Equivalent Transformations for Invariant Parallel Functions

MARK TRAKHTENBROT
Computer Science Department
Holon Academic Institute of Technology
52 Golomb St., POB 305, Holon 58102
ISRAEL

Abstract: - Monotonic parallel functions were extensively studied in research on semantics of programming languages. While most of this research concentrated on expressive power of parallel functions, this paper focuses on development of a rich catalog of equivalent transformations associated with invariant parallel functions. Such functions are independent of interpretation of their definition domain, and they can be naturally used as additional control means to enrich models of sequential programs (such as recursive program schemes). It is shown how the offered transformations can be applied for a variety of goals, such as regularization of terms and reducing the strength of the used operations.

Key-Words: - Program schemes, Parallel functions, Invariant functions, Equivalent transformations.

1 Introduction

The concept of monotonic function is central to research on semantics of programming languages, starting from works by D.Scott in early 70-ies [2]. Parallel monotonic functions were first considered in paper [1]. Over years, research on parallel functions concentrated mainly on their impact on expressive power of languages [1,3-7]. In particular, the work [7] investigated the use of such functions in the language of recursive program schemes. Program schemes use non-interpreted function symbols along with interpreted ones. Behavior of interpreted functions used in program schemes shouldn't depend on interpretation of their definition domain; this was formalized in [7] in terms of invariant monotonic functions. Based on this concept, hierarchies of recursive schemes were studied that are obtained when the language of recursive schemes is augmented with various parallel invariant functions.

This paper puts focus on another important aspect in the study of augmented classes of recursive program schemes, namely on their equivalent transformations. Such transformations can be applied for a wide variety of goals: optimization, simplification, check of equivalence between different implementations, transformation into special "regular" forms, reduction of the strength of used operations, etc. The paper investigates equivalent transformations for a simple and yet powerful language of terms in which every sequential and parallel invariant function can be expressed. The main objective is to develop a rich catalog of transformations for manipulations on terms, in order to match the above goals.

2 Basic Concepts

First, let's shortly recall some well-known concepts. Let D be any domain, extended with the undefined value ω that is viewed as a result of non-terminating computation process. A partial order $\subseteq$ is then defined: non-ω elements are non-comparable, and ω is less than any other element; we write $x \equiv y$ when $x \subseteq y$ and $y \subseteq x$. Finally, function f *is* ***monotonic*** with respect to $\subseteq$ if:

$$\forall i[x_i \subseteq y_i] \rightarrow f(x_1,...,x_n) \subseteq f(y_1,...,y_n).$$

Monotonic functions are classified as ***sequential*** and ***parallel***. Furthermore, a monotonic function f is called ***invariant*** [7] if for any 1-1 mapping $h: D \rightarrow D$ such that $h(\omega) \equiv \omega$, the following holds: $f(h(x_1),...,h(x_n)) \equiv h(f(\bar{x}))$.

All functions below are invariant, and they don't depend on the nature of domain D:

- equality $x = y$; gets value ω when $x \equiv \omega$ or $y \equiv \omega$ (can calculate the arguments in any order)
- conditional function $if(\alpha, x, y) \equiv$ *if* α *then* x *else* y is sequential: first calculate α; if the process terminates then calculate x or y respectively
- parallel $IF(\alpha, x, y)$ with $IF(\omega, x, x) \equiv x$; here all arguments should be calculated in parallel)
- parallel voting function $V_n^m(x_1,...,x_n)$ ($[n/2] < m < n$) : if at least m arguments get the same non-ω value then V_n^m also gets this value; otherwise $V_n^m(\bar{x}) \equiv \omega$.

3 Transformation of Terms Over a Full Basis for Invariant Functions

Recall (see [6]) the following fact:

Theorem 1. Every invariant function is expressible as composition of functions *IF*, V_3^2 (further mentioned just as *V*), equality predicate =, variables and the constant ω.

In other words, functions *IF, V,* = constitute a basis in the class of invariant functions. The set of such terms is denoted as *T(IF, V, =)*. Two terms in *T(IF, V, =)* are called ***equivalent*** if for any assignment of values to their variables they produce the same result in *D*.

Below we present a catalog of transformations for terms in *T(IF, V, =)*, followed by their analysis that includes warnings against some "obvious" simplifications that turn out to be "false friends".

