J. Functional Programming **4** (1): 89–112, January 1994 © 1994 Cambridge University Press

How powerful are folding/unfolding transformations?

HONG ZHU[1]

Institute of Computer Software, Nanjing University, Nanjing, PR China, and Department of Computing, The Open University, UK

Abstract

This paper discusses the transformation power of Burstall and Darlington's folding/unfolding system, i.e. what kind of programs can be derived from a given one. A necessary condition of derivability is proved. The notion of inherent complexity of recursive functions in introduced. A bound on efficiency gain by folding/unfolding transformations is obtained for all reasonable computation models. The well-known partial correctness and incompleteness of the system are corollaries of the result. Examples of underivability are given, e.g. binary searching cannot be derived from linear searching, merge sorting cannot be derived from insert sorting.

Capsule review

This paper is devoted to the proof of a theorem which says, in effect, that a transformation in Burstall and Darlington's unfold-fold system can reduce the depth of recursion in a recursive program by at most a constant multiple. Using this result, the author derives several corollaries. One is that there are two programs, P_1 and P_2, computing the same function but P_2 cannot be derived from P_1 in the system. Another corollary says that if P_2 can be derived from P_1, then the inherent time complexity of P_2 cannot be much better than that of P_1.

1 Introduction

Recent years have seen much attention given to the rapid growth of program transformations (Bauer, 1979; Pepper, 1984; Meertens, 1987), which separates the process of software development into two phases. In the first phase, programmers write the first version (or a formal specification) of the program, concentrating on the correctness of the program. The second phase is concerned with the efficiency of the program. This is achieved by successive manipulations and transformations of the program, whilst preserving correctness. One of the fundamental problems concerning the approach is whether the transformation mechanism is powerful enough to derive efficient programs from formal specifications. It is this subject to which the paper is devoted.

[1] Current address: Department of Computing, Faculty of Mathematics, The Open University, Walton Hall, Milton Keynes MK7 6AA, UK, EMail:h.zhu@uk.ac.open.acsvax

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

The transformation mechanism studied in the paper is the folding/unfolding system of Burstall and Darlington (1977). It is chosen for its simplicity and clarity, as well as generality. Many systems have folding/unfolding as basic rules (Feather, 1982; Darlington, 1981*a*, *b*, 1984*a*; Partsch and Steinbruggen, 1983). Examples of successful transformations have shown that algorithms of exponential time and space complexity can be transformed into algorithms of linear complexity (Burstall and Darlington, 1977; Darlington, 1978, 1984*b*; Pettorossi and Burstall, 1982). Case studies have demonstrated that the system is suitable for various programming purposes (see Darlington, 1985, for a survey). Several strategies and tactics have been proposed for effective derivation of efficient programs (Pettorossi, 1984, 1989; Nielson and Nielson, 1990), and it is widely believed that the system has no limit to improving the efficiency of programs, although its incompleteness and partial correctness are also well-known (Knott, 1985; Yongqian, 1987). But, as will be proved in the paper, there is an upper bound on the efficiency gain that can be obtained by folding and unfolding. Such examples include linear search that cannot be transformed into binary search, and insert sorting that cannot be transformed into merge sorting.

The paper is organized as follows: section 2 briefly describes the notation used in the paper and gives a formal definition of the folding/unfolding system of Burstall and Darlington. In section 3, first we prove a necessary condition for deriving one program from another, without the use of a 'eureka' step. We then investigate the role of eureka in the transformations. Section 4 discusses how much efficiency can be gained by transformations. Here we introduce the notion of the inherent complexity of recursive programs, and prove that the order of inherent complexity of a recursive program cannot be improved by folding/unfolding transformations. Hence, we obtain a bound on the efficiency of programs derivable by transformations. Section 5 concludes the paper, and briefly addresses some related work.

2 Preliminaries

In this section, we define the notation used in the paper, and the folding/unfolding transformation system.

2.1 Notation

We assume that the reader has some knowledge of the theory of complete partially ordered sets (CPO sets) (Gunter and Scott, 1990); we will use the same system of notation as them. In particular, the least element of a CPO set $(D, \leqslant)$ is written as $\perp$, called bottom or undefined. Every ascending chain $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_n \leqslant \ldots$ in a CPO set D has a least upper bound, written as $\sqcup_{i=1}^{\infty} a_i$. We will frequently refer to the following theorem about fixed points of continuous functions on CPO sets:

Kleene's Theorem (*Kleene, 1951; Manna, 1974*)
Let f be a continuous function on a CPO set $(D, \leqslant)$. Then, there is a least fixed point of f, and the least fixed point of f is $\sqcup_{n=1}^{\infty} f^n(\perp)$, where f^n is the composition of f with itself n times. □

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

Programs transformed by the folding/unfolding system are in the form of a set of recursive equations:

$$\begin{cases} L_1 \Leftarrow R_1 \\ L_2 \Leftarrow R_2 \\ \quad \cdots \\ L_k \Leftarrow R_k \end{cases}$$

where the expressions on the left hand sides of equations are in the form of

$$f p_1 p_2 \ldots p_n, \quad n > 0.$$

f is called a *recursive function symbol*, this function being defined by the equations. p_i $(i = 1, \ldots, n)$ are patterns, i.e. expressions constructed from variables and constants by data constructors. The right hand sides are normal expressions which may contain 'where' clauses. Since in the transformation process, instances of an equation may be added into the system, we do not require that the patterns in different equations, defining the same function, are disjoint. However, we do require that the equations are consistent, in the sense that if there is more than one equation applicable to an input, then there is a common upper bound for the values obtained from the applications of the equations to the input. Moreover, the least upper bound of the values considered to be the value of the function applied to the input.

Given a set K of constants, V of variables, F of recursive function symbols, and C of constructors, the expressions are recursively defined as follows:

(i) a constant $c \in K$ is an expression;
(ii) a variable is an expression;
(iii) a function symbol is an expression;
(iv) if e_1, e_2 are expressions, then so is the application of e_1 to e_2, written as $e_1 e_2$;
(v) if $e_1, e_2, \ldots, e_n$ are expressions, and C is a n-ary constructor, then $C(e_1, e_2, \ldots, e_n)$ is an expression;
(vi) if e and d are expressions, then $(e \textit{ where } \langle u, v, \ldots, w \rangle = d)$ is an expression; where $u, v, \ldots, w$ are local variables defined by d.

Without loss of generality, we assume that *where* clauses do not contain recursive definitions. So $u, v, \ldots, w$ must not occur in d. The primitive functions (i.e. higher order constants and constructors) are assumed to be continuous.

An expression may be viewed as a finite ordered tree, the leaves of which are labelled with variables or constants, and the internal nodes of which are labelled with 'application', 'where clause', or a constructor name. An occurrence within an expression may be represented as a sequence of positive integers, describing the path from the outermost 'root' symbol to the head of the subexpression at that position. Let e be an expression. The following notation will be used in the sequel:

$e|\rho$—the subexpression of e at occurrence ρ.
$[x_1 \to e_1, \ldots, x_n \to e_n]$—the substitution which maps the variable x_i to the expression e_i, $i = 1, \ldots, n$.
$\sigma[x \to e]$—the substitution σ' which agrees with σ except that $\sigma'(x) = e$.

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

$e\sigma$—the expression obtained by the application of the substitution σ to the expression e, i.e. the expression e with free occurrences of x_i replaced by the expression $\sigma(x_i)$, $i = 1, 2, \ldots, n$.
$e[d\backslash\rho]$—the expression e with the subexpression at occurrence ρ replaced by d.
$e(f_1, f_2, \ldots, f_n)$—$\{f_1, f_2, \ldots, f_n\}$ is the set of function symbols occurring in e.
$E(f_1, f_2, \ldots, f_n)$—the system E of equations contains recursive function symbols f_1, $f_2, \ldots, f_n$.

