Warm Fusion for the Masses: Detailing Virtual Data Structure Elimination in Fully Recursive Languages

Patricia Johann*
John Launchbury†

June 22, 1998

Abstract

In functional programming, small programs are often combined to construct larger, more complex ones. The component reuse encouraged by this modular style of programming yields many benefits, but, unfortunately, modular programs also tend to be less efficient than their monolithic counterparts. Inefficiency is significantly attributable to the construction of intermediate data structures which "glue" together smaller program components into larger ones.

Fusion is the process of removing intermediate data structures from modularly constructed programs. Two particularly successful approaches to achieving fusion in functional languages have emerged in recent years. The first is a *catamorphic fusion* technique based on the promotion theorems of category theory, while the second is a *shortcut* based on parametricity which fuses compositional programs via canned applications of traditional fold/unfold program transformation steps. Both techniques apply only to programs written in terms of certain special language constructs, but each contributes significantly to the elimination of intermediate data structures in such programs. *Warm fusion* combines catamorphic fusion and the shortcut to arrive at a two-step method for preparing various components in program compositions — each of which may be written not in some highly stylized form, but instead in the common recursive style — for one-step inter-functional fusion via the shortcut. In essence, catamorphic fusion is used to "preprocess" programs into compositions to which the shortcut applies.

The purpose of this paper is two-fold. Its primary purpose is to report on a prototype implementation of the warm fusion, together with the maturing of ideas from [LS95] its construction has entailed. Discussion of the prototype provides an ideal opportunity to give a detailed and self-contained treatment of the entire warm fusion method, however, and so the prototype and its properties are investigated within this broader context; in particular, the evolution of the warm fusion method is traced from early work in program transformation. The discussion here builds on the original presentation of warm fusion by Launchbury and Sheard, and resolves a number of outstanding issues from their 1995 paper. Warm fusion is presented here in the context of a fully polymorphic higher-order language, in which the role played by type information — which is essential in eliminating from programs intermediate data structures other than lists — can be made precise. Considerable effort is taken throughout this paper to place all aspects of the warm fusion method on a secure logical foundation.

1 Introduction

Program reuse has long been regarded a desirable goal of software development, and a *modular* approach to programming — in which solutions to complex computational problems are constructed from smaller solutions to simpler such problems — is an effective means of achieving it. Modularity is also associated with greater reliability, clarity, and maintainability of software components, and, since modular programs are typically easier to construct than their monolithic cousins, has been

*Department of Mathematics, Bates College, Lewiston, ME 04240, *pjohann@bates.edu*
†Pacific Software Research Center, Oregon Graduate Institute, P.O. Box 91000, Portland, OR 97291, *jl@cse.ogi.edu*

found to improve programmer productivity as well. Modular program construction is, therefore, widely regarded as an integral part of any reasonable software development process.

One classic mechanism for supporting code reuse and modularity is to arrange for the construction of a variety of widely applicable software components and their storage for later use in a library served by a well-defined application program interface (API). In such a scheme, stored components become available for service as fundamental building blocks in the development of new software artifacts, since new software components can be constructed simply by combining existing ones as appropriate and augmenting the resulting programs as necessary with small amounts of customized code. A number of software companies, Microsoft among them, develop products in precisely this way. They further capitalize on modularity by making individual program components, as well as the larger products they comprise, available to other developers.

The API approach to reuse is a direct outgrowth of the software engineering principle of *product line development*, according to which software solutions to computational problems are deliberately designed in such a way that they can easily be modified to provide solutions to related problems as well. In product line development, an entire family of software systems is built around a common core of modules constructed specifically with long-term variation in mind; each system need, therefore, contain just sufficient customization to meet the requirements of the particular installation. This is quite similar to the manner in which a factory line of cars sharing many common components, and yet also possessing a number of model-specific features, might be built.

Product line development is fairly widespread, and has emerged as a particularly suitable approach to reuse in well-understood and well-defined problem domains. But in fact a considerable amount of software development takes place in domains which meet neither of these criteria. API-based reuse has, therefore, been less effective in practice than originally expected, with a number of factors undermining its promise in more general settings. Perhaps most obviously, program modules resident in existing API libraries need not necessarily be suitable for use in particular applications. It will clearly not be possible, for example, to reuse in a given application modules which lack the functionality that the application demands. On the other hand, modules which have been designed to effect particular computations in rather diverse situations may be too general to perform specific instances of those computations with the required efficiency. Or it may simply be that all existing modules have been defined in accordance with last year's interface specification, and that it is no easier to alter program interfaces than it is to construct entire new software components from scratch.

1.1 Beyond APIs : The Importance of Good Glue

But even when existing modules exhibit appropriate functionality, are written at levels of abstraction suitable for use in a variety of applications, and interface properly with ambient code, there may still be limits to their suitability for a given application. Indeed, modules commonly provide so much functionality all at once that extracting only that which is required for a given application becomes either impractical or impossible. Fortunately, it is possible to focus instead on the reuse of software components, each of which is smaller than a whole module and is designed to provide only a small amount of very specific, well-defined functionality. This is the approach favored in object oriented languages, for instance, which encourage reuse through language constructs — such as object classes and instantiation, class inheritance, polymorphism, genericity, and strong typing — specifically designed to exploit the encapsulation of certain relatively small program fragments.

Merely restricting attention to smaller software components than entire modules is not, however, sufficient to guarantee that significant reuse will be achievable. Successful modular program development depends not only on having code modules — of whatever size — to combine, but also on having effective means for combining them. This point is convincingly made in John Hughes' 1989 paper, "Why Functional Programming Matters" ([Hug89]). There Hughes observes that any programming language which aims to improve programmer productivity must provide strong support for modularity — which he hails as "the key to successful programming"— and that as software becomes increasingly complex, and particularly as the number of programmers required to author and maintain a given software product increases, the well-structuring of software becomes a matter

of utmost importance. But Hughes insists that modularity is important for the solitary programmer, as well. After all, he argues, successfully structuring a solution to a computational problem often requires the ability to conceptualize that problem in terms of subproblems whose individual solutions can be combined to produce its solution. And since the ways in which it is fruitful to conceptually modularize computational problems depends critically on the language constructs provided for modularizing computations themselves, he concludes that a truly useful programming language must support powerful constructs for "gluing together" computations, as well as for performing them.

1.2 Intermediate Data Structures: The Price of Modularity

One very general means of combining computations, and therefore of achieving program modularity, is to construct large programs as compositions of smaller components which communicate via data structures. Each component in such a composition generates a data structure representing its output, and this data structure is immediately consumed by the next component in the composition. This leads to a style of modular programming in which intermediate data structures serve as the primary means of conveying information between — or "gluing together" — component programs, and which is quite prevalent in functional programming, especially in demand-driven *lazy* languages. This style of programming is also seen increasingly in imperative languages, although these do not, at first glance, seem to lend themselves particularly well to it.

Consider, by way of illustration, a routine problem like that of appending one file onto another, searching for all lines in the resulting file which contain the word "jones," sorting these lines according to some predefined ordering, and then displaying the result of this process. While modules coding computational solutions to each of the constituent subproblems (append, search, sort, display) are easily constructed, and although a solution to the larger problem consists precisely in solving these subproblems one after the other and appropriately combining their solutions, there need not exist any way to prescribe the flow of control necessary to achieve their combination. In imperative languages, for example, a solution to each of the relevant subproblems can be realized in terms of a `for` loop, and yet these loops cannot automatically be "composed" in any way to yield a solution of the larger problem. In general, constructing a single loop having the computational effect of a composition of smaller loops requires detailed knowledge of those smaller loops, and so "loop composition" must be achieved manually. The fact that component loops cannot somehow be naively "plugged in" to some ready-made construct which composes them automatically renders the construction of "composed" loops quite difficult to achieve in imperative languages. A programmer using an imperative language would therefore be unlikely even to consider constructing a solution to a computational problem like that given above in terms of individual `for` loops, despite the increases in clarity and, most probably, productivity, to be gained by doing so.

One approach to achieving the effect of module composition in imperative languages is typified by Unix's pipe-and-filter model, in which flow of control is mimicked by explicitly passing data structures between code modules. The Unix command line

```
cat file1 file2  |  grep "jones"  |  sort  |  more
```

for example, uses pipes and filters to generate and pass around data structures to which the individual modules they combine are applied. A single code module solving the originally posed problem is in no way constructed in this process, but the effect is precisely the same as if one had been. That is, automatic composition of modules can be successfully simulated by the pipe-and-filter model of data structure creation and manipulation.

Unfortunately, however, the kind of compositionality engendered by the use of Unix pipes and filters does not come without computational cost. Here, the pipe must exist as a special process, and the data from the output end of each pipe must, in general, be read and copied so that it becomes available for consumption by subsequent code modules. Thus, in addition to requiring a number of procedure calls, command lines like that above also cause many intermediate data structures to be constructed and manipulated. Of course, this is in no way evident from the command lines themselves or from the results of the computations they effect, a rather disturbing fact, given that

the hidden construction and manipulation of data structures exacts considerable overhead, especially if performed on any reasonably large scale.

1.3 Virtual Data Structures

The situation is similar with functional languages. Functional programming languages are well-known to provide powerful means — perhaps most notably function composition and curried higher-order functions — for combining computations, and so support forms of modularity which are difficult to achieve in other programming paradigms. Functional programs are highly compositional, being constructed exclusively as compositions and applications of first- and higher-order built-in functions and smaller programs which are likewise constructed by function composition and application. As does the pipe-and-filter model in Unix, computation in a functional setting relies quite heavily on the manipulation of intermediate data structures. Lists in particular frequently emerge as computational intermediaries; indeed, functional languages strongly encourage a compositional style of programming based on lists by providing large libraries of predefined functions for manipulating them, as well as by supporting special syntax, such as that of list comprehensions, for describing them. Of course, non-list data structures can serve equally well as computational glue in functional programs although — particularly in the case of user-defined types — programmers may have to construct functions for manipulating these by hand.

But whether built-in or user-defined, the construction of intermediate data structures in functional settings incurs the same efficiency costs as in imperative ones. It is therefore precisely because functional languages so strongly encourage and support a compositional style of programming based on intermediate data structures that their proliferation and manipulation emerges as such a serious source of efficiency concerns in functional languages. Each time a list is created in the course of a computation, for instance, each `cons` cell in that list must be allocated, filled, deconstructed, and deallocated, all of which consumes resources and lengthens execution time. Thus, whenever an intermediate data structure is constructed by one component of a program at one stage of a compositionally specified computation only to be consumed by another at the next stage, a considerable amount of work which is not essential to the result of the computation is being performed. For this reason, computations specified compositionally often consume far more time and space than do their noncompositional counterparts, and than is, strictly speaking, necessary.

To see how such a situation can arise, consider the following simple program for computing the sum of the first n squares[1]:

```
sos n = sum (map sqr(upto 1 n)).
```

Here, `upto` p q generates a list whose elements are the integers $p, p+1, ..., q$, `sqr` denotes the squaring function, and `sum` adds the (numerical) elements of a list.[2] This implementation of the sum-of-squares function is straightforward, modular, and easily maintainable. The only drawback is that, although it computes a function over natural numbers, it *literally* constructs, traverses, and discards *two* intermediate lists. In particular, `sos` executes significantly more slowly than the following accumulating-parameter version of the sum-of-squares function, in which parts from the component functions of `sos` are intermingled:

```
sos' n =
  let h m n a  =  if m > n
                    then a
                    else h (m + 1) n (m² + a)
  in
    h 1 n 0
  end
```

No intermediate lists at all are processed in `sos'`. Moreover, the local function `h` is tail recursive and so can be implemented by a loop using only constant stack space. But this computational success has been achieved at the cost of the clarity, conciseness, and modularity of the original implementation.

[1] All programs in this paper will be written in Haskell.

[2] Definitions of `sum`, `map`, and `upto` are given in Section 2.4.

The above example illustrates that although the kind of compositional program development which functional languages naturally support does indeed boast all of the benefits usually associated with modular programming, the explicit construction of intermediate data structures simply to bridge cleanly separated program components can — and often does — result in programs which are considerably less efficient than corresponding ones in which pieces from different components are intermingled. Fortunately, however, such data structures appear as neither inputs nor outputs of the top-level programs whose construction they facilitate, and so need have no actual, physical manifestation. That is, although they are abstractions which are extremely useful to programmers in structuring programs, these data structures need not actually exist. It is precisely because they play no essential computational role in the execution of programs that these "unnecessary" data structures are often referred to as *virtual* data structures.

1.4 Modularity vs. Efficiency: An Ideal Solution

Ideally, then, we would like to retain the benefits of programs like `sos` while achieving the efficiency of those fashioned after `sos'`. More specifically, we wish to enable programmers to program in the compositional style without sacrificing computational efficiency by arranging for compilers to automatically transform compositions of program components to eliminate the virtual data structures which glue them together. Of course, since some data structures do actually serve true computational purposes — i.e., since some data structures are *not* virtual — we should not anticipate the elimination of *all* intermediate data structures from computations. We may reasonably expect, however, that "unnecessary" virtual data structures are removed from computations whenever possible, and that their removal is accomplished automatically by optimizing compilers. Unfortunately, few currently existing compilers are capable of performing such program transformation.

To achieve this goal, fully automatic techniques for removing virtual data structures from compositionally defined programs are needed. It is significant, then, that program components can often automatically be *fused* together to yield functionally equivalent programs containing no such data structures. When this is possible, programmers may feel liberated to write in a compositional style, confident that compilers can remove virtual data structures from their programs, and so eliminate performance penalties associated with their increased expressiveness. Because of its potential for automatic program improvement, a substantial number of papers on virtual data structure elimination, or program *fusion*, has appeared in the literature over the past twenty years. While initially exclusively the province of functional language researchers, fusion is increasingly of interest to those working in more traditional languages as well. But whether intended to effect the improvement of functional or imperative programs, each fusion technique addresses the long-standing tension between program modularity on the one hand, and program efficiency on the other, precisely by describing techniques for transforming modular programs into more efficient, fused, functionally equivalent ones.

It has recently been demonstrated that programs which can be represented in a highly stylized form, called `build-cata` form, are particularly amenable to fusion. Two fundamental observations underlie this result. The first is that functions which consume (finite) data structures in a uniform manner can be written in terms of a language construct, `cata`, which captures a kind of "regular recursion" over elements of a datatype by replacing each datatype constructor in those elements with an appropriate function. The second is that functions which produce (finite) data structures in a uniform manner can be written in terms of a dual construct, `build`, which applies to the constructors of a datatype the results of abstracting uniform data structure-producing functions with respect to them. Once a program component has been written in `build-cata` form, virtual data structures on input or output can be eliminated in one program transformation step ([GLPJ93]). This transformation describes one precise way in which compilers can take advantage of regularity in the production and consumption of data structures to optimize programs which manipulate them.

Writing simple data structure-consuming and -producing functions in `build-cata` form is, in fact, reasonably straightforward. But, unfortunately, most programs are not written in `build-cata` form, since doing so is not particularly convenient for programmers who have grown accustomed to using iterative constructs and recursion, and who seek the benefits of modular programming mentioned

above. To accommodate this majority of programmers, a "preprocessing" phase is required to prepare programs written in the common recursive style for one-step fusion via the shortcut by transforming them into `build-cata` forms. Although transforming arbitrary recursive programs into `build-cata` form appears to be quite a difficult problem, in a 1995 paper entitled *Warm Fusion: Deriving Build-Catas from Recursive Definitions* ([LS95]), Launchbury and Sheard describe a method for transforming a large class of programs computing recursively defined programs into `build-cata` forms. It is precisely the "preprocessing" method described in [LS95] for recursively defined programs which is the subject of this paper (see Section 1.5).

At this point, some benefits of program fusion other than program improvement are worth mentioning. We have already discussed at some length the ways in which the availability of effective fusion tools can encourage programmers to develop large software artifacts by constructing, and then combining, smaller program components while avoiding the degradations in program efficiency usually associated with modularity. Components to be combined are typically designed specifically to exhibit high degrees of internal cohesion and to present uniform interfaces to facilitate combination with other such components. But when modules don't interface well, fusion can play an additional role in their effective combination. In such cases, a fusion algorithm can act as a sort of "impedance matcher," since transformations can be constructed for mapping the intermediate data structures manipulated by these modules into intermediate forms which both modules understand, and the fusion engine can then be employed to remove the inefficiencies inherent in the translations.

Finally, programs written in the compositional style are easily decomposed into key components, and this can often highlight similarities and relationships between computations which might otherwise not be apparent. A number of superficially quite different graph algorithms, for example, can each be expressed as depth-first graph traversals with some data accumulation at each node; such algorithms can therefore be realized as compositions of very similar program components. Ample evidence suggests that when computations can easily be structured as compositions of basic program components whose behaviors are well-understood, algorithmic solutions to computational problems are more readily constructed. When such compositions are, in addition, amenable to it, fusion can provide insight into more efficient, if also less transparent, renderings of these solutions. In such situations, fusion can provide a tool for answering the question, "Where do algorithms come from?"

1.5 Overview of This Paper

This paper presents a detailed, comprehensive, and self-contained treatment of the warm fusion method, tracing its evolution from early work in program transformation, and situating it clearly within the larger context of virtual data structure elimination. The discussion here is considerably more robust than that in [LS95], and a number of outstanding issues from that paper are resolved. In particular, the important role of type information in warm fusion is recognized by considering the method in the context of a fully polymorphic higher-order language, and the application of the method to the removal of virtual data structures other than lists are explored in detail. That the ideas originally put forth in [LS95] have matured into the basis of a realizable and fully automatic algorithm for program fusion is also demonstrated, as this paper reports in addition on a prototype implementation of a warm fusion-based virtual data structure elimination method.

Having already provided a context for the study of virtual data structure elimination in general, and the warm fusion method in particular, the remainder of this paper is organized as follows. Section 2 describes the technical precursors to [LS95], from the fold/unfold transformations on which the earliest program improvement methods were based, through Wadler's work on listlessness and, later, treelessness, to the shortcut and catamorphic fusion rules which are so central to warm fusion. Section 3 details the warm fusion method itself, offers examples of the kinds of program transformation it can effect, provides a solid theoretical grounding for the method, and describes our implementation. A comparison of warm fusion with recently developed techniques for fusing hylomorphisms is given in Sections 4.5 and 4.6, while other related work is discussed in the remainder of Section 4. Section 5 concludes with some suggestions for further research.

2 A Technical Context for Warm Fusion

There have, in fact, been some attempts to add a pipe-and-filter model like that described in Section 1.3 to languages like Pascal and C, the latter quite recently ([Wat91]). Essential to this work have been efforts to completely transform pipes away, so that constituent loops are effectively composed. Interestingly, however, the key ideas on which such transformations are based have existed in the functional programming community from the mid 1970's, when Burstall and Darlington first brought them to light in their work on improving functional programs via fold/unfold transformations.

In the intervening two decades, fold/unfold transformations have played a significant role in work on supercompilation and partial evaluation (see Section 4.1), as well as in techniques for eliminating virtual data structures. Since the first conception of such transformations as means of eliminating computational overhead associated with composition in functional languages, a substantial body of work on new and better methods for achieving virtual data structure elimination has appeared in the literature. Fusion work in the functional programming community has since proceeded apace, and list fusion is now a standard optimization in the Glasgow Haskell Compiler (GHC). It is precisely because all virtual data structure elimination techniques to date have been, fundamentally, alternative characterizations, refinements, or extensions of the basic fold/unfold process, that we begin our discussion of warm fusion by rehearsing Burstall and Darlington's seminal paper.

2.1 First Steps Toward Automating Program Fusion

In their 1977 paper, "A Transformation System for Developing Recursive Programs" ([BD77]), Burstall and Darlington demonstrated that an instantiate/unfold/simplify/fold transformation process can be used to eliminate virtual data structures from programs written in the form of higher-order recursion equations. Their system was designed as part of an overall effort to help programmers write clear and correct programs which are easily altered. Recognizing that this goal is often achieved only at the high price of computational inefficiency, Burstall and Darlington concerned themselves with the question of how "a lucid program can be transformed into a more intricate but efficient one in a systematic way, or indeed in a way which could be mechanized."

The program transformation system presented in [BD77] is systematic, but unfortunately is not fully automatable. In fact, it requires human guidance for two reasons. First, the method requires user-supplied *eureka* steps during the folding of function calls. *Eureka* steps are so named because they often require creativity and a deep understanding of the particular program being transformed; on the other hand, they can also contribute significantly to the power of a program transformation system, effecting algorithmic changes such as can turn, for example, an algorithm of quadratic complexity into one of linear complexity. Secondly, the transformation method of Burstall and Darlington necessitates keeping track of which function calls have occurred, and uses clever control mechanisms to avoid the possibility of performing infinite sequences of transformations by repeatedly unfolding function definitions. Although the kind of bookkeeping required is well-understood, it introduces substantial cost and complexity into the method. Furthermore, given an arbitrary program, it is entirely possible that no pattern of function calls will repeat itself, so that the resulting infinite unfolding of function definitions will lead to the nontermination of any compiler into which fold/unfold transformations are incorporated. And neither is the correctness of fold/unfold transformations guaranteed: folding function definitions can introduce recursion into transformed programs, which may therefore have termination properties quite different from the programs from which they came.

