

# A Transformation Method for Dynamic-Sized Tabulation

Wei-Ngan Chin<sup>1</sup> and Masami Hagiya<sup>2</sup>

<sup>1</sup> Dept of Information Systems & Computer Science, National University of Singapore, Kent Ridge, Singapore 0511. Email: chinwn@iscs.nus.sg.

<sup>2</sup> Dept of Information Science, University of Tokyo, Hongo, Bunkyo-ku, Tokyo 113, Japan. Email: hagiya@is.s.u-tokyo.ac.jp.

Received November, 1993 / April, 1994

**Abstract.** Tupling is a transformation tactic to obtain new functions, without redundant calls and/or multiple traversals of common inputs. It achieves this feat by allowing each set (tuple) of function calls to be computed recursively from its previous set. In previous works by Chin and Khoo [8, 9], a safe (terminating) fold/unfold transformation algorithm was developed for some classes of functions which are guaranteed to be successfully tupled.

However, these classes of functions currently use *static-sized* tables for eliminating the redundant calls. As shown by Richard Bird in [3], there are also other classes of programs whose redundant calls could only be eliminated by using *dynamic-sized* tabulation. This paper proposes a new solution to dynamic-sized tabulation by an extension to the tupling tactic. Our extension uses *lambda abstractions* which can be viewed as either dynamic-sized tables or applications of the higher-order generalisation technique to facilitate tupling. Significant speedups could be obtained after the transformed programs were vectorised, as confirmed by experiment.

## 1. Introduction

In [8, 9], we proposed a safe automated tupling method to help transform functions with redundant calls and/or multiple traversals to equivalent functions without them. This method applies to both functions with a single recursion parameter per function [8], as well as those with multiple recursion parameters per function [9]. A recursion parameter of a function is a parameter which is *strictly increasing* or *strictly decreasing* down the recursion.

A classic example of a function with a single recursion parameter is the fibonacci function shown next. (Warning: for brevity, we have frequently used the word “function” to denote a syntactic object, synonymous to “function definition”, as opposed to the usual mathematical (extensional) sense of the word.)

```

fib :: Int → Int;
fib 0      = 1;
fib 1      = 1;
fib (n+2)  = (fib(n+1))+(fib n);

```

The above program is given in **Haskell**, a non-strict purely functional language[15]. It contains redundant *fib* calls which could be transformed with the help of a new tuple function, shown below.

```

fib_tup :: Int → (Int,Int);
fib_tup n  = ((fib(n+1)),(fib n));

```

Applying the safe tupling method yields the following efficient transformed program with time-complexity  $\mathcal{O}(n)$ .

```

fib 0      = 1;
fib 1      = 1;
fib (n+2)  = u+v where (u,v) = (fib_tup n);
fib_tup 0  = (1,1);
fib_tup (n+1) = (u+v,u) where (u,v)=(fib_tup n);

```

Similarly, functions with multiple recursion parameters such as *zip* (shown below) could also be optimised by the safe tupling method as shown in [9].

```

data List A = Nil | Cons A (List A);
zip :: ((List A),(List B)) → (List (A,B));
dupl :: (List A) → (List (A,A));
dupl xs                = zip(xs,xs)
zip (Cons x xs, Cons y ys) = Cons (x,y) (zip (xs,ys))
zip (xs,ys)            = Nil;

```

In the case of the above program, the tupling transformation could eliminate multiple traversals of the variable *xs* by transforming *dupl* to:

```

dupl (Cons x xs) = Cons (x,x) (dupl xs)
dupl xs         = Nil;

```

The problem of finding eureka tuples (cf. static-sized tables) that allow each set of function calls to be computed from its previous set down the recursion has now been solved for several classes of functions. However, there exist other classes of functions whose redundant calls could only be eliminated by a dynamic-sized tabulation technique. An example of such a program is the following recursive version of the function to compute the binomial coefficient  $n!/(k!(n-k)!)$  where  $0 \leq k \leq n$ . (Note: This recursive version of binomial function does not overflow as easily as the usual definition  $n!/(k!(n-k)!)$ .)

```

bin (0,k)      = 1
bin (n+1,k)    = if k≤0 or k≥(n+1) then 1 else (bin(n,k-1)) + (bin(n,k))

```

The call dependency graph (DG) of this function is given in Fig. 1. From the DG, it is quite clear that dynamic-sized tables would be needed to tabulate (eliminate the redundant calls of) the function. This is because each row of function calls could only be computed without redundancy from a slightly larger row of calls down the recursion. As far as the authors are aware, it was previously thought that tabulation techniques, other than tupling, will be required to eliminate the redundant calls [3, 10].

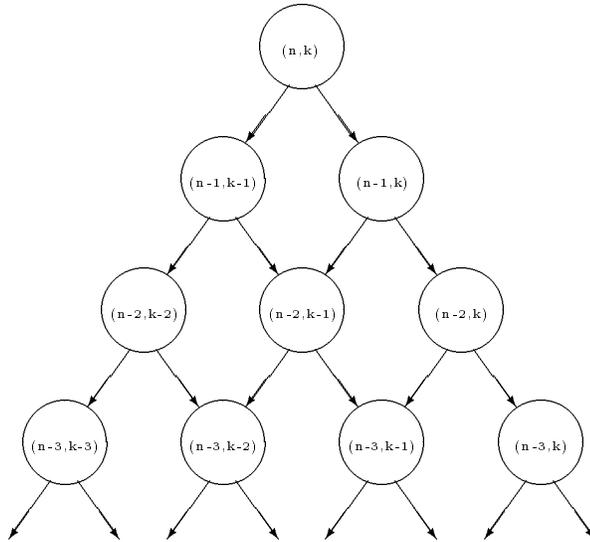


Fig. 1. Top Portion of DG for  $\text{bin}(n, k)$  (only arguments shown)

This paper proposes a new method for dynamic-sized tabulation. It is based on an extension of the tupling tactic which uses lambda abstractions to represent dynamic-sized tables. The use of lambda abstraction in tupling transformation is not new. Pettorossi has used it to eliminate multiple traversals of data structures in [22, 21]. Our contribution is to show that some of these lambda abstractions could be converted to vectors and be used to remove redundant calls via dynamic-sized tables. A key result is that such a dynamic-sized tabulation method could be done automatically via the safe tupling tactic.

An overview of this paper follows. In Section 2, we introduce the safe tupling tactic and present a class of functions, called the **T0** class. This class of functions has a single recursion parameter per function and is guaranteed to be safely tupled. Section 3 outlines the overall dynamic-sized tabulation method which is made up of a set of smaller transformation tactics described in detail in subsequent sections. Section 4 looks at how lambda abstraction could be used by safe tupling for functions which require dynamic-sized tabulation. The introduction of suitable lambda abstractions is guided by the need to conform to the **T0** grammar form. Section 5 outlines a technique to convert suitable lambda abstractions into dynamic-sized tables (or vectors). Section 6 introduces local recursion as another mechanism (in addition to lambda abstraction) to facilitate the safe tupling transformation. Section 7 shows how the tail-recursion transformation could be used to obtain memory-efficient tabulation. Section 8 reports on the potential improvements which could be achieved by the tabulated programs. Related works of Bird, Takeichi and Pettorossi are discussed in Section 9, before

a conclusion in Section 10. Appendix A briefly outlines a bound determination analysis that is needed by our vector conversion technique.

## 2. Tupling Tactic and the T0 Class

To find the eureka tuples needed to eliminate redundant calls and/or multiple traversals, we have formulated a tupling transformation which searches through a *tree-of-cuts* for matching tuples. The term *cut* was introduced by Pettorossi in [20] and is treated synonymously to *tuples* in this paper. The original use of *cut* is to denote a set of calls in the call dependency graph of a function which when removed will divide the dependency graph into two disjoint halves.

Our tupling tactic is structured to obtain such cuts as tuples during transformation. The tactic uses the basic fold/unfold transformation rules of Burstall and Darlington [6]. Its transformation algorithm repeats the process of *instantiating*, *unfolding* and *splitting* to each cut until it obtains a tree-of-cuts for which each branch is found to match (*fold*) with an earlier cut up the tree. The tupling transformation algorithm is given below in Definition 1. The search of the transformation branches out whenever there are (i) different instantiations to the recursion parameters or (ii) distinct recursion arguments among the calls in the cut.

**Definition 1 (Tupling Transformation Algorithm).** *An informal description of the transformation procedure is as follows:*

1. Start with an initial function call.
2. From the current tuple, get the next tuple(s) by different minimal **instantiations** (which permit one or more unfolds) to the recursion parameter.
3. Perform **unfolding** (without instantiations) and eliminate duplicate calls.
4. **Split** the function calls into separate tuples according to the different recursion variables. For each tuple, perform:
  - a) Attempt a fold match with the previously constructed tuples up the tree.
  - b) IF successful, we have found an eureka tuple and could terminate for this branch;  
*OTHERWISE* repeat from (2) with this tuple as the current tuple.