In the sequel, instead of $IF(\alpha,x,y)$ we write *IF* α *then x else y*. Latin letters are used to denote terms in *T(IF, V, =)* that get values in *D* , while Greek letters are used for term with Boolean values. For example, *t* could denote the term

V(IF (V(x,y,z)=u) then x else v , u, z)

while α could stand for

$V(\omega,x,y)$ = *IF* $(V(x,y,z)=v)$ *then u else x*

The transformations are:

(1) $x=y \leftrightarrow y=x$

(2) $x=\omega \leftrightarrow \omega$

(3) *IF* α *then x else x* $\leftrightarrow$ *x*

(4) *IF* ω *then x else y* $\leftrightarrow$ *IF* x=y *then x else* ω

(4') *IF* ω *then x else y* $\leftrightarrow$ *IF* x=y *then y else* ω

(5) *IF* α *then x else* (*IF* α *then y else z*) $\leftrightarrow$
IF α *then* (*IF* α *then x else y*) *else z*

(6) *IF* α *then* (*IF* β *then x else y*)
else (*IF* β *then t else z*) $\leftrightarrow$
IF β *then* (*IF* α *then x else t*)
else (*IF* α *then y else z*)

(7) *IF* (*IF* α *then* β *else* γ) *then x else y* $\leftrightarrow$
IF α *then* (*IF* β *then x else y*)
else (*IF* γ *then x else y*)

(8) *x = (IF* α *then y else z)* $\leftrightarrow$
IF α *then x=y else*
(*IF* α *then* (*IF z=y then x=y else* ω) *else x= z*)

(9) $V(x,x,y) \leftrightarrow x$

(10) $V(x,y,z) \leftrightarrow V(x,z,y)$

(10') $V(x,y,z) \leftrightarrow V(y,x,z)$

(11) $V(x,t,V(x,y,z)) \leftrightarrow V(x,y,V(x,t,z))$

(12) $V(x,y,\omega) \leftrightarrow$ *IF x=y then x else* ω

(13) *IF* $(V(x,y,z)=x)$ *then t else u* $\leftrightarrow$
IF x=y then t else (IF x=z then t else
(IF y=z then u else (IF t=u then t else ω*)))*

(14) *V(IF* α *then x else y, t, z)* $\leftrightarrow$
IF α *then* $V(x,t,z)$ *else* $V(y,t,z)$

(15) *IF* $(V(x,y,z)=t)$ *then u else v* $\leftrightarrow$
V(
IF $(V(x,y,z)=x)$ *then (IF x=t then u else v) else u,*
IF $(V(x,y,z)=y)$ *then (IF y=t then u else v) else u,*
IF $(V(x,y,z)=z)$ *then (IF z=t then u else v) else u*
)

Theorem 2. For any terms τ_1 and τ_2 belonging to *T(IF, V, =)*, if τ_2 is obtained from τ_1 by application of one or more of the transformations 1-15, then τ_1 and τ_2 are equivalent.

Proof: Straightforward.

Let's consider now some nuances related to the above transformations:

1-2) Note that $x=x \leftrightarrow$ *true* can't be added here, because if *x* is equal to ω then *x=x* is also equal to ω, and not to *true.*

4-4') It follows from these rules that:

IF x=y *then x else* $\omega \leftrightarrow$ *IF* x=y *then y else* ω

Note that a similar "rule":

IF x=y *then x else* z $\leftrightarrow$ *IF* x=y *then y else* z

is not correct: if *x* is equal to ω and *y=z*, then the left side equals ω, while the right side is equal to *z*.

5) For sequential function $if(\alpha,x,y)$ the following holds: *if* α *then x else (if* α *then y else z)* $\leftrightarrow$ *if* α *then x else z* . However, a similar "rule" for parallel conditional $IF(\alpha,x,y)$ is not correct (consider, for example, the case when α is undefined, while *x, y, z* differ from ω and $x=z$, $x\neq y$). Hence instead our rule (5) is required.

7-8) These rules use two types of function *IF*: one that gets values in domain *D* and another one that gets Boolean values. Note that terms in *T(IF, V, =)* are built using only the first type of *IF*; boolean version of *IF* appears only in intermediate stages of transformation, when rule (8) is applied.

8) The "obvious" rule *x = (IF* α *then y else z)* $\leftrightarrow$ *IF* α *then x=y else x=z* is not correct. Indeed, if α is undefined and *x, y, z* have pair wise distinct values that differ from ω, then the term on the left side equals ω while the one on the right side is *false*.

8, 14) In general, if we consider composition of an arbitrary monotonic function *f* with parallel conditional *IF*, then the correct way to commute between the two is described by the following rule:

f(...,IF α then x else y, ...) ↔
IF α then f(...,x,...) else
(IF α then (IF x=y then f(...,x,...) else f(..., ω,...))
else f(...,x,...))

Transformation (8) is an instance of this more general rule, with the equality predicate standing for *f*. Note that sometimes getting the *IF* out of *f* can be done in a simpler way, as seen in our rule (14), where the voting function *V* stands for *f*.