Burstall and Darlington address neither the issue of correctness nor that of termination for the transformation system in [BD77]; indeed, that paper is offered specifically as an empirical study of the kinds of program improvement which are achievable using their system. In spite of this — or perhaps because of it — the search for fully automatable, correct, and terminating program improvement transformations for languages both natural and expressive enough to be of practical interest has been the impetus behind most work in virtual data structure elimination to date. This search has not only led to the development of calculation-based program transformations (e.g., [Fok92], [Jeu93],

[Mee92], [Mei92]), but has inspired more general work on fold/unfold transformations as well (see, e.g., [Amt92], [San94]). Those investigations which have most directly impacted the development of the warm fusion method are described in the next subsections.

2.2 Listlessness and Deforestation

Wadler's work on deforestation, which grew out of his earlier work on listlessness, provides one example of a program improvement algorithm based on fold/unfold transformations. The original listless transformer ([Wad83]) used many of the same ideas that Turchin used in developing his supercompiler (see section 4.1) to design a semidecision procedure for transforming recursive programs which can be lazily evaluated in constant bounded space into equivalent *listless* programs resembling finite automata. The transformations comprising the listless transformer were not, however, source-to-source, since they actually transformed functional programs into equivalent imperative ones.

Wadler later realized that his program transformation techniques could be expressed entirely within the functional language with which he was working, and subsequent modification of the listless transformer led to a decision procedure for converting compositions of listless programs into single listless programs provided the semantic condition of preorder traversal could be verified for each of the component programs ([Wad86]). Refining his ideas on listlessness even further, Wadler shortly thereafter developed a set of program transformations capable of eliminating a considerably larger class of tree-like intermediate data structures from certain programs specified in first-order functional languages ([Wad90]). In order to make precise which intermediate data structures his transformations could remove, a *treeless form* for function definitions was specified. Treeless forms are easy to identify syntactically and, unlike listless forms, do not require the verification of a semantic condition. Moreover, whereas functions with listless forms must evaluate lazily in constant bounded space, those with treeless forms may use space bounded by the depths of their intermediate trees. On the other hand, treelessness requires not only that function definitions are first-order, but also that all variables are used linearly and that no internal data structures are permitted (no nested applications of functions are allowed). Due to the nature of the restrictions defining listless and treeless forms, direct comparison of their computational power is not possible.

A set of explicit transformations on compositions of treeless forms, together with implicit, but essential, fold steps in which previously encountered expressions are recognized, comprise the *treeless transformer*. The virtual data structure elimination method they induce was dubbed *deforestation* early on, and, for better or for worse, the pun has stuck. In fact, "deforestation" is often somewhat erroneously used to refer to the general process of eliminating virtual data structures from programs. In this paper, however, "deforestation" will designate only the virtual data structure elimination technique embodied by the treeless transformer; the unqualified word "fusion" will be used here to refer to the process of eliminating virtual data structures from programs more generally.

The core of the treeless transformer consists of seven rules. Four of these focus on expressions built using the `case` construct, but the only rules which actually remove computation from programs are those for unfolding function definitions appearing either as selectors of case expressions or at the top level of programs, and the "case-on-constructor" rule. Using the latter, the expression

```
case (C2 t1 t2 t3) of
    C1 x1 x2       → e1
    C2 x1 x2 x3    → e2
    C3 x1          → e3
```

for example, is transformed into

```
let  x1  =  t1
     x2  =  t2
     x3  =  t3
in
     e2
```

Note that the heap object represented by the constructor `C2` is neither built nor examined in the resulting expression. Indeed, it is precisely in applications of the case-on-constructor transformation that intermediate data structures are eliminated. All of the other transformation rules, including those for unfolding function definitions, exist merely to generate opportunities for applying this one.

As with both the transformation system of Burstall and Darlington and the original listless transformer, it is possible to perform infinite sequences of treeless transformations by repeatedly unfolding function definitions; this is the case particularly if no restrictions are imposed on the structure of input programs to the treeless transformer. This difficulty is alleviated for the treeless transformer in precisely the same way as described in Section 2.1 for the original fold/unfold transformations, i.e., by keeping track of which function calls have occurred previously and generating suitable recursive definitions when calls are repeated. Unsurprisingly, this solution has precisely the same associated costs in the first-order setting that it does in the higher-order one in which Burstall and Darlington worked.

While treeless form is a particularly restrictive form of function definition which has the unfortunate consequence of severely limiting the applicability of Wadler's algorithm, the restrictions it embodies do permit some important properties of the treeless transformer to be established. First, although the treeless transformer applies only to programs which are compositions of programs in treeless form, it is fully automatable and does apply to *all* such programs. Secondly, the composition of programs in treeless form must always be another program in treeless form. This guarantees that no new intermediate data structures are constructed during deforestation, so that compositions are effectively fused. Third, the restrictions on input programs to the treeless transformer are essential to the establishment of Wadler's *efficiency theorem,* which ensures that the performance characteristics of deforested programs are no worse than those of the programs from which they are derived. That it not degrade the performance of acceptable input programs is, of course, critical to any practical virtual data structure elimination method. Finally, it has been shown that applications of Wadler's transformations to compositions of treeless forms always terminate ([FW88]).

Observing that intermediate data structures of atomic types need not be removed from programs, Wadler ([Wad90]) introduced a second treeless form for programs, called *blazed treeless form.* Blazed treeless form is defined in terms of an annotation scheme which marks each atomic-type subexpression with $\ominus$ and each tree-type subexpression with $\oplus$; the intention is that intermediate trees marked with $\oplus$ should be eliminated whenever possible, while those marked $\ominus$, since they are in some precise sense "small," can remain without computational penalty. The latter are abstracted using a `let` constructor and transformed independently. As a result, blazed treeless forms need only be linear in variables blazed $\oplus$.

The blazed treeless transformer is obtained by augmenting the (pure) treeless transformer with four additonal transformations. Two of these supersede the transformations involving unfolding of function definitions in cases where the result and all arguments of the function are blazed with $\ominus$, and the other two manage occurrences of `let`. The blazed treeless transformer can be seen to deforest compositions of blazed treeless forms just as the (pure) treeless transformer does for compositions of pure treeless forms, and an efficiency result analogous to that for the pure treeless transformer has also been established.

A fully automatic version of the pure treeless transformer has been implemented by Wadler, although a blazed counterpart is not currently available. There have been attempts to extend Wadler's deforestation techniques to programs in higher-order languages ([Chi90], [Mar95]), but this seems to be quite difficult and many problems remain to be solved.

2.3 A Practical Approach to Fusion

Wadler's algorithm for virtual data structure elimination based on fold/unfold transformations restricts attention to a syntactically distinguished class of first-order programs written in the common recursive style. A quite different approach to deforestation is explored by Sheard and Fegaras ([SF93]). Like Burstall and Darlington, Sheard and Fegaras work in the context of a higher-order language, but they reject efforts to eliminate virtual data structures from arbitrary programs, and focus instead on the fusion of programs built up only from certain constructors designed to repre-

sent "regular recursion" over data structures. In particular, the language of [SF93] does not support general recursion.

The main recursion operators with which Sheard and Fegaras work are called *catamorphisms*; their list versions are known variously as `fold` in ML, `reduce` in early versions of Miranda, and `foldr` in Haskell. Catamorphisms are instances of more general structure-preserving operators called *hylomorphisms*, which have their origins in the study of *constructive algorithmics* ([Fok92]). In constructive algorithmics, (regular) datatypes are defined categorically in terms of initial algebras associated with certain functors, functions between datatypes are represented as hylomorphisms, and the structure that hylomorphisms impose on program components becomes the basis for fusing them together. Sheard and Fegaras capitalize on the properties of catamorphic programs to define a *normalization algorithm* which is capable of eliminating virtual data structures from them. This normalization algorithm transforms compositions of catamorphic programs (which are strong analogues of Wadler's treeless programs) into new programs which are canonical, in the sense that they contain the smallest possible number of catamorphisms. Normalization of programs is achieved by repeatedly applying transformations encoding the catamorphic Promotion Theorem (see Theorem 3.13) to fuse nested compositions of two catamorphisms into single catamorphisms; the Promotion Theorem describes the precise conditions under which catamorphisms can be so fused. Definitions for other recursive forms, such as generalized catamorphisms and primitive recursion, are given together with their corresponding promotion theorems, but normalization algorithms for these forms are not supplied.

The key observation underlying Sheard and Fegaras' approach to program fusion is that program optimization techniques must search for syntactic structure in programs to ensure that particular transformations are applicable, and so rendering the structure of programs more explicit than implicit makes the job of the transformation engine considerably easier. Accordingly, their normalization algorithm calculates program improvement based on the explicit *local* structure of catamorphic programs, rather than relying on more global program analysis to discover their implicit structure and determine applicability of transformations. In particular, keeping track of function calls, as must Burstall and Darlington, and even Wadler, is not necessary for normalization. In addition to insuring that no global program analysis is required to determine applicability of the normalizing transformations, the restriction to catamorphic recursion in [SF93] guarantees termination of programs obtained by normalization. Moreover, since all programs expressible in the language of [SF93] necessarily terminate, the correctness of the normalization algorithm is immediate.

But even when attention is restricted to languages permitting only catamorphic recursion, a problem arises in practice: without user-supplied guidance, the normalization algorithm unfolds all function definitions in order to propagate producer information to consumer sites. Because programs contain only very controlled recursion, this unfolding always seems to terminate, but it can lead to rather severe combinatorial explosion: the normalization algorithm of Sheard and Fegaras actually requires exponential effort to deforest an arbitrary legal input program because fusion of each pair of catamorphic program components is attempted. And, since the transformations entail no guarantee against code duplication, program performance can actually be substantially degraded by normalization.

Despite these difficulties — and although unaccompanied by any formal guarantee of termination — the normalization algorithm of Sheard and Fegaras has proved to be considerably more practical than its predecessors. The results of [SF93] demonstrate convincingly that catamorphic fusion can in fact be achieved using fully automated tools, and so really is suitable for incorporation into real compilers. Although Sheard did develop a catamorphic fusion tool based on the normalization algorithm, it relied rather heavily on user-supplied guidance; a fully automatic version of the normalization algorithm has, unfortunately, never been implemented.

2.4 A Shortcut to Deforestation

At about the same time that Sheard and Fegaras were developing their normalization algorithm, Gill, Launchbury, and Peyton Jones came up with another solution to the difficulties associated with applying fold/unfold transformations to recursively defined higher-order programs. In *A Short Cut to Deforestation* ([GLPJ93]), a one-step fusion algorithm is given which completely by-passes the

combinatorial explosion problem of [SF93]. Like Sheard and Fegaras' normalization algorithm, the shortcut is a calculation-based, rather than a search-based, program improvement technique, and it relies on programs to be transformed being written in a highly stylized form called `build-cata` *form*. Although it is in no way realistic to expect applications programmers to write code in such a form, all of the list-processing functions from the Haskell standard prelude have actually been hand-coded in this style. This makes it possible for GHC to automatically fuse compositions of standard functions, such as the function `sos` from Section 1.3, via the shortcut.

The use of the shortcut to remove intermediate lists from programs is more practical than some earlier deforestation techniques, but it is also somewhat less purist. The shortcut does not guarantee the removal of *all* intermediate data structures from programs (indeed, this is provably impossible to achieve), but, like the treeless transformer, it does accept all legal programs as input. Moreover, termination of the method is guaranteed. In fact, termination of the shortcut method is trivial, since recursive function definitions are not explicitly unfolded. Instead, a similar effect is achieved by performing one or more algebraic transformations on higher-order programs, each of which can be regarded as a canned application of the fold/unfold transformations prevalent in earlier and more traditional virtual data structure elimination techniques. In the discussion of the shortcut in this section, we follow [GLPJ93] and focus initially on its use in removing only intermediate lists from compositionally defined programs. Later, in Section 3.2, we describe how the shortcut can be extended to accommodate the elimination of arbitrary algebraic virtual data structures.

Short Cut relies on two fundamental observations to describe how programs represented in `build-cata` form can be fused in a single transformation step. The first is that programs which consume lists in a uniform manner can always be written in terms of the language construct `cata`, which, as its name suggests, captures the notion of a catamorphism. Operationally, given an appropriate function $\oplus$, a value `z`, and a list `xs`, `cata` replaces all occurrences of `cons` in `xs` with $\oplus$ and replaces the occurrence of `nil` at the end of `xs` by `z`. The second, new insight offered by [GLPJ93] is that programs which produce lists in a uniform manner can always be written in terms of a construct called `build`, which is dual to `cata` in a sense to be made precise below. `Build` acts essentially by applying its functional argument to the list data constructors `(:)` and `[]`, i.e., `build` satisfies `build g = g (:) []` for every `g`. Thus,

$$\texttt{build}\ (\backslash \texttt{c}\ \texttt{n} \rightarrow \texttt{c}(7, \texttt{c}(3, \texttt{n}))),$$

for example, applies the function argument to `build` to the list constructors `(:)` and `[]`, thereby reconstructing the list `7:(3:[])`. The `cata-build` rule

$$\texttt{cata}\ (\oplus)\ \texttt{z}\ (\texttt{build}\ \texttt{g}) = \texttt{g}\ (\oplus)\ \texttt{z},$$

which relates the list-consuming construct `cata` and the list-producing construct `build`, can be thought of as describing one precise way in which compilers can take advantage of regularity in the production and consumption of lists to achieve one-step fusion of programs.

Our sum-of-squares example makes clear the benefit of putting programs in `build-cata` form. It is not hard to see that the list-consuming function `sum` can be written as

$$\texttt{sum}\ \texttt{xs} = \texttt{cata}\ (+)\ 0\ \texttt{xs},$$

the list-producing function `upto` can be written as

```
upto low high  =  build (upto' low high)
                    where upto' low high c n =
                            if low < high
                            then n
                            else c low (upto' (low + 1) high c n),
```

and the function `map`, which both produces and consumes lists, can be written in `build-cata` form as

$$\texttt{map}\ \texttt{f}\ \texttt{xs} = \texttt{build}\ (\texttt{map}'\ \texttt{f}\ \texttt{xs}),$$

where

$$\texttt{map'\ f\ xs} = (\backslash \texttt{c\ n} \rightarrow \texttt{cata}\ (\backslash \texttt{a\ b} \rightarrow \texttt{c\ (fa)\ b})\ \texttt{n\ xs}).$$

Denoting the squaring function by `sqr`, as usual, the sum-of squares function `sos` from Section 1.3 can therefore be written as

$$\begin{aligned} \texttt{sos} &= \texttt{sum\ (map\ sqr\ (upto\ 1\ n))} \\ &= \texttt{cata\ (+)\ 0\ (build\ (map'\ sqr\ (upto\ 1\ n)))} \end{aligned}$$

This stylized version of `sos` can now be fused using the `cata-build` rule, as follows:

$$\begin{aligned} \texttt{sos} &= \texttt{cata\ (+)\ 0\ (build\ (map'\ sqr\ (upto\ 1\ n)))} \\ &= \texttt{map'\ sqr\ (upto\ 1\ n)\ (+)\ 0} \\ &= \texttt{cata\ (\backslash a\ b\ \rightarrow\ (sqr\ a)\ +\ b)\ 0\ (upto\ 1\ n)} \\ &= \texttt{cata\ (\backslash a\ b\ \rightarrow\ (sqr\ a)\ +\ b)\ 0\ (build\ (upto'\ 1\ n))} \\ &= \texttt{upto'\ 1\ n\ (\backslash a\ b\ \rightarrow\ (sqr\ a)\ +\ b)\ 0.} \end{aligned}$$

Inlining the parameters `c`, `n`, and `high` in the definition of `upto'`, gives a version of `sos` which produces no intermediate data structures:

```
sos n = upto'' 1
          where upto'' low =
            if low < n
            then 0
            else (sqr low) + (upto'' (low + 1)).
```

(A more elaborate scheme described in [LS95] will derive the tail recursive version given in Section 1.3.)

Generally speaking, the shortcut causes the `cata`s at the input ends of list-consuming programs to "cancel" with the `build`s at the output ends of list-producing programs. This cancellation may in turn bring together other `cata-build` pairs for fusion, and so on, since nested list-transforming programs give rise to compositions of `build`s and `cata`s in essentially the following pattern:

...cata][build cata][build...

Such compositions can be reassociated to yield compositions of the form

...[cata build][cata build]...,

in which the inner `cata-build` pairs "cancel" by application of the `cata-build` rule.

The only problem with the `cata-build` rule as just described is that it is, in fact, false. As observed in [GLPJ93],

$$\texttt{cata}\ (\oplus)\ \texttt{z}\ (\texttt{build}\ (\backslash \texttt{c\ n}\ \rightarrow\ [\texttt{True}])) \not\equiv (\backslash \texttt{c\ n}\ \rightarrow\ [\texttt{True}])\ (\oplus)\ \texttt{z},$$

for instance: using the definitions of `cata` and `build`, it is easily seen that the left-hand side of this equation evaluates to $\oplus$ `True z`, whereas its right-hand side evaluates to [`True`]. These two expressions need not even have the same type, let alone be identical! The problem here is that the functional argument to `build` constructs its result without using `c` and `n`, and so cannot possibly satisfy the criteria insuring correct application of the `cata-build` rule. In fact, the `cata-build` rule holds only when the argument to `build` is a *uniform* list-consuming function, i.e., when the argument to `build` is obtained by uniformly abstracting a list over all its list constructors.

Fortunately, the kind of uniformity required for correct application of the `cata-build` rule can be guaranteed simply by restricting the types of the arguments to `build`. Indeed, the precise version of the `cata-build` rule for lists is a direct consequence of the "free theorem" ([Rey83], [Wad89]) for the type $\forall\beta.(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta$ which correct applications of the `build-cata` rule require functional arguments to `build` to have. The intuition here is that if g has type $\forall\beta.(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta$,

then, because g is polymorphic in β and has result type β, it can only manufacture its result from its arguments $\oplus$ and `z`. But since $\oplus$ must have type $\alpha \to \beta \to \beta$ and `z` must have type β, we see that the only expressions of type β which can be constructed from them are of the form

$$(\oplus\ \mathtt{a}_1\ (\oplus\ \mathtt{a}_2\ \ldots\ (\oplus\ \mathtt{a}_n\ \mathtt{z})\ldots)).$$

For a language like Haskell, in which explicit type abstraction and application is suppressed, this intuition is made precise in the following theorem from [GLPJ93]:

Theorem 2.1 *If α is a fixed type and $g : \forall\beta.(\alpha \to \beta \to \beta) \to \beta \to \beta$, then*

$$\mathtt{cata}\ (\oplus)\ \mathtt{z}\ (\mathtt{build}\ \mathtt{g})\ =\ \mathtt{g}\ (\oplus)\ \mathtt{z}.$$

The correctness of the `cata-build` rule below is an immediate consequence of this theorem. In fact, Theorem 2.1 guarantees that as long as `build` is applied only to functions of the appropriate type, fusion via the `cata-build` rule may proceed with complete security. Moreover, since the `cata-build` rule preserves the types of expressions, the type-correctness of programs obtained from applications of the `cata-build` rule can be insured simply by arranging that when programs are put into `build-cata` form, the introduced `build`s and `cata`s are type-correct.

The ideal way to proceed in an implementation would thus be to define `build` (for lists) to have the type

$$\forall\alpha.(\forall\beta.(\alpha \to \beta \to \beta) \to \beta \to \beta) \to \mathit{List}\ \alpha,$$

and then to have the compiler's type checker confirm the well-typedness of all applications of `build` in programs. Unfortunately, however, languages with type systems which permit local quantification do not support type inference. In GHC, the issue of correctness is effectively circumvented; as described at the beginning of this subsection, all of the standard list-processing functions have been hand-coded as `build-cata` forms, so that both the need and the opportunity for programmers to introduce `build`s into programs themselves, and their attendant difficulties, are eliminated. User introduction of `build` is similarly proscribed in our prototype warm fusion engine. Some of the issues raised by the incorporation of the `cata-build` rule into GHC are mentioned briefly in Section 4.3, and the topic is discussed quite thoroughly in [Gil96].

There are instances of `cata` and `build` for algebraic datatypes other than lists, and results analogous to the ones described in this section do indeed hold for them. To make these results precise, it is necessary to describe accurately the conditions under which more general instances of the `cata-build` rule hold, i.e., to express the type restrictions on the functional arguments to arbitrary instances of `build`. Doing this turns out to require the expressivity of a fully polymorphic higher-order language.

In the next section, we describe how the warm fusion method builds upon Sheard and Fegaras' normalization algorithm and the shortcut of [GLPJ93] to provide a virtual data structure elimination method for programs expressing certain recursively defined higher-order polymorphic functions. Particular care is taken to put the warm fusion method on a theoretical foundation secure enough to support its implementation. The section concludes with a discussion of our prototype fusion engine.

