A LINEAR SPACE TRANSLATION OF FUNCTIONAL PROGRAMS TO TURNER COMBINATORS *

F. Warren BURTON

School of Computing Studies and Accountancy, University of East Anglia, Norwich, NR4 7TJ, England

Received 24 March 1982; revised version received 13 May 1982

Keywords: Combinators, functional programming, lambda calculus

1. Introduction

A functional program is a sugared λ-calculus expression without free variables. If we include the three functions

$\mathbf{S} = \lambda f \cdot \lambda g \cdot \lambda x \cdot (fx)(gx)$,

$\mathbf{K} = \lambda x \cdot \lambda y \cdot x$,

$\mathbf{I} = \lambda x \cdot x$

as atom, or primitive constant, then any λ-expression may be translated to an equivalent one which is free from variables and λ's. The translation consists of repeatedly applying the transformations

$\lambda v \cdot EF \to \mathbf{S}(\lambda v \cdot E)(\lambda v \cdot F)$,

$\lambda v \cdot w \to \mathbf{K}w \quad \text{if } v \neq w$,

$\lambda v \cdot v \to \mathbf{I}$

where v and w are arbitrary variable names and E and F are arbitrary expressions. Innermost expressions always are transformed first. Special cases of **S** (called **B** and **C**) may be used when E and F do not both reference the variable v.

This transformation leads to a combinatorial growth in the size of the expression. Turner has used the above combinators together with some others [3, 4] to reduce the combinatorial growth. However, Kennaway [2] has shown that a λ-expression of length n, measured in total number of occurrences of variables and constants, may still grow to length $\theta(n^2)$.

Turner has used a combinatory representation of λ-expression as an object code for the functional language SASL. With the elimination of the λ's, beta reduction is avoided except for the simple cases of specific combinators. The combinators used by Turner (as opposed to those used by Abdali [1], for example) selectively ship to each subexpression exactly those values required in the evaluation of the subexpression. We are interested in a distributed evaluation of function programs, and so find Turner combinators particularly attractive for ensuring that each expression receives exactly that part of the environment required for its evaluation. Unfortunately, we do not regard object code of length $\theta(n^2)$ as acceptable.

We must make one important restriction on the class of functional programs we consider. No function may refer to a global variable. (This seems reasonable in the context of a language for the distributed evaluation of function programs, since a global variable reference is likely to result in a hidden communication cost.) Any functional program may be transformed to one where no function refers to a global variable by adding extra parameters. In reasonable cases this will usually not greatly increase the size of a program. In the worst case a program of length n will grow to length $\theta(n^2)$ with this transformation.

In order to get linear growth, we take the

* This work was supported by a grant from the United Kingdom Science and Engineering Research Council.

0020-0190/82/0000-0000/$02.75

lengths of identifiers into account when measuring the length of a program. If a program has more than 26 variables in any one function, they cannot all be identified by a single letter identifier. The length of the longest identifier must grow at least logarithmically with the number of identifiers. Even with these restrictions, Kennaway's $\theta(n^2)$ result holds, reduced to $\theta((n/\log n)^2)$ when the identifier lengths are taken into account. This worst case growth is caused by unbalanced expression trees. By partly balancing the expression tree before applying Turner's abstraction algorithm, we get only linear growth. This holds even when an arbitrary mix of variables and constants is allowed (so identifiers need not be of $\theta(\log n)$ length on average), when short identifiers are used more often than long identifiers, etc. An outline of the proof of this result is given.

We will first consider the method for improving the balance of a λ-expression.

2. Definitions and notations

Let the vector $\overline{X} = (x_1, x_2, \ldots, x_n)$. We will use $\lambda\overline{X}\cdot$ to mean $\lambda x_1 \cdot \lambda x_2 \cdots \lambda x_n \cdot$, and $A\overline{X}$ to mean $(\cdots((Ax_1)x_2)\cdots x_n)$.

If A is an expression, B is a subexpression of A, and v is any variable occurring in A, then $[v/B]A$ is the expression produced by replacing B by v in A.

We define a *simple function* to be a λ-expression of the form $\lambda\overline{X}\cdot A$ where A contains no λ's and no variables not in $\overline{X}$. We will call A the body of $\lambda\overline{X}\cdot A$ and think of A as the usual expression tree. We recursively define an *autonomous function* to be any finite λ-expression which may be produced from a simple function by replacing zero or more leaves of the body with autonomous functions. Autonomous functions are exactly the class of λ-expressions in which we are interested (i.e., functional programs where no function contains a global variable reference).

3. Balancing simple function bodies

We will regard the body of a simple function as a tree. Internal nodes will be applications and leaves will be constants and variables. Additional variables and λ's will be introduced when balancing a tree, so the final result will no longer be a simple function body. After balancing, in the worst case, a subtree may contain up to four times as many nodes as its brother. (In practice, a better balance normally can be expected.) For brevity we will call such a partially balanced tree a balanced tree.

