Program Transformation by Solving Recurrences

Beatrice Luca Stefan Andrei Hugh Anderson Siau-Cheng Khoo
Department of Computer Science, School of Computing, National University of Singapore
{lucabeat,andrei,hugh,khoosc}@comp.nus.edu.sg

Abstract

Recursive programs may require large numbers of procedure calls and stack operations, and many such recursive programs exhibit exponential time complexity, due to the time spent re-calculating already computed sub-problems. As a result, methods which transform a given recursive program to an iterative one have been intensively studied. We propose here a new framework for transforming programs by removing recursion. The framework includes a unified method of deriving low time-complexity programs by solving recurrences extracted from the program sources. Our prototype system, `APTSR1`, is an initial implementation of the framework, automatically finding simpler *"closed form"* versions of a class of recursive programs. Though in general the solution of recurrences is easier if the functions have only a single recursion parameter, we show a practical technique for solving those with multiple recursion parameters.

Categories and Subject Descriptors D.1.2 [*Automatic Programming*]: Program transformation; D.3.3 [*Language Constructs and Features*]: Recursion; F.3.1 [*Specifying and Verifying and Reasoning about Programs*].

General Terms Languages, Performance.

Keywords Program transformation, recurrences with one or multiple parameters, efficient time complexity.

1. Introduction

Linear recurrences play a significant role in many areas of computer science. For example, in the field of complexity analysis, recurrence relations can be used to compactly express complexity measures. The solution of the recurrences provides a closed form expression which can be evaluated to decide if mobile agents can be allowed to run in a given context [16], assist programmers to reason about the behaviour of the programs, optimize their programs [22], and even discover bugs that reduce the efficiency of a program, that may be otherwise difficult to detect. In addition, many algorithms are naturally expressed in a recursive form, although there may be more efficient closed forms of the algorithms.

There is a significant need for efficient software systems that incorporate recursive functions. A common concern with such software is to reduce the time or space complexity of the recursive functions, and since much of the complexity is due to the time spent in recalculating already computed sub-problems, an improvement is to compute each sub-problem only once, remembering a specific number of previous values.

Traditional techniques for program transformation are *tupling* [4, 21], *memoization* [1, 6, 7, 14, 19, 24], *tabulation* [2, 3, 13, 23] and *static incrementalization* [18].

Tupling aims to compute multiple values together in an efficient way, leaving open the possibility for parallel evaluation. It can be made automatic on subclasses of problems [5] and to work on more general forms [8]. It is also extended to store lists of values [23], but such lists are generated in a fixed way, which is not the most appropriate way for many programs. A special form of tupling can eliminate multiple data traversals for many functions [13].

Memoization has the advantage that the original recursive program needs little change and only values needed for the solution are computed. However, separate table management has an interpretative overhead and a general strategy cannot be efficient for all problems.

The *tabulation* method involves a compiled table management, no interpretative overhead, specialized computation strategy and also optimized space usage. Tabulation manually rewrites the original program and may compute values that are not needed for the solution.

Static incrementalization has the advantage of a transformation based on computation increment. Compared with previous methods that perform memoization or tabulation, the method based on incrementalization is more powerful and systematic and it stores only values that are necessary for the optimization.

Program transformation by solving recurrences is a novel and powerful technique that is capable of obtaining low time-complexity run-time *closed form* (in terms of algebraic operations) non-recursive programs from exponential recursive programs. Here is an example, the *"Domino Puzzle"*, which can be efficiently transformed to a closed form by our technique, but not by the traditional techniques. This puzzle asks the question: How many ways, $\mathrm{U}(n)$ are there to tile a $3 \times n$ rectangle with dominos? A solution program ($\mathcal{O}(e^n)$) derived using the approach suggested in [12] consists of two mutually recursive functions:

```
U(n) = if (n = 0) then 1
       else if (n = 1) then 0
       else U(n − 2) + 2 ∗ V(n − 1);
V(n) = if (n = 0) then 0
       else if (n = 1) then 1
       else U(n − 1) + V(n − 2);
```

A transformed *"Domino Puzzle"* program ($\mathcal{O}(\log n)$) obtained automatically by our method is:

$$\mathrm{U\_s}(n) = -\tfrac{1}{12}(1+(-1)^n)((2-\sqrt{3})^{\frac{n}{2}}(-3+\sqrt{3})-(2+\sqrt{3})^{\frac{n}{2}}(3+\sqrt{3}));$$

$$\mathrm{V\_s}(n) = \frac{(1+-1^n)((2-\sqrt{3})^{\frac{1+n}{2}}-(2+\sqrt{3})^{\frac{1+n}{2}})}{4\sqrt{3}};$$

PEPM '06 January 9–10, Charleston, South Carolina, USA.