3 The Warm Fusion Method

Fusing compositions of standard programs in `build-cata` form via the `cata-build` rule is reasonably straightforward. The difficulty comes in preparing programs written in the common recursive style for fusion via the shortcut by transforming them into `build-cata` form. In fact, transforming arbitrary recursive programs into `build-cata` form is quite a difficult problem, and so, therefore, is eliminating virtual data structures from compositions of them.

One method for automatically deriving equivalent programs in `build-cata` form from recursively defined programs was nevertheless recently reported. In the 1995 paper *Warm Fusion: Deriving Build-Catas from Recursive Definitions* ([LS95]), Launchbury and Sheard demonstrate that a simple syntax-to-syntax translation introducing appropriate `build`s and `cata`s into programs, together

with a generalization of the catamorphic fusion technique of Sheard and Fegaras which is capable of fusing compositions of *arbitrary* (strict) functions with the newly-introduced catamorphisms, can form the basis of a workable method for transforming a large class of recursively defined programs into `build-cata` forms. Their warm fusion method thus yields a two-phase process for eliminating virtual data structures from compositions of recursively defined programs. In the first phase, the programs in a composition to be fused, each of which is typically written in the common recursive style, are "preprocessed" into `build-cata` form via warm fusion. In the second phase, `build-cata` forms resulting from the first phase are assembled into compositions corresponding to the original ones, and these then undergo fusion via the `cata-build` rule from [GLPJ93]. This two-step approach to program fusion can be seen as bringing together the algebraic strand and the recursion equation strand of previous methods. In fact, it seems rather remarkable that techniques for eliminating virtual data structures from catamorphic programs can be used to achieve the same for many programs in languages which express recursion equationally as well.

Like the normalization algorithm of Sheard and Fegaras and the shortcut of [GLPJ93], warm fusion is calculation-based rather than search-based, and so at no time in the warm fusion process must arbitrary patterns of recursive calls be spotted. Instead, it is necessary only to detect recursive calls to the function currently being processed, and so warm fusion is achievable exclusively by local transformation. The scope of warm fusion is limited to the body of the program being processed, and warm fusion is, in fact, incapable of performing virtual data structure elimination across function boundaries (the shortcut is used, perhaps repeatedly, for this once programs in a composition of interest have been preprocessed into `build-cata` forms by warm fusion). But, as pointed out in [LS95], it is often possible to take advantage of having already transformed into `build-cata` form definitions of functions called by programs currently being processed. Some interaction between the warm fusion and shortcut phases of program fusion will thus naturally result from this optimization, but within the lifecycle of any given program, these two phases are distinct.

3.1 Illustrating Warm Fusion

By means of illustrating the warm fusion process, consider the problem of deriving the `build-cata` form in Section 2.4 from the usual recursive definition of `map :: (a → b) → List a → List b` as

```
map = \f xs → case xs of
                Nil → Nil
                Cons z zs → Cons (f z) (map f zs).
```

Recalling that `build` constructs data structures and `cata` consumes them, we observe that only programs which consume input data structures to produce new output data structures can possibly have `build-cata` forms. When such a program takes as input a number of arguments, all parameters other than the one actually being consumed in the program are essentially carried along throughout the processing by the fusion engine. It is therefore the *body* of the program — obtained by deleting the parameters not consumed — upon which transformations are actually performed. In the case of *map*, it is

```
body_map = \xs → case xs of
                   Nil → Nil
                   Cons z zs → Cons (f z) (map f zs)
```

— which codes a function from `List a` to `List b` — that is the program of primary concern; the parameter `f` of *map* is automatically carried along explicitly by the fusion engine during processing. Note that we may now express our original definition of *map* quite succinctly as

```
map = \f xs → body_map xs.
```

The warm fusion process for all programs begins with the observation that, according to the defining properties of `build` and `cata`,

```
build (\c n → cata c n ls) = ls
```

and

$$\texttt{cata (:) [] ls} = \texttt{ls}$$

always hold. Composing the body of the program to be processed on the left and on the right by these highly-stylized identity functions yields an equivalent definition of that program. The resulting program for *map* is

$$\texttt{map} = \backslash \texttt{f xs} \rightarrow \texttt{build} (\backslash \texttt{c n} \rightarrow \texttt{cata c n} (\texttt{body}_{\texttt{map}} (\texttt{cata (:) [] xs})));$$

for a general program

$$\texttt{foo} = \backslash \texttt{x}_1 ... \texttt{x}_n . \texttt{body x}_i,$$

composition gives

$$\texttt{foo} = \backslash \texttt{x}_1 ... \texttt{x}_n \rightarrow \texttt{build}(\backslash \texttt{c}_1 ... \texttt{c}_k \rightarrow \texttt{cata c}_1 \ldots \texttt{c}_k (\texttt{body} (\texttt{cata d}_1 \ldots \texttt{d}_l \texttt{ x}_i)),$$

where $\texttt{c}_1, ..., \texttt{c}_k$ are parameters corresponding to the data constructors of the result type of `foo` and $\texttt{d}_1, ..., \texttt{d}_l$ are the data constructors of the input type for `foo`.

These compositions have the effect of introducing a `build-cata` pair on the left of the body being processed, and a particular catamorphism — namely, the appropriate *copy function* — on its right. The introduced `build` will appear in the derived `build-cata` form for the body being processed; its purpose is to provide a functional representation of the data structure produced by the body by abstracting over the constructors of the result datatype. The appropriate version of `build`, as well as of its `cata` partner, is therefore completely determined by the result type of the data structure-producing program being processed. For the uniform list-producer `map :: (a -> b) -> List a -> List b`, the appropriate version of `build` is the one which produces data structures of type `List b`, and the appropriate version of `cata` is the one which consumes those same data structures.

The introduced copy function has a different purpose altogether. In order to write in `build-cata` form the body of a program being processed, we need to replace by a *single* catamorphism an application of the form

$$\texttt{cata c n (body (cata (:) [] xs))}.$$

Composing `body` on the right with the copy function makes possible the application(s) of catamorphic fusion which may accomplish that task. Observe, however, that `body` and the copy function must have the same domain, so that the particular copy function introduced in processing a function definition is completely determined by the domain type of its body. In the case of *map*, the appropriate copy function therefore maps elements of the datatype `List a` to elements of the datatype `List a`. When the fusion rewriting process succeeds in transforming a composition such as that above into a new catamorphism, the result is a program in `build-cata` form which is completely equivalent to the original one, and which can therefore replace the original in compositions eligible for virtual data structure elimination via the shortcut. Otherwise, the input program can be returned unchanged by the fusion engine, since expressions involving calls to it will not participate in any `cata-build` fusions. In the case of *map*, fusion with the copy function will indeed be successful.

A number of things need to be pointed out before continuing with our warm fusion example. First, having acknowledged the need to work in a polymorphic language, we must also recognize that the type of *map* is incorrectly — or, at best, incompletely — specified above. Indeed, *map* actually has the polymorphic type $\forall\alpha\beta.(\alpha \rightarrow \beta) \rightarrow \mathit{List}\ \alpha \rightarrow \mathit{List}\ \beta$. Moreover, since all subterms of *map* must also be typable, in particular, the data constructors *Nil* and *Cons* must be. In fact, *Nil* and *Cons* turn out to have types $\forall\alpha\beta.1$ and $\forall\alpha\beta.\alpha \rightarrow \beta \rightarrow \beta$, respectively, where 1 is a distinguished base type called the *unit type* (see Section 3.2.1).

Secondly, had we been transforming into `build-cata` form the map function over trees rather than that over lists, we would have introduced on the left of its body a version of `build` which produces, and a version of `cata` which consumes, trees with data of the appropriate types. Similarly, the copy function for trees, rather than that for lists, would have been introduced on the right of this map function's body. For the function *flatten*, which converts a tree into a list via (say) preorder traversal, the `build` and `cata` introduced on the left of its body would be those which process lists of

the appropriate type, while the copy function introduced on the right of *flatten*'s body would again be an appropriate copy function for trees.

Third, notice that `build` and `cata` are always introduced in pairs, and that these pairs are always instantiated at the same datatype. This is, of course, necessary if the introduced `build` and `cata` are to be used in the construction of an identity function, but has the additional purpose of insuring that the functional arguments to introduced instances of `build` are sufficiently polymorphic for the resulting term to be well-typed.

Although it is very important to the understanding and implementation of fusion algorithms generally, and to those of our implementation of warm fusion in particular, type information will continue to be suppressed throughout the remainder of this subsection (but will resurface in the next). Indeed, having pointed out some of the subtleties involved in properly typing and transforming terms in a language supporting elimination of a variety of algebraic virtual data structures, we can equally well continue, in the remainder of our informal description of the warm fusion method, to write programs just as we would in Haskell or, in any other language which suppresses explicit type abstraction and application. A more formal development of warm fusion, in which full account of type information is taken, begins in Section 3.2.

It turns out that there are two potentially quite different ways to derive a single catamorphism which is functionally equivalent to a composition such as that above. On the one hand, some heuristics can be used to "distribute" the outermost `cata` over `body`, and then the resulting program can be fused with the copy function it contains via the Promotion Theorem (see Theorem 3.13). This is essentially the first approach to fusion described on page 320 of [LS95]. On the other hand, `body` can be fused with the copy function via the Promotion Theorem to arrive at a new catamorphism, with which the outermost `cata` in the composition can in turn be fused. This is essentially the second approach to fusion described on page 320 of [LS95]. Launchbury and Sheard report that although the fusion steps in the second approach sometimes fails, it seems to yield better resulting programs when it succeeds. By contrast, the second approach seems to be more robust than the first. On many common examples, however, both methods are reportedly completely equivalent, in the sense that they both succeed and produce the same `build-cata` form. The first method requires only first-order fusion, but the second method typically involves higher-order fusion (described in [LS95]), which is undoubtedly more difficult to implement. For this reason, we have chosen initially to implement only the first method for producing a catamorphic argument to the introduced `build`. It would, of course, ultimately be quite useful to implement higher-order fusion as well, so that experiments comparing the two methods on a range of examples could be performed and analyzed.

For the first catamorphic production method, there is actually no need to introduce a copy function when first processing a program. That is, introduction of the copy function can be delayed without penalty until the type abstracted argument to the `build` introduced into the program's body is constructed. Delaying the introduction of the copy function gives a program completely equivalent to that obtained by its immediate introduction, but has the advantage of producing smaller programs to be manipulated at various stages during the warm fusion process. Our implementation, and the description of the warm fusion method below, reflect these observations: the `build-cata` form of the identity function is composed on the left of the program to be transformed, and the newly introduced `cata` is immediately distributed over the case statement which comprises the boby of that program.

For the prototype implementation of warm fusion reported on here, we actually restrict successful virtual data structure elimination to a smaller class of first-order-fusable programs than permitted in [LS95]. This is done by accepting for non-trivial transformation only programs of a particular syntactic form; fortunately, this form is general enough to accommodate many programs of interest, including those defined by of pattern matching on their arguments. Specifically, the prototype requires for successful fusion that bodies of programs to be transformed are abstracted case statements. This condition is met in the definition of *map* above, as well as in standard definitions of many other functions which arise commonly in practice, such as *append*, *reverse*, *zip*, *flatten*, etc.

Composing the `build-cata` form of the identity with $\texttt{body}_{\texttt{map}}$ and then distributing the newly-introduced, outermost `cata` over its outermost case statement yields this equivalent definition for

map:

```
map = \f xs → build (\c n → case xs of
                              Nil → cata c n Nil
                              Cons z zs → cata c n (Cons (f z) (map f zs))).
```

Distribution of `cata`s introduced into programs which are abstracted case statements is, in fact, straightforward, since

$$\texttt{f (case x of } \begin{array}{l} \texttt{e}_1 \rightarrow \texttt{p}_1 \\ \vdots \\ \texttt{e}_n \rightarrow \texttt{p}_n \end{array} \texttt{)}$$

is functionally equivalent to

$$\texttt{case x of } \begin{array}{l} \texttt{e}_1 \rightarrow \texttt{f (p}_1\texttt{)} \\ \vdots \\ \texttt{e}_n \rightarrow \texttt{f (p}_n\texttt{)} \end{array}$$

whenever f is strict, and catamorphisms are indeed strict. Of course, it would be advantageous eventually to allow distribution of `cata`s over more general program bodies, but this is supported neither by theory nor practice at present.

Now, whether or not attention is restricted only to programs which are abstracted case statements, some ideas from work on strictness analysis by Peyton Jones and Launchbury ([PJL91]) can be used to good effect in the production of catamorphic arguments to introduced `build`s. To illustrate, suppose that the data structure-producing function `foo` is expressed in the form of a call to `build`:

```
foo = \ x → build (c n → < foo⊥body >),
```

where `foo-body` is the possibly-very-large body of `foo`. If there are many sites in a program where `foo` is called as an argument to `cata`, then in order to use the `cata-build` rule to eliminate, at each such call, the virtual data structure produced by `foo`, `foo`'s body must meet the `cata` at that call site. One way to arrange for this is simply to inline `foo` bodily at each call site, but this may result in a huge amount of code explosion via duplication, especially if the body of `foo` is large. A more promising possibility is to split `foo` into two functions, a *wrapper* and a *worker*, thus:

```
foo  = \x → build (\ c n → foo# x)
foo# = \x c n → < foo⊥body > .
```

Here, `foo` is the wrapper, and `foo#` is the worker. The former is always small — being just an abstracted application of `build` to an abstracted call to `foo`'s worker — while the latter contains the whole of `foo`'s original body. Now the wrapper can be inlined at every call site without the risk of code explosion, so that `foo`'s `build` will meet the `cata`s at the various call sites in the calling program. This simple decomposition technique elegantly propagates from a program's definition to its call sites information about the way in which it produces its results. It is suitable for use in either the intrafunctional fusion which occurs during warm fusion itself, or during interfunctional fusion, via the shortcut, of compositions of programs already in `build-cata` form. Warm fusion is concerned with intrafunctional fusion, of course, but it requires the production of both a wrapper and a worker for every program it processes. A technique similar to the worker-wrapper method for data structure production can be used to propagate from a program's definition to its call sites information about the way in which it processes its arguments, but this is explored neither in [LS95] nor in our implementation.

One difficulty which may arise from overzealous use of wrapper-worker decompositions in virtual data structure elimination is that if the `build` from a wrapper happens *not* to cancel with a `cata` at a call site, then the code produced by warm fusion is slightly worse than if no fusion had been done at all. This is because putting a program in `build-cata` form parameterizes its body over the constructors of its result datatype, and so the resulting code is certain to be somewhat less efficient than it would be if the datatype constructors were used directly in the program's body. This problem

is not addressed in [LS95], and it is not even clear whether or not the code degradation is sufficiently great to deserve attention. But if it is, a reasonable remedy might simply be to maintain two versions of `foo`, one to be used at call sites at which the `cata-build` rule applies, and one to be used at call sites where it does not.

For the programs which our implementation successfully fuses, wrappers and workers of function definitions are computed as follows. Given

$$\texttt{foo} \;=\; \backslash\, \texttt{x}_1 \;...\; \texttt{x}_n \rightarrow \texttt{case}\; \texttt{x}_k\; \texttt{of}\; \mathit{pats},$$

we redefine `foo` in wrapper form as

$$\texttt{foo} \;=\; \backslash\, \texttt{x}_1 ... \texttt{x}_n \rightarrow \texttt{build}\; (\backslash\, \texttt{c}\; \texttt{n} \rightarrow \texttt{foo\#}\; \texttt{x}_k\; \texttt{v}_1\; ...\; \texttt{v}_m),$$

where the worker `foo#` of `foo` is given by

$$\texttt{foo\#} \;=\; \backslash\, \texttt{x}_k\; \texttt{v}_1\; ...\; \texttt{v}_m \rightarrow \texttt{case}\; \texttt{x}_k\; \texttt{of}\; \mathit{pats}.$$

Here $x_k \in \{x_1, ..., x_n\}$, and x_k must not appear free in *pats*; $v_1, ..., v_k$ are the free variables of *pats* other than x_k which are not also free in `foo` itself. The restriction on x_k prevents a case of full primitive recursion. Note also that the case selector of `foo` becomes the outermost argument to its worker; workers are constructed in this way specifically to meet the strictness requirement in the hypotheses of the Promotion Theorem (Theorem 3.13), which is used to actually transform them into catamorphisms. When `foo` is our "distributed" definition of *map*, for example, its wrapper is given by

$$\texttt{map} \;=\; \backslash\, \texttt{f}\; \texttt{xs} \;\rightarrow\; \texttt{build}\; (\backslash\, \texttt{c}\; \texttt{n} \rightarrow \texttt{map\#}\; \texttt{xs}\; \texttt{f})$$

and its worker is

$$\begin{aligned}\texttt{map\#} \;=\; \backslash\, \texttt{xs}\; \texttt{f} \;\rightarrow\; &\texttt{case}\; \texttt{xs}\; \texttt{of}\\ &\quad \texttt{Nil} \rightarrow \texttt{n}\\ &\quad \texttt{Cons}\; \texttt{z}\; \texttt{zs} \rightarrow \texttt{cata}\; \texttt{c}\; \texttt{n}\; (\texttt{Cons}\; (\texttt{f}\; \texttt{z})\; (\texttt{map}\; \texttt{f}\; \texttt{zs})).\end{aligned}$$

Inlining a program's wrapper in its worker gives a recursive definition of the worker, which can then be fused with an appropriate copy function in hopes of obtaining a catamorphic rendering of it. The copy function introduced here is the one whose introduction was delayed earlier; fusing the worker on the right with an appropriate copy function has precisely the same effect as performing this fusion on the original program. If the fusion is successful, substitution of the catamorphic worker back into the wrapper yields a `build-cata` form for the original program. For example, inlining gives the following recursive definition of `map#`:

$$\begin{aligned}\texttt{map\#} \;=\; \backslash\, \texttt{xs}\; \texttt{f} \;\rightarrow\; &\texttt{case}\; \texttt{xs}\; \texttt{of}\\ &\quad \texttt{Nil} \rightarrow \texttt{n}\\ &\quad \texttt{Cons}\; \texttt{z}\; \texttt{zs} \rightarrow \texttt{cata}\; \texttt{c}\; \texttt{n}\; (\texttt{Cons}\; (\texttt{f}\; \texttt{z})\; (\texttt{build}\; (\backslash\, \texttt{c}\; \texttt{n} \rightarrow \texttt{map\#}\; \texttt{zs}\; \texttt{f}))).\end{aligned}$$

Fusing with the copy function gives

$$\texttt{map\#} \;=\; \backslash\, \texttt{xs}\; \texttt{f} \rightarrow \texttt{cata}\; \texttt{h}_1\; \texttt{h}_2\; \texttt{xs},$$

where

$$\texttt{h}_1 \;=\; \backslash\, \texttt{z}_1\; \texttt{z}_2 \rightarrow \texttt{c}\; (\texttt{f}\; \texttt{z}_1)\; (\texttt{z}_2\; \texttt{f})$$

and

$$\texttt{h}_2 \;=\; \texttt{n}.$$

Exactly how h_1 and h_2 are calculated is described in Section 3.2.4; the machinery for doing calculating them is developed in Section 3.2.5.

At several stages in this process — such as after the case distribution at its beginning — standard simplifications based on the definitions of the various language constructs can be performed on programs. These are coded in the rules of the warm fusion calculus, which are given in their entirety in Section 3.2.

3.2 Theoretical Underpinnings

The language with which we work in this paper is a full polymorphic lambda calculus with first-order skolem constants, a `case` construct, and two distinguished higher-order term-building constructs called `build` and `cata`. The usual construction of types from type variables by closure under $\rightarrow$ and type abstraction is extended to permit, in addition, the definition of certain kinds of recursive datatypes. Recursive function definitions are also permitted, so that the language supports both the structural recursion embodied by `cata`, and general recursion. This section defines and motivates our particular language choice.

In order to describe the terms in our language, we first need to define the types that terms may have. This requires an investigation of polynomial functors, the physical and virtual datatypes to which they give rise, and the connections between these two kinds of datatypes. Once the (polymorphically typed) terms of the language are defined in Section 3.2.3, the calculus defined over them is given in Section 3.2.5.

3.2.1 Polynomial Functors, Least Fixed Points, and Physical Datatypes

The category in which we will work in this paper is the category $\mathcal{SD}$ of Scott domains. Thus, the types of our language will be modeled by algebraic, pointed, consistently complete, complete partially ordered sets, and the terms of our language will be modeled by elements of the Scott domains which interpret their types. It is well-known that the polymorphic lambda calculus has no classical set-theoretic models; working in the category of Scott domains will, however, guarantee the existence of non-trivial models of the polymorphic lambda calculus ([CGW89]). While it is possible that we could be more general in our choice of underlying semantic category, the category $\mathcal{SD}$, which we adopt mainly for concreteness, is certainly sufficient for our purposes.

For any given category $\mathcal{C}$, endofunctors on $\mathcal{C}$ are commonly used to capture the signatures of algebraic datatypes over $\mathcal{C}$. The endofunctors on $\mathcal{SD}$ which are of interest to us here are the *polynomial functors*, i.e., are those functors which are definable solely in terms of $\mathcal{SD}$'s identity functor, constant functors, product functor, and sum functor. These elementary functors are described immediately below. In the definitions that follow, the collection of objects of $\mathcal{SD}$ is denoted $obj(\mathcal{SD})$ and the collection of arrows of $\mathcal{SD}$ is denoted $arr(\mathcal{SD})$. We write $f \in arr(A, B)$ to indicate that $f : A \rightarrow B$ is a continuous function from the Scott domain A to the Scott domain B.