The semantics of the recursive equations are defined to be the least fixed point of the functional defined by the equations. Let E be a set of equations defining recursive functions $f, g, \ldots, h$. When E is considered as a functional, the result of the application of E to functions $a, b, \ldots, c$, written as $E(a, b, \ldots, c)$, is the function defined by the set of equations:

$$\{L \Leftarrow (R[f \to a, g \to b, \ldots, h \to c]) \mid L \Leftarrow R \text{ in } E\}$$

Without loss of generality, we assume that functionals are continuous (Schmidt, 1988). Then, the semantics of a set of recursive equations is the least fixed point of the functional, i.e. the function $f^*, g^*, \ldots, h^*$ so that

$$(f^*, g^*, \ldots, h^*) = E(f^*, g^*, \ldots, h^*)$$

and for all functions $f', g', \ldots, h'$,

$$(f', g', \ldots, h') = E(f', g', \ldots, h') \Rightarrow f^* \leqslant f', g^* \leqslant g', \ldots, h^* \leqslant h'$$

By Kleene's Theorem, we have

$$(f^*, g^*, \ldots, h^*) = \bigsqcup_{n=1}^{\infty} E^n(\bot, \bot, \ldots, \bot)$$

where $\bot$ is the function which always gives the value undefined.

Since the programming language used here is functional, we will not distinguish the term 'program' from 'function' in the paper. And if no confusion arises, we do not distinguish a system of recursive equations from the functional defined by the equations. For the sake of convenience, subsequently, f will be used to denote $(f_1, f_2, \ldots, f_n)$, $E(f)$ to denote $E(f_1, \ldots, f_n)$, and so on.

2.2 *The folding/unfolding system*

Let E be a set of equations. The folding/unfolding system consists of the following six rules.

(1) Instantiation
Let $L \Leftarrow R$ be an equation in the system E of equations. By applying a substitution to both sides of the equation, we obtain an instance of the equation. The application of the instantiation rule adds an instance of an equation to the system. Formally, let $\sigma = (x_1 \to e_1, \ldots, x_n \to e_n)$ be a substitution, we define the instantiation rule as a binary relation $-\mathrm{I}\to$ on systems of equations as follows

$$E \overset{L \Leftarrow R}{\underset{\sigma}{-\mathrm{I}\to}} E; \quad (L\sigma \Leftarrow R\sigma)$$

where $E; x = E \cup \{x\}$.

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

(2) Unfolding

Let $L \Leftarrow R$ and $U \Leftarrow V$ be equations in E. Suppose that a subexpression of V is an instance of L. By replacing the subexpression (i.e. the instance of L) with the corresponding instance of R, we obtain an expression W. The application of the unfolding rule adds the equation $U \Leftarrow W$ into the system of equations. Formally, we define the rule as a relation $-\mathrm{U}\rightarrow$ that

$$E \underset{U \Leftarrow V, \rho}{\overset{L \Leftarrow R}{-\mathrm{U}\rightarrow}} E; \quad (U \Leftarrow V[R\sigma \backslash \rho]), \quad \text{if } V|\rho = L\sigma.$$

(3) Folding

Let $L \Leftarrow R$ and $U \Leftarrow V$ be equations in the system E. By applying a folding rule, we add the equation $U \Leftarrow W$ into E, where W is the expression V, with a subexpression which is an instance of R replaced with the corresponding instance of L. Formally, we define that

$$E \underset{U \Leftarrow V, \rho}{\overset{L \Leftarrow R}{-\mathrm{F}\rightarrow}} E; \quad (U \Leftarrow V[L\sigma \backslash \rho]), \quad \text{if } V|\rho = R\sigma.$$

(4) Abstraction

The abstraction rule can be applied to the right hand side R of an equation $L \Leftarrow R$ in E. Some new local variables are introduced by adding a where clause. Then, the equation is added into the system of equations. Formally, we have

$$E \underset{(\rho_1, \ldots, \rho_n)}{\overset{L \Leftarrow R}{-\mathrm{A}\rightarrow}} E; \quad (L \Leftarrow R')$$

where

$R' = (R[u_1 \backslash \rho_1, \ldots, u_k \backslash \rho_k, v_1 \backslash \rho_{k+1}, \ldots, v_{n-k} \backslash \rho_n]$ *where* $(u_1, \ldots, u_k) = (R|\rho_1, \ldots, R|\rho_k))$,

$n \geqslant k$,

$$u_i \neq u_j, \text{ if } i \neq j,\ i,j = 1,2,\ldots,k,$$

$$v_i \in \{u_1, \ldots, u_k\}, \text{ for all } i = 1,2,\ldots,n-k,$$

and

$$R|\rho_{i+k} = R|\rho_j, \text{ if } v_i = u_j,\ i = 1,2,\ldots,n-k,\ j = 1,2,\ldots,k.$$

(5) Application of law

A law is an assertion of the equivalence of pairs of expressions. Laws are written in the form $P = Q$, where P, Q are expressions. For example, $x+y = y+x$ is a law about the primitive function $+$. Laws must be distinguished from the equations which define functions. By a law we mean that the equality holds for all of its instances. In other words, '$P = Q$ is a law' means $P\sigma = Q\sigma$ for all substitutions σ, but this is not necessarily true for equations.

By applying a law $P = Q$ to the right hand side R of the equation $L \Leftarrow R$, we replace a subexpression of R which is an instance of P with the corresponding instance of Q.

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

The resulting equation is then added into the system. Formally, we define the rule as follows.

$$E \xrightarrow[P=Q,\rho]{L \Leftarrow R} E; \quad (L \Leftarrow R[Q\sigma \backslash \rho]), \quad \text{if } R|\rho = P\sigma.$$

For the sake of convenience, we will omit some or all of the subscripts and superscripts of the rules provided no confusion arises.

(6) Selection of a subset

The transformation processes are the successive applications of the above rules to the system of equations. A subset of the equations are selected as the result. Therefore, the selection of a subset of the equations should also be considered as a rule. Formally, we define that

$$E - \mathrm{S} \rightarrow F, \quad \text{if } F \subseteq E.$$

In the original proposal of Burstall and Darlington (1977) there is another rule called definition. It allows the introduction of new equations during the process of transformation. We avoid the definition rule by considering separately those programs that can be derived from $E \cup H$ and those that can be derived from just E, where H are new equations. Obviously, any program which can be derived by adding an equation H into consideration during the process of transformation, can also be derived from the program together with the additional equation H without use of the definition rule. The definition rule is much more practical because various techniques can be applied to generate the right equations during transformation. However, it is obvious that without restrictions on what kind of equations can be added, the transformation system would have no limit to its power and would certainly be incorrect. Unfortunately, such restrictions are not given explicitly in the original proposal or elsewhere. This will be explored in the next section.

Now, define the relation $\rightarrow$ on programs as the union of the above relations, i.e.

$$\rightarrow = -\mathrm{I} \rightarrow \cup -\mathrm{F} \rightarrow \cup -\mathrm{U} \rightarrow \cup -\mathrm{A} \rightarrow \cup -\mathrm{L} \rightarrow \cup -\mathrm{S} \rightarrow.$$

Notice that, the relation $\rightarrow$ is reflexive because the relation $-\mathrm{S} \rightarrow$ is reflexive.

A program E can be transformed into a program F by folding/unfolding, written as $E - {}^{*} \rightarrow F$, if and only if, there is a finite sequence $E_1, E_2, \ldots, E_k, k \geqslant 1$, of systems of equations so that

$$E = E_1 \rightarrow E_2 \rightarrow \ldots \rightarrow E_k = F.$$

Therefore, $-{}^{*} \rightarrow$ is the transitive closure of $\rightarrow$.

3 A necessary condition of transformability

3.1 Fundamental theorem

Since the semantics of a system of recursive equations is the least fixed point of the functional defined by the equations, the transformation rules can be considered as operators on functionals. Therefore, we investigate the properties of the operators by studying their effects on the corresponding functional. This leads to the following lemmas about the properties of the operators. The proofs of the lemmas are given in the appendix because they are straightforward but tedious.