4 Transformation of Terms into a Regular Form

Denote by *T(V)* the set of all terms obtained by composition of the voting function *V*, variables and the constant ω. We define the set *TreeT(IF, V, =)* of ***tree-terms*** as follows:

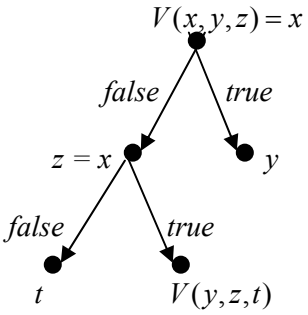
1) $x \in T(V) \rightarrow x \in TreeT(IF,V,=)$

2) $x,y \in T(V)$, $z,t \in TreeT(IF,V,=) \rightarrow$
IF x=y then t else z $\in TreeT(IF,V,=)$

Clearly, every term in *TreeT(IF, V, =)* can be represented as a tree in which leaves correspond to terminals of the term (that belong to *T(V)*), while other nodes correspond to the term's tests. For example, for term:

IF(V(x,y,z) = x) then y else
(IF z=x then V(y,z,t) else t)

we get the following tree:



Here $V(x,y,z) = x$ and $z = x$ are tests, while t, y and $V(y,z,t)$ are terminals.

Let's now define a set of so called ***regular*** terms. It contains all terms $\tau \in TreeT(IF,V,=)$ such that every test in τ is either the constant ω, or has the form $x = y$ where x, y are variables over *D*.

Theorem 3. Every term in *T(IF, V, =)* can be transformed into an equivalent regular term using the transformation rules 1-15 defined in section 3.

Proof. In fact, to transform a term into equivalent regular term it is enough to apply the rules 1,2,7,8,10,10',13-15. This is achieved in several steps:

(a) The goal of the first one is to eliminate all occurrences of function *IF* appearing in argument positions of function *V* or of the equality predicate. For this, rules 8 and 14 are applied while possible (taking into account that arguments of *V* and = can be switched using rules 1, 10 and 10'.

(b) Rule 7 is then applied while possible; clearly, this results in a tree-term from *TreeT(IF, V, =)*.

(c) Occurrences of function *V* in tests of a tree-term are eliminated during the bottom-up traverse of the tree (from its leaves to the root). Each time occurrence of *V* in a test is found, we either apply rule 13, or rule 15 followed by three applications of rule 13. After that, rule 14 is applied repeatedly to eliminate those occurrences of *IF* inside *V* that could have been created in the case when rule 15 is used. Note that after processing of one test containing *V*, several new such tests can appear (e.g. when rule 13 is applied, and $x \equiv V(t,u,w)$), However, the number of occurrences of *V* in each of these tests as at least by one smaller than the number of its occurrences in the original test. This ensures termination of this process after a finite number of iterations.

(d) The obtained tree is again traversed bottom-up, and so on. This way it is guaranteed that each time a term of the form *IF (V(x,y,z) = t) then u else v* is processed, there is no occurrence of *V* in tests of terms *u* and *v*.

(e) In the obtained term, simplifications according to rule 2 are performed, finally leading to a regular term, as required. Q.E.D.

5 More Transformations: Relying on Relations Between Variables

Based on Theorem 3, only regular terms are considered in the sequel.

Recall [6] that to compute the value of term τ under interpretation *I* (i.e. under some assignment of values to all variables), a so called computing sub-tree Q_I of the term's tree *Q* is considered. The root Q_I coincides with the root of *Q*; if value of a test already included in Q_I is equal to *true, false* or ω, then its *true*-descendant, *false*-descendant, or both descendants are also included in Q_I. If all terminals in Q_I get the same value then τ gets this value; otherwise its value under *I* is ω.

Note that every node *p* in *Q* is reachable from its root under some interpretation *I*, i.e. is included in Q_I for some *I* (at least when all variables have value ω, i.e. are undefined). Moreover, if node *p* contains a test $x = y$ then it is possible to determine

those values that this test can get when p is reached. This clearly depends on the tests traversed in the path from Q's root to p, and on descendants of these tests selected to continue the traverse. Without going into further details, we'd like only to note the following:

(a) Assume that the *true-(false-)* descendant of a test $x = y$ was selected in the path leading from this test to p. If p is reachable under interpretation I then the value of $x = y$ in p under I may equal *true (false)* or ω, but not *false (true)*.

(b) Assume that the *true*-descendant of a test $x = y$ was selected in the path leading from this test to p, and then later in this path the *false*-descendant of another occurrence of $x = y$ was selected. If p is reachable under interpretation I then the value of $x = y$ in p under I equals ω.

(c) Generalizing the above, assume that selection of descendant for some test s contradicts to the selection already made earlier in this path for other tests $s_1,...,s_k$ (for example, *true*-descendants of $x = y$ and $y = z$ were selected, but *false*-descendant was selected for $x = z$). If p is reachable under interpretation I then the value of at least one of the tests $s_1,...,s_k$ in p under I equals ω.