Definition 3.1 The *identity functor* $\mathcal{I}$ on $\mathcal{SD}$ is given by

- $\mathcal{I}X = X$ for all $X \in obj(\mathcal{SD})$, and
- $\mathcal{I}f = f$ for all $f \in arr(\mathcal{SD})$.

Definition 3.2 If A is any Scott domain, then the *constant functor* $\mathcal{K}_A$ on $\mathcal{SD}$ is given by

- $\mathcal{K}_A X = A$ for all $X \in obj(\mathcal{SD})$, and
- $\mathcal{K}_A f = id_A$ for all $f \in arr(\mathcal{SD})$.

Definition 3.3 The *product functor* $\times$ on $\mathcal{SD}$ is given by

- $X_1 \times ... \times X_n = \{\langle x_1, ..., x_n\rangle \mid x_i \in X_i\}$ for all $X_1, ..., X_n \in obj(\mathcal{SD})$ and all integers n, and
- for any integer n and for any $f_i \in arr(X_i, Y_i)$, $f_1 \times ... \times f_n$ is defined by

$$(f_1 \times ... \times f_n)\langle x_1, ..., x_n\rangle = \langle f_1 x_1, ..., f_n x_n\rangle$$

  for all $\langle x_1, ..., x_n\rangle \in X_1 \times ... \times X_n$.

When $n = 0$, the resulting product is denoted by 1, and called the *unit type*. In what follows, context should easily disambiguate the unit type from the number 1.

Definition 3.4 The *(lifted) sum functor* $|$ on $\mathcal{SD}$ is given by

- $X_1|...|X_n = (\{1\} \times X_1) \cup ... \cup (\{n\} \times X_n) \cup \perp$ for all $X_1, ..., X_n \in obj(\mathcal{SD})$ and all integers n, and
- for any integer n and for any $f_i \in arr(X_i, Y_i)$, $f_1|...|f_n$ is defined by

$$(f_1|...|f_n)\langle i, x\rangle = \langle i, f_i x\rangle,$$

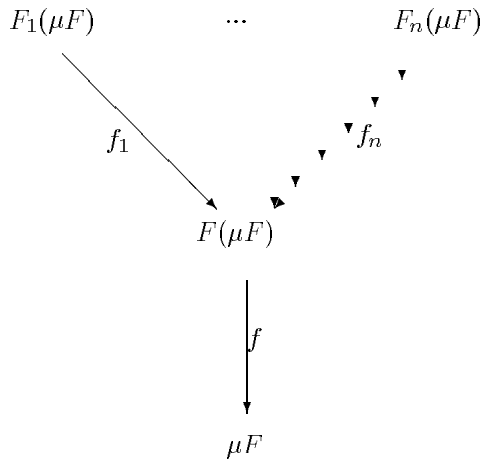
for all $i \in \{1, ..., n\}$ and $x \in X_i$, and

$$(f_1|...|f_n)(\perp_{X_1|...|X_n}) = \perp_{Y_1|...|Y_n}.$$

The category $\mathcal{SD}$ does not have true categorical sums.

Throughout the remainder of this paper, let F be a fixed but arbitrary polynomial functor unless otherwise specified. Given F, we single out for special attention the least Scott domain $\mathcal{D}$ satisfying $\mathcal{D} = F\mathcal{D}$. Such a least Scott domain always exists and is unique up to isomorphism when F is a polynomial functor ([Sch86]). We call the least Scott domain $\mathcal{D}$ such that $\mathcal{D} = F\mathcal{D}$ the *physical datatype* for F, and denote it by μF; this terminology reflects the fact that μF is obtainable by means of a standard colimit construction in which the elements of μF are built using injective tags corresponding to the physical tags on data structures in the heap. Note carefully that equality, rather than mere isomorphism, is required in the definition of μF.

If we write the domain FX as a sum-of-products $F_1X|...|F_nX$, then when X is μF we have the following n-ary sum diagram:



From the definition of (lifted) sums and the definition of μF, we know that to every n-tuple of functions $[f_1, ..., f_n]$ with $f_i : F_i(\mu F) \to \mu F$, there corresponds a unique $f : F(\mu F) \to \mu F$ such that $f = [f_1, ..., f_n]$. In particular, there exists a one-to-one correspondence between *strict* arrows from $F(\mu F)$ to μF and such n-tuples. Thus, since $\mu F = F(\mu F)$, and since $id_{\mu F}$ can be considered a strict mapping from $F(\mu F)$ to μF, there must exist unique f_i's such that $id_{\mu F} = [f_1, ..., f_n]$. These f_i's are called *data constructors* for μF, and they are precisely the injective tags mentioned above from which the elements of the physical datatype for F are built.

Example 3.5 If $FX = 1|X$ then we traditionally write Nat for μF, and write the components of $id_{\mu F}$ as $Zero : 1 \to \mu F$ and $Succ : \mu F \to \mu F$. This means that an element of μF is either $Zero\ 1$, of the form $Succ\ x$ for some $x \in \mu F$, or $\perp$. Abusing notation, we might reasonably tag the summands of μF with the data constructors $Zero$ and $Succ$, and so write $\mu F = Zero\ 1|Succ\ \mu F$ rather than $\mu F = 1|\mu F$.

Example 3.6 On the other hand, if $F_A X = 1 \mid (A * X)$ for some fixed type A, then common practice is to write $List\ A$ for μF and Nil and $Cons$ for the component functions of $id_{\mu F_A} : F_A(\mu F_A) \to \mu F_A$.

An element of μF_A is therefore either of the form $Nil\ 1$, of the form $Cons\ x\ y$ for some element $x \in A$ and some $y \in \mu F_A$, or $\perp$. We normally write $\mu F_A = Nil\ 1 \mid Cons(A * \mu F_A)$ rather than $\mu F_A = 1 \mid (A * \mu F_A)$.

We will use this "tagged" sum notation below. Example 3.6 demonstrates that functors can be parameterized over types. The definition of functors for our language will indeed support this kind of parameterization.

3.2.2 Types for Warm Fusion

Fix a set *TVar* of *type variables* $\alpha, \beta, ...$ and a disjoint set *TConst* of *type constants* such as `Int, Bool,` etc. The types of our language are defined over *TVar* and *TConst* as described in Definition 3.7. This definition is mutually recursive with that of type constructors in Definition 3.8.

Definition 3.7 The set *Types* of *types* (over *TVar* and *TConst*) is defined inductively by:

- every type variable in *TVar* and every type constant in *TConst* is a type,
- if s and t are types, then $s \to t$ is also a type,
- if t is a type and α is a type variable, then $\forall\alpha.t$ is also a type, and
- if $T = \forall\alpha_1...\forall\alpha_n.\texttt{Rec}\,\beta\,.\,\texttt{C}_1 t_1 | ... | \texttt{C}_m t_m$ is a type constructor, $\tau_1, ..., \tau_n$ are types, then $T\tau_1...\tau_n$ is also a type.

The types satisfying the four clauses of Definition 3.7 are called *atomic types*, *arrow types*, *abstracted types*, and *constructed types*, respectively. Following common practice, we write $\forall\alpha_1...\alpha_n.T$ in place of $\forall\alpha_1...\forall\alpha_n.T$.

Definition 3.8 A *type constructor* is an expression of the form

$$\forall\alpha_1...\alpha_n.\texttt{Rec}\,\beta\,.\,\texttt{C}_1 t_1 | ... | \texttt{C}_m t_m,$$

where each *data constructor* $\texttt{C}_i$ is a unique tag into the sum $t_1|...|t_m$ as described in Section 3.2.1, and each t_i is either an atomic type or a constructed type. Here, each t_i, $i \in \{1, ..., m\}$, is a list $[t_{i1}, ..., t_{ik_i}]$ representing the product of the types $t_{i1} \times ... \times t_{ik_i}$. Such a type constructor T is said to have *arity* n.

A type variable is *free* in the type t (resp., the type constructor T) if it does not appear in the scope of any occurrence of $\forall$ or `Rec` in t (resp., T), and is *bound* in t (resp., T) otherwise. As is customary, we identify types which are identical up to renaming of bound variables. We write $t[\alpha := t']$ for the type obtained by replacing all free occurrences of the type variable α in the type t by the type t', with the usual proviso regarding free variable capture. We will be interested only in type constructors containing no free variables, and so make the assumption throughout the remainder of this paper that all type constructors appearing are closed.

Of course, it is only sensible to talk about sums of types by virtue of our intended semantics. In this semantics, types are interpreted as Scott domains as described above. In a type constructor definition $T = \forall\alpha_1...\alpha_n.\texttt{Rec}\beta\,.\,\texttt{C}_1 t_1 | ... | \texttt{C}_m t_m$, the expression $\texttt{C}_1 t_1 | ... | \texttt{C}_m t_m$ is a "type schema," $\forall\beta.\texttt{C}_1 t_1 | ... | \texttt{C}_m t_m$ denotes a unary functor on β which is parametrized over $\alpha_1, ..., \alpha_n$, and the expression $\texttt{Rec}\beta\,.\,\texttt{C}_1 t_1 | ... | \texttt{C}_m t_m$ is interpreted as the least fixed point of that unary functor, i.e., as its physical datatype. The expression defining the type constructor itself thus denotes a functor from Scott domains interpreting the α_is to physical datatypes instantiated at the α_is. Since type constructors are functors they not only have actions on Scott domains interpreting types, but on arrows between such domains as well. The natural action of the functor denoted by a type constuctor $T = \forall\alpha_1...\alpha_n.\texttt{Rec}\beta\,.\,\texttt{C}_1 t_1 | ... | \texttt{C}_m t_m$ on arrows $f_1, ..., f_n$ of types $u_1 \to v_1, ..., u_n \to v_n$, respectively, is to map them recursively across the structure of T to get a map from $Tu_1...u_n$ to $Tv_1...v_n$. That is, the "arrow part" of the functor denoted by type constructor T is just the usual *map* functor associated with T. We write map^T for the "arrow part" of the functor T.

We see, therefore, that it is precisely because polynomial functors over $\mathcal{SD}$ have least fixed points in that category that expressions of the form $\texttt{Rec}\beta\,.\,\texttt{C}_1 t_t | ... | \texttt{C}_m t_m$ have well-defined semantics, and because $\mathcal{SD}$ is a model of the polymorphic lambda calculus that their type-abstracted versions are sensible. Since β need not actually appear in $\texttt{C}_1 t_t | ... | \texttt{C}_m t_m$, this explicitly recursive notation is sufficient to describe both recursive and non-recursive datatypes.

Note that "twisting" is not allowed in type constructor definitions. This prohibition rules out the definition of type constructors — such as $T = \forall\alpha.\texttt{Rec}\beta.\texttt{C}_1\,[\,] \mid \texttt{C}_2[\alpha, T\beta\alpha]$ — in which the arguments to the type constructor being defined are permuted; it is enforced via the recursive variable in the expression defining T. Observe further that neither arrow types nor abstracted types can appear as the type arguments to data constructors, and that type constructors appearing in constructed types must be fully applied. As a consequence of the latter restriction, datatypes built using function space are not considered here. Recent work by Meijer and Hutton ([MH95]) may indicate strategies for relaxing this restriction.

3.2.3 Terms for Warm Fusion

Fix a set *Var* of raw term variables, a set *Const* of raw constants, and a set S of raw skolem constants such that the sets *TVar*, *TConst*, *Var*, *Const*, and S are all mutually disjoint. The terms of our warm fusion calculus are constructed over these primitives as in the next definition, which is mutually recursive with Definition 3.10. Write $[t_1, ..., t_n]$ for the list of terms $t_1, ..., t_n$. Although we use the same notation for lists and type applications in this paper, which of these is meant in any given instance should always be discernible from the context in which it appears.

Definition 3.9 The set *Terms* of typed *terms* (over *Var*, *Const*, and S) is defined inductively by:

- If a is a raw term variable, raw constant, or raw skolem constant, then $a :: t$ is a term variable, constant, or skolem constant, respectively, of type t, and so is a term of type t.
- If $T = \Lambda\alpha_1...\alpha_n.\texttt{Rec}\beta\,.\,\texttt{C}_1 t_1 | ... | \texttt{C}_m t_m$ is a type constructor, then each data constructor $\texttt{C}_i$ for T is a term of type $\forall\alpha_1...\alpha_n.t_{i1}[\beta := T\alpha_1...\alpha_n] \to ... \to t_{ik_i}[\beta := T\alpha_1...\alpha_n] \to T\alpha_1...\alpha_n$.
- If e is a term of type t', and $a_1, ..., a_n$ are all arms of type t, then $\texttt{case}\ e\ [a_1, ..., a_n]$ is a term of type t.
- If $x :: t$ is a term variable of type t and e is a term of type t', then $\lambda x :: t.e$ is a term of type $t \to t'$.
- If e_1 is a term of type $t \to t'$ and e_2 is a term of type t, then $e_1 e_2$ is a term of type t'.
- If α is a type variable and e is a term of type t, then $\Lambda\alpha.e$ is a term of type $\forall\alpha.t$, provided that, for every term variable $x :: t'$ which does not appear in the scope of some occurrence of λ in e, α is not free in t'.
- If e is a term of type $\forall\alpha.t$ and t' is a type, then $e[t']$ is a term of type $t[\alpha := t']$.
- The expression $\texttt{cata}\ [t_1, ..., t_n]\ t\ T\ [e_1, ..., e_m]$ is a term of type t.
- If e is a term of type $\forall\alpha_1...\alpha_n.(T\alpha_1...\alpha_n \to \alpha_1 \to ... \to \alpha_n) \to \alpha_1 \to ... \to \alpha_n$ for some type constructor T of arity n, then the expression $\texttt{build}\ [t_1, ..., t_n]\ T\ e$ is a term of type $Tt_1...t_n$.

This definition insists that atoms are always annotated with their types. We write $e :: t$ to indicate that e is a term of type t. The set $Terms_t$ of *terms of type* t is defined to be $\{e \in Terms \mid e :: t\}$. Because term variables, constants, and skolem constants carry explicit type information, every term has a unique type. We will often abuse terminology and refer simply to a term variable, constant, or skolem constant a — rather than $a :: t$ — when the type of such a term is not germane to the discussion at hand. According to the second clause of Definition 3.9, partially applied — and even unapplied — data constructors are terms. This is in contrast to our insistence that only fully applied type constructors are valid types.

Definition 3.10 An *arm of type* t is an expression of the form $a = \langle \mathtt{C}, \langle [v_1, ..., v_n], e \rangle \rangle$ where `C` is a data constructor for T with type $\forall \alpha_1 ... \alpha_n . t_1[\beta := T\alpha_1 ... \alpha_n] \rightarrow ... \rightarrow t_k[\beta := T\alpha_1 ... \alpha_n] \rightarrow T\alpha_1 ... \alpha_n$, v_j is a variable of type $t_j[\beta := T\alpha_1 ... \alpha_n]$ for each $j \in \{1, ..., k\}$, and e is a term of type t.

A type variable α is *free* in the term e provided it does not appear in the scope of any occurrence of Λ or $\forall$ in e. A term variable x is *free* in the term e if it does not appear in the scope of any occurrence of λ in e and is free in every case subterm of e; x is free in the arm $\langle \mathtt{C}, \langle [v_1, ..., v_n], e' \rangle \rangle$ if it is free in e' and is distinct from each v_i, and free in `case` $e\ [a_1, ..., a_n]$ if it is free in e and free in each arm a_i, for $i \in \{1, ..., n\}$. A term or type variable which is not free in a term is said to be *bound* in that term. We adopt the usual conventions regarding variable renaming, identifying those terms which differ only in the names of their bound type and term variables. A term with no free type or term variables is said to be *closed*. When $x :: t$ is a term variable and e' is a term of type t, we write $e[x := e']$ for the term obtained by replacing all free occurrences of the term variable x by the term e' in the term e (observing the usual proviso regarding free variable capture). We similarly write $e[\alpha := \tau]$ for the result of replacing all free occurrences of the type variable α in e by the type τ.

In the prototype warm fusion tool described in this paper, we require that all term variables and constants appearing in terms are explicitly and fully polymorphically typed at input, and this fact is reflected in the definition of our calculus. Together with a primitive type checker this ensures that all terms on which the tool works are well-formed according to the clauses of Definition 3.9. Although we would expect to implement a type inference algorithm for terms for use in succeeding versions of the tool, none is available at present.

Terms other than data constructors and those constructed using `cata`, `build`, and `case` comprise a polymorphic lambda calculus with (in this case, two distinguished sets of) constants. The computational intuition underlying the polymorphic lambda calculus has been discussed at length elsewhere (see, e.g., [Bar92] or [Gir89]), and the intuition behind, as well as the typing rule for, terms constructed using `case` should be self-explanatory. In a term of the form `cata` $[t_1, ..., t_n]\ t\ T\ [e_1, ..., e_m]$, the list $[t_1, ..., t_n]$ of types denotes the nonrecursive type arguments to the n-ary type constructor T over which the catamorphism is defined, $[e_1, ..., e_m]$ represents the functional arguments to the catamorphism, and t denotes the result type of the catamorphism. In a term of the form `build` $[t_1, ..., t_n]\ T\ e$, the list $[t_1, ..., t_n]$ of types denotes the nonrecursive type arguments to the n-ary type constructor T for the result type of this instance of `build` and e denotes its functional argument. The type of the resulting `build` term is $Tt_1 ... t_n$. We assume that all terms appearing throughout the remainder of this paper are formed in accordance with Definition 3.9.

Careful inspection of the typing rule for `build` terms reveals that they do not have Hindley-Milner types. This observation necessitates some care in adding `build` as a primitive construct in any functional programming language. The most straightforward way to resolve the typing difficulties surrounding `build` is to restrict its use so that it is available internally to the compiler but not as a language construct for programmers. Of course, warm fusion is intended precisely to allow programmers to code in the common — and, for most programmers, more natural — recursive style while automatically reaping the benefits of algebraic programming; the unavailability of `build` at design time should, therefore, in no way be seen as an obstacle to effective programming practice. The algorithm we give in Section 3.3 for automatically introducing `build`s into programs guarantees that `build` terms are always typed in accordance with Definition 3.9. This, in turn, insures that an appropriate generalization of Theorem 2.1 holds for each type constructor.

A class of terms with which we will be particularly concerned is the class of `build-cata` forms. They are described formally in

Definition 3.11 A term is in `build-cata` *form* if it is of the form

$$Lq_1 ... Lq_k . \mathtt{build}\ [t_1, ..., t_n]\ T\ (Lp_1\ ...\ Lp_l . \mathtt{cata}\ [t_1, ..., t_n]\ t'\ T\ [e_1, ..., e_m]),$$

where $q_1, ..., q_k, p_1, ..., p_l$ are either type or term variables, and L denotes λ or Λ as appropriate.

As discussed in Section 1, the goal of the warm fusion process is to produce `build-cata` forms from recursive function definitions so that the resulting equivalent programs can then be fused with other

programs — which have also been put into `build-cata` form — to eliminate virtual data structures from their compositions.

It is via declarations that the programmer may specify recursive function definitions.

Definition 3.12 A *declaration* is an expression of the form $v :: t = e$, where $v :: t$ is a term variable and e is a term of type t.

We write $v = e$ for the declaration $v :: t = e$ when the particular identity of the type t is not pertinent to the discussion at hand. The declaration $v :: t = e$ is intended to denote the least fixed point of the (continuous) function, from the Scott domain interpreting t to itself, which is denoted by $\lambda v.e$. Of course, the term variable v need not appear in e. Mutually recursive declarations are not permitted.

The language we describe here differs from that presented in [LS95]. One superficial difference is that we permit tuples at neither the type nor the term level, preferring instead to pass bundled arguments as lists. More significantly, the polymorphism of the expression language of [LS95] is not made explicit. In particular, type abstractions and applications are not included in the abstract syntax of the language there, although this is, of course, necessary in order for terms involving datatype constructors, `cata`, and `build` to be well-formed and to behave as expected.

3.2.4 Virtual Datatypes, Build, and Kata

At present, `cata` and `build` are mere syntactic constructs in our term language. In this section, we'll consider their semantics and the properties they satisfy, and, thereby, come to understand the motivation for the computation rules involving them which appear in the warm fusion calculus of Section 3.2.5.

In our intended semantics, types are interpreted as Scott domains, and a term t of type α is interpreted as an element of the Scott domain denoted by α. If, as is common practice, we identify both types and terms with their interpretations, then a type can be thought of as representing all terms which have that type. In particular, the type $\forall\alpha.(F\alpha \to \alpha) \to \alpha$ can be thought of as the set of all $f \in arr(\mathcal{SD})$ which map type variables α to types of the form $(F\alpha \to \alpha) \to \alpha$. We write $\forall F$ to abbreviate $\forall\alpha.(F\alpha \to \alpha) \to \alpha$, and call $\forall F$ the *virtual datatype* for F. This terminology is reasonable in light of the observation that, for each endofunctor F on $\mathcal{SD}$, the elements of $\forall F$ can serve as "instructions" for constructing elements of the physical datatype for F. (To see this, consider the application of an element of $\forall F$ to the type μF.) The elements of $\forall F$ should not themselves be mistaken for elements of the physical datatype, however.