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

Lemma 1 (Selection of a subset)
Let $E(f) - \mathtt{S} \rightarrow F(f)$. *Then,*
(i) *for all functions* $g, F(g) \leqslant E(g)$;
(ii) *for all* $n = 1, 2, \ldots, F^n(\perp) \leqslant E^n(\perp)$. □

Lemma 2 (Instantiation)
Let $E(f) - \mathtt{I} \rightarrow F(f)$. *Then*
(i) *For all functions* $g, E(g) = F(g)$;
(ii) *For all* $n = 1, 2, \ldots, E^n(\perp) = F^n(\perp)$. □

Lemma 3 (Unfolding)
Let $E(f) - \mathtt{U} \rightarrow F(f)$. *Then for all* $n = 1, 2, \ldots, F^n(\perp) \leqslant E^{2n}(\perp)$. □

Lemma 4 (Folding)
Let $E(f) - \mathtt{F} \rightarrow F(f)$. *Then for all* $n = 1, 2, 3, \ldots, F^n(\perp) \leqslant E^n(\perp)$. □

Lemma 5 (Abstraction)
Let $E(f) - \mathtt{A} \rightarrow F(f)$. *Then we have*
(i) *For all functions* $g, F(g) = E(g)$;
(ii) *For all* $n = 1, 2, \ldots, F^n(\perp) = E^n(\perp)$. □

Lemma 6 (Application of law)
Let $E(f) - \mathtt{L} \rightarrow F(f)$. *Then we have*
(i) *For all functions* $g, F(g) = E(g)$;
(ii) *For all* $n = 1, 2, \ldots, F^n(\perp) = E^n(\perp)$. □

From the above lemmas, we can obtain the following necessary condition on the transformability of the folding/unfolding system:

Theorem 1 (Fundamental Theorem)
If $E - * \rightarrow F$, *then there is a constant* $K \geqslant 1$ *so that for every* $n = 1, 2, \ldots, F^n(\perp) \leqslant E^{Kn}(\perp)$.

Proof (sketch)
By induction on the length of transformations, using Lemmas 1–6. □

The theorem can be used to prove underivability between programs. For example,

Example 1. From the program

$$E: \begin{cases} f(0) \Leftarrow 0, \\ f(n+1) \Leftarrow f(n), \end{cases}$$

we cannot derive the program

$$F: \begin{cases} f(0) \Leftarrow 0, \\ f(n+1) \Leftarrow 0. \end{cases}$$

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

Proof
For every $i = 1, 2, \ldots,$ we have

$$F^i(\perp) = \begin{cases} f(0) \Leftarrow 0, \\ f(n+1) \Leftarrow 0. \end{cases} \quad \text{and} \quad E^i(\perp) = \begin{cases} f(0) \Leftarrow 0 \\ f(1) \Leftarrow 0 \\ f(2) \Leftarrow 0 \\ \cdots \\ f(i) \Leftarrow 0 \end{cases}$$

Obviously, for all $s, t = 1, 2, \ldots,$ we have

$$F^t(\perp) \geqslant E^s(\perp), \quad \text{but} \quad F^t(\perp) \neq E^s(\perp).$$

Thus, for all $s, t = 1, 2, \ldots, F^t(\perp) > E^s(\perp)$, i.e. there is no constant $K \geqslant 1$ so that for all $i = 1, 2, \ldots, F^i(\perp) \leqslant E^{Ki}(\perp)$. By theorem 1, F cannot be derived from E. □

Notice that the two functions defined in Example 1 are equivalent one to another under eager evaluation. Whether they are equivalent under lazy evaluation depends on whether or not the constructor '$+1$' is strict. Normally, constructors in lazy functional programming languages, such as Miranda and Haskell, are non-strict. In this case, the two functions are not equivalent, because under the definition of E,

$$f(\perp + 1) = f(\perp) = \perp$$

But under the definition of F,

$$f(\perp + 1) = 0$$

However, if the constructor $+1$ is strict, we can prove that they are equivalent under both eager and lazy evaluation. A simpler example of two functions equivalent under eager evaluation but inequivalent under lazy evaluation can be obtained by replacing the definition of F with the following equation:

$$F': \quad f(n) \Leftarrow 0.$$

The same proof given above can be applied to the underivability of F' from E. The following is a counter-example of transformability of equivalent functions under lazy evaluation:

Example 2. Consider the following two functions:

$$E \begin{cases} f(0) \Leftarrow 0 \\ f(n+1) \Leftarrow f(n) + n + 1 \end{cases}$$

$$F \begin{cases} f(0) \Leftarrow 0 \\ f(1) \Leftarrow 0 \\ f(2n) \Leftarrow 2f(n) + n^*n \\ f(2n+1) \Leftarrow f(2n) + 2n + 1. \end{cases}$$

Similar to Example 1, we can prove that there is no constant K so that for all $i = 1, 2, \ldots, F^i(\perp) \leqslant E^{Ki}(\perp)$. Therefore, F cannot be derived from E by Theorem 1. By

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

induction on n, we can prove that both of the least fixed points of F and E compute the following function:

$$f(n) = \sum_{i=1}^{n} i$$

and no matter whether they are evaluated eagerly or lazily, we have

$$f(\perp) = \perp.$$

That is, they are equivalent under eager and lazy evaluation. □

The following is an example of underivability of higher order functions:

Example 3. Consider the following two higher order functions which are equivalent under eager evaluation:

$$\text{Map}\begin{cases} \text{map}\, f\, \text{nil} \Leftarrow \text{nil} \\ \text{map}\, f(x::x1) \Leftarrow (f x)::\text{map}\, f\, x1 \end{cases}$$

$$\text{Tree}\begin{cases} \text{map}\, f\, \text{nil} \Leftarrow \text{nil} \\ \text{map}\, f(x::\text{nil}) \Leftarrow (f x)::\text{nil} \\ \text{map}\, f(x_1::(x_2::x1)) \Leftarrow (\text{map}\, f u) \parallel (\text{map}\, f v) \text{ where } \langle u, v\rangle = \text{split}\ (x_1::(x_2::x1)) \end{cases}$$

Where $\parallel$ is list concatenation, split is the primitive function on list which splits a list into two lists of equal length:

$$\text{split}\ \langle x_1, x_2, \ldots, x_n\rangle = \langle\langle x_1, x_2, \ldots, x_s\rangle, \langle x_{s+1}, \ldots, x_n\rangle\rangle \text{ where } s = [n/2]$$

Now, let f be id, the identity function, then we have:

$$\text{Map}^n(\perp)\,\text{id}\,\langle x_1, x_2, \ldots, x_k\rangle = \begin{cases} \langle x_1, x_2, \ldots, x_k\rangle, & \text{if } k \leqslant n; \\ \perp, & \text{if } k > n. \end{cases}$$

$$\text{Tree}^n(\perp)\,\text{id}\,\langle x_1, x_2, \ldots, x_k\rangle = \begin{cases} \langle x_1, x_2, \ldots, x_k\rangle, & \text{if } k \leqslant 2^n; \\ \perp, & \text{if } k > 2^n. \end{cases}$$

Therefore, there is no constant $K \geqslant 1$ so that for all $n = 1, 2, \ldots,$

$$\text{Tree}^n(\perp)\,\text{id} \leqslant \text{Map}^{Kn}(\perp)\,\text{id}$$

That is, there is no constant $K \geqslant 1$ such that for all $n = 1, 2, \ldots,$

$$\text{Tree}^n(\perp) \leqslant \text{Map}^{Kn}(\perp)$$

By Theorem 1, Tree cannot be derived from Map by folding/unfolding. □

3.2 'Eureka'

The use of 'eureka' in folding/unfolding transformations is of particular importance. Little transformation can be done without the help of eureka. The following is an example:

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

Example 4. (Burstall and Darlington, 1977)
The program

$$E:\begin{cases} f0 \Leftarrow 1 \\ f1 \Leftarrow 1 \\ f(n+2) \Leftarrow f(n+1)+fn \end{cases}$$

with the help of the eureka

$$H: gn \Leftarrow \langle f(n+1), fn \rangle$$

can be transformed into

$$F:\begin{cases} f1 \Leftarrow 1 \\ f(n+2) \Leftarrow u+v \quad \text{WHERE} \quad (u,v) = gn \\ g0 \Leftarrow \langle 1,1 \rangle \\ g(n+1) \Leftarrow \langle v, u+v \rangle \quad \text{WHERE} \quad \langle u, v \rangle = gn. \end{cases} \quad \square$$

Now, let us consider when eureka is helpful. Given two systems of equations E and F, if the condition of Theorem 1 does not hold for them, we cannot derive F from E. The question is whether we can derive it by adding some eureka. More formally, is there any set H of equations, such that F is derivable from E and H? However, this question is meaningful only if there is some restriction on the equations which can be added. Otherwise, every program F can be derived from a given one by just adding the equations of F into the system of equations, and then applying the subset selection rule. This causes a serious problem of incorrectness and is obviously undesirable.