Another example that cannot be transformed to a closed form using the traditional techniques is this Fibonacci-like function, which has *two* recursion parameters:

```
fib1(m, n) = if ((m ≤ 1) or (n ≤ 3)) then 1
             else fib1(m − 1, n − 2) + fib1(m − 2, n − 4);
```

The transformed program obtained automatically by our method is:

```
fib1_s(m, n) =
  1/2(1 + √5/5)((1+√5)/2)^min(m, n/2) + 1/2(1 − √5/5)((1−√5)/2)^min(m, n/2);
```

For efficiency, in practice we may retain the base cases of the functions in the final programs. The transformed functions have no recursive calls, and specify a compact and efficient form for calculating the results. The most time-costly component of the efficient form is the calculation of an exponentiation a^b, but such a computation can be done in $\mathcal{O}(\log b)$ time. We use mathematical notation in the final program for clarity, for example, $\sqrt{a}$ instead of `sqrt(a)` and a^b instead of `pow(a,b)`, as the use of the programming notation results in long and unclear programs. The use of real-valued variables (for example $\sqrt{3}$) may introduce concern about the precision of values, however sufficient precision is easily achieved using (for example) an arbitrary precision package such as the MPFR library [26].

The class of programs that can be transformed effectively is the class of programs that have corresponding recurrences that may be solved using automatic tools such as MATHEMATICA. This permits a wide range of functions, including:

- Functions with one recursion parameter and other symbolic parameters.
- Recursive calls on one branch - *single recursive branch* (SRB), or more than one branch - *multiple recursive branches* (MRB).
- Different types: linear recurrences of finite order with constant coefficients, linear recurrences with variable coefficients or divide and conquer recurrences.
- Simple conditionals containing polynomial expressions or ones containing a single recursive call. SRB programs with such a recursive test are called SRB-RT.
- Functions with multiple recursion parameters.

In this paper we show useful function templates and the types of recurrences that can be extracted from them, but we avoid showing how the recurrences are solved, and just name the particular solution method used. The emphasis is on program transformation rather than solving recurrences.

This paper's contributions are:

- We present a new technique, program transformation by solving recurrences, for obtaining low-complexity non-recursive programs. This technique solves the implicit recursion equation found on each path and eliminates any recursive conditionals.
- We describe and target an important class of recursive programs; the ones that contain linear recurrences of finite order with constant or variable coefficients and divide and conquer recurrences. The class can be extended to programs with many parameters but only one recursion parameter. Moreover, we admit an extension of the SRB functions; those with recurrences in conditionals, such as in this function:

```
f2(n) = if (n = 0) then 1
        else if (f2(n − 1) > 10) then f2(n − 1) + 4
        else 2 ∗ n;
```

- We present code transformation techniques corresponding to the types of the programs that we consider, SRB, MRB and SRB-RT. We present experimental results that demonstrate the effectiveness of the algorithms on a set of benchmarks.
- We show how a specific useful class of recursive functions with multiple recursion parameters may be methodically reduced to a more complex function with one recursion parameter. In turn this function may be expressed as a recurrence using the techniques outlined, leading to an efficient program for the original recursive function. There is no general method for solving recurrence relations with more than one parameter, but the class of functions that can be solved by our technique appears useful.

Section 2 introduces the terminology used in this paper, the formal basis of the approach, and characterizes the class of recursive functions analyzed in terms of their corresponding recurrences. Section 3 explains the analysis strategies for each of the categories of programs. Section 4 shows how functions with more than one recursion parameter may be methodically reduced to a more complex function with one recursion parameter. Section 5 describes APTSR1, and presents some preliminary results, and we conclude in Section 6 with discussion about the method.

2. Recurrences in programs

In this section we explore the terminology and definitions used by our approach. The transition from functions to corresponding recurrences is straightforward. For example, the recurrences corresponding to the functions U and V from the *Domino Puzzle* are:

$$\begin{aligned} \mathrm{U}_n &= \mathrm{U}_{n-2} + 2 \cdot \mathrm{V}_{n-1} \\ \mathrm{V}_n &= \mathrm{U}_{n-1} + \mathrm{V}_{n-2} \end{aligned}$$

Initial conditions for the Domino Puzzle are $\mathrm{U}_0 = 1$, $\mathrm{U}_1 = 0$, $\mathrm{V}_0 = 0$ and $\mathrm{V}_1 = 1$. For the recursive calls we may have corresponding linear recurrences of finite order with constant coefficients, linear recurrences of order 1 with variable coefficients or divide and conquer recurrences. We allow self and mutual recursive calls in function definitions. The recurrences may be *linear* or *non-linear recurrences*, and each of these can be in turn, *self*, *mutual* or *auxiliary recursive*.