To illustrate this tupling transformation algorithm, consider the following set of mutually recursive functions which each has a single recursion parameter (of possibly different types) per function.

```

data Weird A = Empty A | Pair (List(Weird A)) (Tree(Weird A));
data Tree A = Leaf A | Node (Tree A) (Tree A)
f (Empty a)           = ...;
f (Pair ws n)        = ...[(g ws),(h ws),(t n),(u n)]...;
g Nil                = ...;
g (Cons w ws)        = ...[(f w),(g ws)]...;
h Nil                = ...;
h (Cons w ws)        = ...[(f w),(h ws)]...;
t (Leaf w)           = ...[(f w)]...;
t (Node l r)         = ...[(t l),(u r)]...;

```

$$\begin{aligned} u \text{ (Leaf } w) &= \dots[(f w)]\dots; \\ u \text{ (Node } l r) &= \dots[(t l), (t r)]\dots; \end{aligned}$$

When the tupling transformation algorithm is applied to the function call  $(f a)$ , it results in a tree-of-cuts shown in Fig. 2.

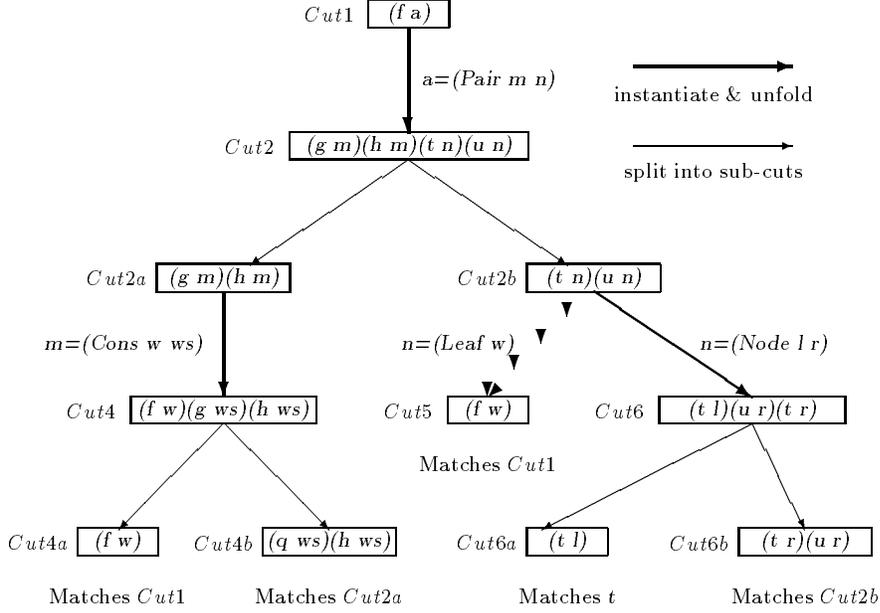


Fig. 2. A Tree-of-Cuts for  $(f a)$

Starting with  $(f a)$  as  $Cut1$ , we unfold it to obtain four subsidiary calls  $((g m), (h m), (t n), (u n))$  as  $Cut2$ . There are two different recursion arguments in  $Cut2$  which would be split up into two sub-cuts,  $Cut2a ((g m), (h m))$  and  $Cut2b ((t n), (u n))$ . In the case of  $Cut2b$ , two different recursive instantiations (followed by unfolding) are possible which result in  $Cut5$  and  $Cut6$ . We terminate the branch at  $Cut5$  because it matches with  $Cut1$ , and repeat the safe transformation process of *instantiating, unfolding and splitting* for the other branches until they match with some cuts higher up in the tree. Eureka tuples found are  $((g m), (h m))$  and  $((t n), (u n))$ , while  $(f w)$  and  $(t l)$  represent cuts of a single call each. Single function calls are allowed to match with their original functions rather than be subjected to further transformation which will simply re-define<sup>1</sup> these functions.

These matching tuples enable the tupling transformation to obtain the following transformed program without redundant calls.

$$\begin{aligned} f \text{ (Empty } a) &= \dots; \\ f \text{ (Pair } ws n) &= \dots[G, H, T, U]\dots \text{ where} \end{aligned}$$

<sup>1</sup> Provide a similar function definition via a different name.

$$\begin{aligned}
& \{ (G,H)=(gh\_tup\ ws); (T,U)=(tu\_tup\ n) \}; \\
gh\_tup\ Nil &= (\dots, \dots); \\
gh\_tup\ (Cons\ w\ ws) &= (\dots[F,G]\dots, \dots[F,H]\dots) \mathbf{where}\ (G,H)=(gh\_tup\ ws), F=(f\ w); \\
tu\_tup\ (Leaf\ w) &= (\dots[F]\dots, \dots[F]\dots) \mathbf{where}\ F=(f\ w); \\
tu\_tup\ (Node\ l\ r) &= (\dots[Tr,Ur]\dots, \dots[Tr,Ur]\dots) \mathbf{where} \\
& \{ Tr=(t\ l); (Tr,Ur)=(tu\_tup\ r) \}; \\
t\ (Leaf\ w) &= \dots[(f\ w)]\dots; \\
t\ (Node\ l\ r) &= \dots[(t\ l),(u\ r)]\dots; \\
u\ (Leaf\ w) &= \dots[(f\ w)]\dots; \\
u\ (Node\ l\ r) &= \dots[(t\ l),(t\ r)]\dots;
\end{aligned}$$

In general, the above tupling transformation may fail to terminate for some programs. However, we have managed to identify several classes of (mutually and auxiliary) recursive functions for which it is guaranteed to terminate. These classes of functions include those with a single recursion parameter per function, as well as those with multiple recursion parameters per function. The extension to be proposed in this paper for handling dynamic-sized tabulation is applicable to functions with single as well as multiple recursion parameters. However, for simplicity, we shall only consider the class of functions with a single recursion parameter per function. This class of functions, called the **T0** class, can be formally defined as follows:

**Definition 2 (The T0 Class).** *A set of mutually recursive functions,  $f_1, \dots, f_h$ , satisfies the T0 class if the following conditions are fulfilled.*

1. Each equation (the  $s$ -th one) of function  $f_i$  has the form:

$$f_i(p, v_1, \dots, v_n) = \dots[C_1, \dots, C_k] \dots$$

where  $\dots[\dots]$  represents a context with zero or more holes to place the (mutually) recursive calls,  $C_1, \dots, C_k$ . The pattern of the first parameter must be of the form  $p \equiv v' \mid c_s(v'_1, \dots, v'_m)$ . (Note: symbol  $\equiv$  denotes syntactic equivalence.)

2. Each of the recursive calls,  $C_j$ , is of the form:  $f_{i_j}(w', w_1, \dots, w_a)$ . This call must satisfy the following:
  - a)  $\forall x \in 1..a. w_x \in \{v_1, \dots, v_n\}$  or a constant.
  - b)  $p \equiv c_s(v'_1, \dots, v'_m) \rightarrow (w' \equiv v'_{j_1}) \wedge (j_1 \in \{1..m\})$ .
  - c)  $p \equiv v' \rightarrow (w' \equiv v') \wedge (f_i >_{fn} f_{i_j})$
3. The function name ordering ( $>_{fn}$ ) introduced in 2(c) must not be cyclic.

Condition (1) states that the first parameter of each **T0** function is a recursion parameter whose pattern  $p$  has at most a single constructor (cf. simple pattern of [1]). Condition (2a) states that the other non-recursion parameters (namely  $v_1, \dots, v_n$ ) of the function are non-accumulating with either variables or constants for the recursive calls  $C_1, \dots, C_k$ . Note that no new variables are introduced. Condition (2b) ensures that the corresponding recursion argument  $w'$ , of each recursive call  $C_j$ , is taken from a variable of the recursion pattern  $p$ . Conditions (2c) & (3) ensure that each recursion argument will always decrease by at least one constructor when it cycles back to the same function. In other words, it should not be possible for a recursion argument to remain unchanged after going through a sequence (cycle) of calls back to the same function.

Some examples of invalid **T0** functions are given below. The various places where the violations against the above grammar form occur are shown underlined.

$$\begin{aligned} f(C1(C2(v)),x,y) &= \dots[f(v,x,y)]\dots; \\ f(C2(v),x,y) &= \dots[f(v,x,\underline{v}),f(\underline{x},x,y),f(\underline{C3},x,y),f(v,\underline{acc}(x),y)]\dots; \\ f(v,x,y) &= \dots[f(\underline{y},y,x)]\dots; \end{aligned}$$

Even the fibonacci function given earlier does not satisfy the **T0** form because of its use of the complex pattern  $n+2$  in the third equation. However, we could make use of the pattern-matching compilation technique of Augustsson [1] to obtain the following **T0** definition.

$$\begin{aligned} fib\ 0 &= 1; \\ fib\ (n+1) &= fib'\ (n); \\ fib'\ 0 &= 1; \\ fib'\ (n+1) &= (fib'\ n)+(fib\ n); \end{aligned}$$

This technique can be considered as a pre-processing step to obtain more **T0** functions for tupling transformations. A number of such pre-preprocessing techniques are highlighted in [8].