Intuitively, the use of eureka means introduction of auxiliary functions. Therefore, the additional equations should meet the following two conditions. Firstly, they should not include additional equations for the definition of the existing functions, otherwise the equations should be considered as a redefinition of the program instead of a 'eureka'. Secondly, they should not include additional equations for the definition of the primitive functions occurring in the program, otherwise they would change the meaning of the program. These conditions lead to the following syntactical restriction of eureka: a system H of equations is a eureka of E, if the recursive function symbols defined by H do not occur in E. Formally,

Definition 1
Let E be a system of equations, $f = \{f_1, \ldots, f_n\}$ be the function symbols in E. A set of equations H which contains function symbols $h = \{h_1, \ldots, h_m\}$ is said to be a eureka of E, if

(i) *f does not occur on any left hand side of H,*

(ii) *$h - f$ does not occur in E, where $h - f$ is set difference.*

Subsequently, without loss of generality, we will write $H(f,g)$ for $f \cup g$ containing all the function symbols in H, and $f \cap g = \varnothing$.

Since eureka are auxiliary functions, when we discuss whether a system E of equations can be transformed into another system F of equations with the help of eureka, we must deal with the possibility that they use different sets of recursive function symbols. Among the recursive function symbols defined by a system of

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

equations, we will always take the first one as the main function subsequently. Thus, we need the following corollary of Theorem 1:

Corollary 1
If $E -\!\!*\!\!\rightarrow F$, *then there is a constant* $K \geqslant 1$ *such that for all* $i = 1, 2, \ldots,$

$$F^i(\perp)\downarrow 1 \leqslant E^{Ki}(\perp)\downarrow 1, \quad \text{where } (x_1, x_2, \ldots, x_n)\downarrow s = x_s, \quad 1 \leqslant s \leqslant n$$

Proof
By Theorem 1, we have that for all $i = 1, 2, \ldots,$

$$F^i(\perp) \leqslant E^{Ki}(\perp)$$

Let $F^i(\perp) = (f_1, f_2, \ldots, f_n)$ and $E^{Ki}(\perp) = (g_1, g_2, \ldots, g_n)$. By the definition of the partial ordering on product CPO sets, we have

$$(f_1, f_2, \ldots, f_n) \leqslant (g_1, g_2, \ldots, g_n) \Rightarrow f_i \leqslant g_i, \quad \text{for all } i = 1, 2, \ldots, n$$

Therefore, $F^i(\perp)\downarrow 1 = f_1 \leqslant g_1 = E^{Ki}(\perp)\downarrow 1$. □

Lemma 7
Let $E(f)$ *be a system of equations and* $H(f, g)$ *be a eureka of* E. *Then, for all* $n = 1, 2, 3, \ldots,$ *we have*

$$F^n(\perp) = (E^n(\perp), H(x, y)) \qquad (*)$$

where $F(f, g) = E(f) \cup H(f, g)$, x, y *are expressions obtained by applications of* E *and* H *to* $\perp$. □

Theorem 2 (Eureka)
Let $E(f)$ *and* $F(f')$ *be two sets of equations. If there is a eureka* H *such that* F *can be derived from* $E \cup H$, *then there is a constant* $K \geqslant 1$ *such that for every* $n = 1, 2, \ldots,$

$$F^n(\perp)\downarrow 1 \leqslant E^{Kn}(\perp)\downarrow 1$$

Proof
Suppose that $H(f, g)$ is a set of equations such that F can be derived from $E \cup H$, the recursive function symbols f do not occur in the left hand sides of the equations in H, and g is the recursive function symbols introduced by H. Let $G(f, g) = E(f) \cup H(f, g)$. Then by Corollary 1, there is a constant $K \geqslant 1$ such that for every $n = 1, 2, \ldots,$

$$F^n(\perp)\downarrow 1 \leqslant G^{Kn}(\perp)\downarrow 1 \qquad (1)$$

By Lemma 7, we have for all $n = 1, 2, \ldots,$

$$G^n(\perp) = (E^n(\perp), E'(x, y)) \qquad (2)$$

where x, y are expressions constructed from $\perp$ by E and H. Hence, for all $n = 1, 2, \ldots,$

$$G^n(\perp)\downarrow 1 = E^n(\perp)\downarrow 1 \qquad (3)$$

Then, (1) and (3) imply that for all $n = 1, 2, \ldots.$

$$F^n(\perp)\downarrow 1 \leqslant E^{Kn}(\perp)\downarrow 1. \quad \square$$

Notice that Theorem 2 is a necessary condition of derivability but not sufficient. Therefore, it is possible that we cannot derive a program even if the condition is

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

satisfied. In these cases, eureka may be helpful. But, if the condition is not satisfied, we cannot derive the program no matter what eureka is added.

Subsequently, by saying 'F can be transformed into E' we mean that there is a eureka H such that $F; H - * \rightarrow E$; and by saying 'F cannot be transformed into E', we mean that there is no eureka H such that $F; H - * \rightarrow E$.

3.3 Some corollaries of the fundamental theorems

To conclude this section, we look at some corollaries of the theorems. Firstly, from Examples 1–3 we have the following incompleteness theorem for the folding/unfolding system:

Corollary 2 (Incompleteness theorem)
There are two programs f and g such that $f = g$, but there is no eureka u of f such that with the help of u, $f - * \rightarrow g$.

Proof
The equivalence of recursive functions may have different meanings according to the evaluation strategy. But the incompleteness theorem holds for both of the eager and lazy evaluation strategies. Example 1 gives a counter example for equivalence under eager evaluation. Example 2 gives a counter example for equivalence under lazy evaluation. □

Theorem 2 also implies the partial correctness of the folding/unfolding system.

Corollary 3 (Partial correctness theorem)
If a program g can be derived from the program f, then $g \leqslant f$.

Proof
Let E and F be systems of equations, f and g be the least fixed points of E and F, respectively. Assume that F is derived from E by folding/unfolding. Then by Theorem 2, there is a constant $K \geqslant 1$ such that for every $n = 1, 2, \ldots,$

$$F^n(\perp)\downarrow 1 \leqslant E^{Kn}(\perp)\downarrow 1.$$

Therefore,

$$\begin{aligned} g &= \bigsqcup_{n=1}^{\infty} \{F^n(\perp)\downarrow 1\} && \text{(Kleene's theorem)} \\ &\leqslant \bigsqcup_{n=1}^{\infty} \{E^{Kn}(\perp)\downarrow 1\} && \text{(Theorem 2)} \\ &= \bigsqcup_{n=1}^{\infty} \{E^n(\perp)\downarrow 1\} \\ &= f. && \text{(Kleene's theorem)} \quad \square \end{aligned}$$

An interesting corollary of the partial correctness theorem is that the folding/unfolding transformations preserve the strictness of functions.

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

Corollary 4
Let f be transformed into g by folding/unfolding. If f is a strict function (i.e. $f(\perp) = \perp$), then g is also a strict function.