DEFINITION 1. *(Self, Auxiliary and Mutual Recursion): A function f is said to be self-recursive if its recursive set of functions, from its call graph, is simply itself $\{f\}$. Mutual recursion is a form of recursion where two functions are defined in terms of each other. A function is said to be auxiliary recursive if the expressions contain auxiliary functions that have the recursive calls as parameters. An example of the use of an auxiliary function is* $\max(\mathrm{f2}(n-1), 10)$. *We also use the terms self, auxiliary and mutual when talking about the corresponding recurrences.*

If the recursive functions contain conditional constructs (guards), prior to their recursive calls, they are called *conditional recursive*.

DEFINITION 2. *(Conditional Recursive): A function call is said to be conditional recursive if there exists one or more guards prior to its recursive call(s). A recurrence is said to be conditional recursive if its corresponding function call is conditional recursive.*

Another classification, based on the structure of the conditionals is as follows:

DEFINITION 3. *(Recursive Calls in Conditionals): A function, or a set of mutually recursive functions, is said to have recursive calls in conditionals if at least one of the guards from its definition contains a self recursive call.*

v	$\in$	`Var`		⟨ Variables ⟩
f, g, h	$\in$	`FName`		⟨ Function names ⟩
n	$\in$	$\mathbb{Z}$		⟨ Integer constants ⟩
β	$\in$	`Guard`		⟨ Boolean expressions ⟩
		β ::=	$\delta \mid \beta_1$ or $\beta_2 \mid \beta_1$ and β_2	
		δ ::=	$e_1 = e_2 \mid e_1 \neq e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid e_1 \leq e_2 \mid e_1 \geq e_2$	
			$t = e_2 \mid t \neq e_2 \mid t < e_2 \mid t > e_2 \mid t \leq e_2 \mid t \geq e_2$	
e	$\in$	`AExp`		⟨ Arithmetic Expressions ⟩
		e ::=	$n \mid v \mid n \star e \mid e_1 + e_2 \mid -e$	
fe	$\in$	`FExp`		⟨ Functional Expressions ⟩
		fe ::=	$e \mid t \mid n \star fe \mid fe_1 + fe_2 \mid -fe$	
t	$\in$	`FTerm`		⟨ Function Terms ⟩
		t ::=	$f(v_1 + n_1, \ldots, v_m + n_m) \mid f(v_1/n_1, \ldots, v_m/n_m)$	
s	$\in$	`Stat`		⟨ Statements ⟩
		s ::=	`if` β `then` s_1 `else` $s_2 \mid s_1\ s_2 \mid fe$	
d	$\in$	`Decl`		⟨ Declarations ⟩
		d ::=	$f(x_1, \ldots, x_m) = s;$	
$\mathcal{P}$	$\in$	`Prog`		⟨ Program ⟩
		P ::=	$d*$	

Table 1. The language syntax

An example of a function with recursive calls in conditionals is the `f2` function given previously. The analysis of these functions is more difficult, but still tractable.

DEFINITION 4. *(Recursion and symbolic parameters): We distinguish two groups of parameters: recursion parameters whose size could change via recursive calls and symbolic parameters whose values do not change during recursive calls.*

More in depth classifications of the recursion parameters can be done (*constant*, *accumulative* or *roving* parameters) but we focus here on first-order recursive functions with one recursive parameter only.

2.1 The approach

Our goal is to transform exponential recursive programs in the simple first-order recursive language defined in Table 1 into equivalent programs with lower complexity. Note that the suffix d^* in Table 1 denotes a list of one or more distinct syntactic terms. A *Program* is a set of function definitions; each function definition is a set of base cases followed by one or more recursive calls on single or multiple `if` branches. Recursion is the only iterative construct.

We limit the form of programs to those which match templates. These templates are general enough to capture a wide class of programs, but each template has a corresponding set of recurrences, and hence extracting the recurrences from the programs that match the template is a trivial task.