Let's now add more transformation rules, denoting by t_p the sub-term of t that corresponds to the sub-tree of t's tree with root p).

(16) Let t_p be of the form $IF\ x = y\ then\ z\ else\ \omega$, and value of $x = y$ in p is not equal to *true*. Then $t_p \leftrightarrow \omega$.

(16') Let t_p be of the form $IF\ x = y\ then\ \omega\ else\ z$, and value of $x = y$ in p is not equal to *false*. Then $t_p \leftrightarrow \omega$.

(17) Let t_p be of the form $IF\ x = y\ then\ u\ else\ w$, and value of $x = y$ in p is equal to ω. Then $t_p \leftrightarrow IF\ \omega\ then\ u\ else\ w$.

(18) Let p be such terminal of term t that contains sub-term *V(x,y,z)*, and value of $x = y$ in p is not equal to *true*. Then the following holds in p:

$$V(x,y,z) \leftrightarrow IF\ x = z\ then\ z\ else\ (IF\ y = z\ then\ z\ else\ \omega)$$

Note that application of rule 18 may lead to a non-tree and even non-regular term (for example if terminal p contains term *V(V(x,y,z),u,w)*) . However, in such cases regularization can again be performed, according to Theorem 3.

6 Reducing the Strength of Used Operations

As was shown in [6], function V can not be expressed as composition of conditional *IF* and the equality predicate = , so that *T(IF, =)* is a proper subset of *T(IF, V, =)*. In other words, in general case it is impossible to eliminate occurrences of V from terms in *T(IF, V, =)*. The following Theorem 4, along with Theorem 3, establishes an important class of terms in which such elimination is possible.

Theorem 4. Suppose that V appears only in tests of term $\tau \in TreeT(IF, V, =)$. Then τ can be transformed into an equivalent regular term with no occurrences of V in it.

Proof. Application (as described above) of rules 15, 13 and 14 to a term of the form $IF\ (V(x,y,z) = t)\ then\ u\ else\ w$ leads to a regular term in which all terminals are of the form *V(u,u,w)* or *V(w,w,u)* – up to the order of the arguments. But then rule 9 (taking into account 10 and 10') allows to eliminate such occurrences of V. Hence if there were no V's in terms u and w then the obtained regular term also doesn't contain any occurrence of V. Q.E.D.

In some cases, even though full elimination of V can not be achieved, it is possible at least to reduce the number of occurrences of V . For example, this can be done when rule 18 is applicable. Another example is shown below (for simplicity we skip all steps where rules 10 and 10' are applied):

$$V(V(x,y,z,),y,V(x,y,t)) \xleftrightarrow{11} V(t,y,V(x,y,V(x,y,z))) \xleftrightarrow{11} V(t,y,V(x,z,V(x,y,y))) \xleftrightarrow{9} V(V(x,y,z),y,t)$$

7 Conclusion

Invariant parallel functions can be naturally used in various types of program schemes (e.g. in recursive schemes) to enrich their control means. As shown in author's earlier publications, use of such functions in sequential recursive schemes leads to a rich hierarchy of extended schemes. This paper studies equivalent transformations associated with invariant parallel functions. A rich catalog of transformations is developed for terms over a full basis in the class of invariant functions. It is then shown how they can be applied for a variety of goals, such as regularization of terms and reducing the strength of the used operations. The offered transformations can be further used in program schemes for optimization, simplification, check of equivalence between different implementations, etc.

References:

[1] M. Paterson M. and C.Hewitt, Comparative schematology. *Record of Project MAC Conference on Concurrent Systems and Parallel Computation*, ACM, 1970, pp. 119-128

[2] D. Scott, *Outline of a mathematical theory of computation.* Technical Monograph PRG-2, Oxford University, Oxford, 1970

[3] A.Stoughton, Interdefinability of parallel operations in PCF. *Theoretical Computer Science*, Vol. 79, No.2, 1991, pp.357-358.

[4] M.Escardo, M.Hofmann, T.Streicher, On the non-sequential nature of the interval-domain model of exact real-number computation, *Workshop on Domains V*, Darmstadt, September 1999

[5] M.Trakhtenbrot, On representation of sequential and parallel functions. *Proceedings of Fourth Symposium on Mathematical Foundations of Computer Science*, LNCS Vol.32, 1975, pp. 411-417.

[6] M. Trakhtenbrot, Relationships between classes of monotonic functions, *Theoretical Computer Science*, Vol.2, No.2, 1976, pp.225-247

[7] M. Trakhtenbrot, Parallel functions in recursive program schemes, *WSEAS Transactions on Mathematics*, Vol.2, No.2, 2003, pp.151-155.