Program Transformation in Calculational Form

AKIHIKO TAKANO
Advanced Research Laboratory, Hitachi, Ltd., Saitama, Japan
<takano@harl.hitachi.co.jp>
ZHENJIANG HU and MASATO TAKEICHI
Department of Information Engineering, University of Tokyo, Tokyo, Japan
<hu@ipl.t.u-tokyo.ac.jp>; <takeichi@u-tokyo.ac.jp>

Introduction

Correctness-preserving program transformation has recently received a particular attention for compiler optimization in functional programming [Kelsey and Hudak 1989; Appel 1992; Peyton Jones 1996]. By implementing a compiler using many passes, each of which is a transformation for a particular optimization, one can attain a modular compiler. It is no surprise that the modularity would increase if transformations are structured, i.e. constructed in a modular way. Indeed, the *program transformation in calculational form* (or *program calculation*) can help us to attain this goal.

Program calculation is a kind of program transformation based on the theory of *Constructive Algorithmics* [Bird 1987; Malcolm 1990; Meijer et al. 1991; Fokkinga 1992], which is a calculus of program derivation based on the theory of *algebra of programming* [Bird and De Moor 1997]. In Constructive Algorithmics, calculation is a series of applications of calculational laws (i.e. rules) that describe some properties of programs. Theorems may be used to capture larger steps in calculation in which there is ample opportunity for machine assistance. Basically, these laws and theorems follow from the properties of homomorphism over algebraic data types. Categorical functors are used to describe the algebraic structure of data types, and the generic structure of homomorphism (called *catamorphism*, or generalized fold) is defined for each algebraic data type.

It has been shown that many important program transformations such as *deforestation* (or *fusion*), *tupling transformation* and *parallelization* can effectively and elegantly be formalized in calculational form [Takano and Meijer 1995; Hu et al. 1996; Hu et al. 1997; Hu et al. 1998]. In general, program transformation in calculational form enjoys the nice properties such as *modularity*, *cheap implementation*

and *compatibility*.

Structuring Recursions

Recall the structured programming methodology for imperative language, where the use of arbitrary *goto*'s is abandoned in favor of structured control flow primitives such as conditionals and while-loops so that program construction and reasoning becomes easier and elegant. For functional programs, recursive definitions provide a powerful control mechanism for specifying programs. Consider, for example, the following function definition on lists:

$$\begin{array}{ll} f\ [\,] & = \cdots \\ f\ (x : xs) & = \cdots . \end{array}$$

Here, we usually do not care about the form of the right hand side; it can be any expression where recursive calls to f may appear in any form. This, in fact, resembles the arbitrary use of goto's in imperative programs, and makes recursive definitions difficult to be manipulated. In contrast, the calculational approach imposes restrictions on the right hand side to make the recursion structure explicit, resulting in suitable forms such as *catamorphisms*, *anamorphisms* and *hylomorphisms* [Meijer et al. 1991] to which a number of general but modular transformation theorems can be applied.

However, it is unrealistic to force programmers to write their programs with catamorphisms, anamorphisms, or hylomorphisms which are quite abstract and require some acquaintance with category theory. Rather, we would like to propose a way to structure recursions into such forms. This is possible as seen in [Hu et al. 1996; Onoue et al. 1997] where most recursive definitions (on algebraic data types) of interest can be automatically transformed into *hylomorphisms*, a most general form. Informally, hylomorphism is a function defined in the following form of recursion:

$$f = \phi \circ (\eta \circ F\, f) \circ \psi.$$

The right hand side of this definition can be read as follows: generate some F-structure from the input by ψ; apply f to all recursive components in the F-structure by $F\, f$; manipulate the F-structure into some G-structure by η; and finally fold the G-structure by ϕ to give the result. In case of hylomorphic function f on lists, the right hand side of the definition of $f(x : xs)$ may have the term $f\ xs$, but never contains the terms like $f(f\ xs)$.

This structuring process is so important that it paves the way for us to make use of the theory of Constructive Algorithmics in formalizing useful program transformations in calculational form.

Procedure

The procedure to formalize a program transformation in calculational form usually consists of the following three major steps.

(1) *Capturing program structure in a specialized form*
Although hylomorphisms are the most general form enjoying general calculational properties, it is too general for a certain optimizing transformation. In

most cases, we first need to transform them into more restrictive form, in order to handle the specific optimization effectively. For example, in fusion calculation [Hu et al. 1996], we introduce *structural hylomorphisms* to describe the production and consumption of data in a more explicit way. Similarly, in tupling calculation [Hu et al. 1997], we need to define a class of *tuplable functions*.

(2) *Establishing basic calculational laws*
Second, we should establish specialized calculational laws for manipulating our specialized form accordingly. For example, *Acid Rain Theorem* [Takano and Meijer 1995] shows how to manipulate structural hylomorphisms, and *Tupling Theorem* [Hu et al. 1997] guides us in manipulating tuplable functions.