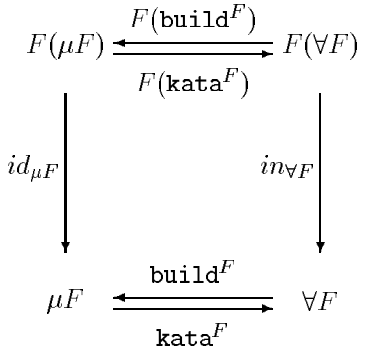
Given an endofunctor F on $\mathcal{SD}$, an *F-algebra* over $\mathcal{SD}$ (or, simply, an *F-algebra*) is a pair $\langle A, f\rangle$ such that A is a Scott domain and $f : FA \to A$ is a continuous function. The domain A is called the *carrier* of the algebra. An *F-homomorphism* from an F-algebra $\langle A, \varphi\rangle$ to an F-algebra $\langle B, \psi\rangle$ is a continuous function $h : A \to B$ which satisfies $h \circ \varphi = \psi \circ Fh$. $ALG(F)$ is defined to be the category whose objects are F-algebras over $\mathcal{SD}$ and whose arrows are F-homomorphisms.

An F-algebra $\langle A, \varphi\rangle$ is said to be *initial* if for every F-algebra $\langle B, \psi\rangle$ there exists a unique F-homomorphism $h : A \to B$ such that $h \circ \varphi = \psi \circ Fh$. We say that such an h is a *catamorphism* from A to B, and write $(\!|\,\psi\,|\!)_{A,B}$ for f. When A and B are clear from context we simply write $(\!|\,\psi\,|\!)$ for h.

Consider the collection of F-algebras $\langle \forall F, in_{\forall F}\rangle$ and $\langle \mu F, id_{\mu F}\rangle$, where F ranges over all endofunctors on $\mathcal{SD}$, μF is the physical datatype for F. Here $in_{\forall F} = \lambda x.\Lambda\alpha.\lambda f.f(map^F(\lambda\ z.z[\alpha]f)x)$ maps $F(\forall F)$ to $\forall F$. Each algebra of the form $\langle \mu F, id_{\mu F}\rangle$ is initial, and each can be identified with the physical datatype for its endofunctor F. For any F-algebra $\langle B, \psi\rangle$ we may therefore write $\texttt{cata}^F\,\psi$ for $(\!|\,\psi\,|\!)_{\mu F,B}$, reflecting our intention that the language construct `cata` in Definition 3.9 provide a means of denoting catamorphisms out of physical datatypes. It is the fact that catamorphisms can be defined categorically, with no specific reference to lists whatsoever, which makes possible the generalization of the programming language notion of a catamorphism to other physical dataytypes.

In the special case when $\langle B, \psi\rangle$ is $\langle \forall F, in_{\forall F}\rangle$, we write $\texttt{kata}^F$ for $\texttt{cata}^F in_{\forall F}$. Thus, for each F, $\texttt{kata}^F$ provides a mapping from the physical datatype μF for F to the virtual datatype $\forall F$ for F. If we could also establish that $\langle \forall F, in_{\forall F}\rangle$ is initial, then the existence of a unique homomorphism

$(\!|\, id_{\mu F} \,|\!) : \forall F \to \mu F$ which is a counterpart to $\mathtt{kata}^F$ would also be guaranteed. If we called this homomorphism $\mathtt{build}^F$, then $\mathtt{kata}^F$ and $\mathtt{build}^F$ would be related as in the following figure:



By uniqueness of catamorphisms, we would then see easily that $\mathtt{kata}^F$ and $\mathtt{build}^F$ are in fact mutual inverses, so that each would be an isomorphism. That is, for any data structure d in μF we would have

$$\mathtt{build}^F(\mathtt{kata}^F d) = d,$$

and for any function g in $\forall F$ we would have

$$\mathtt{kata}^F(\mathtt{build}^F g) = g.$$

The existence of these isomorphisms would provide a means of shifting points of view between physical and virtual datatypes — and, hence, of trading stack space for heap space, and vice-versa, in computations — at will. Since any two initial algebras are isomorphic, neither μF nor $\forall F$ would be able to be considered "the" datatype associated with F — they would just be two different representations of the same Scott domain. In certain computational settings using one representation might be advantageous, while in other circumstances using the other might be more effective. What would be important in such a scenario would be our ability to pass from one representation to the other, using $\mathtt{build}^F$ and $\mathtt{kata}^F$ as explicit tags indicating which representation of an element of the fixed point of F we wished to work with at any given time in a computation. Being able to change representations in mid-computation would in some instances lead to efficiency gains in programs which construct physical data structures for passing information between components of programs by enabling them to forego actual the construction of such datatypes in favor of building their virtual counterparts.

Unfortunately F-algebras of the form $\langle \forall F, in_{\forall F} \rangle$ are not necessarily initial, and so, given an F-algebra $\langle B, \psi \rangle$, there need not be a unique homomorphism h such that $h \circ in_{\forall F} = \psi \circ Fh$. In fact, in models of the polymorphic lambda calculus which are not parametric, initiality of $\langle \forall F, in_{\forall F} \rangle$ cannot necessarily be guaranteed. In particular, for $\langle \mu F, id_{\mu F} \rangle$ there need not be a unique homomorphism h such that $h \circ in_{\forall F} = id_{\mu F} \circ Fh$.

The program outlined above thus appears to fail. But even in the absence of parametricity the *existence* of a homomorphism $\mathtt{build}^F$ from $\langle \forall F, in_{\forall F} \rangle$ to $\langle \mu F, id_{\mu F} \rangle$ can always be established, even if its uniqueness cannot. In fact, if F is a polynomial functor, then it is not hard to see that the mapping h from $\forall F$ to μF given by

$$hg = g[\mu F] id_{\mu F}$$

is indeed an F-homomorphism. We may therefore take $\mathtt{build}^F$ to be precisely this h. That is, we may *define* $\mathtt{build}^F$ by

$$\mathtt{build}^F\ g = g[\mu F] id_{\mu F},$$

reflecting the intention that the language construct `build` in Definition 3.9 provide a mechanism for denoting, for each endofunctor F, this particular homomorphism from $\forall F$ to μF. This definition of $\mathtt{build}^F$ is, of course, consistent with the description of `build` for lists in Section 2.4, although it is appropriate for more general physical datatypes as well.

The categorical definition of catamorphisms, meanwhile, insures that the operational behavior of the language construct `cata` generalizes that described in Section 2.4 on lists. Indeed, for any $f : FA \to A$, $\texttt{cata}^F\ f$ denotes the unique F-homomorphism h satisfying $h = f \circ Ff$. That is, $\texttt{cata}^F\ f$ denotes the unique F-homomorphism which repeatedly applies f down the recursive structure of its argument. This means that although the definition of $\texttt{build}^F$ above does not necessarily guarantee that $\texttt{kata}^F\ (\texttt{build}^F\ g) = g$, it does insure that

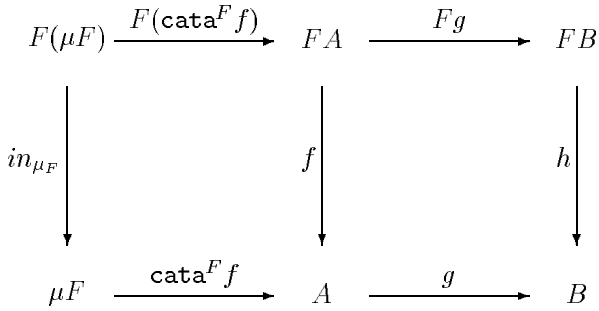
$$\texttt{kata}^F\ (\texttt{build}^F\ g) = g\ in_{\forall F}$$

for all appropriate g. More generally, we have that

$$\texttt{cata}^F\ f\ (\texttt{build}^F\ g) = g\ f$$

for all appropriate F, g, and f. In addition, as suggested by the following diagram, the *Promotion Theorem* for catamorphisms must also hold:

Theorem 3.13 *(Promotion Theorem) If g is strict and $g \circ f = h \circ Fg$, then $g \circ \texttt{cata}^F f = \texttt{cata}^F h$.*



The `cata-build` rule for our term language appears in Definition 3.22 and that language's counterpart to the Promotion Theorem is given in Definition 3.23.

3.2.5 The Warm Fusion Calculus

The reduction rules which are used to transform terms of our calculus break naturally into two groups. First we present the *basic reduction rules*, which describe interactions between the various language constructs. To do this, we need to have a few preliminary notions at our disposal.

Definition 3.14 A term M is an *argument subterm* of the term N if either

- N is exactly PM for some term P,
- N is of the form $\lambda x.P$ and M is an argument subterm of P, or
- N is of the form PQ and N is an argument subterm of either P or Q.

Definition 3.15 A variable y *appears under an abstraction* in an expression e provided there exists a subexpression $M \equiv \lambda x.b$ of e such that y occurs at least once in an argument subterm of M.

Example 3.16 1. The variable y appears under an abstraction in $M(\lambda x.xy)$.

2. The variable y appears under an abstraction in $(\lambda x.xy)b$ iff it appears under an abstraction in b.

Definition 3.17 A variable y is *linear* in e if y occurs in e no more than once and y does not appear under any abstraction in e.

Example 3.18 1. y is not linear in $\lambda x.xy$.

2. y is linear in b iff y does not occur in b.

The intuition underlying the notion of linear terms is that they should entail no duplication of work during β-reduction. In Definition 3.22 below, we will require that a linearity condition be satisfied in order for an expression to be β- or case-reducible. According to this criterion, $(\lambda y.f(\lambda x.xy))e$ will not be β-reducible, for example, whereas $(\lambda y.(\lambda x.xy)b)e$ will be. This is precisely as desired. In the first situation, β-reduction yields $f(\lambda x.xe)$; if f duplicates e, further β-reduction could lead to a significant increase in work to be done. In the second, β-reduction gives $(\lambda x.xe)b$, in which duplication of e has occurred iff y occurs in b. In both β- and case-reduction, terms meeting certain syntactic criteria are literally copied whether or not they appear linearly, but the copying of other arguments which do not appear linearly is disallowed.

We also need a mechanism for describing the recursive structure of type constructors. To describe the action of a catamorphism over lists, for example, the recursive structure of the list type constructor must be known so that the catamorphism can be properly distributed over its recursive parts. The following function captures the recursive structure of data constructors; the recursive structure of a type constructor is determined by the structure of its data constructors.

Definition 3.19 Let $T = \forall\alpha_1...\alpha_n.\mathtt{Rec}\beta.\mathtt{C}_1 t_1|...|\mathtt{C}_m t_m$ be a type constructor, and let g be any term of type $(T\tau_1...\tau_n) \to \tau$ for some types $\tau, \tau_1, ..., \tau_n$. If α is any type, write id_α for the term $\lambda x :: \alpha.x$.

1. The function $\mathcal{E}_g^T$ from *Types* to *Terms* is given by
   - $\mathcal{E}_g^T\ \beta\ =\ g$,
   - $\mathcal{E}_g^T\ \alpha\ =\ id_\alpha$ for α distinct from β,
   - $\mathcal{E}_g^T\ c\ =\ id_c$ for each type constant c,
   - $\mathcal{E}_g^T\ T't'_1...t'_k\ =\ map^{T'}[\delta_1]...[\delta_k][\rho_1]...[\rho_k][\mathcal{E}_g^T t'_1, ..., \mathcal{E}_g^T t'_k]$, where $\mathcal{E}_g^T t'_i$ has type $\delta_i \to \rho_i$ for all $i \in \{1, ..., k\}$.
2. For each $i \in \{1, ..., m\}$, the function $E_{\mathtt{C}_i}^T g$ from lists of *Terms* to lists of *Terms* is defined to be $[\mathcal{E}_g^T t_{i1}, ..., \mathcal{E}_g^T t_{ik_i}]$ when $t_i = [t_{i1}, ..., t_{ik_i}]$ is the list of arguments to $\mathtt{C}_i$.

The function $map^{T'}$ above has type $\forall\delta_1...\delta_k\rho_1...\rho_k.(\delta_1 \to \rho_1) \to ... \to (\delta_k \to \rho_k) \to T\delta_1...\delta_k \to T\rho_1...\rho_k$. Note that such map functions need not be directly expressible in the term language.

Although β is bound in T, the intent here is that $\mathcal{E}_g^T$ is defined with respect to the particular identity of that bound variable. That is, if the bound variables in T are renamed, then the definition of $\mathcal{E}_g^T$ should change accordingly.

The example below illustrates how the function $E_{\mathtt{Nil}}^{List}$ and $E_{\mathtt{Cons}}^T$ capture the recursive structure of the list type constructor.

Example 3.20 If $List = \forall\alpha.\mathtt{Rec}\beta.\mathtt{Nil}[\ ]|\mathtt{Cons}[\alpha, \beta]$ and g is a term of type $(List\ \gamma) \to \delta$ for some types γ and δ, then $E_{\mathtt{Nil}}^{List} g = [\ ]$ and $E_{\mathtt{Cons}}^{List} g = [\mathcal{E}_g^T \alpha, \mathcal{E}_g^T \beta] = [id_\alpha, g]$. Note that the actions of $E_{\mathtt{Nil}}^{List}$ and $E_{\mathtt{Cons}}^{List}$ are the same on every function g, no matter which instance of the *List* datatype is the domain of g.

In defining the basic reduction rules for warm fusion we will use the standard vector notation for a list of terms and types, so that an expression of the form $\overline{x}$ will stand for a list of the form $[x_1, ..., x_n]$. If $\overline{x}$ and $\overline{y}$ are lists, then $e[\overline{x} := \overline{y}]$ is used as shorthand for the simultaneous substitution $e[x_1 := y_1, ..., x_n := y_n]$. In the fifth rule of Definition 3.23, the list $\overline{x}$ includes both term and type variables, but the list $\overline{v}$ in the third includes only term variables. The omission of type variables in the third rule is consistent with the representation of case statements in GHC.

According to the basic reduction rules, β- and case-reduction steps are allowed under certain conditions involving linearity of variables and when certain subterms of the β- and case-redices are values. Values are given by

Definition 3.21 A *value* is a term of one of the following forms:

- every variable, constant, skolem constant, or constructor is a value,
- every term of the form $\lambda x :: t.e$ is a value,
- every term of the form $\Lambda\alpha.e$ is a value,
- every term of the form $\texttt{cata}\ [t_1, ..., t_n]\ t\ T\ [e_1, ..., e_m]$, where $e_1, ..., e_m$ are values, is also a value, and
- every term of the form $\texttt{build}\ [t_1, ..., t_n]\ T\ e$, where e is a value, is also a value.

We are now in a position to define the rules which will provide the backbone of the warm fusion calculus.

Definition 3.22 The set $\mathcal{R}$ of *basic reduction rules* of the warm fusion calculus are:

$(\lambda v.b)e$	$\perp\!\!\rightarrow$	$b[v := e]$ if v linear in b or e is a value
$(\Lambda\alpha.e)t$	$\perp\!\!\rightarrow$	$e[\alpha := t]$
$\texttt{case}\ C_i\overline{x}\ \texttt{of}\ C_1\overline{v_1} \rightarrow e_1\ \vert ...\vert\ C_n\overline{v_n} \rightarrow e_n$	$\perp\!\!\rightarrow$	$e_i[\overline{v_i} := \overline{x}]$ if v_{ij} linear in e_i for all i, j or each $x_j \in \overline{x}$ is a value
$(\texttt{cata}_t^{T\overline{ts}}\ \overline{fs})(\texttt{build}^{T\overline{ts}} g)$	$\perp\!\!\rightarrow$	$g\ [t]\ \overline{fs}$
$(\texttt{cata}_t^{T\overline{ts}}\ \overline{fs})(C_i\overline{x})$	$\perp\!\!\rightarrow$	$fs_i(E_{C_i}^T(\texttt{cata}_t^{T\overline{ts}}\ \overline{fs})\ \overline{x})$
$(\texttt{case}\ e\ \texttt{of}\ C_1\overline{v_1} \rightarrow e_1\ \vert ...\vert\ C_n\overline{v_n} \rightarrow e_n)\ e'$	$\perp\!\!\rightarrow$	$\texttt{case}\ e\ \texttt{of}\ C_1\overline{v_1} \rightarrow e_1e'\ \vert ...\vert\ C_n\overline{v_n} \rightarrow e_ne'$
$\lambda x.fx$	$\perp\!\!\rightarrow$	f

We intend, of course, that any subterm of any term which is a substitution instance of the left-hand side of any of these rules can be rewritten, and so we are actually interested in the *term rewriting system* generated by these rules. That is, we are interested in the closure of these rules with respect to monotonicity and substitution. At present, whether or not this term rewriting system — let alone its extension by the inference rules in Definition 3.23 — is either confluent or terminating is an open question; see Section 5 for a discussion of these issues. All of the rules of Definition 3.22 are, however, type-preserving and correct with respect to our intended semantics. The form of the β-reduction rule appearing there indicates that this semantics is nonstrict.

The linearity conditions in this definition prevent β- and case-reductions which would lead to duplication of work, and this will in turn prevent some functions with catamorphic representations from being transformed — by the basic reduction rules and the inference rule of Definition 3.23 below — into catamorphic form. Note that the case-application rule (the next-to-last rule) entails the risk of code explosion, but does not introduce any extra work into a computation since evaluating the case actually selects exactly one of its branches.

The η-reduction rule for the polymorphic lambda calculus is not explicitly included in the calculus of [LS95], but repeated use of the rule is, in fact, made throughout that paper. We therefore include it explicitly in $\mathcal{R}$. The β-reduction rule for types is also not present in calculus of [LS95] although certainly this was intended. The η-rule for types, on the other hand, is incompatible with the requirement that type constructors always be fully applied, and so we do not include it in our set of basic rewrite rules; no similar problem arises for the η-rule for terms, though, since even partially applied data constructors are valid terms.

Although the basic rewrite rules are not alone capable of transforming an arbitrary recursive program into `build-cata` form, they are suitable for simplifying program bodies written in terms of other programs, some or all of which may already be in `build-cata` form. But in order to actually put programs into `build-cata` form, a transformation encoding first-order catamorphic fusion is still needed. Like the higher-order equational logic for the polymorphic lambda calculus with shortcut, the Promotion Theorem, on which catamorphic fusion is based, can also be described in terms of rewrite rules. In fact, for each catamorphic fusion to be performed, a new set of rewrite rules is generated "on the fly" and added to the set of basic rules. Precisely which new rules are dynamically generated for a given fusion is determined by the type constructor over which the catamorphism involved in the fusion is defined, together with the program to be fused with

that catamorphism. One rule is generated for each data constructor of the catamorphism's type constructor, and together these rules are used to rewrite certain terms constructed from the program to be fused with the catamorphism and the functional arguments to that catamorphism. When the terms this rewriting produces satisfy a certain "purity" condition, functional arguments to the new catamorphism resulting from the fusion can be extracted.

To illustrate, suppose a function g is to be fused with an application of the form $\texttt{cata}\,\texttt{f}_1...\texttt{f}_\texttt{m}\,\texttt{x}$, where the type constructor over which the catamorphism is defined is

$$\Lambda\alpha_1...\alpha_n.\texttt{Rec}\,\beta.\texttt{C}_1 t_1|...|\texttt{C}_m t_m,$$

and, for each i, t_i is a list $[t_{i1},...,t_{ik_i}]$ representing the types of the arguments to $\texttt{C}_i$. For each data constructor $\texttt{C}_i$, a set of rules of the form

$$E_{\texttt{C}_i}(g)y_{ij} \perp\!\!\!\rightarrow z_{ij},$$

for $j \in \{1,...,k_i\}$, is dynamically generated, where y_{ij} and z_{ij} are new skolem variables. Skolem variables are not true variables; the intent is that skolem variables never instantiated since they serve as mere place holders for the term arguments of data constructors which record the recursive structure of type constructors.

For each $i \in \{1,...,m\}$, the set of rewrite rules dynamically generated from g and $\texttt{cata}\,\texttt{f}_1...\texttt{f}_\texttt{m}\,\texttt{x}$ is now used, together with the basic rewrite rules, to rewrite the terms

$$\lambda z_{i1}...z_{ik_i}.g(f_i y_{i1}...y_{ik_i})$$

which code up the hypotheses of the Promotion Theorem. The goal is to thereby arrive at a collection of terms $h_1,...,h_m$ which represent the functional arguments to the catamorphism obtained by successful fusion.