Proof
By the partial correctness theorem, $g \leqslant f$. Therefore, for all x we have

$$g(x) \leqslant f(x)$$

In particular, $$g(\perp) \leqslant f(\perp) = \perp$$

That is, g is strict. □

Using the partial correctness theorem and Corollary 4, Example 1 can be proved very simply because under lazy evaluation $f(\perp + 1) = \perp$ by definition E, whilst $f(\perp + 1) = 0$ by definition F. But, Corollaries 3 and 4 cannot distinguish the two functions in Example 2.

4 A bound on efficiency improvement

A lot of work has been done to improve program efficiency and synthesize efficient algorithms by folding/unfolding (Darlington, 1978, 1984*b*; Pettrossi and Burstall, 1982). Example 4 is a typical example. It is pointed out that the time and space complexity of the function E in Example 4 are exponential while the function F is linear. However, these complexity measures are in the model of sequential computation. In a parallel computation model, the time complexities of the two programs are all linear. Therefore, in different computation models, the efficiency gained by transformation may be different.

In this section, we are going to apply the results obtained in the previous sections to obtain a model independent bound on the efficiency gain by folding/unfolding. Firstly, we will introduce a notion of inherent complexity of recursive functions, which is independent of the computation model. Then, we will discuss the effect of folding/unfolding transformations on the inherent complexity. Finally, we will discuss the relationship between time complexity and inherent complexity. The notion of reasonable models is introduced, and it is proved that inherent complexity is always less than or equal to the time complexity in reasonable models. Therefore, a bound of efficiency gain by transformations is obtained for all reasonable models.

4.1 Description of context

Under lazy evaluation, the time complexity of a function also depends on the context the function is called. To give a general treatment of computational complexity we must take context into account. A formulation of the context provided by Wadler and

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

Hughes (1987) uses projections on domains. A projection is a continuous function α so that

$$\alpha \circ \alpha = \alpha; \quad \text{and} \quad \alpha \leqslant \text{id}.$$

In other words, given an object u, a projection removes information from the object, ($\alpha u \leqslant u$), but once this information has been removed further application has no effect, ($\alpha(\alpha u) = \alpha u$). The information removed represents information not needed by the context. Thus, a projection describes what information is sufficient. To describe what information is necessary, Wadler and Hughes introduce a new domain element $\lightning$, called 'abort'. The interpretation of $\alpha u = \lightning$ is that context α requires a value more defined than u. To make this work, we must have $\lightning \leqslant \perp$ and all functions are naturally extended to be strict in $\lightning$. Therefore, the domain of a function f in the context α is defined to be the set of objects x so that $f(x)$ is sufficiently defined in the context. Formally, we have

$$\text{Dom}_\alpha(f) = \{x \mid (\alpha \circ f)\, x \neq \lightning\}$$

For the sake of convenience, we will omit the superscript if it is the projection STR,

$$\text{STR}\, u = \begin{cases} \lightning, & \text{if } u = \lightning, \text{ or } u = \perp; \\ u, & \text{otherwise.} \end{cases}$$

4.2 The notion of inherent complexity

Let $|.|$ be the size of input data (i.e. a function $|.|: D \to N$), where D is the input domain, $N = \{1, 2, 3, \ldots\}$. The inherent complexity of a recursive function is defined as follows:

Definition 2

The function $IC_\alpha: N \to N$ is called an inherent complexity of the recursive function $f \leqslant E(f)$ w.r.t. $|.|$ in the context α, where $IC_\alpha(n)$ is defined as follows:

$$IC_\alpha(n) = \max\{\text{depth}_\alpha(x) \mid |x| = n\},$$

where $\text{depth}_\alpha(x) = \min\{t \mid x \in \text{Dom}_\alpha(E^t(\perp) \downarrow 1)\}$, $\max X = \infty$, *if X is infinite or $\infty \in X$.*

If there are constants C_1, $C_2 > 0$ and natural number n_0 so that for all $n > n_0$ that

$$C_1 g(n) \leqslant IC_\alpha(n) \leqslant C_2 g(n),$$

we say that $IC_\alpha(n)$ is of order $g(n)$, written as $IC_\alpha(n) = O(g(n))$.

Intuitively, $x \in \text{Dom}_\alpha(E^t(\perp) \downarrow 1)$ means that when the value $f(x)$ is needed in the context α, the tth approximation $E^t(\perp) \downarrow 1$ of f, the least fixed point $E^\infty(\perp) \downarrow 1$, is able to produce an approximate result for the use under context α. The $\text{depth}_\alpha(x)$ is the minimum of such approximations. Then the inherent complexity $IC_\alpha(n)$ is the least m

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

such that the mth approximation is sufficient for all input x of size n. Since the mth approximation of the least fixed point corresponds to the depth of recursive calls being limited to be less than or equal to m, roughly speaking, the inherent complexity of a recursive function is the depth of recursive calls. Subsequently, we will call the number

$$\min\{t \mid x \in \mathrm{Dom}_\alpha(E^t(\bot)\downarrow 1)\}$$

the depth of recursion of $f(x)$ with respect to the context α, written as $\mathrm{depth}_{f,\alpha}(x)$.

Example 5

Let $|n| = n$. Let the context be STR.

(1) The inherent complexity of the function E in example 1 w.r.t. $|.|$ is $IC(n) = n$, because for every $i = 1, 2, \ldots,$

$$\min\{t \mid i \in \mathrm{Dom}(E^t(\bot))\} = i.$$

But, the inherent complexity of the function F is $O(1)$.

(2) The function E in Example 2 is of linear inherent complexity, but the inherent complexity of the function F is $O(\log_2 n)$.

(3) The inherent complexities of the two functions in Example 4 are linear, i.e. of the order $O(n)$.

(4) Define the function *iota* as follows:

$$\begin{cases} \mathrm{iota}\, 0 \Leftarrow \langle 0 \rangle \\ \mathrm{iota}\,(n+1) \Leftarrow (n+1) :: (\mathrm{iota}\, n) \end{cases}$$

Let $|\langle x_1, x_2, \ldots, x_n \rangle| = n$. Define the projection *take_k* to be the function

$$take\_k\ \langle x_1, x_2, \ldots, x_m \rangle = \begin{cases} \langle x_1, x_2, \ldots, x_k \rangle, & \text{if } m \geqslant k; \\ \lfloor, & \text{if } m < k. \end{cases}$$

The inherent complexity of iota in the context *take_k* is constant, i.e. $O(k)$. However, the inherent complexity of iota in the context *take_all* is $O(n)$, where

$$take\_all\ \langle x_1, x_2, \ldots, x_m \rangle = \begin{cases} \langle x_1, x_2, \ldots, x_m \rangle & \text{if } x_i \neq \bot, \text{ for all } i = 1, 2, \ldots, m; \\ \lfloor, & \text{otherwise.} \end{cases}$$ □

The following theorem shows that folding/unfolding transformations cannot improve the order of inherent complexity of recursive functions:

Theorem 3

Let α be a given context. Assume that the program F is derived from E by folding/unfolding transformation, and that their inherent complexities are $f(n)$ and $g(n)$ in the context α, respectively. Then, there is a constant $C > 0$ so that

$$f(n) \geqslant C * g(n).$$

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

Proof

Let $\text{depth}_\alpha(x) = \min\{t \mid x \in \text{Dom}_\alpha(E^t(\perp)\downarrow 1)\}$,

$\text{depth}'_\alpha(x) = \min\{t \mid x \in \text{Dom}_\alpha(F^t(\perp)\downarrow 1)\}$.