Thus, from the initial specification, program P^{init}, we want to obtain a final program P^{final} which computes the same value, that is for all functions f in P^{init} and values $x \in \text{dom}(f)$, $\mathcal{E}[\![f^{\text{init}}(x)]\!] = \mathcal{E}[\![f^{\text{final}}(x)]\!]$. Note that $\mathcal{E}[\![P]\!]$ is the *evaluation* function defined inductively over the (functional) programming language. Moreover, we intend to preserve this equivalence between each step of the program transformation, such that

$$\mathcal{E}[\![f^i(x)]\!] = \mathcal{E}[\![f^{i+1}(x)]\!]$$

for all i such that $\text{init} \leq i < \text{final}$.

We define a space/time cost function $\mathcal{C}_1[\![P]\!]$ which measures the number of calls needed for a function. The principal reduction rule we use removes all recursion from a program and so

$$\mathcal{C}_1[\![P^{\text{final}}]\!] = 1 < \mathcal{C}_1[\![P^{\text{init}}]\!]$$

A second cost function $\mathcal{C}_2[\![P]\!]$ measures the number of arithmetic operations in a program. For all transformations in our framework, $\mathcal{C}_2[\![P^{i+1}]\!] \leq \mathcal{C}_2[\![P^i]\!]$. We *improve* a program by reducing either (or both) of these costs:

DEFINITION 5. *(Rule): A rule $P \longrightarrow P'$ is a basic step in the transformation of a program P to P' that preserves the above $\mathcal{E}$ equivalence and either improves the program, or leaves it unchanged.*

The rules are applied according to some strategy:

DEFINITION 6. *(Strategy): A strategy is an algorithm for choosing a sequence of rule reductions in the program transformation process for a particular type of program.*

Our transformation applies to three sub-classes of recursive functions, SRB - single recursive branch, MRB - multiple recursive branches and SRB-RT - recursive test(s) programs. SRB and MRB recursive programs may look limited, but they are able to express all integer linear recursive functions having one recurrence parameter.

These three classifications of the recursive programs are not disjoint; we may have programs with multiple recursive branches and with recursive calls in conditionals.

A new idea in our transformation is to exploit the use of symbolic computation. Symbolic computation, in contrast to numeric computation, deals with manipulating formulas instead of calculating numbers and thus the obtained generalized solutions can contain variables and parameters. `MATHEMATICA` allows for symbolic computations, and explains why it is used as the engine for our transformer.

For each program in our language, there is a corresponding system of recurrence equations, composed of the union of the base cases (BaseEq) and recurrences (RecEq). For the *Domino Puzzle* program P_{UV} consisting of the functions U and V, the system is:

$$P_{\text{UV}} \rightarrow \{\text{RecEq} \cup \text{BaseEq}\}_{\text{UV}} = \left\{ \begin{array}{l} \text{U}_n = \text{U}_{n-2} + 2 \cdot \text{V}_{n-1}, \\ \text{U}_0 = 1, \ \ \text{U}_1 = 0, \\ \text{V}_n = \text{U}_{n-1} + \text{V}_{n-2}, \\ \text{V}_0 = 0, \ \ \text{V}_1 = 1 \end{array} \right\}_{\text{UV}}$$

Note the clear correspondence between the programs and the system of recurrence equations. This correspondence is so direct and immediate that sometimes we swap from the recurrence view to the functional program view without comment.

The *solution* of this system of recurrence equations for U_n and V_n is a set of mappings $\{\mathrm{U}_n \to e_\mathrm{U}, \mathrm{V}_n \to e_\mathrm{V}\}$ from our functions U and V to closed form expressions e which express the same set of evaluations as the original system.

It is clear that the mappings correspond to a new program P' which computes the same values as the original P, and so we can now rewrite our program as

```
U_rs(n) = e_U;
V_rs(n) = e_V;
```

More formally we can express this as:

DEFINITION 7. *For a program P composed from a set of functions $\overline{\mathrm{F}} = \{f_1, \ldots, f_n\}$ with a corresponding system of recurrence equations $\{\mathrm{RecEq} \cup \mathrm{BaseEq}\}_{\overline{\mathrm{F}}}$, we define the $\xrightarrow{\mathrm{RS}}$ reduction rule*

$$\{\mathrm{RecEq} \cup \mathrm{BaseEq}\}_{\overline{\mathrm{F}}} \xrightarrow{\mathrm{RS}} \left\{\bigcup_i f_i \to e_i\right\}_{\overline{\mathrm{F}}}$$

such that for each $f_i \in \overline{\mathrm{F}}$ and $n \in \mathrm{dom}(f_i)$, $\mathcal{E}[\![f_i(n)]\!] = \mathcal{E}[\![e_i]\!]$, and e_i is a closed form expression (without recursion).