Well-formed definitions $h_1,...,h_m$ will be obtainable precisely when the above terms can be rewritten into closed terms, i.e., into function definitions. Note, however, that an essential feature of the way catamorphisms act on data structures involves recursively applying themselves to similarly constructed substructures, and so we can expect to arrive at closed terms defining the functional arguments to an actual *catamorphism* representing the composition $g \circ \texttt{cata}\,\texttt{f}_1...\texttt{f}_\texttt{m}\,\texttt{x}$ precisely if g is applied to every "recursive" argument to every data constructor of the type constructor over which $\texttt{cata}\,\texttt{f}_1...\texttt{f}_\texttt{m}\,\texttt{x}$ is defined. Intuitively speaking, the rewrite rules generated for the constructor $\texttt{C}_i$ can be thought of as rewriting occurrences of y_{ij} which are "non-recursive" arguments to $\texttt{C}_i$, and subexpressions of the form gy_{ij} for y_{ij} occurring as "recursive" arguments to $\texttt{C}_i$, into their corresponding skolem constants z_{ij}. In particular, these rewrite rules eliminate exactly those recursive variables to which g has already been applied. A definition for each h_i, $i \in \{1,...,m\}$, can therefore be expected iff rewriting removes from the corresponding term $\lambda z_{i1}...z_{ik_i}.g(f_i y_{i1}...y_{ik_i})$ all occurrences of the variables y_{ij} in favor of occurrences of $z_{i1},...,z_{ik_i}$.

To make the above precise, we augment the set of basic reduction rules with the inference rule of the next definition.

Definition 3.23 The set $\mathcal{WF}$ of rules for the warm fusion calculus comprises the set $\mathcal{R}$ of basic reduction rules, together with the following inference rule encoding the Promotion Theorem. If $\texttt{cata}^T\ f_1...f_m$ and $\texttt{cata}^T\ h_1...h_m$ are catamorphisms over the type constructor

$$T = \forall\alpha_1...\alpha_n.\texttt{Rec}\,\beta.\texttt{C}_1 t_1|...|\texttt{C}_m t_m,$$

then

$$\frac{\forall i \in \{1,...,m\},\ \forall j \in \{1,...,k_i\}:\ \ \mathcal{R} \cup \{E_{\texttt{C}_i}(g)y_{ij} \perp\!\!\!\rightarrow z_{ij}\} \vdash \lambda z_{i1}...z_{ik_i}.g(f_i y_{i1}...y_{ik_i}) \perp\!\!\!\perp\!\!\!\twoheadrightarrow h_i}{\mathcal{R} \vdash g(\texttt{cata}^T\ f_1...f_m\ x) \perp\!\!\!\rightarrow \texttt{cata}^T\ h_1...h_m\ x,}$$

provided y_{ij} does not appear free in h_l for $l \in \{1,...,m\}$.

It is the monotonic closure of this inference rule, together with the term rewriting system generated by $\mathcal{R}$, in which we are actually interested. In fact, if we agree that skolem variables appearing in the bodies of terms in the hypothesis of the inference rule are to be treated like constants and never be instantiated, then we can actually look at the term rewriting system generated by $\mathcal{WF}$ itself. We'll adopt this latter point of view in this paper.

When applying warm fusion to an input program, our engine must dynamically generate the variables y_{ij} and z_{ij}, together with their types. These types will be determined by the recursive structure of the type constructor T, together with the result type of g. They must be such that all of the terms appearing in any inference rules in which they appear must be well-typed. If $\texttt{cata}^T\ f_1...f_m$ consumes data of type $List|alpha = \texttt{Rec}\beta.\texttt{Nil}[\ ]|\texttt{Cons}[\alpha,\beta]$, for example, then z_{12} (which corresponds to the first argument to `Cons` at type α) has type α, and z_{22} (which corresponds to the second argument to `COns` at type α) has type $List\beta \rightarrow \forall\gamma.\gamma \rightarrow$ *help!*

Like the rules of $\mathcal{R}$, the inference rule of Definition 3.23 is correct with respect to our intended semantics; whether or not the term rewriting system generated by $\mathcal{WF}$ is confluent or terminating remains an open question. In the inference rule, the derivation of *any* terms h_i not containing (free occurrences of) the skolem variables y_{ij} suffices to insure that the composition $g\,(\texttt{cata}f_1...f_mx)$ has a catamorphic representation. But from a practical, computational point of view, it is best if these terms are $\mathcal{WF}$-normal forms.

In the applications of the Promotion Theorem in which we are interested, the functions playing the role of g in the above inference rule will always be the workers of programs undergoing warm fusion. In our encoding of the Promotion Theorem, these workers are always unfolded exactly once. But since the workers of the programs which are eligible for non-trivial processing by our implementation are always abstracted case statements, this simply means that the branches of the case statements which apply to the various terms $f_iy_{i1}...y_{ik_i}$ are determined, and that appropriately instantiated versions of their right hand sides are made available for transformation by their corresponding dynamically generated sets of rewrite rules. This makes available for scrutiny the recursive structure of the program being processed, which is, of course, essential to deriving a catamorphic form for its worker.

Incidentally, unfolding the definition of a function being fused with a catamorphism more than once might possibly lead to a larger number of successful fusions than are obtainable via single unfolding. There is, however, no way to predict ahead of time how many unfoldings will suffice, or even if some finite number of them ultimately will.

3.3 An Implementation of Warm Fusion

In this section we describe our implementation of a warm fusion-based virtual data structure elimination method. Recall that our implementation currently eliminates virtual data structures only from programs which can be represented as abstracted case statements (the input program is returned unchanged if it does not meet this criterion). Moreover, since we have not implemented higher-order catamorphic fusion, our implementation achieves only the first warm fusion method from [LS95], as explained in Section 3.1 of this paper. Finally, as noted there, there is no need to also compose the program to be processed with the appropriate copy function at the beginning of the warm fusion process, since delaying this composition actually reduces the complexity of the intermediate programs to be manipulated by the fusion engine. For programs in the language of Definition 3.9 with the required syntactic form, we achieve warm fusion via the following steps:

1. *Composing, on the left, the program being processed with the* `build-cata` *representation of an appropriate identity function.* This is achieved by introducing a `build-cata` pair according to the following specification, which extends that in [LS95] to accommodate explicit polymor-

phism.

$$\begin{array}{lcl}
\mathcal{B}(\lambda x.e) & = & \lambda x.\mathcal{B}e \\
\mathcal{B}(\Lambda t.e) & = & \Lambda t.\mathcal{B}e \\
\mathcal{B}e & = & \texttt{build}\ [\tau_1,...,\tau_n]\ T\ (\Lambda t.\lambda p_1...p_m.\texttt{cata}\ [\tau_1,...,\tau_n]\ t\ T\ [p_1,...,p_m]\ e) \\
 & & \quad \text{where } e \text{ has type } T\tau_1...\tau_n \text{ and } T \text{ has arity } n \text{ and } m \text{ summands.} \\
\mathcal{B}e & = & e \text{ otherwise}
\end{array}$$

In the third clause, p_i has type $t_{i1} \rightarrow ... \rightarrow t_{ik_i} \rightarrow t$, where data constructor $\texttt{C}_i$ in T takes arguments of types $t_{i1},...,t_{ik_i}$. Note that the particular `build-cata` pair introduced by $\mathcal{B}$ is completely determined by the type of e.

2. *Distributing the newly introduced* `cata` *over the body of the program to be processed.* The fact that the catamorphism is strict justifies this manipulation.

3. *Reducing, via the basic reduction rules, the "distributed" program obtained in Step 2.*

4. *Breaking the resulting reduced, distributed program into its wrapper and its worker.* Consider a program body

$$f\ =\ Lq_1...Lq_n.\texttt{build}[\tau_1,...,\tau_n]T(\Lambda t.\lambda p_1...p_m\texttt{cata}[\tau_1,...,\tau_n]tT[P_1,...,p_m](\texttt{case}\ q_k\ \texttt{of}\ pats),$$

where $q_1,...,q_n$ are either type or term variables, L denotes λ or Λ as appropriate, and q_k is a term variable. Its wrapper is given by

$$f\ =\ Lq_1...Lq_n\texttt{build}[\tau_1,...,\tau_n]T(\Lambda t.\lambda p_1...p_m\texttt{cata}[\tau_1,...,\tau_n]tT[P_1,...,p_m](f\#q_kv_1...v_l),$$

and its worker is given by

$$\texttt{f\#}\ =\ \backslash \texttt{q}_\texttt{k} \rightarrow \texttt{Lv}_1...\texttt{Lv}_1 \rightarrow \texttt{case}\ \texttt{q}_\texttt{k}\ \texttt{of}\ pats,$$

where $\{v_1,...,v_l\} = FV(pats) \perp FV(f) \perp \{q_k\}$.

5. *Unfolding the program's wrapper in the worker and reducing its resulting recursive definition of its worker.*

6. *Inlining, in the recursive representation of the worker, called functions which are already in* `build-cata` *form, and reducing the resulting expanded representation of the worker.* Inlined functions can be any from the same input file as the program to be processed. In fact, our fusion engine will process into `build-cata` form an entire file of programs in input order, and, when processing each program, will inline all other programs in the file which are already in `build-cata` form.

7. *Composing, on the right, the expanded representation of the worker with the appropriate copy function, and fusing the resulting composition to obtain a catamorphic representation of the worker.* The particular copy function used in the composition is determined by the domain type of the worker (and, hence, by the domain type of the input program). If the worker has domain type $T\tau_1...\tau_n$, where $T = \forall\alpha_1...\alpha_n.\texttt{C}_1t_1|...|\texttt{C}_mt_m$, and, for each $i \in \{1,...,m\}$, $t_i = [t_{i1},...,t_{ik_i}]$ is a list of the types of arguments the data constructor $\texttt{C}_i$ takes, then the copy function used in the composition is

$$\forall\alpha_1...\alpha_n.\texttt{cata}\ [\alpha_1,...,\alpha_n]\ (T[\alpha_1]...[\alpha_n])\ T\ [\texttt{C}_1[\alpha_1]...[\alpha_n],...,\texttt{C}_m[\alpha_1]...[\alpha_n]],$$

which has type $\forall\alpha_1...\alpha_n\,.T\alpha_1...\alpha_n \rightarrow T\alpha_1...\alpha_n$. As explained at the end of the preceding section, our implementation unfolds a function exactly once during its fusion with a catamorphism.

8. *Inlining, in the wrapper, the catamorphic representation of the worker and reducing the resulting expression via the set $\mathcal{WF}$ of rules to obtain a* `build-cata` *representation for the original input program.*

As in the implementation of any deterministic algorithm, choices have been made here as to the order in which the transformations on which the algorithm is based are to be performed (they are performed in a leftmost outermost fashion). In the absence of confluence and termination results for the warm fusion calculus — which are the subjects of ongoing work (see Section 5) — we must expect that this choice may dramatically impact the behavior of our implementation.

Our fusion engine takes as input some number of datatype definitions and function definitions. The datatype definitions can be recursive, as in

$$\texttt{List}\ \alpha\ =\ \texttt{Cons}\ \alpha\ (\texttt{List}\ \alpha)\ |\ \texttt{Nil},$$

or non-recursive, as in

$$\texttt{Pair}\ \alpha\ \beta\ =\ \texttt{MkPair}\ \alpha\ \beta.$$

From these, expressions representing the corresponding type constructors —

$$\forall\alpha.\texttt{Rec}\ \beta.\texttt{Cons}\ \alpha\ \beta\ |\ \texttt{Nil}$$

and

$$\forall\alpha\beta.\texttt{Rec}\ \gamma.\texttt{MkPair}\ \alpha\ \beta,$$

for example — are automatically generated. Note that the term variables and constants appearing in function definitions must be explicitly and fully polymorphically typed at input. For instance, *map*, which really has type $\forall\,\alpha\,\beta.\,(\alpha \to \beta) \to \mathit{List}\ \alpha \to \mathit{List}\ \beta$, must be defined as

```
map :: (forall alpha . (forall beta .
            (alpha -> beta) -> (List alpha) -> (List beta))) =
        (/\ alpha . /\ beta . \ f::(alpha -> beta) . \ xs::(List alpha) .
              (case (xs::(List alpha)) of {
                 Cons (z::alpha) (zs::(List alpha)) ->
                     (Cons [beta])((f::(alpha -> beta))(z::alpha))
                        (((((map::(forall alpha . (forall beta .
                              (alpha -> beta) -> (List alpha) -> (List beta))))
                             [alpha])
                             [beta])
                             (f::(alpha -> beta)))
                             (zs::(List alpha)));
                 Nil -> (Nil [beta])}))
```

rather than simply as in Section 3.1. The actual processing by our fusion engine for *map* yields the following intermediate programs at the various stages of manipulation:

1.

```
map :: (forall alpha.(forall beta.
                ((alpha -> beta) -> ((List alpha) -> (List beta))))) =
          /\alpha. /\beta. \f::(alpha -> beta). \xs::(List alpha).
            build {beta} List
              (/\t0. \p1::(beta -> (t0 -> t0)). \p2::t0.
                 cata {beta} t0 List
                    [(p1::(beta -> (t0 -> t0))),
                     (p2::t0)]
                   (case xs::(List alpha) of
                      Cons::(forall alpha.(alpha ->
                           ((List alpha) -> (List alpha))))
                              (z::alpha) (zs::(List alpha)) ->
                         Cons::(forall alpha.(alpha ->
                              ((List alpha) -> (List alpha))))
```

```
                          beta
                          (f::(alpha -> beta)
                          (z::alpha))
                          (map::(forall alpha.(forall beta.((alpha -> beta) ->
                                  ((List alpha) -> (List beta)))))
                             alpha
                             beta
                             (f::(alpha -> beta))(zs::(List alpha)))
                       Nil::(forall alpha.(List alpha)) ->
                          Nil::(forall alpha.(List alpha)) beta))
```

2.

```
map :: (forall alpha.(forall beta.
                ((alpha -> beta) -> ((List alpha) -> (List beta))))) =
          /\alpha. /\beta. \f::(alpha -> beta). \xs::(List alpha).
            build {beta} List
              (/\t0. \p1::(beta -> (t0 -> t0)). \p2::t0.
                 cata {beta} t0 List
                    [(p1::(beta -> (t0 -> t0))),
                     (p2::t0)]
                   (case xs::(List alpha) of
                      Cons::(forall alpha.(alpha ->
                           ((List alpha) -> (List alpha))))
                              (z::alpha) (zs::(List alpha)) ->
                         p1::(beta -> (t0 -> t0))
                           (f::(alpha -> beta)
                           (z::alpha))
                           (cata {beta} t0 List
                             [(p1::(beta -> (t0 -> t0))),
                              (p2::t0)]
                               (map::(forall alpha.(forall beta.((alpha ->
                                     beta) -> (List alpha) -> (List beta)))))
                                alpha
                                beta
                                (f::(alpha -> beta))(zs::(List alpha)))
                      Nil::(forall alpha.(List alpha)) ->
                           p2 :: t0))
```

3.

```
map::(forall alpha.(forall beta.((alpha -> beta) -> ((List alpha) -> (List beta)))))
  = /\alpha./\beta.\f::(alpha -> beta).
                   \xs::(List alpha).
                   build {beta} List
                   (/\t0.
                    \p1::(beta -> (t0 -> t0)).
                    \p2::t0.
                    case xs::(List alpha) of
                      Cons::(forall alpha.(alpha -> ((List alpha) -> (List alpha)))) (z::alpha)
                                                                                     (zs::(List alpha
                                                                                     (z::alpha))
                                                                                    (cata {beta} t0 L
                                                                                     [(p1::(beta -> (
                                                                                      (p2::t0)]
                                                                                     (map::(forall al
```

```
                                                                                  beta
                                                                                  (f::(alpha -> beta
                                                                                     (zs::(List alph
                    Nil::(forall alpha.(List alpha)) -> p2::t0)
```

4.

```
wrdef =

map::(forall alpha.(forall beta.((alpha -> beta) -> ((List alpha) -> (List beta)))))
  = /\alpha./\beta.\f::(alpha -> beta).
                   \xs::(List alpha).
                   build {beta} List
                   (/\t0.
                    \p1::(beta -> (t0 -> t0)).
                    \p2::t0.
                    map4::((List alpha) -> (forall t0.((beta -> (t0 -> t0)) -> ((alpha -> beta) -> (
                                                                                 t0
                                                                                 (p1::(beta -> (t0 -
                                                                                 (f::(alpha -> beta)
                                                                                 (p2::t0))


wodef =
map::(forall alpha.(forall beta.((alpha -> beta) -> ((List alpha) -> (List beta)))))
  = \xs::(List alpha).
      /\t0.\p1::(beta -> (t0 -> t0)).
           \f::(alpha -> beta).
           \p2::t0.
           case xs::(List alpha) of
             Cons::(forall alpha.(alpha -> ((List alpha) -> (List alpha)))) (z::alpha)
                                                                           (zs::(List alpha)) -> p1
                                                                                  (z::alpha))
                                                                                  (cata {beta} t0 Li
                                                                                   [(p1::(beta -> (t
                                                                                    (p2::t0)]
                                                                                   (map::(forall alp
                                                                                 beta
                                                                                 (f::(alpha -> beta)
                                                                                    (zs::(List alpha
             Nil::(forall alpha.(List alpha)) -> p2::t0
```

5.

```
map::(forall alpha.(forall beta.((alpha -> beta) -> ((List alpha) -> (List beta)))))
  = \xs::(List alpha).
      /\t0.\p1::(beta -> (t0 -> t0)).
           \f::(alpha -> beta).
           \p2::t0.
           case xs::(List alpha) of
             Cons::(forall alpha.(alpha -> ((List alpha) -> (List alpha)))) (z::alpha)
```

```
                                                                    (zs::(List alpha)) -> p1
                                                                           (z::alpha))
                                                                          (map4::((List alph
                                                                              t0
                                                                              (p1::(beta ->
                                                                              (f::(alpha ->
                                                                           (p2::t0))
            Nil::(forall alpha.(List alpha)) -> p2::t0
```

6.

```
map::(forall alpha.(forall beta.((alpha -> beta) -> ((List alpha) -> (List beta)))))
  = \xs::(List alpha).
      /\t0.\p1::(beta -> (t0 -> t0)).
           \f::(alpha -> beta).
           \p2::t0.
           case xs::(List alpha) of
             Cons::(forall alpha.(alpha -> ((List alpha) -> (List alpha)))) (z::alpha)
                                                                    (zs::(List alpha)) -> p1
                                                                           (z::alpha))
                                                                          (map4::((List alph
                                                                              t0
                                                                              (p1::(beta ->
                                                                              (f::(alpha ->
                                                                           (p2::t0))
             Nil::(forall alpha.(List alpha)) -> p2::t0
```

7.

```
map::(forall alpha.(forall beta.((alpha -> beta) -> ((List alpha) -> (List beta)))))
  = \y35::(List alpha).
    cata {alpha} t0 List
    [(\z39::
      alpha.
      \z40::(forall t0.((beta -> (t0 -> t0)) -> ((alpha -> beta) -> (t0 -> t0)))).
        /\t41.\p42::(beta -> (t41 -> t41)).
              \f43::(alpha -> beta).
              \p44::t41.
              p42::(beta -> (t41 -> t41)) (f43::(alpha -> beta)
                                           (z39::alpha))
                                          (z40::(forall t0.((beta -> (t0 -> t0)) -> ((alpha -> beta)
                                                                                     (p42::(beta ->
                                                                                     (f43::(alpha ->
                                           (p44::t41)))
     (/\t55.
      \p56::(beta -> (t55 -> t55)).
      \f57::(alpha -> beta).
      \p58::t55. p58::t55)] (y35::(List alpha))
```

8.

```
map::(forall alpha.(forall beta.((alpha -> beta) -> ((List alpha) -> (List beta)))))
  = /\alpha./\beta.\f::(alpha -> beta).
                   \xs::(List alpha).
                   build {beta} List
                   (/\t0.
                    \p1::(beta -> (t0 -> t0)).
                    cata {alpha} t0 List
                    [(\z63::
                      alpha.
                      \z64::(forall t65.((beta -> (t65 -> t65)) -> ((alpha -> beta) -> (t65 -> t65))
                        /\t66.\p67::(beta -> (t66 -> t66)).
                              \f68::(alpha -> beta).
                              \p69::t66.
                              p67::(beta -> (t66 -> t66)) (f68::(alpha -> beta)
                                                           (z63::alpha))
                                                          (z64::(forall t65.((beta -> (t65 -> t65))
                                                                                (p67::(beta -> (t6
                                                                                (f68::(alpha -> be
                                                           (p69::t66)))
                     (/\t71.
                      \p72::(beta -> (t71 -> t71)).
                      \f73::(alpha -> beta).
                      \p74::t71.
                      p74::t71)] (xs::(List alpha))
                                 t0
                                 (p1::(beta -> (t0 -> t0)))
                                 (f::(alpha -> beta)))
```

We don't optimize for static parameters We don't generate maps.

3.4 Warm Fusion Examples

We present in this section examples of `build-cata` forms obtained by our implementation of warm fusion.