Let B be a λ-expression containing no λ's. We define *select*(B) to be a subtree of B, consisting of a subtree root together with all of its descendants, having as close to $\frac{1}{2}(n+1)$ leaves as possible, where n is the number of leaves in B. Such a subtree can be found in O(n) time. We first store the number of leaves contained in each subtree with the subtree root. We then start at the root of the tree and repeatedly consider the larger subtree, while keeping track of the size of the largest smaller subtree found so far, until we find that both subtrees have fewer than $\frac{1}{2}(n+1)$ nodes. The subtree *select*(B) will contain not more than $\frac{2}{3}$ of the total number of leaves in B, and strictly more than $\frac{1}{3}$ of the leaves. To see that such a subtree exists, we need only consider the two immediate subtrees of the smallest subtree containing strictly more than $\frac{2}{3}$ of the leaves. At least one of these subtrees will satisfy the above condition.

We can now define *balance*(B), the function which returns a balanced expression equivalent to B, as follows:

If B contains three or fewer nodes
  then return B unchanged
else if neither immediate subtree of B contains more than two-thirds of the leaves **then return** B with both immediate subtrees balanced
else return $(\lambda v \cdot \text{balance}([v/E]B))\ \text{balance}(E)$
  where v is any variable not containing in B and E = select(B).

Example 3.1. $(a_1(a_2(a_3\,a_4)))$ contains one leaf, a_1, in its left subtree, and three leaves in its right subtree. Hence we must rebalance with $E = (a_3\,a_4)$. The result of balancing is

$$((\lambda v \cdot (a_1(a_2\,v)))(a_3\,a_4)).$$

Example 3.2. balance(((((a$_1$ a$_2$)a$_3$)a$_4$)a$_5$)a$_6$) will result in two balance operations. The final result will be

$$(\lambda v \cdot (\lambda w \cdot ((w\, a_5)a_6))(v\, a_4))((a_1\, a_2)a_3).$$

In practice, there is little or no benefit in balancing such small trees, but big trees make awkward examples.

4. Balancing autonomous functions

Since an autonomous function shares no variables with any internal function, the function and any internal function do not interact when being translated to Turner combinatory form. Hence we can treat any internal function as a simple leaf when balancing an autonomous function.

Internal functions are balanced independently.

5. Overview of analysis

Clearly we need only analyse the growth associated with translating a simple functon. We will outline a proof of linear growth stating theorems without proof. (In most cases a simple induction proof is possible.)

We will use a result in [2] which says that each combinator may be associated with a particular variable, and that the total number of combinators associated with a variable is the number of nodes in the smallest connected subgraph of the original expression tree which contains both the root and every occurrence of the variable.

To show linear growth, different combinators will be associated with (or charged to) different parts of the expression. We will charge some of the combinators to specific occurrences of variables, and the rest to general overhead. The number of combinators charged to each occurrence of a variable will be proportional to the length of the identifier for the variable. The total number of combinators charged to general overhead will be at most proportional to the number of leaves in the expression tree. The linear growth follows.

Theorem 5.1. *Balancing a tree with* m *nodes produces a tree with no more than* $2m - 1$ *nodes.*

Theorem 5.2. *A tree of depth* d *must contain at least* 1.25^d *leaves.*

Assume a program has k variables. Let S be the size of the symbol set for identifiers and let p_i be the length of the identifier for variable v_i. Since there can be at most S^{p_i} identifiers of length p_i, we get the following result.

Theorem 5.3

$$\sum_{i=1}^{k} \frac{1}{(2S)^{p_i}} < 1.$$

We will associate a number of special nodes, called *service points*, with each variable. All combinators associated with nodes at or above a service point are charged to general overhead. We require that each leaf be within distance $d_i = \lceil c\, p_i \rceil$ of a service point for v_i, for $1 \leqslant i \leqslant k$, with $c = \log 2S / \log 1.25$. Clearly we need at most d_i combinators to tie any leaf to the nearest service point for v_i. Since each occurrence of v_i increases the length of the program by p_i, these combinators, when required, may be charged to the specific occurrence of v_i. All that remains is to show that, in a tree with m leaves, O(m) combinators are sufficient to tie all service points for all variables to the root. This follows from the following lemma and theorems.

Lemma 5.4. *A subtree containing fewer than* $1.25^{d_i} \geqslant (2S)^{p_i}$ *leaves needs no service points for* v_i.

Theorem 5.5. *A tree with* m *leaves needs at most* $2m/(2S)^{p_i} - 1$ *combinators to support the service points for* v_i.

Theorem 5.6. *Fewer than* 2m *combinators are required to support all service points for all variables.*

Acknowledgement

The author would like to acknowledge helpful suggestions from M.S. Joy, J.R. Kennaway, M.R. Sleep and M.M. Huntbach.

References

[1] S.K. Abdali, An abstraction algorithm for combinatory logic, J. Symbolic Logic 41 (1976) 222–224.

[2] J.R. Kennaway, The complexity of a translation of λ-calculus to combinators, to appear.

[3] D.A. Turner, A new implementation technique for applicative languages, Software Practice and Experience 9 (1979) 31–49.

[4] D.A. Turner, Another algorithm for bracket abstraction, J. Symbolic Logic 44 (1978) 67–70.