By Theorem 3, there is a constant $K \geqslant 1$, such that for every $t = 1, 2, \ldots$,

$$F^t(\perp)\downarrow 1 \leqslant E^{Kt}(\perp)\downarrow 1.$$

By the monotonicity of projections, we have

$$\alpha \circ F^t(\perp)\downarrow 1 \leqslant \alpha \circ E^{Kt}(\perp)\downarrow 1.$$

Hence, by the definition of Dom, we have that for every x,

$$x \in \text{Dom}_\alpha(F^t(\perp)\downarrow 1) \Rightarrow x \in \text{Dom}_\alpha(E^{Kt}(\perp)\downarrow 1),$$

i.e.
$$\text{depth}'_\alpha(x) \geqslant (1/K)\,\text{depth}_\alpha(x).$$

By the definition of inherent complexity, we have

$$\begin{aligned} f(n) &= \max\{\text{depth}'_\alpha(x) \mid |x| = n\} \\ &\geqslant \max\{(1/K)\,\text{depth}_\alpha(x) \mid |x| = n\} \\ &= (1/K)\max\{\text{depth}_\alpha(x) \mid |x| = n\} \\ &= (1/K)\,g(n). \quad \square \end{aligned}$$

4.3 Reasonable computation models and a bound of efficiency gain

A computation model is said to be reasonable if the amount of time needed in computing a recursive function is always greater than or equal to the maximal depth of recursion in the execution. More formally, let f be a recursive function, $\text{depth}_{f,\alpha}(x)$ be the depth of recursion of execution $f(x)$ under context α, and $T_{f,\alpha}(x)$ be the time needed to compute $f(x)$ in a computation model under context α. Then,

Definition 3

A computation model is said to be reasonable, if, for all functions f and data x, and context α

$$\text{depth}_{f,\alpha}(x) \leqslant T_{f,\alpha}(x)$$

Obviously, a sufficient condition for a computation model being reasonable is that each function call consumes at least one time unit, no matter whether the function call is in parallel with the evaluation of other expressions of the same program. Therefore, parallel computation, sequential computation, string reduction and graph reduction are all reasonable, if function calls and reductions are time consuming.

Lemma 8

Let α *be any given context. In reasonable models, the following inequality holds for the inherent complexity* $IC_\alpha(n)$ *and time complexity* $T_\alpha(n)$ *of recursive programs in the context* α

$$IC_\alpha(n) \leqslant T_\alpha(n)$$

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

Proof
By the definitions. □

Now we can get a bound on efficiency gain by transformations for all reasonable models.

Theorem 4 (Bound on efficiency gain)
Let α *be any given context, and* $O(g(n))$ *be the order of the inherent complexity of a recursive program E in the context* α. *Then the order of the time complexity in the context* α *of the program derivable from E by folding/unfolding transformations will not be less than* $O(g(n))$.

Proof
Let F be a program derived from E by folding/unfolding transformations, $IC_E(n)$ be the inherent complexity of E, $IC_F(n)$ and $T(n)$ be the inherent complexity and time complexity of F, respectively. Then, by Lemma 8, $T(n) \geqslant IC_F(n)$. By Theorem 3, there is a constant $C > 0$ such that $IC_F(n) \geqslant C * IC_E(n) = O(g(n))$. □

Notice that if in a computation model the units of space needed in computing a recursive function are always greater than or equal to the maximal depth of recursion in the execution, than a similar result can be obtained for the space complexity of the recursive program. However, on reduction machines it is possible that recursive calls do not consume any space resource. For example, by string rewriting, the space needed to compute $f(n)$ given in Example 1 does not depend on the depth of recursion.

4.4 Some examples

Example 6 (Search algorithms)
The following is a linear search algorithm:

$$\begin{cases} \text{linear\_search}(n, a) \Leftarrow f(1, n, a) \\ f(i, i, a) \Leftarrow \textbf{if } (A[i] = = a) \textbf{ then } 1 \textbf{ else } 0 \\ f(i, i+k, a) \Leftarrow \textbf{if } (A[i] = = a) \textbf{ then } i \textbf{ else } f(i+1, i+k, a) \end{cases}$$

It cannot be transformed into the binary search algorithm given below, because the former is of linear inherent complexity while the latter is of the order $O(\log_2 n)$, where n is the number of elements in the table to be searched

$$\begin{cases} \text{binary\_search}(n, a) \Leftarrow f(1, n, a) \\ f(i, i, a) \Leftarrow \textbf{if } A[i] = = a \textbf{ then } i \textbf{ else } 0 \\ f(i, i+k, a) \Leftarrow \textbf{if } (A[i+k/2] = = a) \textbf{ then } i+k/2 \\ \qquad\qquad\qquad\qquad \textbf{else if } (A[i+k/2] < a) \textbf{ then } f(i+k/2+1, i+k, a) \\ \qquad\qquad\qquad\qquad \textbf{else } f(i, i+k/2-1, a) \end{cases}$$

where we assume that $A[i] \leqslant A[i+1]$, for all $i = 1, 2, \ldots, n$. Here, to avoid syntactic

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

reasons for untransformability, we make the two functions have the same functionality. □

Example 7 (Sorting algorithms)
This example will show underivability between sorting algorithms.

(1) Consider the following two sorting programs:

$$\text{insert\_sorting}: \begin{cases} \text{sort}(\text{nil}) \Leftarrow \text{nil} \\ \text{sort}(x::x1) \Leftarrow \text{insert}(x, \text{sort}(x1)) \\ \text{insert}(x, \text{nil}) \Leftarrow \langle x \rangle \\ \text{insert}(x, y::y1) \Leftarrow \textbf{if } x > y \textbf{ then } y::\text{insert}(x, y1) \textbf{ else } x::(y::y1) \end{cases}$$

$$\text{merge\_sorting}: \begin{cases} \text{sort nil} \Leftarrow \text{nil} \\ \text{sort } x::x1 \Leftarrow \text{merge (sort } u, \text{ sort } v) \text{ where } \langle u, v \rangle = \text{split } (x::x1) \\ \text{merge}(\text{nil}, y) \Leftarrow \langle y \rangle \\ \text{merge}(x, \text{nil}) \Leftarrow \langle x \rangle \\ \text{merge}(x::x1, y::y1) \Leftarrow \textbf{if } x > y \textbf{ then } y::\text{merge}(x::x1, y1) \\ \qquad\qquad\qquad\qquad\qquad\qquad \textbf{else } x::\text{merge}(x1, y::y1) \end{cases}$$

If we consider the functions insert and merge as primitives, the inherent complexity of insert sorting is $O(n)$, but the inherent complexity of merge sorting is $O(\log_2 n)$. By Theorem 3, we cannot transform insert sorting into merge sorting.

Now the question is whether we can transform insert sorting into merge sorting if insert and merge are not considered as primitive functions? The answer is no. There is no call to the function sort in the definitions of insert and merge, so we can consider the definitions of insert and merge as laws about primitives. If there were a transformation with the definitions of insert and merge, by replacing each use of the definitions in the transformation with an application of a law rule, we could have obtained a transformation without the definitions of insert and merge.

(2) The following bubble sorting algorithm

for $i = n$ **to** 1 **do**
  for $j = 0$ **to** $i-1$ **do**
    if $A[j] > A[j+1]$ **then** $(A[j], A[j+1] := (A[j+1], A[j])$ **endif**.

can be translated into the following equations:

$$\text{bubble\_sort}: \begin{cases} \text{sort}(A) \Leftarrow \textbf{for}(\|A\|, 0, A) \\ \textbf{for}(0, 0, A) \Leftarrow A \\ \textbf{for}(i+1, i+1, A) \Leftarrow \textbf{for}(i, 0, A) \\ \textbf{for}(j+k+1, j, A) \Leftarrow \textbf{for}(j+k+1, j+1, \text{exchange}(j, A)) \end{cases}$$

where *exchange* is considered as a primitive function which computes the if-statement in the loop body.

The inherent complexity of bubble sorting is $O(n^2)$, where $n = \|A\|$ is the length of

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

the list A to be sorted. Therefore, from the system of equations, we cannot derive any sorting algorithm which has time complexity less than $O(n^2)$.