All mutually recursive functions and their auxiliary functions which fit the **T0** form are guaranteed to have eureka tuple(s). Our safe tupling algorithm is based on the tree-of-cuts tupling algorithm. This can be proved to be terminating for the **T0** class, as follows.

**Theorem 1 (Termination Property of the T0 Class).** *Given a set of mutually (and auxiliary) recursive **T0** functions, its transformation by the tree-of-cuts tupling algorithm will terminate in finite time.*

*Proof.* Given a set of **T0** functions, the tupling transformation will only have to deal with finitely many different tuples. Firstly, there is a finite number of function names. Secondly, each cut has only one recursion variable. Also, the number of different variables for the other parameters is finite, as no new variables are introduced by unfolding. Thus, if there are  $n$  different functions and the largest number of parameters for the functions is  $m$  and the number of different variables and constants used is  $s$ , then the maximum number of distinct function calls which could be obtained is  $n \times s^m$ . As the number of distinct calls is bounded, the number of different tuples (modulo variable renaming) encountered will also be bounded (at most  $2^{n \times s^m}$ ). Hence, a re-occurring (matching) tuple is bound to happen after a finite number of steps by the tupling algorithm.

### 3. Overall Method

This section gives an outline of the dynamic-sized tabulation method. The basic method is made up of four smaller transformation tactics which are applied in the order listed below.

- Lambda Abstraction Tactic
- Tupling Tactic
- Vector Conversion Tactic
- Tail-Recursion Tactic

The lambda abstraction transformation tactic is used to convert certain non-**T0** functions with accumulating parameters to **T0** functions. The accumulating parameters are removed by applying higher-order generalisation to these parameters (changing them to bound variables of lambda abstractions). This step is similar to the well-known currying technique and is described in detail in Section 4. Its main use is to obtain **T0** class functions which could hopefully be transformed by the tupling tactic.

The tupling tactic has already been outlined earlier in Section 2.

The vector conversion tactic (described in Section 5) is used to convert lambda abstractions that satisfy some pre-conditions into vectors. This step is needed for effective sharing of results from function-type calls (lambda abstractions). Without explicit conversion of lambda abstractions to vectors, we need a run-time technique, called memoisation<sup>2</sup> [19], to reuse identical calls to the same lambda abstraction. Vector conversion can be viewed as a compiled (more efficient) version of the memoisation technique.

The last tactic which might be applied is tail-recursion transformation. This tactic (described in Section 7) could be used to obtain an iterative version of our vectorised program. With the iterative form, more savings on both time and storage utilisation are possible.

An outline of the above four transformation tactics is illustrated in Fig. 3 using the binomial function as an example. In the example, we have used lists (formed by list comprehension notation) to represent vectors, and the index operator (!), to select an element from a given list, as follows  $[x_0, x_1, \dots, x_n]!!i = x_i$ .

Apart from the four transformation tactics, another tactic which is often useful is the *circular variable* introduction tactic. This tactic could be used to handle a class of functions with cycles among the dependency graphs of their recursive calls. Such cycles among recursive calls of DGs violate the **T0** definition but fortunately could be eliminated with the help of circular variables. This tactic is described in Section 6 and must be performed prior to the tupling tactic.

#### 4. Lambda Abstraction Transformation

The binomial function given in Section 1 and reproduced below is a function with two parameters.

$$\begin{aligned} \text{bin}(0,k) &= 1 \\ \text{bin}(n+1,k) &= \text{if } k \leq 0 \text{ or } k \geq (n+1) \text{ then } 1 \text{ else } (\text{bin}(n,k-1)) + (\text{bin}(n,k)) \end{aligned}$$

When it is compared with the **T0** grammar form, we note that the first parameter is a valid recursion parameter (assuming a peano-style integer where  $+1$  is viewed as a constructor). However, the second parameter is neither a suitable recursion parameter nor is it non-accumulating. This parameter is currently the reason why the binomial function does not belong to the **T0** class. Often, such problematic accumulating parameters are handled by the parameter generalisation technique (e.g. as done in the safe fusion tactic [7]) which would replace

---

<sup>2</sup> Memoisation is a run-time technique for avoiding the recomputation of redundant calls. It does this by storing pairs of argument/result values of each evaluated function call into a table, whose result could then be retrieved from the table (rather than recomputed) whenever the same function call is re-encountered.

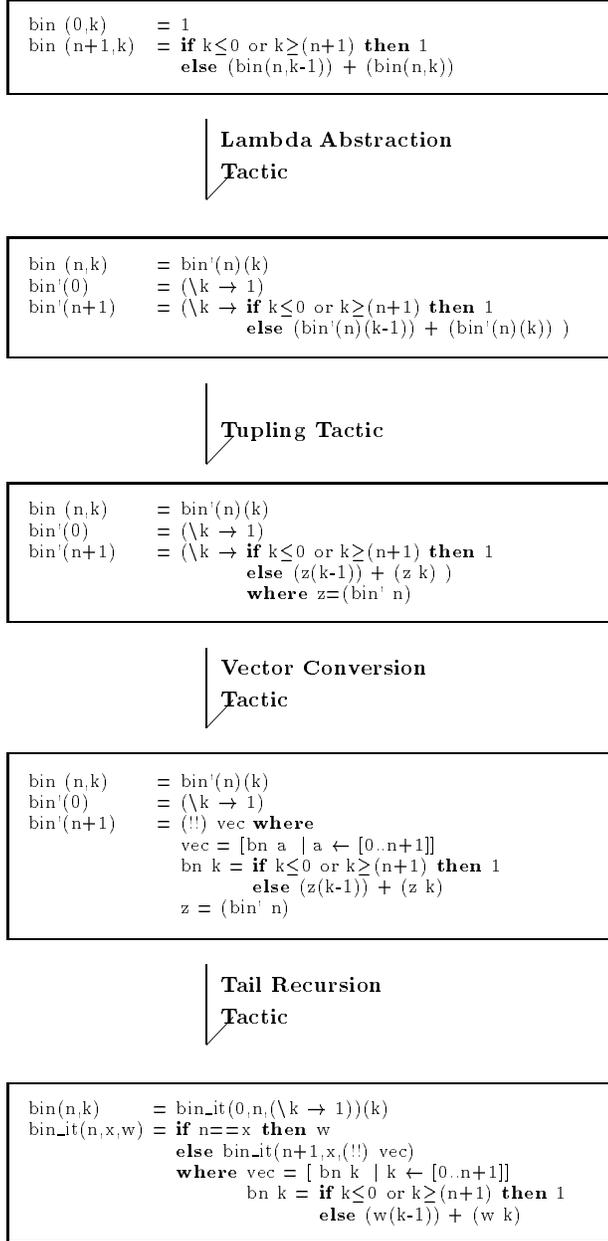


Fig. 3. Four Tactics of Dynamic-Sized Tabulation

each such argument by a new parameter variable. However, this simple technique does not work for the tupling transformation when redundant calls have to be eliminated. The reason is that infinitely many tuples with an increasing number of calls still arise after generalisation. The first-order generalisation technique could limit the size of function calls encountered, but not the number of different calls which may arise during tupling.

The more powerful generalisation technique that is needed is the higher-order generalisation technique of Pettorossi and Skowron [22] which would introduce the following lambda abstraction function.

$$bin'(n) = (\lambda k \rightarrow bin(n,k))$$

This new function represents a curried equivalent of  $bin$  which could be transformed by a sequence of unfold/fold steps (corresponding to the currying technique) to obtain:

$$\begin{aligned} bin'(0) &= (\lambda k \rightarrow 1) \\ bin'(n+1) &= (\lambda k \rightarrow \text{if } k \leq 0 \text{ or } k \geq (n+1) \text{ then } 1 \\ &\quad \text{else } (bin'(n)(k-1)) + (bin'(n)(k)) ) \end{aligned}$$

With this application of the lambda abstraction transformation, the new  $bin'$  function now has only one parameter (arity of 1) and is a member of the **TO** class. Note that the second parameter is no longer a parameter of the **TO** function. Instead, it has become a bound variable of the lambda abstraction. Applying the tupling tactic to it would yield the following function.

$$\begin{aligned} bin'(0) &= (\lambda k \rightarrow 1) \\ bin'(n+1) &= (\lambda k \rightarrow \text{if } k \leq 0 \text{ or } k \geq (n+1) \text{ then } 1 \\ &\quad \text{else } (z(k-1)) + (z k) ) \text{ where } z = (bin' n) \end{aligned}$$

Notice that the eureka tuple used,  $(bin' n)$ , here consists of only a single call.