(3) *Designing a calculational algorithm*
Finally, we must propose a calculational algorithm, which clarifies how to convert a given program into our specific form and how to apply the newly established calculational laws in a systematic way, as seen in [Onoue et al. 1997; Hu et al. 1997; Hu et al. 1998].

Characteristics

The main characteristics of the program transformation in calculational form can be summarized as follows.

—*Modularity*
Program transformation in calculational form does not require any global analysis which other transformation methods often need. Instead, it only uses a local program analysis to obtain the specialized form, and the applicability of their calculational rules can be checked locally. Therefore, it can be implemented in a modular way, and it is guaranteed to terminate.

—*Cheap Implementation*
Transformations in calculational form are more practical than the well-known *fold/unfold* transformation [Burstall and Darlington 1977]. Fold/unfold transformation basically has to keep track of all occurrences of function definitions and introduce new function definitions, which are searched in the folding steps. This housekeeping process and the clever controll to avoid infinite unfolding at the specialization points introduce a substantial cost and complexities, which prevent the practical implementation. Though being less general than fold/unfold transformation, transformations in calculational form can be implemented in a cheap way [Gill et al. 1993; Sheard and Fegaras 1993; Takano and Meijer 1995; Launchbury and Sheard 1995; Hu et al. 1997] by means of local program analysis and application of rules. No heuristics is necessary in calculational transformations.

—*Compatibility*
It is usually difficult to make several different transformations coexist well in a single system, but transformations in calculational form can solve this problem well. For instance, fusion calculation can coexist well with tupling calculation [Hu et al. 1997]. There are two reasons. First, each transformation is based on the same theoretical framework, Constructive Algorithmics. Second, the locality of the program analysis and the application of laws make it easier to check compatibility of transformations.

Conclusion

In this short article we explained the basic principle of program transformation in calculational form, summarized the procedure to formalize it, and clarified its characteristics. We believe that the calculational approach is a promising research direction to study program transformation in a more essential and systematic way.

REFERENCES

Appel, A. 1992. *Compilation with Continuations*. Cambridge University Press.

Bird, R. 1987. An introduction to the theory of lists. In M. Broy Ed., *Logic of Programming and Calculi of Discrete Design* (1987), pp. 5–42. Springer-Verlag.

Bird, R. and De Moor, O. 1997. *Algebra of Programming*. Prentice Hall.

Burstall, R. and Darlington, J. 1977. A transformation system for developing recursive programs. *JACM 24*, 1 (Jan.), 44–67.

Fokkinga, M. 1992. *Law and Order in Algorithmics*. Ph. D. thesis, University of Twente, The Netherlands.

Gill, A., Launchbury, J., and Peyton Jones, S. 1993. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark, June 1993), pp. 223–232. ACM Press.

Hu, Z., Iwasaki, H., and Takeichi, M. 1996. Deriving structural hylomorphisms from recursive definitions. In *Proceedings of the 1st ACM International Conference on Functional Programming* (Philadelphia, PA, May 1996), pp. 73–82. ACM Press.

Hu, Z., Iwasaki, H., Takeichi, M., and Takano, A. 1997. Tupling calculation eliminates multiple data traversals. In *Proceedings of the 2nd ACM International Conference on Functional Programming* (Amsterdam, The Netherlands, June 1997), pp. 164–175. ACM Press.

Hu, Z., Takeichi, M., and Chin, W. 1998. Parallelization in calculational forms. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages* (San Diego, CA, Jan. 1998). ACM Press.

Kelsey, R. and Hudak, P. 1989. Compilation by program transformation. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages* (Austin, Texas, Jan 1989), pp. 281–292. ACM Press.

Launchbury, J. and Sheard, T. 1995. Warm fusion: Deriving build-cata's from recursive definitions. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (La Jolla, CA, June 1995), pp. 314–323. ACM Press.

Malcolm, G. 1990. Data structures and program transformation. *Science of Computer Programming 14*, 2–3 (Aug.), 255–279.

Meijer, E., Fokkinga, M., and Paterson, R. 1991. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (LNCS 523) (Cambridge, MA, Aug. 1991), pp. 124–144. Springer Verlag.

Onoue, Y., Hu, Z., Iwasaki, H., and Takeichi, M. 1997. A calculational fusion system HYLO. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi* (Le Bischenberg, France, Feb. 1997), pp. 76–106. Chapman&Hall.

Peyton Jones, S. 1996. Compiling Haskell by program transformation: A report from the trenches. In *Proceedings of the 6th European Symposium on Programming* (LNCS 1058) (Linköping, Sweden, April 1996), pp. 18–44. Springer Verlag.

Sheard, T. and Fegaras, L. 1993. A fold for all seasons. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark, June 1993), pp. 233–242. ACM Press.

Takano, A. and Meijer, E. 1995. Shortcut deforestation in calculational form. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture* (La Jolla, CA, June 1995), pp. 306–313. ACM Press.