Note that since there is a clear correspondence between the programs and the system of recurrence equations, we can also view this reduction rule as one from program to program: $P \xrightarrow{\mathrm{RS}} P'$, and since the final program P' has no recursion $\mathcal{C}_1[\![P']\!] = 1 < \mathcal{C}_1[\![P]\!]$.

DEFINITION 8. *We define the $\xrightarrow{\mathrm{S}}$ simplification rule*

$$\left\{\bigcup_i f_i \to e_i\right\}_{\overline{\mathrm{F}}} \xrightarrow{\mathrm{S}} \left\{\bigcup_i f_i \to e'_i\right\}_{\overline{\mathrm{F}}}$$

such that $\mathcal{C}_2[\![e'_i]\!] \le \mathcal{C}_2[\![e_i]\!]$ and $\mathcal{E}[\![e_i]\!] = \mathcal{E}[\![e'_i]\!]$.

All recurrence types that we consider in this paper have decision procedures supported by MATHEMATICA. We use the RSolve built-in function to implement an $\xrightarrow{\mathrm{RS}}$ rule, and the Simplify function to implement an $\xrightarrow{\mathrm{S}}$ rule:

- RSolve[$\{\mathrm{RecEq} \cup \mathrm{BaseEq}\}_{\overline{\mathrm{X}}}$,{$\overline{X}$[n]},n] solves for a set of recurrences $\overline{\mathrm{X}}$.
- Simplify[e] simplifies expression e.

We also use the Reduce[e, dom] from MATHEMATICA to solve simple (in)equalities over a domain. The RecEq equations can involve terms of the form X_{n+i} where i is any fixed integer. Equations such as $\mathrm{X}_0 = c_0$ can be given to specify base conditions. If not enough base conditions are specified, RSolve will give solutions with undetermined constants.

We now present the forms of the recurrences examined in this paper, showing how each is solved by MATHEMATICA, and with examples of corresponding functions.

2.2 Linear recurrences of finite order with constant coefficients

A linear recurrence of finite order with constant coefficients has the form

$$X_n = a_1 \cdot X_{n-1} + \ldots + a_k \cdot X_{n-k} + p(n) \qquad (1)$$

for $a_k \neq 0$, $n \ge k$, where k is a positive integer. The coefficients a_j are real numbers, and if a_k is not zero, we say that the recurrence has order k. The function $p(n)$ is definable on $\mathbb{N} \cap [k, \infty)$, and is the *non-homogeneous* part of this recurrence. If $p(n) = 0$, then we say that the recurrence (equation 1) is *homogeneous*, otherwise it can be a sum of polynomials and exponentials. An example of a function with a corresponding homogeneous recurrence is the Fibonacci function.

Solutions to linear recurrences of finite order with constant coefficients are found by systematic means, by using *generating* functions or by noticing that r^n is a solution for particular values of r. If the recurrence is non-homogeneous, a particular solution can be found by the *method of undetermined coefficients*. The final solution is the sum of the solutions of the homogeneous and particular solutions. MATHEMATICA can provide an exact symbolic solution using *matrix powers* [25].

An example with symbolic parameters is in the computation of Chebyshev polynomials containing one symbolic parameter, $a = cos(t)$. These polynomials find practical application in the field of computer graphics when calculating boxing and intersections of parametric curves and surfaces [11]. A sample is:

```
cheb(n, a) = if (n = 0) then 1
             else if (n = 1) then a
             else 2 * a * cheb(n - 1, a) - cheb(n - 2, a);
```

The transformed function is much faster to compute:

```
cheb_s(n, a) = 1/2((a - Sqrt(-1 + a^2))^n + (a + Sqrt(-1 + a^2))^n);
```

2.3 Linear recurrences with variable coefficients

Linear recurrences with variable coefficients that have a solution in closed form are more difficult to classify. Exponential generating functions may be used, which after taking derivatives, lead to an ordinary differential equation (ODE). Unfortunately there are no general methods for finding closed form solutions to the resultant ODEs. However, an algorithm to solve the first-order case (equation 2) exists [20]. The general form is:

$$X_n = \alpha(n) \cdot X_{n-1} + p(n) \qquad (2)$$

where $\alpha(n)$ and $p(n)$ are polynomials in n. For these kind of recurrences there is only one initial value, usually it is X_o, but depending on the domain of $\alpha(n)$ and $p(n)$.