The lambda abstraction of  $bin'$  can be viewed as an application of higher-order generalisation. It has helped to facilitate the safe tupling tactic by permitting the calls to be shared. However, unlike ground-type function calls, the sharing of function-type calls (lambda abstractions) is only effective if a suitable tabulation technique such as memoisation is available. Unfortunately, the memoisation technique is not an efficient method for avoiding redundant calls because of the potential overheads associated with the searching and storage of large tables of calls.

We shall be advocating the selective use of lambda abstraction as a step towards more efficient tabulation of redundant calls. Our final target is to convert these lambda abstractions into dynamic-sized vectors containing the results of the needed function calls. Such a vector conversion technique is better than memoisation because it could provide efficient direct access to the tabulated calls. However, this conversion is not always feasible. We shall be outlining the criteria for successful conversion and the conversion technique itself later.

Another example which requires dynamic-sized tabulation to remove redundant calls is the knapsack problem. Given  $n$  items of positive weights and a knapsack which can be filled with selected items up to a total weight of  $w$ , maximise the value of the knapsack. The maximum value for this knapsack can be specified by  $knap(n,w)$  with the following definition for  $knap$  where (*weight*  $i$ ) and (*value*  $i$ ) specify the weight and value of the  $i$ -th item.

```

knap(0,w) = 0
knap(j+1,w) = if w < weight(j+1) then knap(j,w)
              else max (knap(j,w)) ((knap(j,w-weight(j+1)))+(value(j+1)));

```

If we compare this function to the **T0** grammar form, we could again tell that the second parameter does not conform to the non-accumulating form. To rectify this situation, we could introduce a new lambda function:

```

knap'(j) = (\w → knap(j,w))

```

which enables *knap* to be transformed to its curried equivalent.

```

knap'(0) = (\w → 0)
knap'(j+1) = (\w → if w < weight(j+1) then knap'(j)(w)
              else max (knap'(j)(w))
              ((knap'(j)(w-weight(j+1)))+(value(j+1)))) ;

```

The *knap'* function is now a **T0** class function. It could be safely tupled to the following.

```

knap'(0) = (\w → 0)
knap'(j+1) = (\w → if w < weight(j+1) then (z w)
              else max (z w) ((z (w-weight(j+1))) + (value(j+1))))
              where z=(knap'j) ;

```

Again the lambda abstraction used is only effective if it could be eventually converted to a vector of calls that are reused to avoid redundant computation. To explicitly convert lambda abstractions to vectors, we require a further transformation tactic which is outlined next.

## 5. Vector Conversion Tactic

When executed, the earlier functions using lambda abstractions are actually no more efficient than their original definitions. Theoretically, we would like calls with identical arguments to each lambda abstraction to be shared. Unfortunately, this requires the lambda abstractions to be memoised.

To obtain properly tabulated programs where repeated calls are re-called from tables (rather than re-computed), we shall propose another transformation tactic, called *vector conversion*, which explicitly replaces each lambda abstraction by an equivalent vector.

There are three pre-conditions for the vector conversion tactic. First of all, this tactic is only applicable to lambda abstractions whose parameters are based on either integer or enumerated types suitable for use as array indices. If the parameters are based on other kinds of data types, then some prior data type transformations would have to be applied to convert them into integer types. (This paper does not address such data type transformation techniques). Secondly, we need to determine the bounds for our lambda abstractions (vectors) so that efficient memory allocation and direct access to the repeated calls could be provided. Lastly, the parameter of the lambda abstraction must be strict if the semantics of lazy programs are to be preserved. (This is because vector indices are always strictly evaluated.)

An extended type-inference technique of Hagiya [13] could be used to help determine the lower and upper bounds of each lambda abstraction's parameter,

where possible. This technique allows linear constraints to be attached to integer types so that their bounds could be determined. An outline of such a bound analysis is given in Appendix A.

To make things simple, we shall use a list  $[x_0, x_1, \dots, x_n]$  to represent a one-dimensional vector with the range  $[0..n]$ . The  $i$ -th element of the list,  $x_s$ , could be accessed by  $x_s !! i$ . If the lower bound,  $l$ , is not 0, we simply use function  $conv(l)(i)=i-l$  to obtain indexes that start from 0. If the upper bound is unknown, it is possible to take advantage of lazy evaluation and infinite list (based on index  $[0..]$ ) to represent vectors with unknown upper bounds. If the lower bound is not known, we could reverse the index using  $rev(u)(i)=u-i$  where  $u$  represents the upper bound. However, at least one of the bounds must be known if list-based vectors are to be used. (If both bounds are unknown, it is still possible to use an association list of indexes and values to represent the vectors. However, this approach, similar to memoisation, requires searching and is rather inefficient. We shall not consider it here.) If real vectors are to be used instead of the list-based vectors, both the lower and upper bounds must be known.

With known bounds, each lambda abstraction could be converted into a vector of calls by three simple steps, namely:

1. Name the lambda abstraction.
2. Introduce a bounded vector of calls.
3. Replace the lambda abstraction by a corresponding lookup to the vector.

In the case of  $bin'$ , its lambda abstraction could be converted to a vector using bound  $0 \leq k \leq n$  for the second parameter, as follows.

```

bin'(n+1)  = (\k → if k ≤ 0 or k ≥ (n+1) then 1 else (z(k-1)) + (z k) )
              where z = (bin' n)
              ; name the lambda abstraction
            = bn where
              bn k = if k ≤ 0 or k ≥ (n+1) then 1 else (z(k-1)) + (z k)
              z = (bin' n)
              ; introduce a new vector of calls
            = bn where
              vec = [bn a | a ← [0..n+1]]
              bn k = if k ≤ 0 or k ≥ (n+1) then 1 else (z(k-1)) + (z k)
              z = (bin' n)
              ; substitute bn = (!) vec
            = (!) vec where
              vec = [bn a | a ← [0..n+1]]
              bn k = if k ≤ 0 or k ≥ (n+1) then 1 else (z(k-1)) + (z k)
              z = (bin' n)

```

The above transformation sequence manages to introduce a vector of calls for each of the recursive  $bin'$  function calls. Each of these vectors may be of different sizes and permits its values to be reused when computing the next vector of calls up the recursion. As a result, they are able to avoid redundant calls by providing a dynamic-sized tabulation mechanism.

A similar sequence of steps could also be provided for the knapsack function to obtain its dynamically-tabulated program. In this case, the  $knap'$  function will have the following bounds for the second parameter:  $0 \leq w \leq w0$ , where the upper bound  $w0$  is the value of the  $w$  parameter for the first recursive call. As

this upper bound is not presently included in the *knapsack*' function, we apply a transformation to supply it as an additional parameter, as follows.

**Define:**

$$\begin{aligned} \text{knapsack}(j, w) &= \text{knapsack}''(j, w) \ w \\ \text{knapsack}''(j, w0) &= \text{knapsack}'(j) \end{aligned}$$

**Transform *knapsack*'' to:**

$$\begin{aligned} \text{knapsack}''(0, w0) &= (\wedge w \rightarrow 0) \\ \text{knapsack}''(j+1, w0) &= (\wedge w \rightarrow \text{if } w < \text{weight}(j+1) \text{ then } (z \ w) \\ &\quad \text{else } \max(z \ w) \ ((z \ (w - \text{weight}(j+1))) + (\text{value}(j+1)))) \\ &\quad \text{where } z = (\text{knapsack}''(j, w0)) ; \end{aligned}$$

After adding in a new parameter to hold the upper bound, we could now proceed to obtain the vectorised version of *knapsack*'', shown below.

$$\begin{aligned} \text{knapsack}''(0, w0) &= (\wedge w \rightarrow 0) \\ \text{knapsack}''(j+1, w0) &= (!!) \ \text{vec} \ \text{where} \\ &\quad \text{vec} = [\text{knapsack}'' \ a \mid a \leftarrow [0..w0]] \\ &\quad \text{knapsack}'' \ w = \text{if } w < \text{weight}(j+1) \text{ then } (z \ w) \\ &\quad \quad \text{else } \max(z \ w) \ ((z \ (w - \text{weight}(j+1))) + (\text{value}(j+1))) \\ &\quad \quad z = (\text{knapsack}''(j, w0)) \end{aligned}$$

If both the upper and lower bounds are known, the above list-based functions could be converted to use real vectors where constant-access time to the elements are possible. Where feasible, this approach will result in more efficient tabulated programs.