However, the same bubble sorting algorithm defined by the following equations is of linear inherent complexity:

$$\text{bubble\_sort}': \begin{cases} \text{sort nil} \Leftarrow \text{nil} \\ \text{sort}\,(x::x1) \Leftarrow m::\text{sort}\,(\text{delete}\,(m, x::x1))\ \textit{where}\ m = \min(x::x1) \\ \min(x::\text{nil}) \Leftarrow x \\ \min(x::(y::y1)) \Leftarrow (\textit{if}\ x > z\ \textbf{then}\ z\ \textbf{else}\ x)\ \text{where}\ z = \min(y::y1). \end{cases}$$

where *delete* is a primitive function which deletes an element from a list.

Notice that the function

$$\text{Min}(A) = \text{head}(\text{bubble\_sort}(A))$$

is of $O(n^2)$ inherent complexity. Therefore, by Theorem 4, we can neither transform it into a linear time complexity program, nor evaluate the program in linear time complexity under lazy or eager evaluation. But,

$$\text{Min}(A) = \text{head}(\text{bubble\_sort}'(A))$$

has a linear inherent complexity. In fact, it cannot only be transformed into a program of linear time complexity, but can also be evaluated in linear time complexity under lazy evaluation (Wadler, 1988). □

5 Conclusion

It is well-known that the folding/unfolding system is partially correct but incomplete (Burstall and Darlington, 1977). Formal proofs of the partial correctness can be found in Knott (1985) and Yongqian et al. (1987). From their proofs, we cannot get the results of this paper. However, the partial correctness and incompleteness are corollaries of our results.

Example 1 of this paper comes from Burstall and Darlington (1977), in which the following counter example of completeness is given. The function

$$f(0) \Leftarrow 0$$
$$f(n+1) \Leftarrow f(n)$$

cannot be transformed into

$$f(n) \Leftarrow 0$$

by folding/unfolding. The reason of underivability given by Burstall and Darlington is that the pattern n cannot be derived from $n+1$ and 0. No formal proof is given, and whether there exist some eureka making the transformation possible is also not answered. Here, we changed the definition of the second function so that their argument does not apply to our example, but it is still not transformable as formally proved in the paper. And by Theorem 2, we know that there is no eureka which can make the transformation possible. Moreover, by Theorem 4 of the paper, any program derivable from the first program has a linear time complexity.

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

The necessary conditions of derivability obtained in the paper is strong enough for practical use, especially the result about the bound on efficiency gain. It is so tight that many pairs of algorithms can be distinguished. Moreover, it is possible to determine the transformability without knowing the details of the program it has to be transformed to. For example, from the bubble sorting program in the Example 7, it is impossible to derive any sorting algorithm whose order of time complexity is less than $O(n^2)$, hence quick sorting is underivable since its time complexity is $O(n \log_2 n)$. The result is also very general in that it is true for all reasonable computation models. As far as we know, there is no similar work on such conditions of derivability, especially the power of eureka and the bound on efficiency gain.

Acknowledgement

This work started when I was in Nanjing University, P R China, and continued when I was visiting Brunel University, UK, sponsored by British Council. I am most grateful to Professor Xu Jiafu of Nanjing University and Dr Chris Reade of Brunel University for their invaluable comments and corrections. Many other people have contributed in various ways; thanks to Dr John Launchbury, Professor John Hughes of Glasgow University, Dr Alan Hutchinson of King's College of London, Dr Peter Burton of Queen Mary and Westfield College of University of London, Val Kirby of The Open University, and the referees of the paper.

Appendix: Proofs of lemmas

Proof of Lemma 1 (sketch)

(i) Obvious.

(ii) By induction on n and use of (i) repeatedly. □

Proof of Lemma 2 (sketch)

(i) Notice that whenever the rule $L\sigma \Leftarrow R\sigma$ is applicable to an expression e, the rule $L \Leftarrow R$ is also applicable and the results are the same. Therefore, by induction on the length of the computation, we can prove that every computation sequence in $F(g)$ can be transformed into a computation sequence in $E(g)$ which leads to the same result by replacing the application of $L\sigma \Leftarrow R\sigma$ by $L \Leftarrow R$. Thus, we have

$$F(g) \leqslant E(g)$$

On the other hand, E is a subset of F, hence by Lemma 1, we have

$$E(g) \leqslant F(g)$$

(ii) By induction on n and use of (i) repeatedly. □

Proof of Lemma 3

Without loss of generality, we assume that the system E of equations defines two function symbols f and g, F is the subset of equations in E which defines the function symbol f, and G is the subset of equations in E which defines the function symbol g.

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

Assume that E is transformed to H by applying the unfolding rule:

$$E \underset{U \Leftarrow V, \rho}{\overset{L \Leftarrow R}{-\mathrm{U}\rightarrow}} H$$

Without loss of generality, we also assume that $L \Leftarrow R$ is in the form of

$$f p_1 \dots p_m \Leftarrow R$$

and $U \Leftarrow V$ is in the form of

$$g q_1 \dots q_n \Leftarrow V$$

By the applicability condition of the unfolding rule, we have

$$V|\rho = (f p_1 \dots p_m)\sigma$$

for some substitution σ so that $\sigma(f) = f$.

We assume that the σ is the identity substitution. This does not involve loss of generality, because if σ is not the identity substitution we can apply the instantiation rule

$$E \underset{\sigma}{\overset{L \Leftarrow R}{-\mathrm{I}\rightarrow}} E''$$

and obtain an equation $L\sigma \Leftarrow R\sigma$ in E''. Then apply the unfolding rule to E''

$$E'' \underset{U \Leftarrow V, \rho}{\overset{L\sigma \Leftarrow R\sigma}{-\mathrm{U}\rightarrow}} H$$

This application satisfies the assumption. And by Lemma 2,

$$E^i(\bot, \bot) = E''^i(\bot, \bot), \quad \text{for all } i = 1, 2, \dots.$$

So the proof of the Lemma can be done by proving E''.

Then, by the definition of the unfolding rule, H is

$$H: \begin{cases} E; \\ g q_1 \dots q_n \Leftarrow V[R(f,g)\backslash\rho] \end{cases}$$

Now, we prove that $H^n(\bot, \bot) \leqslant E^{2n}(\bot, \bot)$ by induction on n. When $n = 1$, we have

$$H(\bot, \bot) = \begin{cases} E(\bot, \bot); \\ g q_1 \dots q_n \Leftarrow V[R(f,g)\backslash\rho](\bot, \bot) \end{cases}$$

$$= \begin{cases} E(\bot, \bot); \\ g q_1 \dots q_n \Leftarrow V' \end{cases}$$

where $V' = V[R(\bot, \bot)\backslash\rho, \bot\backslash\rho', \quad \text{for all } \rho' \neq \rho \text{ so that } V|\rho' = f \text{ or } V|\rho' = g]$.

Because $\bot \leqslant F(\bot, \bot)$, $\bot \leqslant G(\bot, \bot)$, and by Lemma 1, $R(\bot, \bot) \leqslant F(\bot, \bot)$, the following inequality follows from the monotonicity of functionals,

$$(g q_1 \dots q_n \Leftarrow V') \leqslant (g q_1, \dots, q_n \Leftarrow V'')$$

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

where

$$V'' = V[F(\bot,\bot)\backslash\rho, \quad \text{for all } \rho \text{ so that } V|\rho = f,$$
$$G(\bot,\bot)\backslash\rho', \quad \text{for all } \rho' \text{ so that } V|\rho' = g].$$

Obviously, $V'' = V(F(\bot,\bot), G(\bot,\bot))$.

The satatement follows from the inequality that $E(\bot,\bot) \leqslant E^2(\bot,\bot)$.