MATHEMATICA can derive solutions for this type of recurrence [25]. The example below is an average-case time analysis of quicksort:

```
qs(n) = if (n = 1) then 1
        else (1 + 1/n) * qs(n - 1) + 2 - 1/n;
```

The transformed function is presented below, where γ is Euler's constant (which can be evaluated to arbitrary precision in MATHEMATICA), and $\psi(a, b)$ is the Polygamma function, the $(n+1)^{th}$ logarithmic derivative of the Gamma function, which is computable in constant time.

```
qs_s(n) = -3n + 2γ(1 + n) + 2(1 + n)ψ(0, 1 + n);
```

2.4 Divide and conquer recurrences

Divide and conquer algorithms are a powerful tool for solving conceptually difficult problems, and also providing a natural way to design efficient programs, as the algorithms may often be executed on multi-processor machines, making efficient use of shared memory and so on. The recurrences have the form:

$$X_n = a \cdot X_{\frac{n}{b}} + p(n) \qquad (3)$$

where a, b are positive, and $p(n)$ a function whose domain is the set of positive integers.

In order to solve divide and conquer recurrences, we could use either the *rewrite* or *substitution* techniques. Depending on the form of the recurrence, we can *rewrite* the terms in easier-to-solve forms. For recurrence $X_i = X_{\frac{i}{2}} + X_{\frac{i}{4}}$, we let $Y_i = X_{2^i}$, and the recurrence becomes $Y_i = Y_{i-1} + Y_{i-2}$. In the *substitution* method, the idea is to guess the solution and then try to prove it explicitly, by induction.

There are good reasons to make divide and conquer algorithms more efficient. Besides the slowness of recursion, it may also be more complicated than some iterative approach, especially if large base cases are to be implemented for performance reasons.

Strassen's algorithm multiplies two $n \times n$ matrices in $\mathcal{O}(n^{2.81})$ time, instead of $\mathcal{O}(n^3)$. The complexity function of the algorithm runs in $\mathcal{O}(\log n)$ time, but we can improve it to be approximately $\mathcal{O}(1)$. The function is:

$$\text{strassen}(n) \texttt{ = if } (n = 1) \texttt{ then } 1 \texttt{ else } 7 * \text{strassen}(\tfrac{n}{2}) + 18 * (\tfrac{n}{2})^2;$$

The equivalent transformed function is the following:

$$\text{strassen\_s}(n) \texttt{ = } 7^{\frac{\log(2n)}{\log 2}} - 6n^2;$$

3. Strategies for transforming functions

Here we define the strategies used for each of the types of functions, SRB, MRB and SRB-RT. For each type of function, we define the syntax first, and then give the strategy, along with a worked example.

3.1 Strategy for SRB functions

An SRB function is a recursive function f with one or more base cases and with a single recursive branch, containing one or more recursive calls. A template for this is:

$$f(x) \texttt{ = if } c_0(x) \texttt{ then } b_0(x) \texttt{ else if } c_1(x) \texttt{ then } b_1(x) \texttt{ else if ... else } \text{rec\_calls}(x);$$

where the $\text{rec\_calls}(x)$ expression has one of the following forms:

$$a_1 \cdot g(x-1) + \ldots + a_k \cdot g(x-k) + p(x)$$
$$p(x) \cdot g(x-1) + p(x)$$
$$a \cdot g(x/b) + p(x)$$

and g is any function (including f) that belongs to this SRB program. We name the set of all these functions as SRB *functions*. An SRB *program* contains one or more SRB functions.

To ensure that the recurrence is well defined, all base cases relevant to a function must appear in the function body. This simplifies the process of constructing the equations for the base cases BaseEq and recursive cases RecEq. We distinguish 3 steps in transforming SRB functions.

Given an SRB program P, we apply firstly the $\xrightarrow{\text{RS}}$ rule and then the $\xrightarrow{\text{S}}$ rule:

(Step 1) Extract the recurrence equations and base cases from the body of the functions of the program:

$$P_{\overline{\text{F}}} \rightarrow \{\text{RecEq} \cup \text{BaseEq}\}_{\overline{\text{F}}}$$

(Step 2) Apply the $\xrightarrow{\text{RS}}$ rule:

$$\{\text{RecEq} \cup \text{BaseEq}\}_{\overline{\text{F}}} \xrightarrow{\text{RS}} \left\{\bigcup_i f_i \rightarrow e_i\right\}_{\overline{\text{F}}}$$