Our vector conversion tactic uses a set of vectors which are returned by the various recursive calls. An alternative approach (as used in [26]) would be to regard the original recursive functions (e.g. *bin* and *knapsack*), as multi-dimensional vectors after the bounds have been determined. However, there are a number of reasons for not doing so. Firstly, our scheme is more memory efficient because the set of vectors are allowed to be of different sizes. Secondly, multi-dimensional vectors correspond to the large table technique (like full memoisation) where the whole table of calls has to be stored. In contrast, our approach need only use one or more previous vectors to compute the next vector. As a result, we may sometimes be able to apply further transformation to utilise small tables of calls. A suitable technique which facilitates this is tail-recursion transformation. Where applicable, a tail-recursion transformation could transform our recursive programs into an iterative loop where the dynamic-sized vectors are allowed to be updated in-place. This technique is described in Section 7. Lastly, this scheme allows the tupled functions to be transformed which the alternative scheme doesn't.

As an example of how the tupling tactic coexists with the lambda abstraction (and vector conversion) tactic, consider the following contrived program.

$$\begin{aligned} f(0, k) &= 1 \\ f(n+1, k) &= \text{if } k \leq 0 \text{ or } k \geq (n+1) \text{ then } 1 \ \text{else } f(n, k-1) + g(n, k) \\ g(0, k) &= 1 \\ g(n+1, k) &= \text{if } k \leq 0 \text{ or } k \geq (n+1) \text{ then } 1 \ \text{else } g(n, k-1) * f(n, k) \end{aligned}$$

To obtain a **T0** program, we must apply the lambda abstraction transformation which yields the following.

```

f'(0)      = (\k → 1)
f'(n+1)   = (\k → if k≤0 or k≥(n+1) then 1 else (f'(n)(k-1)) + (g'(n)(k)))
g'(0)      = (\k → 1)
g'(n+1)   = (\k → if k≤0 or k≥(n+1) then 1 else (g'(n)(k-1)) * (f'(n)(k)))

```

The redundant calls of the above program can be eliminated by tupling up the calls  $(f' n, g' n)$ . This transformation would define a new tupled function:

```
ftup' n    = (f' n, g' n)
```

which is later transformed to the following.

```

ftup' 0    = ((\k → 1), (\k → 1))
ftup'(n+1) = (fn, gn) where
fn = (\k → if k≤0 or k≥(n+1) then 1 else (u (k-1)) + (v (k)))
gn = (\k → if k≤0 or k≥(n+1) then 1 else (v (k-1)) * (u (k)))
(u,v) = ftup'(n)

```

Here, the transformed function returns a tuple of two lambda abstractions. Applying the vector conversion technique to both lambda abstractions would yield the following program.

```

ftup' 0    = ((\k → 1), (\k → 1))
ftup'(n+1) = ((!!) fvec, (!! ) gvec) where
fvec = [fn a | a ← [0..n+1]]
gvec = [gn a | a ← [0..n+1]]
fn k = if k≤0 or k≥(n+1) then 1 else (u (k-1)) + (v (k))
gn k = if k≤0 or k≥(n+1) then 1 else (v (k-1)) * (u (k))
(u,v) = ftup'(n)

```

Hence, tupled functions could also be transformed by our vector conversion tactic.

## 6. Circular Variable Tactic

So far, the examples of vectorised programs are those where one vector of calls is being computed solely from the next vector of calls down the recursion. However, there are also other programs where a vector needs to be computed in-situ from other elements of the *same* vector as well as from elements of the *next* vector down the recursion. To help allow that, we shall make use of circular vectors that are introduced by local recursion variables.

Consider the following function to find the longest common sub-sequence of two strings,  $XS$  and  $YS$ , where  $N=(length\ XS)$ ,  $M=(length\ YS)$ .

```

csub(j,k)  = if j>N or k>M then 0
            else if XS !! j == YS !! k then 1+(csub(j+1,k+1))
            else max (csub(j,k+1)) (csub(j+1,k));

```

Ignoring the recursive call  $csub(j,k+1)$  for the moment, the first parameter could be taken as the recursion parameter, while the second parameter has to be abstracted. Applying the lambda abstraction transformation, we obtain the following curried function.

```

csub'(j)   = (\k → if j>N or k>M then 0
            else if XS !! j == YS !! k then 1+(csub'(j+1)(k+1))
            else max (csub'(j)(k+1)) (csub'(j+1)(k)));

```

Even with the lambda abstraction transformation, the function  $csub'$  still does not belong to the **T0** class yet. This is because the recursive call,  $csub'(j)$ , has a parameter which is unchanged (across successive recursive calls) when it is supposed to be strictly increasing to qualify as a recursion parameter. As a result, it causes a cycle among its DG of recursive calls. However, this  $csub'(j)$  call is also a re-occurrence of its equation's LHS. It could be eliminated with the introduction of a local recursion variable (circular variable) as follows.

$$\begin{aligned}
 csub'(j) &= r \text{ where } r = (\backslash k \rightarrow \\
 &\quad \text{if } j > N \text{ or } k > M \text{ then } 0 \\
 &\quad \text{else if } XS !! j == YS !! k \text{ then } 1 + (csub'(j+1))(k+1) \\
 &\quad \text{else max } (r(k+1)) (csub'(j+1)(k));
 \end{aligned}$$

With this use of a local recursion variable, the  $csub'$  function is now a **T0** class function and could be safely tupled to obtain the following.

$$\begin{aligned}
 csub'(j) &= r \text{ where } r = (\backslash k \rightarrow \\
 &\quad \text{if } j > N \text{ or } k > M \text{ then } 0 \\
 &\quad \text{else if } XS !! j == YS !! k \text{ then } 1 + (z(k+1)) \\
 &\quad \text{else max } (r(k+1)) (z k) \\
 &\quad z = csub'(j+1)
 \end{aligned}$$

The vector conversion tactic could now be applied to obtain the following dynamically-tabulated program.

$$\begin{aligned}
 csub(j,k) &= (csub' j) k \\
 csub' j &= r \text{ where } \\
 &\quad r = (!! ) \text{ vec} \\
 &\quad \text{vec} = [cn \ a \ | \ a \leftarrow [0..M]] \\
 &\quad cn \ k = \text{if } j > N \text{ or } k > M \text{ then } 0 \\
 &\quad \quad \text{else if } XS !! j == YS !! k \text{ then } 1 + (z(k+1)) \\
 &\quad \quad \text{else max } (r(k+1)) (z k) \\
 &\quad z = csub'(j+1)
 \end{aligned}$$

Notice that each new vector  $r$  is computed from the previous vector  $z$  down the recursion; as well as from other elements of the current vector,  $r$ .

The circular variable introduction tactic is used to convert a sub-class of non-**T0** functions to their **T0** equivalent. This sub-class of functions have cycles in the dependency graphs of their recursive descendant calls. Tupling transformation tactic is unable to cope with such cycles directly. To overcome this problem, we replace each such re-occurring recursive call (explicit cycle) by an equivalent local recursion variable (implicit cycle). This step helps us obtain **T0** class functions which could be transformed by the tupling tactic.

The circular variable tactic could also be used in the tabulation of the  $bin$  function. Instead of taking the first parameter of  $bin$  as the recursion parameter, it is possible to regard its second parameter  $k$  as the recursion parameter. In this case, we would introduce:

$$bin2(k) = (\backslash n \rightarrow bin(n,k))$$

which could be transformed to:

$$\begin{aligned}
 bin2(0) &= (\backslash n \rightarrow 1) \\
 bin2(k) &= (\backslash n \rightarrow \text{if } k \leq 0 \text{ or } k \geq n \text{ then } 1 \text{ else } (bin2(k-1)(n-1)) + (bin2(k)(n-1)))
 \end{aligned}$$

Based on the pattern of the second equation we have  $k > 0$ . This knowledge could be used to omit the test  $k \leq 0$  as it will always be false. Also, the function  $bin2$  is still not a **T0** class function but could be made into one with the help of a circular variable. These transformations yield the following.

$$\begin{aligned} bin2(0) &= (\backslash n \rightarrow 1) \\ bin2(k) &= z \text{ where } z = (\backslash n \rightarrow \text{if } k \geq n \text{ then } 1 \text{ else } (bin2(k-1)(n-1)) + (z(n-1))) \end{aligned}$$

This function could then be subjected to the vector conversion technique. As the value of  $k$  is always smaller than  $n$ , this alternative program will use a smaller outer recursion (outer vector).

Like the lambda abstraction transformation, the introduction of a circular variable is based on the need to conform to the **T0** grammar form.

## 7. Tail-Recursion Transformation

Compared to the full memoisation technique, our vectorised programs can yield better performances because the *search* for repeated calls has been replaced by *direct access* using vector indices. However, as all the vectors are constructed on the heap space, the total memory used for storing these calls may not be asymptotically better than that for full memoisation.

It is possible to improve the space behaviour of the vectorised programs. For this, we require another technique, called the tail-recursion transformation, which could convert some of the linear recursive programs to their iterative counterparts. This technique could help obtain programs which are more memory-efficient because their vectors would be carried as parameters that could be updated in-place once they are made strict.