The following type constructor declarations are input to our warm fusion tool for the examples. We present the declarations used in the various examples all at once simply to avoid code repetition in our presentation of the examples. Of course, only the declarations used in a given example need to be input when that example is run. Following the type constructor declarations, we give a list of input definitions and the corresponding output generated by our tool. Outputs are given twice, once with full type information, and once in a more digestible form in which type information has been suppressed; `build-cata` forms obtained earlier are sometimes used in processing programs which are presented later.

```
data String = List Char

data Int = Zero | Succ Int

data Pair alpha beta = MkPair alpha beta

data List a = Cons a (List a) | Nil
```

```
data Tree alpha = Node (Tree alpha) alpha (Tree alpha) | Tip

data Exp = Num Int | Id String | Plus Exp Exp

data Code = LoadI Int | LoadV String | Add
```

3.4.1 Append

Given input

```
append :: (forall alpha. (List alpha) -> (List alpha) -> (List alpha)) =
   (/\ alpha . (\ xs :: (List alpha) . (\ ys :: (List alpha) .
      (case (xs::(List alpha)) of {
          Cons (z::alpha)(zs::(List alpha)) ->
               (((Cons [alpha])(z::alpha))
                  ((((append::(forall alpha. (List alpha) ->
                                             (List alpha) ->
                                             (List alpha)))
                     [alpha])
                     (zs::(List alpha)))
                     (ys::(List alpha))));
           Nil -> (ys::(List alpha))}))))
```

our warm fusion tool constructs the `build-cata` form

```
append :: (forall alpha.((List alpha) -> ((List alpha) -> (List alpha)))) =
    /\alpha. \xs::(List alpha). \ys::(List alpha).
       build {alpha} List
         (/\t0. \p1::(alpha -> (t0 -> t0)). \p2::t0.
              cata {alpha} (List alpha) List
                   [(\z40::alpha.
                     \z41::recvarList.
                     \p42::(alpha -> (t0 -> t0)).
                     \p43::t0.
                     \ys44::(List alpha).
                     p42::(alpha -> (t0 -> t0)) (z40::alpha)
                            (z41::recvarList (p42::(alpha -> (t0 -> t0)))
                                             (p43::t0)
                                             (ys44::(List alpha))))
                    (\p45::(alpha -> (t0 -> t0)).\p46::t0.
                        cata {alpha} t0 List
                             [(p45::(alpha -> (t0 -> t0)))
                              (p46::t0)])]
                        (xs::(List alpha))
                        (p1::(alpha -> (t0 -> t0)))
                        (p2::t0)
                        (ys::(List alpha)))
```

or, suppressing type information,

```
append = \xs.\ys.build List
            (\p1.\p2.cata List
                      (\z40.\z41.\p42.\p43.\ys44.p42 z40
                                                     (z41 p42
                                                          p43
                                                      ys44))
```

```
                    (\p45.\p46.cata List
                                p45 p46) xs
                                         p1
                                         p2
                                         ys)
```

3.4.2 Concat

Given input

```
rec version of concat goes here
```

our warm fusion tool constructs the `build-cata` form
   *

Help!

or, suppressing type information,

```
type-suppressed version of b-c form for concat goes here.
```

In the examples below, the following `build-cata` forms for *map*, *append*, and *concat* are used. These are completely equivalent to, but somewhat simpler than, the forms derived by our implementation. *

We don't use the build-cata forms deriv from our implementation becaus they don't optimize fo static parameters, and the ones we generate hand do.

```
map :: (forall alpha . (forall beta .
            (alpha -> beta) -> (List alpha) -> (List beta))) =
  (/\ alpha . /\ beta . \ f::(alpha -> beta) . \ xs::(List alpha) .
     (build {[beta] List
        (/\ gamma . \c::(beta -> gamma -> gamma) . \n::gamma .
            (cata {[alpha] gamma List
                [\a::alpha . \b::gamma .
                      (c::(beta -> gamma -> gamma))
                      ((f::(alpha -> beta)) (a::alpha))
                      (b::gamma),
                 n::gamma]})
            xs::(List alpha))})),

append :: (forall alpha . (List alpha) -> (List alpha) -> (List alpha)) =
  (/\ s . \xs::(List s) . \ys::(List s) .
     (build {[s] List
        (/\t . \c::(s -> t -> t) . \n::t .
          (cata {[s] t List
                   [c::(s -> t -> t),
                     (cata {[s] t List
                         [c::(s->t->t), n::t]}) (ys::(List s))]})
          (xs::(List s)))})),

concat :: (forall alpha . (List (List alpha)) -> (List alpha)) =
  (/\ alpha . \xs::(List (List alpha)) .
      (build {[alpha] List
         (/\ gamma . \c::(alpha -> gamma -> gamma) . \n::gamma .
           (cata {[(List (List alpha))] gamma List
               [\ys::(List alpha) . \y::gamma .
                  (cata {[alpha] gamma List
```

```
                  [c::(alpha -> gamma -> gamma),
                   y::gamma]} (ys::(List alpha))),
              n::gamma]})
          xs::(List (List alpha)))}))
```

3.4.3 Zip

Given input

```
zip :: (forall a. (forall b. (List a) -> (List b) -> (List (Pair a b)))) =
     /\ a . /\ b . \ x::(List a) . \ y::(List b) .
          (case (x::(List a)) of {
             Nil -> (Nil [(Pair a b)]);
             Cons (u::a) (v::(List a)) ->
                     (case (y::(List b)) of {
                          Nil -> (Nil [(Pair a b)]);
                          Cons (c::b) (d::(List b)) ->
                            (Cons [(Pair a b)])
                               (((MkPair [a]) [b]) (u::a) (c::b))
                               ((((zip :: (forall a. (forall b. (List a) ->
                                          (List b) -> (List (Pair a b)))))
                                  [a]) [b])
                                     (v::(List a))
                                  (d::(List b)))})})
```

our warm fusion tool constructs the `build-cata` form

```
zip :: (forall a.(forall b.((List a) -> ((List b) -> (List (Pair a b))))))) =
    /\a. /\b. \x::(List a). \y::(List b).
      build {(Pair a b)} List
        (/\t0. \p1::(alpha -> (t0 -> t0)). \p2::t0.
           cata {a} (List a) List
             [(\z69::\alpha. \z70::recvarList. \p71::t0.
                 \p72::(alpha -> (t0 -> t0)). \y73::(List b).
               cata {(Pair a b)} t0 List
                  [(p72::(alpha -> (t0 -> t0)))
                   (p71::t0)]
               (case y73::(List b) of
                 Nil::(forall alpha.(List alpha)) ->
                    Nil::(forall alpha74.(List alpha74))(Pair a b)
                 Cons::(forall alpha.(alpha ->
                           ((List alpha) -> (List alpha))))
                       (c75::b)(d76::(List b)) ->
                    Cons::(forall alpha77.(alpha77 ->
                              ((List alpha77) -> (List alpha77))))
                       (Pair a b)
                       (MkPair::(forall alpha78.(forall beta79.
                           (alpha78 -> (beta79 -> (Pair alpha78 beta79)))))
                       a
                       b
                       (z69::alpha)
                       (c75::b))
        (build {(Pair a b)} List
          (/\t80. \p81::(alpha -> (t80 -> t80)). \p82::t80.
             z70::recvarList (p82::t80)
```

```
            (p81::(alpha -> (t80 -> t80)))
            (d76::(List b))))))
            (\p83:: t0. \p84::(alpha -> (t0 -> t0)). \y85::(List b).
            p83::t0)] (x::(List a))
            (p2::t0)
            (p1::(alpha -> (t0 -> t0)))
            (y::(List b)))
```

or, suppressing type information,

```
zip = \x.\y.build List
         (\p1.\p2.cata List
           (\z69.\z70.\p71.\p72.\y73.cata List
                                    p72
                                    p71 (case y73 of
                                           Nil -> Nil
                                           Cons c75 d76 ->
                                              Cons (MkPair z69 c75)
                                        (build List
                                          (\p81.\p82.z70 p82 p81 d76))))
                              (\p83.\p84.\y85.p83) x
                                                   p2
                                                   p1
                                                   y)
```

3.4.4 Flatten

Given input

```
flatten :: (forall alpha . (Tree alpha) -> (List alpha)) =
   /\ alpha . \ t :: (Tree alpha) .
       (case (t::(Tree alpha)) of {
           Node (a::(Tree alpha)) (b::alpha) (c::(Tree alpha)) ->
              ((append :: (forall alpha . (List alpha) ->
                             (List alpha) -> (List alpha)))
              [alpha])
              (((flatten :: (forall alpha . (Tree alpha) -> (List alpha)))
               [alpha])
               a :: (Tree alpha))
              ((Cons [alpha])
                     (b::alpha)
                     (((flatten :: (forall alpha . (Tree alpha) ->
                                      (List alpha)))
                       [alpha])
                       (c :: (Tree alpha))));
         Tip -> (Nil [alpha])
        })
```

our warm fusion tool constructs the `build-cata` form

```
flatten :: (forall alpha.((Tree alpha) -> (List alpha))) =
         /\alpha.\t::(Tree alpha).
          build {alpha} List
            (/\t0. cata {alpha} (Tree alpha) Tree
                          [(\z66::recvarTree. \z67::alpha.
                             \z68::recvarTree. \p69::(alpha -> (t0 -> t0)).
```

```
                          \p70::t0.
             z66::recvarTree (p69::(alpha -> (t0 -> t0)))
            (p69::(alpha -> (t0 -> t0)) (z67::alpha)
                      (z68::recvarTree (p69::(alpha -> (t0 -> t0)))
                      (p70::t0))))
            (\p71::(alpha -> (t0 -> t0)). \p72::t0. p72::t0)]
          (t::(Tree alpha)))
```

or, suppressing type information,

```
flatten = \t.build List
            (cata Tree
              (\z66.\z67.\z68.\p69.\p70.z66 p69
                                            (p69 z67(z68 p69 p70)))
                             (\p71.\p72.p72)
                            t)
```

3.4.5 Reverse

Given input

```
reverse :: (forall alpha . ((List alpha) -> (List alpha))) =
(/\ alpha . (\ xs :: (List alpha) .
    (case (xs :: (List alpha)) of {
        Cons (z :: alpha) (zs :: (List alpha)) ->
              ((append ::
                (forall alpha.(List alpha) -> (List alpha) -> (List alpha)))
              [alpha])
              (((reverse:: (forall alpha . ((List alpha) -> (List alpha))))
                [alpha]) (zs :: (List alpha)))
                ((Cons [alpha]) (z :: alpha) (Nil [alpha]));
        Nil  -> (Nil [alpha])
      })))
```

our warm fusion tool constructs the `build-cata` form

```
reverse :: (forall alpha.((List alpha) -> (List alpha))) =
  /\alpha.\xs::(List alpha).
    build {alpha} List
      (/\t0. cata {alpha} (List alpha) List
                [(\z55::alpha. \z56::recvarList.
                    \p57::(alpha -> (t0 -> t0)). \p58::t0.
                       z56::recvarList (p57::(alpha -> (t0 -> t0)))
                       (p57::(alpha -> (t0 -> t0))
                       (z55::alpha)
                       (p58::t0))),
                 (\p59::(alpha -> (t0 -> t0)). \p60::t0. p60::t0)]
              (xs::(List alpha)))
```

or, suppressing type information,

```
reverse = \xs.build List
          (cata List
             (\z55.\z56.\p57.\p58.z56 p57 (p57 z55 p58))
             (\p59.\p60.p60)
           xs)
```

3.4.6 Postfix

Given input

```
postfix :: (Exp -> (List Code)) =
   \x::Exp . case (x::Exp) of {
       Num (n::Int) -> (Cons [Code]) (LoadI (n::Int)) (Nil [Code]);
       Id (s::String) -> (Cons [Code]) (LoadV (s::String)) (Nil [Code]);
       Plus (x::Exp) (y::Exp) ->
             ((append :: (forall alpha . (List alpha) ->
                                  (List alpha) -> (List alpha)))
                 [Code])
                ((postfix :: Exp -> (List Code)) (x::Exp))
                (((append :: (forall alpha . (List alpha) ->
                                  (List alpha) -> (List alpha)))
                [Code])
                ((postfix :: Exp -> (List Code)) (y::Exp))
                ((Cons [Code]) Add (Nil [Code])))}
```

our warm fusion tool constructs the `build-cata` form

```
postfix :: (Exp -> (List Code)) =
    \x::Exp.
    build {Code} List
    (/\t0. cata {} Exp Exp
             [(\z113:: Int. \p114::(alpha -> (t0 -> t0)).
                 p114::(alpha ->(t0 -> t0))
                 (LoadI::(Int -> Code)(z113::Int)))
              (\z115::String. \p116::(alpha -> (t0 -> t0)).
                 p116::(alpha -> (t0 -> t0))
                 (LoadV::(String -> Code)(z115::String)))
              (\z117::recvarExp. \z118::recvarExp.
               \p119::(alpha -> (t0 -> t0)). \p120::t0.
                  z117::recvarExp (p119::(alpha -> (t0 -> t0)))
                 (z118::recvarExp (p119::(alpha -> (t0 -> t0)))
                                  (p119::(alpha -> (t0 -> t0))
                                      (Add)::Code
                                      (p120::t0))))]
          (x::Exp))
```

or, suppressing type information,

```
postfix = \x.build List
       (cata Exp
          (\z113.\p114.p114 (LoadI z113))
          (\z115.\p116.p116 (LoadV z115))
          (\z117.\z118.\p119.\p120.z117 p119
                                        (z118 p119 (p119 (Add) p120)))
                        x)
```

3.4.7 Unlines

Given input

```
unlines :: (List (List Char)) -> (List Char) =
  (\ ls :: (List (List Char)) .
```

```
    (((concat :: (forall alpha . (List (List alpha)) -> (List alpha)))
       [Char])
       ((((map :: (forall alpha . (forall beta .
                   (alpha -> beta) -> (List alpha) -> (List beta))))
           [(List Char)])
           [(List Char)])
           (\ l :: (List Char) .
                ((append :: (forall alpha .
                           (List alpha) -> (List alpha) -> (List alpha)))
                [(List Char)])
                (l::(List Char))
                (newlineList::(List Char)))
           (ls::(List (List Char))))))
```

our warm fusion tool constructs the `build-cata` form

```
unlines::((List (List Char)) -> (List Char)) =
    \ls::(List (List Char)).
     build {Char} List
       (/\gamma6. \c7::(Char -> (gamma6 -> gamma6)). \n55::gamma6.
           cata {(List Char)} gamma6 List
                    [(\a56::(List Char). \n86::gamma6.
                         cata {(List Char)} gamma6 List
                            [(c7::(Char -> (gamma6 -> gamma6))),
                             (cata {(List Char)} gamma6 List
                                 [(c7::(Char -> (gamma6 -> gamma6))),
                                  (n86::gamma6)]
                             (newlineList::(List Char)))]
                        (a56::(List Char))),
                    (n55::gamma6)]
           (ls::(List (List Char))))
```

or, suppressing type information,

```
unlines = \ls.build List
        (\c7.\n55.cata List
                    (\a56.\n86.cata List
                                c7
                                (cata List c7 n86
                                 newlineList) a56)
                    n55 ls)
```

4 Related Work

The investigations described in this paper build on a long tradition of using transformations on programs to simplify them and improve their efficiency. We briefly summarize some major threads of related work.

4.1 Supercompilation

Wadler's work on deforestation was inspired by earlier work of Turchin, who also developed a program transformation tool based on fold/unfold transformations ([Tur88]). The *super*vising *compiler* symbolically executes programs, tracing generalized histories of the computations they perform, generating residual programs when computations cannot be performed, and constructing new recursive function definitions when patterns in the nesting of recursive function calls are observed. In

supercompilation, the original program is never actually transformed, however. Instead, it is run, observed, and analyzed by the supercompiler, which then compiles an entirely new but functionally equivalent program and reduces in the process certain redundancies present in the original one.

The fundamental step of supercompilation — called *driving* — can be described roughly as an application of the unfold rule of Burstall and Darlington, together with information propagation. As with the transformations of Burstall and Darlington, the possibility of infinitely unfolding function calls arises. Turchin addressed the issue of termination of the supervising compiler by incorporating into it a *generalization* phase. Generalization performs a kind of abstraction of function definitions which enables folding; the transformation it performs is akin to strengthening the induction hypothesis in an inductive proof, and can be viewed as a means of systematizing the *eureka* steps of Burstall and Darlington.

Supercompilation can achieve the effects of both program fusion and partial evaluation. Indeed, deforestation is very similar to a variant of supercompilation known as *positive supercompilation*, except that *i*) deforestation employs constant propagation, rather than unification-based information propagation, and *ii*) deforestation does not include a generalization phase to avoid infinite unfolding. Of course, as discussed in Section 2.2, Wadler's deforestation transformations are terminating for the treeless programs on which they are designed to work even in the absence of generalization. Modulo the issue of termination, driving thus completely subsumes deforestation.

Partial evaluation also performs *program specialization*, which can be seen as a special case of deforestation, and hence of positive supercompilation. The connections between fold/unfold methodologies and partial evaluation are explored in Chapter 17 of [JGS93]. The recent paper [GS96] is a nice exposition of the ideas underlying Turchin's work; [SGJ94] compares and classifies four important instances of the fold/unfold program transformation methodology of Burstall and Darlington, namely program fusion, partial evaluation, supercompilation, and generalized partial computation.

4.2 The Squiggol School and Calculation-based Programming

In the middle 1980's and early 1990's, a new style of programming that would later heavily influence future work in virtual data structure elimination was developed. The *calculational* style of programming, first introduced by Bird and Meertens ([Bir97], [Mee86], [Mee92], stresses calculating programs via algebraic identities over list programs in much the same manner that high school students calculate with the laws of algebra. Inherent in the calculational style are techniques for transforming clear but inefficient programs into ones which use resources efficiently.

The calculational style of programming was later extended by Malcolm to accommodate datatypes other than lists ([Mal89]). Malcolm introduced a generic promotion theorem, which has its origins in a categorical description of programming ([Hag87]), and which describes conditions under which catamorphisms over regular datatypes may be fused. The significance of the theorem derives primarily from the facts that most of the types used in functional programming are regular, and that many functions of interest are expressible as catamorphisms over them.

4.3 An Implementation of the Shortcut

In the dissertation [Gil96], Gill details his implementation of the shortcut to deforestation described in Section 2.4, and assesses its performance on a suite of real application programs ([Par92]). This implementation includes a translation scheme for list comprehensions which guarantees that intermediate lists between them and their producers and consumers are eliminated. Gill's approach to program fusion is a very practical one, and although his implementation effects only the elimination of virtual *lists*, it is significant in that it represents "the first non-trivial deforestation system to be included as an active part of a production quality functional language compiler."

His investigation of the failure of the shortcut to deforest `append` led Gill to a generalization of `build` which requires no nested copy functions for list creation. In fact, this new list producer — called `augment` — generalizes both `append` and `build`, and its associated one-step fusion algorithm

— called the `foldr-augment` rule — extends the original `cata-build` rule for lists to support deforestation of programs whose results contain recursively produced `cons` cells.

Gill's additions to GHC include transformations implementing the `foldr-augment` rule, as well as its special case embodied by the `cata-build` rule. The incorporation of these transformations required considerable analysis to determine how they would interact with existing program simplification transformations, and how the original simplifier could best be modified. It also required the development of supporting tools — such as an enhanced compile-time inlining mechanism and a function arity expansion scheme — to facilitate program fusion via the generalized shortcut.

To support detailed profiling, Gill has produced several different versions of GHC, each implementing some suitable subset of the new transformations, and is thereby able to reliably analyze fused programs for execution speed, heap size, heap residency, and stack usage. The frequency of use of each new transformation, the amount of time the different compilers take to compile the benchmarks, and the size of the object files each produced can also be determined. From his measurements, Gill was able to conclude that the effects of program fusion via the generalized shortcut can sometimes be quite substantial, and that although the effects on a large set of benchmarks evidence a more modest benefit, this benefit is indeed still tangible. He further reports that some program improvement was discernible in about half of the benchmark examples.

4.4 Fusion with Exponential Types

A central tenet of functional programming is that functions are first-class values. This implies, among other things, that functions should be able to be used freely in datatype definitions, and that the resulting datatypes involving exponentials are in some sense fundamental to functional programming.

As its title suggests, the paper *Bananas in Space: Extending Fold and Unfold to Exponential Types* shows how to extend the fold and unfold operators for polynomial datatypes to corresponding operators for types involving exponentials. Such an extension is less than straightforward, however: the type constructor $\rightarrow$ is contravariant in its first argument and covariant in its second, making it impossible to interpret certain type constructors involving $\rightarrow$ as functors in CPO. This leads to the unpleasant conclusion that functionals such as catamorphisms and anamorphisms cannot always be used to define functionals on recursive datatypes involving exponentials.

The method Meijer and Hutton develop to render exponential datatypes as fixed points of functors draws heavily on work of Freyd ([Fre90]). CPO is still taken as the underlying semantic category, although recursive datatypes are no longer modeled as fixed points of functors over that category, but are instead modeled as fixed points of *difunctors* over CPO which are contravariant in their first variable and covariant in their second. The fundamental task in accommodating exponential types thus becomes the generalization of the definitions of catamorphism and anamorphism from datatypes expressed as fixed points of functors to datatypes expressed as fixed points of difunctors (which reduce to the standard definitions for functors if the difunctor is independent of its contravariant parameter). As with datatypes which do not involve exponentials, this task is accomplished by suitably generalizing simple `copy` functions.