(Step 3) Apply the $\xrightarrow{\text{S}}$ rule:

$$\left\{\bigcup_i f_i \rightarrow e_i\right\}_{\overline{\text{F}}} \xrightarrow{\text{S}} \left\{\bigcup_i f_i \rightarrow e_i'\right\}_{\overline{\text{F}}}$$

Note that at any stage we can reconstruct a program, so we can view this as

$$P \xrightarrow{\text{RS}} P\_rs \xrightarrow{\text{S}} P\_s$$

We demonstrate the three steps of the SRB strategy using the Chebyshev function:

(Step 1) Extract the recurrence equations and base cases:

$$\begin{aligned} P_{\text{cheb}_n} &\rightarrow \{\text{RecEq} \cup \text{BaseEq}\}_{\text{cheb}_n} \\ &\rightarrow \{\{\text{cheb}_n = 2 * a * \text{cheb}_{n-1} - \text{cheb}_{n-2}\} \\ &\quad \cup \{\text{cheb}_0 = 1, \text{cheb}_1 = a\}\}_{\text{cheb}_n} \\ &= \{\text{cheb}_n = 2 * a * \text{cheb}_{n-1} - \text{cheb}_{n-2}, \\ &\quad \text{cheb}_0 = 1, \text{cheb}_1 = a\}_{\text{cheb}_n} \end{aligned}$$

(Step 2) Applying the $\xrightarrow{\text{RS}}$ rule gives:

$$\left\{\text{cheb}_n \rightarrow \frac{1}{2}((a - \sqrt{-1+a^2})^n + (a + \sqrt{-1+a^2})^n)\right\}_{\text{cheb}_n}$$

(Step 3) Applying the $\xrightarrow{\text{S}}$ rule gives:

$$\left\{\text{cheb}_n \rightarrow \frac{1}{2}((a - \sqrt{-1+a^2})^n + (a + \sqrt{-1+a^2})^n)\right\}_{\text{cheb}_n}$$

Note that the application of the $\xrightarrow{\text{S}}$ rule has no effect here, since the result from the $\xrightarrow{\text{RS}}$ cannot be (further) simplified. The equivalent transformed program is:

$$\text{cheb\_s}(n, a) \texttt{ = } \tfrac{1}{2}((a - \sqrt{-1+a^2})^n + (a + \sqrt{-1+a^2})^n);$$

3.2 Strategy for MRB functions

An MRB function is a recursive function mrb with one or more base cases and with more than one recursive branch, containing at least one recursive call for each branch. The template is:

$$\text{mrb}(x) \texttt{ = if } c_0(x) \texttt{ then } b_0(x) \texttt{ else if } c_1(x) \texttt{ then } b_1(x) \texttt{ else if } d_0(x) \texttt{ then rec\_calls1}(x) \texttt{ else if } d_1(c) \texttt{ then rec\_calls2}(x) \texttt{ ...}$$

where the `rec_calls1(x)`, `rec_calls2(x)`, ... expressions have the same forms as in SRB functions with the limitation that the recursive calls are only self and not mutual recursive calls. We name the set of all these functions as *MRB functions*. A program formed by at least one MRB function is called an *MRB program*. Note that there is no way to transform MRB programs into SRB programs, so the MRB functions must be treated as a separate case.

The strategy for MRB functions is to consider each of the recursive branches in a specific order. This order is given by ordering the continuous domains over which a particular recursive branch applies. For each of these branches, we apply the $\xrightarrow{\text{RS}}$ and $\xrightarrow{\text{S}}$ rules separately, folding the results from the previous solution into the next.

Given an MRB program P, we transform each MRB function f_i using the following strategy:

(Step 1) Extract a set of pairs $\left\langle \text{CondI}_i, \text{RecEq}_i^{k_i} \right\rangle$, where CondI_i represents the semi-closed interval over which a particular recurrence equation applies, and $\text{RecEq}_i^{k_i}$ represents the recurrence equation. Also extract the base cases from the function:

$$P \rightarrow \left\langle \left\{ \left\langle \text{CondI}_i, \text{RecEq}_i^{k_i} \right\rangle \right\}, \text{BaseEq} \right\rangle$$

Note that here we annotate the recurrence equation with the symbol k_i, which is the order of the recurrence for this branch (the number of initial conditions needed to solve the recurrence). The domain of an MRB function is the union of all the CondI_i intervals. We may have different conditions with the same $\mathrm{RecEq}_i^{k_i}$ expression. We can have simple constraints, for example $\{n \mid n \leq 5\}$, or more complex conditionals, such as $\{x \mid x^2 - 4x + 2 > 0\}$. In the case of a split interval, we generate two $\left\langle \mathrm{CondI}_i, \mathrm{RecEq}_i^{k_i} \right\rangle$ pairs.