The tail-recursion transformation technique is already quite well developed [11, 18] and is usually based on a collection of transformation schemes. One linear recursive program scheme which could be used to obtain tail-recursive equivalent is shown below.

$$f \ x \quad = \text{if } x == L \text{ then } C \text{ else } H \ x \ (f \ (N \ x)) \ \text{st } I \ (N \ x) = x;$$

This scheme has a known base case,  $L$ , and an inverse function,  $I$ , for the parameter's descent function,  $N$ . (Note that **st** is used to carry a required semantic condition.) Programs which satisfy this scheme can always be converted to the following tail-recursive equivalent.

$$\begin{aligned} f \ x &= f\_it \ (L, x, C) \\ f\_it \ (n, x, w) &= \text{if } n == x \ \text{then } w \ \text{else } f\_it \ ((I \ n), x, (H \ (I \ n) \ w)) \end{aligned}$$

All three examples ( $bin'$ ,  $knap'$ ,  $csub'$ ) used earlier satisfy the above linear recursive program scheme and could be transformed to their tail-recursive counterpart. In the case of  $bin'$ , the tail recursive equivalent is:

$$\begin{aligned} bin' \ n &= bin\_it(0, n, (\backslash k \rightarrow 1)) \\ bin\_it(n, x, w) &= \text{if } n = x \ \text{then } w \ \text{else } bin\_it(n+1, x, (!) \ vec) \\ &\quad \text{where } \vec = [bn \ k \mid k \leftarrow [0..n+1]] \\ &\quad \quad bn \ k = \text{if } k \leq 0 \ \text{or } k \geq (n+1) \ \text{then } 1 \ \text{else } (w(k-1)) + (w \ k) \end{aligned}$$

Suitable compilation techniques could then be used to convert the list-based vectors (when both bounds are known) to real vectors before in-place updates of the vectors are incorporated. In-place update of the vector is not possible if the vector parameter has to be lazily evaluated. However, if we are prepared to sacrifice some loss in laziness and make the vector parameter strict, then in-place update could be carried out on the tail-recursive program. This can be seen as a trade-off to get more memory efficient programs.

## 8. Evaluation

To evaluate our proposed transformation techniques, we compare the performance of five different (both original and transformed) versions of the *bin* function over a range of inputs. Test runs for three sample inputs with calls *bin*(10,2), *bin*(20,4), *bin*(20,10) were conducted on a Haskell-like functional language interpreter, called Gofer[16]. This interpreter provides two very simple statistics for each evaluated expression, namely the number of *reduction steps* and the number of *heap cells* used. These two statistics are sufficient for a quick comparison of the different transformation techniques.

The five different versions of the *bin* function are:

1. Original Recursive *bin*
2. Fully-Memoised *bin*
3. Vectorised (list-based) *bin*
4. Tail-Recursive *bin*
5. Circular Vectorised *bin*

The original recursive *bin* function was given in Section 1. The vectorised version of *bin* was obtained from Section 5 after applying the lambda abstraction transformation and the vector conversion technique. The tail-recursive version of *bin* was obtained from Section 7 after applying the tail-recursion transformation. An alternative vectorised *bin* definition which uses circular vector is obtained from Section 6. The fully-memoised version of *bin* was not given earlier. It could be obtained using the memoisation technique of Holst and Gomard [14], which would derive the following program.

$$\begin{aligned}
 \mathit{bin} \ (n,k) &= \mathit{lookup} \ \mathit{vec} \ (n,k) \ \mathbf{where} \\
 &\quad \mathit{vec} = [(d, (\mathit{binm} \ (\mathit{lookup} \ \mathit{vec} \ d)) \mid d \leftarrow \mathit{dom})] \\
 &\quad \mathit{dom} = \mathit{nub} \ (\mathit{Cons} \ (n,k) \ (\mathit{concat} \ [n \ \mid (d,(-n)) \leftarrow \mathit{vec}])) \\
 \mathit{binm} \ \mathit{bn} \ (0,k) &= (1,[(0,k)]) \\
 \mathit{binm} \ \mathit{bn} \ (n,k) &= (\mathbf{if} \ k \leq 0 \ \mathbf{or} \ k \geq n \ \mathbf{then} \ 1 \\
 &\quad \mathbf{else} \ (\mathit{bn}(k-1)) + (\mathit{bn} \ k), [(n-1, k-1), (n-1, k)]) \\
 \mathit{lookup} \ t \ v &= \mathit{fst} \ (\mathit{lk} \ t \ v) \ \mathbf{where} \\
 &\quad \mathit{lk} \ (\mathit{Cons} \ (x,b) \ xs) \ a = \mathbf{if} \ a == x \ \mathbf{then} \ b \ \mathbf{else} \ \mathit{lk} \ xs \ a
 \end{aligned}$$

Note that *nub* is used to eliminate duplicates in a list, while *concat* will flatten a list of lists.

From Table 1, we can see that both the tail-recursive and the vectorised *bin* function definitions give the best overall performance, with their overheads increasing in proportion to the size of the product of the two parameters. Based on the Gofer interpreter, there appears to be little difference in performance between the vectorised and tail-recursive definitions. We expect a much better performance for tail-recursive programs if in-place update of vectors could be

exploited. However, this expectation would have to await the arrival of more sophisticated compilers for functional languages.

The fully-memoised *bin* function has a high fixed overhead, which is especially significant for the two calls with smaller arguments, namely *bin(10,2)* and *bin(20,4)*. For these two calls, the fully-memoised function actually performs worse than the untransformed function. However, for another call *bin(20,8)*, the untransformed case has a run-time cost which shoots up exponentially to (*3233219 steps/7020714 cells*). This is much worse than the fully-memoised version. We also carried out measurements for the alternative *bin2* function definition which uses a circular vector (from Section 6). There appears to be a marginal improvement in heap space utilisation but a general increase in the number of reduction steps needed.

During our evaluation, we experienced two deficiencies in the Gofer interpreter, namely (i) it does not support real arrays, and (ii) actual CPU time is not reported (the number of reduction steps does not allow us to measure certain optimisation e.g. tail-calls). As most functional language compilers are able to optimise tail calls into jumps and reduce stack utilisation, we carried out a separate set of evaluations using Chalmer’s Haskell compiler [2] which also supports real arrays. Calls with small arguments ran too fast to be compared. We therefore ran our *bin* function for larger argument values. The results are displayed in Table 2. In general, heap space utilisations of tail-recursive and vectorised versions are about the same, but tail-recursive version runs faster (e.g. *bin(500,100)* completes in 26.5 seconds, as opposed to 30.3 for the vectorised case).

The alternative *bin* using circular vector runs a lot slower when list-based vectors are used, but runs faster with real arrays when it is compared to the vectorised/tail recursive versions. The reason for this is that the alternative *bin* function uses fewer but longer vectors (smaller outer recursion). This causes an increase in CPU time with list-based vectors because access time is proportional to the size of such vectors. With real arrays, the access time is constant but the alternative version of *bin* still runs faster because it needs only perform one test for every conditional branch rather than two tests needed by the tail-recursive/vectorised versions.

Also, those *bin* calls which use real arrays run faster than those using list-based vectors but we experience a bad space-leak with real arrays (for e.g. *bin(500,100)* encounters out of heap space when real arrays are used). This space-leak is confirmed with the help of a heap profiler. Such space leaks are also experienced by functions which return tuple results. They may be avoided by using a special compilation technique of Sparud [23]. Alternatively, if in-place update of vectors were supported (for tail-recursive functions), the heap space required will also be much lower.

## 9. Related Work

The use of lambda abstraction in conjunction with the tupling tactic is not a new idea. It has been used in [21, 22] as an alternative to circular programs [4] for the elimination of multiple traversals over data structures. We review some of these previous works here.