Assume that the inequality holds when $n = k$. Then when $n = k+1$, we have

$$H^{k+1}(\bot,\bot) = H(H^k(\bot,\bot)) \leqslant H(E^{2k}(\bot,\bot)) \quad \text{(Hypothesis, monotonicity of } H)$$

$$= \begin{cases} E(E^{2k}(\bot,\bot)); \\ g\,q_1 \ldots q_n \Leftarrow V[R(f,g)\backslash\rho](E^{2k}(\bot,\bot)) \end{cases}$$

$$= \begin{cases} E^{2k+1}(\bot,\bot); \\ g\,q_1 \ldots q_n \Leftarrow V') \end{cases}$$

where
$$V' = V[R(E^{2k}(\bot,\bot))\backslash\rho,$$
$$F(E^{2k-1}(\bot,\bot))\backslash\rho' \text{ for all } \rho' \neq \rho \text{ so that } V|\rho' = f,$$
$$G(E^{2k-1}(\bot,\bot))\backslash\rho'' \text{ for all } \rho'' \text{ so that } V|\rho'' = g]$$

Because $E^{2k-1}(\bot,\bot) \leqslant E^{2k}(\bot,\bot)$, and by Lemma 1, we have

$$R(E^{2k}(\bot,\bot)) \leqslant F(E^{2k}(\bot,\bot))$$

By the monotonicity of E, this implies that

$$\begin{aligned} & g\,q_1 \ldots q_n \Leftarrow V' \\ & \leqslant g\,q_1 \ldots q_n \Leftarrow V[F(E^{2k}(\bot,\bot))\backslash\rho \text{ for all } \rho \text{ so that } V|\rho = f, \\ & \qquad G(E^{2k}(\bot,\bot))\backslash\rho \text{ for all } \rho \text{ so that } V|\rho = g] \\ & = g\,q_1 \ldots q_n \Leftarrow V(E^{2k+1}(\bot,\bot)) \\ & \leqslant E(E^{2k+1}(\bot,\bot)) \\ & = E^{2(k+1)}(\bot,\bot). \qquad \text{(Lemma 1(i))} \end{aligned}$$

Moreover, because $E^{2k+1}(\bot,\bot) \leqslant E^{2k+2}(\bot,\bot)$, we have

$$H^{k+1}(\bot,\bot) \leqslant E^{2(k+1)}(\bot,\bot). \quad \square$$

Proof of Lemma 4

By induction on n. Very similar to the proof Lemma 3. $\square$

Proof of Lemma 5 (Sketch)

(i) Let E is transformed into F by applying the law $P = Q$ to the right hand side of an equation $L \Leftarrow R$, i.e.

$$E \xrightarrow[P=Q,\rho]{L \Leftarrow R} F$$

where
$$F = E + (L \Leftarrow R[Q\sigma\backslash\rho])$$

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

and (+) $R \mid \rho = P\sigma$

Notice that by a law we mean that for all substitution σ',

(*) $P\sigma' = Q\sigma'$.

Therefore, for all functions $(a, b, \ldots, c)$

$$\begin{aligned}
&(R[Q\sigma \backslash \rho])(a, b, \ldots, c) \\
&\quad = (R[Q\sigma \backslash \rho])[f \to a, g \to b, \ldots, h \to c] \\
&\quad = (P[\phi \to a, g \to b, \ldots, h \to c])[Q(\sigma[f \to a, g \to b, \ldots, h \to c]) \backslash \rho] \\
&\quad = (R[f \to a, g \to b, \ldots, h \to c])[P(\sigma[f \to a, g \to b, \ldots, h \to c]) \backslash \rho] \qquad \text{(by (*))} \\
&\quad = (R[P\sigma \backslash \rho])(a, b, \ldots, c) \\
&\quad = R(a, b, \ldots, c) \qquad \text{(by (+))}
\end{aligned}$$

Therefore, for all functions $a, b, \ldots, c$,

$$E(a, b, \ldots, c) = F(a, b, \ldots, c).$$

(ii) By induction on n and repeated uses of (i). □

Proof of Lemma 6.
Very similar to Lemma 5. □

Proof of Lemma 7 (By induction on n)

(a) When $n = 1$, we have

$$F(\perp) = (E(\perp), H(\perp, \perp))$$

So, (*) holds when $n = 1$.

(*b*) Assume that (*) holds when $n = k$, then when $n = k+1$, we have that

$$\begin{aligned}
&F^{k+1}(\perp) \\
&\quad = F(F^k(\perp)) \qquad \text{(definition)} \\
&\quad = (E(f), H(f, g))(F^k(\perp)) \qquad \text{(definition of } F\text{)} \\
&\quad = (E(f), H(f, g))(E^k(\perp), F(x, y)) \qquad \text{(Hypothesis)} \\
&\quad = (E(E^k(\perp)), H(E^k(\perp), F(x, y)) \qquad \text{(assumptions)} \\
&\quad = (E^{k+1}(\perp), H(x', y')) \qquad \text{(definition)}
\end{aligned}$$

where $x' = E^k(\perp)$, and $y' = F(x, y)$. So (*) holds when $n = k+1$.

Therefore, the equation holds for all $n = 1, 2, \ldots$. □

References

Bauer, F. L. and Broy, M. (ed) (1979) *Program Construction. Lecture Notes in Computer Science 69*, Springer-Verlag, Berlin.

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

Darlington, J. (1981*a*) An experimental program transformation and synthesis system. *Artificial Intelligence*, **16**: 1–46.

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press

Darlington, J. (1978) A synthesis of several sorting algorithms, *Acta Informatica*, **11** 91): 1–31.
Darlington, J. (1981*b*) The structured description of algorithm derivations. In de Bakker and von Vliet (Editors), *Algorithmic Languages*. IFIP, North Holland, Amsterdam, 1981*b*, pp 221–250.
Darlington, J. (1984*a*) Program transformation in the *ALICE* project. In P. Pepper (Editor), *Program Transformation and Programming Environments*. Springer-Verlag, Berlin.
Darlington, J. (1984*b*) The design of efficient data representations. In A. W. Biermann *et al.* (Editors), *Automatic Program Construction Techniques*. Macmillan.
Darlington, J. (1985) Program transformation. *BYTE*, **10** (8): 201–216.
Feather, M. S. (1982) A system for assisting program transformation. *ACM TOPLAS*, **4** (1): 1–20.
Gunter, C. A. and Scott, D. S. (1990) Semantic domains. In J. van Leeuwen (Editor), *Handbook of Theoretical Computer Science*, pp 633–674, Elsevier.
Kleene, S. C. (1951) *Introduction to Metamathematics*. North Holland.
Kott, L. (1985) Unfold/fold program transformation. In M. Nivat and J. C. Reynolds (Editors), *Algebraic Methods in Semantics*. Cambridge University Press.
Manna, Z. (1974) *Mathematical Theory of Computation*. McGraw-Hill.
Meertens, L. G. T. (ed) (1987) *Program Specification and Transformation*. IFIP, North-Holland, Amsterdam.
Nielson, H. R. and Nielson, F. (1990) Eureka Definition for Free! In *ESOP '90*, pp 291–305.
Partsch, J. and Stenbruggen, R. (1983) Program transformation systems. *Computing Surveys*, **15** (3): 199–236.
Pepper, P. (ed) (1984) *Program Transformation and Programming Environments*. Springer-Verlag, Berlin.
Pettorossi, A. (1984) A powerful strategy for deriving efficient programs by transformation. In *ACM Symposium on Lisp and Functional Programming*, pp 273–281, Austin, Texas.
Pettorossi, A. (1989) Derivation of programs which traverse their input data only once. In G. Cioni and A. Salwicki (Editors), *Advanced Programming Methodologies*, pp 165–184, Academic Press.
Pettorossi, A. and Burstall, R. M. (1982) Deriving very efficient algorithms for evaluating linear recurrence relations using the program transformation technique. *Acta Informatica*, **18**: 181–209.
Schmidt, D. A. (1986) *Denotational Semantics: A methodology for language development*. Allyn and Bacon.
Wadler, P. L. and Hughes, R. J. M. (1987) Projections for strictness analysis. In *Proc. 3rd International Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon.
Wadler, P. (1988) Strictness analysis aids time analysis. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pp 119–132, San Diego, CA.
Yongqian, S., *et al.* (1987) *Termination preserving problem in the transformation of applicative programs. J. of Computer Science and Technology*, **2** (3): 191–201.

https://doi.org/10.1017/S0956796800000964 Published online by Cambridge University Press