The catamorphism and anamorphism functionals for difunctors satisfy a fusion law which arises, perhaps unsurprisingly, as a free theorem. Because the definitions of catamorphism and anamorphism for difunctors are mutually recursive, the fusion law for these two functionals is actually one simultaneous law, rather than two separate laws (one for each functional), as is the case for polynomial datatypes. This single fusion law is a specialization to functions of Pitts' relational induction principle for recursive datatypes.

Gofer, rather than category theory, is used by Meijer and Hutton as a meta-language throughout *Bananas in Space*, and this provides an executable rendering of the ideas explored in the paper. Meijer and Hutton point out as directions for future research 1) expressing as fixed points of functors (or difunctors) nonregular datatypes, i.e., datatypes, like twist, in which the recursive calls in the body are not all of the form of the head of a definition, and 2) the use of the accompanying generalized operators and their fusion theorems in writing and transforming programs.

4.5 Acid Rain

The paper *Shortcut Deforestation in Calculational Form* presents an algorithm for eliminating arbitrary algebraic virtual data structures from programs, based on two fusion rules for hylomorphisms. The point of departure for the development of this algorithm is a generalization of the shortcut of Gill, Launchbury, and Peyton Jones to arbitrary algebraic datatypes — called the Acid Rain Theorem for Catamorphisms — and the development of its categorical dual, the Acid Rain Theorem for Anamorphisms. Although these theorems are together sufficient to describe every situation in which an intermediate data structure of some algebraic type can be eliminated from a program, they are, in fact, too general to serve as the basis of an automatable virtual data structure elimination method. The difficulty comes in actually recognizing expressions to which the computational rules embodied by the theorems apply, as well as in determining in which order those rules should be applied in cases when redices overlap (overlapping redices are actually quite common in practice).

To remedy these problems, Takano and Meijer first note that if the Acid Rain Theorems are instantiated at specific functions whose bodies are described in terms of hylomorphisms, then their automation is indeed possible. (*Hylomorphisms* capture very expressive recursive patterns, and, in fact, generalize both catamorphisms and anamorphisms; at the same time, however, they do impose some restriction on the structure of programs.) Hylomorphisms are therefore introduced by Takano and Meijer as a means of structuring programs to make Acid Rain redices more readily apparent. A generic triplet notation for hylomorphisms — growing out of the observation that reassociation of natural transformations in programs can exchange redices for either Acid Rain Theorem for redices for its dual — is introduced; in this notation, natural transformations are explicitly factored out of hylomorphisms to facilitate such shifting. The Acid Rain Theorems are then restated for hylomorphisms to arrive at the Cata-Hylo Theorem and its dual, the Hylo-Ana Theorem. It is this pair of theorems which forms the basis of Takano and Meijer's virtual data structure elimination algorithm.

The essence of the transformation algorithm is a reduction strategy which controls the order in which Acid Rain redices are reduced. Heuristics for controlling reduction order comprise the main part of the algorithm, and these are of great importance precisely because contracting some redexes can destroy others (that is, the Cata-Hylo and Hylo-Ana reduction rules are not confluent). Some reduction rules for simplifying basic functors are also employed in the algorithm.

Although their algorithm applies to any program containing compositions of hylomorphisms, Takano and Meijer derive maximum benefit from their method by assuming that all programs are written entirely as such compositions. In their computational paradigm, all recursive programs are therefore written in a standardized hylomorphic form. Fortunately, most programs which arise in practice — including all primitive recursive programs — can be represented in this way, despite the fact that doing so is neither natural nor common in current practice.

Takano and Meijer assert that their algorithm supports elimination of virtual data structures of algebraic type "from a much larger class of compositional functional programs" than do earlier algorithms. It successfully eliminates virtual data structures from both input lists to *zip*, for example (the warm fusion method cannot do this unless augmented with fusion theorems specifically for this purpose), and it performs higher-order eliminations as well.

4.6 Deriving Structural Hylomorphisms

Just as effective use of Gill, Launchbury, and Peyton Jones' shortcut requires that recursive programs be expressed in `build-cata` form, so effective use of the virtual data structure elimination algorithm of Takano and Meijer requires that recursive programs be expressed in terms of hylomorphisms. *Warm Fusion* describes one technique for automatically deriving `build-cata` forms from certain classes of recursive programs; *Deriving Structural Hylomorphisms from Recursive Definitions* similarly develops a technique for automatically transforming certain classes of recursive programs into structural hylomorphisms.

Hu, Iwasaki, and Takeichi propose an algorithm which is intended to "automatically turn all practical recursive definitions into structural hylomorphisms," to be guaranteed correct, and always to terminate with a successful hylomorphism. The main observation underlying their algorithm is

that the recursive structure of programs can often be exploited to derive equivalent hylomorphisms suitable for fusion with other programs via the specializations of the Acid Rain Theorems described in the preceding subsection of this paper. Their algorithm for deriving such hylomorphisms comprises a *two-stage abstraction*. In the first stage, recursive function calls are abstracted to derive hylomorphisms from recursive programs. Then, in the second stage, the maximal non-recursive subterms of the non-natural transformation components of the resulting hylomorphisms are abstracted again to derive restructured hylomorphisms to which the Acid Rain Theorems can more effectively be applied.

The required restructuring of hylomorphisms is accomplished by rewriting them so that as much computation as possible is performed outside of their natural transformations. More specifically, it is desirable to shift as much work as possible left (right) in a hylomorphism to be fused on the right (left) with an arbitrary function; this has as a consequence that the natural transformation appearing in the hylomorphism will be as sparse, or simple, as possible. Unfortunately, although categorical specifications of the desired restructured hylomorphisms are given by Hu, Iwasaki, and Takeichi, no calculus for actually achieving that restructuring is given in their paper. The lack of such a calculus presents a severe impediment to the automation of their entire hylomorphism derivation method.

The techniques described in *Deriving Structural Hylomorphisms* are, theoretically speaking, applicable to all programs in which recursive function calls are not nested and occur only in the terms of the alternatives of the function body. But — at least as presented in their paper — the algorithm envisioned by Hu, Iwasaki, and Takeichi is suitable for converting to hylomorphisms only programs which are abstracted case statements. Even for such programs, however, termination of the algorithm is not proved. Neither, unfortunately, is it self-evident.

It is indisputable that more abstracted case statement programs are likely to be expressible in terms of hylomorphisms than in terms of catamorphisms. Thus, when restricted to this class of programs, the program transformation method of Hu, Iwasaki, and Takeichi is indeed more general than the warm fusion method. The warm fusion method applies, however, to a much larger class of programs than just abstracted case statements (even if it has not yet been implemented for more general programs).

In addition, although fusion of abstracted case statement programs may be achieved more often using hylomorphic fusion than warm fusion, programs resulting from the latter may be considerably more efficient than their counterparts obtained from the former. The derivation of a reverse program that runs in linear time from one that is quadratic in complexity, for example, is achievable via warm fusion but not via hylomorphic fusion: in general, hylomorphic fusion does not allow the possibility of turning a recursive program into a hylomorphism at a type other than the type at which the program is originally defined, and in order to transform (quadratic) reverse into `build-cata` form, the catamorphism must be defined at a different type than that of the original function (the catamorphism must actually be defined at a functional type). It is, in fact, the ability to choose the types at which to perform fusion that contributes the potential for achieving significant performance improvements. The fact that the type of the hylomorphism in hylomorphic fusion must always be the same as the original type of the function thus limits the flexibility of that algorithm and impedes its potential computational benefit.

4.7 Series Expressions

About a decade ago, Waters began advocating that his *series* datatype be incorporated into programming languages. Like Burstall and Darlington's fold/unfold transformations, macro packages supporting series are designed specifically to allow programmers to reap the benefits of the compositional style of programming without sacrificing program efficiency. In essence, Waters' series packages support virtual data structure elimination for programs which use linear data structures, such as lists and streams. One was introduced first for Lisp ([Wat87]), and later for Pascal ([Wat91]).

Unfortunately, Waters' series packages were never widely used. Although their widespread adoption may have been prevented by their ability to eliminate only linear data structures, the lack of enthusiasm for them is more likely attributable to the fact that neither the Lisp nor the Pascal pro-

gramming communities are particularly comfortable with the compositional style of programming.

4.8 Stateful Fusion and Other Work

In the paper *Filter Fusion* ([PW96]), Proebsting and Watterson describe a method for fusing programs constructed from pipes and filters. Their fusion algorithm is based on an instantiate/unfold/simplify/fold process reminiscent of that of Burstall and Darlington, but works on a specialized imperative language with *put*s and *get*s rather than on programs written in the form of higher-order recursion equations.

While Proebsting and Watterson's approach to the fusion problem for stateful computation is *ad hoc*, an axiomatic treatment for such computations can be found in [Joh94]. Johnsson uses manual fold/unfold transformations over stateful interpreters to achieve efficient compilation. Similar techniques for arbitrary monadic interpreters have also appeared in the literature; these, however, are as not as explicit in their use of transformations to eliminate virtual data structures from programs as Johnsson's method is.

5 Future Work

The directions for future work on program fusion fall mainly into the four categories listed below.

1. **Implementation** Although this paper details only a prototype implementation of a fusion engine for a fully polymorphic higher-order language, the ultimate goal of the work described here is to design a fusion engine suitable for incorporation into, say, the Glasgow Haskell Compiler. GHC already supports `build-cata` list fusion for its prelude functions. It is desirable to generalize this engine — and the theoretical work which underlies it — to support eliminating arbitrary virtual data structures from programs, and fusing user-defined, as well as prelude, functions. The intention, of course, is that arbitrary function definitions should automatically be transformed into `build-cata` form via extensions of the warm fusion algorithm detailed here.

   This paper details one source-to-source translation effecting warm fusion on a restricted subset of Haskell. Extending this translation to larger classes of programs will require consideration of programs which are not abstracted case statements, in general, and the development of more widely applicable "`cata` distribution" heuristics, in particular. An implementation of higher-order fusion will also be useful in comparing fused variants of programs, and an implementation of fusion with static parameters would make it possible to determine whether or not this particular extension of the warm fusion method is useful in practice.

   Significant new challenges are expected when scaling prototype implementations to "industrial strength" tools. Still, the experiences reported in this paper clearly indicate that the development and implementation of a large-scale fusion engine should be feasible.

2. **Analysis** Since automatic program fusion requires automatic techniques for transforming programs into `build-cata` form, being able to characterize those programs which can be so transformed — particularly via (some extension of) warm fusion — is essential. It is also important to know how often transformable function definitions appear in "real" code, and to measure the fusion behavior of substantial application programs. With this information, it should be possible to determine whether or not the kinds of programs programmers actually write are amenable to virtual data structure elimination on a large scale, and in what sense(s) fused programs are "better" than their unfused counterparts. One might expect, for example, that large programs exhibit less of a gain in efficiency after fusion than do smaller ones because the overhead involved in constructing and then consuming an intermediate data structure represents a less significant fraction of the total resources used for large programs than for small ones. In addition, one interpretation of program fusion is the trading of heap space (to accommodate the intermediate data structure) for closure heap space (to accommodate the fused programs). Whether or not this is always, or even often, a win remains to be discerned.

3. **Extensions** All existing fusion methods — as well as the theoretical work underlying them — restrict attention to programs written in pure functional languages. Recent research ([Lau95], [Joh94]) suggests that fusion techniques can be extended to functional programs written in monadic style, especially to those written using the monad of state. This in turn suggests that `build-cata` fusion can be lifted to imperative programs. It may also be possible to extend fusion to programs with exceptions, intermediate arrays, or controlled forms of dynamic binding. Finally, while fusion for mutually recursive data structures is understood theoretically, an expressive notation for mutually recursive `build-cata` forms needs to be developed, as does a mechanism for "discovering" the mutually recursive structure in ordinary functional programs.

4. **Foundations** Perhaps surprisingly, significant and subtle issues about parametric models of the lambda calculus arise in the consideration of warm fusion. Of primary concern for implementations of warm fusion is whether or not the set $\mathcal{WF}$ of rewrite rules (Definition 3.23) is terminating; the confluence of $\mathcal{WF}$ is also of considerable interest in this context. One approach to investigating the termination and confluence of $\mathcal{WF}$ is to first embed the warm fusion calculus in the polymormphic lambda calculus ([Gir71], [Rey74]), and then try to extend known results about adding first- and higher-order rewriting to various lambda calculi to the special case of higher-order polymorphic rewriting that $\mathcal{WF}$ represents.

   Interestingly, however, embedings for which reduction rules corresponding to those in the subset $\mathcal{R}$ of $\mathcal{WF}$ (Definition 3.22) are derivable via reduction in the polymorphic lambda calculus seem to be possible only for parametric models of the polymorphic lambda calculus. Parametricity thus appears to be absolutely fundamental to the successful embedding of the larger system $\mathcal{WF}$ into any first-order extension of the polymorphic lambda calculus. But non-parametric models of the polymorphic lambda calculus do, of course, exist, since the calculus is not itself strong enough to express this semantic restriction. Unfortunately, no higher-order rewrite system extending the polymorphic lambda calculus and completely characterizing parametricity is currently known to exist. Thus, although the underlying semantics of the warm fusion calculus can be understood by restricting attention to parametric models, at present its operational aspects — including termination and confluence — cannot.

   A good first step toward achieving an understanding of the operational aspects of $\mathcal{WF}$ would be the development of an extension of the polymorphic lambda calculus which is capable of characterizing completely the parametric models of that calculus. Such a calculus would also be capable of expressing the initiality of F-algebras of the form $\langle \forall F, in_{\forall F} \rangle$, and thus of guaranteeing the uniqueness of each homomorphism $\texttt{build}^F$, as discussed in Section 3.2.4.

   Finally, the correctness of the warm fusion calculus with respect to our intended semantics also remains to be demonstrated formally.

Accomplishment of any of the tasks described above will not only move the functional programming community closer to placing existing work on fusion on a secure logical foundation, but would also facilitate the "scaling up" of fusion, both in theory and in practice, so that the effect of virtual data structure elimination on "real" programs can, ultimately, be determined.

Acknowledgement The authors would like to thank Caleb Shor for assistence in testing and debugging the prototype engine.

References

[Amt92] T. Amtoft. Unfold/Fold Transformations Preserving Termination Properties. In *Proceedings, Programming Language Implementation and Logic Programming*, LNCS 631, Springer-Verlag, pp. 187 – 201, 1992.

[Bar92] H.P. Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science, Volume 2*, Oxford University Press, pp. 117 – 309, 1992.

[BD77] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *J. ACM* 24:1 (1977), pp. 44 – 67.

[CGW89] T. Coquand, C. Gunter, and G. Winskel. Domain Theoretic Models of Polymorphism. *Information and Computation* 81 (1989), pp. 123 – 167.

[Fok92] M. Fokkinga. *Law and Order in Algorithmics.* Dissertation, Universiteit Twente, 1992.

[Fre90] P. Freyd. Recursive Types Reduced to Inductive Types. In *Proceedings, IEEE Symposium on Logic in Computer Science*, pp. 498 – 507, 1990.

[FW88] A. Ferguson and P. Wadler. When Will Deforestation Stop? In *Proceedings, Glasgow Workshop on Functional Programming*, pp. 39 – 56, 1988.

[Gil96] A. Gill. *Cheap Deforestation for Non-strict Functional Languages.* Dissertation, Department of Computing Science, Glasgow University, 1996.

[Gir71] J.-Y. Girard. Une Extension de l'Interprétation de Gödel à l'Analyse, et son Application à l'Élimination des coupures dans l'Analyse et la Théorie des Types. In *Proceedings, Second Scandinavian Logic Symposium*, pp. 63 – 92, 1971.

[Gir89] J.-Y. Girard. *Proofs and Types.* Cambridge University Press, 1989.

[GLPJ93] A. Gill, J. Launchbury, and S. Peyton Jones. A Shortcut to Deforestation. In *Proceedings, Conference on Functional Programming and Computer Architecture*, pp. 223 – 232, 1993.

[GS96] R. Glück and M. H. Sorensen. A Roadmap to Metacomputation by Supercompilation. In *Partial Evaluation*, LNCS 1110, Springer-Verlag, pp. 137 – 160, 1996.

[Hag87] T. Hagino. *A Categorical Programming Language.* Dissertation, University of Edinburgh, 1987.

[Hug89] J. Hughes. Why Functional Programming Matters. *The Computer Journal* 32:2 (1989), pp. 98 – 107.

[Jeu93] J. Jeuring. *Theories for Algorithm Calculation.* Dissertation, Universiteit Utrecht, 1993.

[JGS93] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[Joh94] T. Johnsson. Fold-unfold Transformations on State Monadic Transformers. In *Proceedings, Glasgow Functional Programming Workshop*, Springer-Verlag, pp. 127 – 140, 1994.

[Lau95] J. Launchbury. Graph Algorithms with a Functional Flavor. In *Proceedings, First International Spring School on Advanced Functional Programming*, Springer-Verlag, pp. 308 – 331, 1995.

[LS95] J. Launchbury and T. Sheard. Warm Fusion: Deriving Build-Catas from Recursive Definitions. In *Proceedings, Conference on Functional Languages and Computer Architecture*, pp. 314 – 323, 1995.

[Mal89] G. Malcolm. Homomorphisms and Promotability. In *Mathematics of Program Construction*, LNCS 375, Springer-Verlag, pp. 335 – 347, 1989.

[Mar95] S. Marlow. *Deforestation for Higher-order Functional Programs.* Dissertation, University of Glasgow, 1995.

[Mee86] L. Meertens. Algorithmics – Towards Programming as a Mathematical Activity. In *Proceedings of the CWI Symposium on Mathematics and Computer Science*, pp. 289 – 334, 1986.

[Mee92] L. Meertens. Paramorphisms. *Formal Aspects of Computing* 4:5 (1992), pp. 413 – 424.

[Mei92] E. Meijer. *Calculating Compilers.* Dissertation, Universiteit Nijmegen, 1992.

[MH95] E. Meijer and G. Hutton. Bananas in Space: Extending Fold and Unfold to Exponential Types. in *Proceedings, Conference on Functional Programming and Computer Architecture*, pp. 324 – 333, 1995.

[Par92] W. Partain. The *nofib* Benchmarking Suite. In *Proceedings, Glasgow Workshop on Functional Programming*, Springer-Verlag, pp. 195 – 202, 1992.

[PJL91] S. Peyton Jones and J. Launchbury. Unboxed Values as First-class Citizens in a Non-strict Functional Language. In *Proceedings, Functional Programming and Computer Architecture*, LNCS 523, Springer-Verlag, pp. 636 – 666, 1991.

[PW96] T. Proebsting and S. Watterson. Filter Fusion. In *Proceedings, ACM Symposium on Principles of Programming Languages*, pp. 119 – 130, 1996.

[Rey74] J. Reynolds. Towards Theory of Type Structure. In *Proceedings, Paris Colloquium on Programming*, LNCS 19, Springer-Verlag, pp. 408 – 425, 1974.

[Rey83] J. Reynolds. Types, Abstraction, and Parametric Polymorphism. In *Proceedings, Information Processing*, pp. 513 – 523, 1983.

[San94] D. Sands. Total Correctness and Improvement in the Transformation of Functional Programs. Unpublished manuscript, DIKU, University of Copenhagen, 1994.

[Sch86] D.A. Schmidt. *Denotational Semantics*, Wm. C. Brown, 1986.

[SF93] T. Sheard and L. Fegaras. A Fold for All Seasons. In *Proceedings, Conference on Functional Programming and Computer Architecture*, pp. 233 – 242, 1993.

[SGJ94] M. H. Sorensen, R. Gluck, and N. D. Jones. Toward Unifying Partial Evaluation, Deforestation, Supercompilation, and GPC. In *Proceedings, European Symposium on Programming*, LNCS 788, Springer-Verlag, pp. 485 – 500, 1994.

[TM95] A. Takano and E. Meijer. Shortcut Deforestation in Calculational Form. In *Proceedings, Conference on Functional Programming and Computer Architecture*, p. , 1995.

[Tur86] V. F. Turchin. The Concept of a Supercompiler. *ACM Transactions on Programming Languages and Systems* 8:3 (1986), pp. 90 – 121.

[Wad83] P. Wadler. *Listlessness is Better than Laziness.* Dissertation, Department of Computer Science, Carnegie Mellon University, 1983.

[Wad86] P. Wadler. Listlessness is Better than Laziness II: Composing Listless Functions. In *Proceedings, Workshop on Programs and Data Objects*, LNCS 217, Springer-Verlag, pp. 282 – 305, 1985.

[Wad89] P. Wadler. Theorems for Free! In *Proceedings, Conference on Functional Programming and Computer Architecture*, pp. 347 – 359, 1989.

[Wad90] P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science* 73 (1990), pp. 231 – 248.

[Wat87] R.C. Waters. *Obviously Synchronizable Series Expressions: Part I; User's Manual for the OSS Macro Package.* Technical report, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1987.

[Wat91] R. Waters. Automatic Transformation of Series Expressions into Loops. *Transactions on Programming Languages and Systems* 13:1 (1991), pp. 52 – 98.