**Table 1.** Run-times for different versions of *bin* under the Gofer interpreter

<i>Call</i>	<i>Original Recursive</i>		<i>Fully-Memoised</i>	
	<i>Steps</i>	<i>Cells</i>	<i>Steps</i>	<i>Cells</i>
<i>bin(10,2)</i>	736	1665	16079	27469
<i>bin(20,4)</i>	80416	181187	235620	383542
<i>bin(20,10)</i>	3233219	7020714	236478	384930

<i>Call</i>	<i>Vectorised</i>		<i>Tail-Recursive</i>		<i>Circular</i>	
	<i>Steps</i>	<i>Cells</i>	<i>Steps</i>	<i>Cells</i>	<i>Steps</i>	<i>Cells</i>
<i>bin(10,2)</i>	498	995	491	995	370	585
<i>bin(20,4)</i>	1786	3181	1769	3171	2040	2782
<i>bin(20,10)</i>	3472	5565	3455	5555	3609	5124

**Table 2.** Run-times for different versions of *bin* under the LML compiler

<i>List-Based Vectors</i>	<i>Vectorised</i>		<i>Tail-Recursive</i>		<i>Circular</i>	
	<i>Sec</i>	<i>Cells</i>	<i>Sec</i>	<i>Cells</i>	<i>Sec</i>	<i>Cells</i>
<i>bin(100,50)</i>	0.80	1331300	0.79	1331712	1.26	2375200
<i>bin(200,100)</i>	6.39	9301700	6.17	9302512	11.70	17479308
<i>bin(300,100)</i>	13.13	18350500	13.15	18351712	35.72	50558508
<i>bin(300,200)</i>	28.04	35073700	25.85	35074912	41.08	51598900
<i>bin(500,100)</i>	30.29	36448100	26.50	36450112	134.39	164477100
<i>Real Vectors</i>	<i>Sec</i>	<i>Cells</i>	<i>Sec</i>	<i>Cells</i>	<i>Sec</i>	<i>Cells</i>
<i>bin(100,50)</i>	0.28	720488	0.28	708100	0.29	764288
<i>bin(200,100)</i>	1.18	2800088	1.12	2775300	1.05	3027688
<i>bin(300,100)</i>	6.16	6159688	5.96	6122500	3.10	4707688
<i>bin(300,200)</i>	6.23	6159688	6.09	6122500	5.09	8694488
<i>bin(500,100)</i>	-	-	-	-	-	-

In [4], Richard Bird suggested the use of tupled circular programs to help eliminate multiple traversals on data structures. To illustrate this technique, consider a simple function to test if a string is a palindrome, as follows:

```

palindrome xs    = eq xs (rev xs Nil)
eq Nil ys       = ys==Nil
eq (Cons x xs) ys = ((hd ys)==x) ^ (eq xs (tl ys))
rev Nil ys      = ys
rev (Cons x xs) ys = rev xs (Cons x ys)

```

As defined above, the *palindrome* function traverses its input list twice, once by *eq* and another time by *rev*. To eliminate such multiple traversals, Bird introduced a new tupled function, shown below.

```

pal_tup xs ys zs = ((eq xs ys),(rev xs zs))

```

which is then transformed to:

```

pal_tup Nil ys zs      = ((ys==Nil),zs)
pal_tup (Cons x xs) ys zs = ((hd ys)==x ^ u,v) where
                          (u,v) = (pal_tup xs (tl ys) (Cons x zs))

```

The *pal\_tup* function traverses the *xs* variable once. However, the two calls of *eq* and *rev* in the RHS of *palindrome* are nested. To eliminate multiple traversals of *xs* on the original definition of *palindrome*, Bird has to use the above tupled function in conjunction with a circular program, as follows.

```

palindrome xs    = u where (u,v) = ((eq xs (rev xs Nil)),(rev xs Nil))
                  = u where (u,v) = pal_tup xs (rev xs Nil) Nil
                  = u where (u,v) = pal_tup xs v Nil

```

Note that a locally recursive (circular) variable *v* has been introduced. It is needed for the elimination of multiple traversals from nested function calls. Apart from circular programs, Bird's technique also required lazy evaluation. A positive aspect of this technique is that only first-order parameter generalisation is used rather than higher-order generalisation. As a result, the transformed program is also first-order. However, programs with redundant calls (e.g. binomial and knapsack) could not be transformed using just a first-order generalisation technique. In addition, some transformation steps taken by Bird require human-insights.

To avoid the use of circular programs<sup>3</sup>, Takeichi used a different technique, called *lambda hoisting*[24]. This technique requires the introduction of a common (more general) higher-order function together with a full laziness compilation technique. In the case of the *palindrome* example, the lambda hoisting technique will obtain the following transformed code with *zip2* as the common higher-order function to replace both *rev* and *eq* function calls.

```

palindrome xs    = zip2 xs True and (==) (zip2 xs Nil Cons m Nil)
zip2 xs          = n (if (null xs)) (hd xs) (zip2 (tl xs))
                  where m x y = x
                  n p q r a g f y = p a (r (g (f q (hd y)) a) g f (tl y))

```

---

<sup>3</sup> It is generally thought that circular programs are expensive to implement (as well as to comprehend).

With the use of a suitable common higher-order function and sharing via partial parameterisation, neither tupling nor circular programs are required anymore. This is because multiple traversals have been eliminated through the higher degree of sharing achieved by full laziness. It is, however, not obvious how common higher-order functions could be invented automatically.

Subsequently, Pettorossi introduced a more systematic technique for the elimination of multiple traversals. His technique combines lambda abstraction with tupling. The use of lambda abstraction as a higher-order generalisation technique can help avoid both local recursion and lazy evaluation. It is also more systematic because its steps could be guided by the *forced folding* approach of Darlington [12]. In the case of the above example, Pettorossi introduced the following tupled function with lambda abstraction.

$$pal\_tup'(xs,zs) = ((\lambda ys \rightarrow eq(xs,ys)),(rev xs zs))$$

This function could be successfully transformed to the following where the *xs* variable is only traversed once.

$$\begin{aligned} pal\_tup' Nil zs &= ((\lambda ys \rightarrow ys==Nil),zs) \\ pal\_tup'(Cons x xs) zs &= ((\lambda ys \rightarrow (hd ys)==x \wedge (u ys)),v) \\ &\quad \mathbf{where} (u,v)=pal\_tup'(xs, (Cons x zs)) \end{aligned}$$

Using lambda abstraction, a circular program is avoided as the *palindrome* function can now be transformed, as follows.

$$\begin{aligned} palindrome xs &= u v \mathbf{where} (u,v) = ((\lambda ys \rightarrow (eq xs ys)), (rev xs Nil)) \\ &= u v \mathbf{where} (u,v) = pal\_tup' xs Nil \end{aligned}$$

To come up with the right lambda abstractions, Pettorossi suggested the use of *forced folding*. This approach works by applying a suitable generalisation to places where the attempted folds have failed. To automate the technique, powerful (mis-)matching algorithms/heuristics, such as those used in Turchin's supercompiler [25], will be needed. Such an approach has also been referred to as an *on-line* technique in [17] because on-the-fly generalisation during transformation is required.

We follow Pettorossi in the advocacy of lambda abstraction and tupling transformation but use a different approach for introducing the right lambda abstractions. Instead of forced-folding, we make use of the **T0** class grammar form to help guide higher-order generalisation. Our approach would be referred to as an *off-line* generalisation technique as the analysis is performed before the actual transformation. It appears to be more amenable to mechanisation. Also, we have now made use of lambda abstractions to help eliminate redundant calls, while previous works have concentrated mainly on eliminating multiple traversals.

## 10. Conclusion

To summarise, this paper has made three main contributions. Firstly, we have extended the use of lambda abstractions to formalise dynamic-sized tabulation - an aspect which was not covered by previous works [24, 21, 22]. Secondly, we have made a different use of circular programs to allow some vectors to be computed in-situ. Last, but most importantly, we have now obtained an

automatable dynamic-sized tabulation technique. This is made possible by using the grammar form of the **TO** class to help guide the introduction of lambda abstraction and local recursion; followed by the vector conversion technique. Previously, such transformations might be done either manually (using human-guided intuitions) or by trial-and-error (using forced-folding idea).

With a better understanding of the role lambda abstraction (and local recursion) play, we are now able to obtain wider classes of functions which could be tabulated automatically. Our method uses the **TO** form to guide the introduction of lambda abstractions and local recursion. Together with the vector conversion and tail-recursion transformation tactics, we can systematically obtain suitably tabulated programs.

Further work remains to be done. If the parameters of the lambda abstractions are not based on integers, we require suitable data-type transformation techniques. Investigating these techniques could help widen the class of programs which could be dynamically-tabulated.

*Acknowledgement.* This work started during Masami Hagiya's visit to the National University of Singapore under an exchange program of the Japan Society for the Promotion of Science (JSPS). We would like to thank JSPS whose support has helped initiate this collaborative work. We would also like thank Khoo Siau Cheng for some corrections to an earlier draft of the paper, and Arun CB Kumar for helpful feedback and programming assistance. Lastly, we are extremely grateful to the reviewers of Acta Informatica for constructive comments which have helped improve the paper's presentation significantly.

## References

1. Lennart Augustsson. Compiling pattern-matching. In Conference on Functional Programming and Computer Architecture, pages 368–381, Nancy, France, Springer-Verlag LNCS 201, 1985.
2. Lennart Augustsson. Haskell B User's Manual, February 1993.
3. Richard S Bird. Tabulation techniques for recursive programs. ACM Computing Surveys, 12(4):403–417, December 1980.
4. Richard S Bird. Using circular programs to eliminate multiple traversals of data. Acta Informatica, 21:239–250, 1984.
5. W W Bledsoe. A new method for proving certain Presburger formulae. In Proc of ICJAI, pages 15–21, 1975.
6. R M Burstall and John Darlington. A transformation system for developing recursive programs. Journal of Association for Computing Machinery, 24(1):44–67, January 1977.
7. Wei-Ngan Chin. Safe fusion of functional expressions. In 7th ACM Lisp and Functional Programming Conference, pages 11–20, San Francisco, California, June 1992.
8. Wei-Ngan Chin. Towards an automated tupling strategy. In 3rd ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, pages 119–132, Copenhagen, Denmark, ACM Press, June 1993.
9. Wei-Ngan Chin and Siau-Cheng Khoo. Tupling functions with multiple recursion parameters. In 3rd International Workshop on Static Analysis, pages 124–140, Padova, Italy, Springer-Verlag LNCS 724, September 1993.
10. Norman H Cohen. Eliminating redundant recursive calls. ACM Transaction on Programming Languages and Systems, 5(3):265–299, July 1983.
11. D C Cooper. The equivalence of certain computations. The Computer Journal, 9(1):45–52, May 1966.
12. John Darlington. An experimental program transformation and synthesis system. Artificial Intelligence, 16:1–46, 1981.

13. Masami Hagiya. Typed  $\lambda$ -calculus with arithmetical constraints. In Proc of Workshop on Algorithmic Learning Theory, Springer-Verlag LNAI, 1993.
14. C K Holst and C K Gomard. Partial evaluation is fuller laziness. In ACM Symposium on Partial Evaluation and Program Manipulation, pages 223–233, New Haven, Connecticut, August 1991.
15. Paul Hudak et al. Report on the programming language Haskell: A non-strict, v purely functional language. ACM SIGPLAN Notices, 27(5), May 1992.
16. Mark P. Jones. An Introduction to Gofer, 1991.
17. Neil D Jones. Automatic program specialisation: A re-examination from basic principles. In Workshop on Partial Evaluation and Mixed Computations, pages 225–282, G1 Avarnes, Denmark, North-Holland, 1988.
18. Hessam Knoshnevisan. Automatic Transformation Systems based on Function-Level Reasoning. PhD thesis, Imperial College, University of London, July 1987.
19. Donald Michie. Memo functions and machine learning. Nature, 218:19–22, 1968.
20. Alberto Pettorossi. A powerful strategy for deriving programs by transformation. In 3rd ACM Lisp and Functional Programming Conference, pages 273–281, 1984.
21. Alberto Pettorossi. Program development using lambda abstraction. In Intl Conf on Foundations of Software Technology & Theoretical Computer Science, pages 420–434, Pune, India, Springer-Verlag LNCS 287, 1987.
22. Alberto Pettorossi and Andrzej Skowron. Higher-order generalization in program derivation. In Conf on Theory & Practice of Software Development, pages 182–196, Pisa, Italy, Springer-Verlag LNCS 250, 1987.
23. Jan Sparud. How to avoid space-leak without a garbage collector. In ACM Conference on Functional Languages and Computer Architecture, pages 117–122, Copenhagen, Denmark, June 1993.
24. Masato Takeichi. Partial parameterization eliminates multiple traversals of data structures. Acta Informatica, 24:57–77, 1987.
25. Valentin F Turchin. The algorithm of generalisation in the supercompiler. In Workshop on Partial Evaluation and Mixed Computations, pages 531–549, G1 Avarnes, Denmark, North-Holland, 1988.
26. J Allan Yang and Young-il Choo. Parallel-program transformation using a metalanguage. In 3rd ACM Symp on Principles and Practice of Parallel Programming, pages 11–20, Williamsburg, Virginia, April 1991.

## A. Bounds Determination

In [13], a new typed lambda calculus which permits integer types to be restricted by some linear constraints was proposed to facilitate certain proof generalisation techniques. This calculus allows a notation of the form  $\{i;P(i)\}$  to represent a sub-type of integer  $i$  where  $P(i)$  holds. The type-checking problem of this calculus is then reduced to the problem of proving a formula in arithmetic. As the constraints are restricted to linear forms of integers, the arithmetic formula to be proved is always a Presburger formula whose validity is decidable.

The calculus also has a type of the form  $\Pi i;P(i).A$  which represents a function type that returns a value of  $A$  type when it is given an integer  $i$  such that  $P(i)$  holds. By using  $\Pi$ -types, the curried *bin* function could be assigned the type  $\Pi n;0 \leq n.(\Pi k;0 \leq k \leq n.int)$  which could be checked. From this type, we could see that the parameter of the inner lambda abstraction is bounded and is thus a candidate for vector conversion.

However, our proposed dynamic tabulation method does not require functions to be given restricted types of the above form. Therefore, it is necessary to *infer* an appropriate  $\Pi$ -type for each function.

We are currently devising an inference technique to help obtain upper and lower bounds for the parameters of a given recursive function. The full details of

this technique shall appear in a forthcoming paper. In this appendix, we briefly outline the technique using the *bin* function as an example.

Our bounds determination technique attempts to find suitable dependency relationships of the parameters over all recursive calls of the given function. We introduce the following labelling scheme to identify successive recursive calls of each given function. The first recursive call of a function,  $f$ , will be labelled as  $f_0(\dots)$ . Its immediate dependent recursive calls will be labelled as  $f_1(\dots)$ . Subsequently, the dependent (child) calls of  $f_1(\dots)$  will be labelled as  $f_{i+1}(\dots)$ . The suffix given to each recursive call identifies the recursion level of the call. A similar labelling scheme is also provided for the arguments of each recursive function call. Specifically, an argument, called  $x$ , of the recursive call,  $f_i(\dots x \dots)$  will be labelled as  $x_i$ .

In the case of the *bin* function, we could use the above labelling scheme to obtain the following annotated function definition.

$$\begin{aligned} \mathit{bin}_i(0,k) &= 1 \\ \mathit{bin}_i(n,k) &= \text{if } k \leq 0 \text{ or } k \geq n \text{ then } 1 \text{ else } (\mathit{bin}_{i+1}(n-1,k-1)) + (\mathit{bin}_{i+1}(n-1,k)) \end{aligned}$$

We assume the pre-condition  $0 \leq k_0 \leq n_0$  for the first recursive call. We proceed to infer the relationships of the parameters, as follows.

From the LHS of the two equations, we could deduce that  $n_i = 0$  or  $n_i > 0$ . These two dependencies could be combined into one, namely  $n_i \geq 0$ . Under the context of the **else** branch, we also have  $(k_i > 0) \wedge (k_i < n_i)$ . These two dependencies are valid for the two recursive calls  $\mathit{bin}_{i+1}(n-1, k-1)$  and  $\mathit{bin}_{i+1}(n-1, k)$ . The first recursive call has the following additional dependencies:

$$(n_{i+1} = n_i - 1) \wedge (k_{i+1} = k_i - 1)$$

while the second recursive call also has additional dependencies:

$$(n_{i+1} = n_i - 1) \wedge (k_{i+1} = k_i)$$

Based on the above dependencies, we could make use of the SUP-INF method of Bledsoe[5], to help obtain upper and lower bounds of the given parameter variables. In the case of the first recursive call, we could infer:

$$(0 \leq k_{i+1} < n_{i+1}) \wedge (n_{i+1} < n_i) \wedge (n_{i+1} - k_{i+1} = n_i - k_i) \wedge (k_{i+1} < k_i)$$

Similarly, from the second recursive call, we could infer:

$$(0 < k_{i+1} \leq n_{i+1}) \wedge (n_{i+1} < n_i) \wedge (n_{i+1} - k_{i+1} < n_i - k_i) \wedge (k_{i+1} = k_i)$$

These two sets of dependencies could be combined to give:

$$(\max(0, n_{i+1} - n_i + k_i) \leq k_{i+1} \leq \min(k_i, n_{i+1})) \wedge (0 \leq n_{i+1} \leq n_i)$$

The last set of dependencies is over the parameter variables  $k_i, k_{i+1}, n_i, n_{i+1}$ . By a simple induction, we could further obtain the following universally quantified formula:

$$\forall i. ((\max(0, n_i - n_0 + k_0) \leq k_i \leq \min(k_0, n_i)) \wedge (0 \leq n_i \leq n_0))$$

The final set of dependencies contains the lower and upper bounds for the two parameters of the function *bin*. These bounds could be used by the vector conversion technique of Section 5. In particular, for the binomial example given there, we have used a simpler bound  $0 \leq k_i \leq n_i$  that was extracted from the inferred bound<sup>4</sup>  $\max(0, n_i - n_0 + k_0) \leq k_i \leq \min(k_0, n_i)$ .

This article was processed by the author using the L<sup>A</sup>T<sub>E</sub>X style file *cljour1* from Springer-Verlag.

<sup>4</sup> We would like to thank an anonymous reviewer for showing us how a tighter lower bound could be inferred for this binomial example.