(Step 2) Order the elements in the set $\left\{ \left\langle \mathrm{CondI}_i, \mathrm{RecEq}_i^{k_i} \right\rangle \right\}$ by ordering the intervals CondI_i. Determine the cardinality (or length) $\mathrm{card}(\mathrm{CondI}_i)$ of each interval CondI_i.

(Step 3) The following algorithm performs the transformation:

```
foreach ⟨ CondI_i, RecEq_i^{k_i} ⟩ (in order from step 2)
 if (i = 1) then
  { RecEq_i^k ∪ BaseEq }_F
        --RS-->--S--> E_1
 else
  { RecEq_i^k ∪ {f_{i-1} = E_{i-1}(i-1), ..., f_{i-k} = E_{i-i}(i-k)} }_F
        --RS-->--S--> E_i ;
```

In this algorithm we make a reasonable assumption that the order of the recurrence equations is less then the cardinality of the intervals CondI_i. When this assumption is not satisfied, the algorithm requires a search for all E_i expressions in order to compute all the base cases. It is for this reason that we calculate the cardinality of the intervals in step 2.

We demonstrate the steps using `f1`, an example of an MRB program:

```
f1(n) = if (n = 0) then 1
        else if (n = 1) then 1
        else if (n < 50) then f1(n-1) + f1(n-2)
        else if (n < 100) then 2 * f1(n-1) + n
        else f1(n-2) + n;
```

(Step 1) Extract the base cases and the recursive equations. The base cases are $\mathrm{RecEq} = \{\mathrm{f1}_0 = 1, \mathrm{f1}_1 = 1\}$, and we then find all the domains of the branches:

$$\begin{array}{ll} \mathrm{CondI}_1 = \{n \in \mathbb{N} \mid n < 50 \wedge n \notin \{0,1\}\} & \rightarrow [2, 50) \\ \mathrm{CondI}_2 = \{n \in \mathbb{N} \mid n < 100 \wedge n \notin (I_1 \cup \{0,1\})\} & \rightarrow [50, 100) \\ \mathrm{CondI}_3 = \{n \in \mathbb{N} \mid n \notin (I_1 \cup I_2 \cup \{0,1\})\} & \rightarrow [100, \infty) \end{array}$$

and hence we have the following set of recurrences associated with the domains:

$$\left\{ \left\langle \mathrm{CondI}_i, \mathrm{RecEq}_i^{k_i} \right\rangle \right\}$$
$$\downarrow$$
$$\left\{ \begin{array}{llll} \langle (\mathrm{CondI}_1) & [2, 50), & \mathrm{f1}_n = \mathrm{f1}_{n-1} + \mathrm{f1}_{n-2} \rangle, \\ \langle (\mathrm{CondI}_2) & [50, 100), & \mathrm{f1}_n = 2 * \mathrm{f1}_{n-1} + n \rangle, \\ \langle (\mathrm{CondI}_3) & [100, \infty), & \mathrm{f1}_n = \mathrm{f1}_{n-2} + n \rangle \end{array} \right\}$$

(Step 2) The set ordering is $\mathrm{CondI}_1 < \mathrm{CondI}_2 < \mathrm{CondI}_3$.

(Step 3) We then apply successive folding with the $\xrightarrow{\mathrm{RS}}$ and $\xrightarrow{\mathrm{S}}$ rules:

$$\begin{array}{l} \left\{ \mathrm{RecEq}_1^{k_1} \cup \mathrm{BaseEq} \right\}_{\overline{\mathrm{F}}} \xrightarrow{\mathrm{RS}} \xrightarrow{\mathrm{S}} E_1 \\ \quad \downarrow \\ \left\{ \mathrm{RecEq}_2^{k_2} \cup \{\mathrm{f1}_{49} = E_1(49)\} \right\}_{\overline{\mathrm{F}}} \xrightarrow{\mathrm{RS}} \xrightarrow{\mathrm{S}} E_2 \\ \quad \downarrow \\ \left\{ \mathrm{RecEq}_3^{k_3} \cup \{\mathrm{f1}_{99} = E_2(99), \mathrm{f1}_{98} = E_2(98)\} \right\}_{\overline{\mathrm{F}}} \xrightarrow{\mathrm{RS}} \xrightarrow{\mathrm{S}} E_3 \end{array}$$

Each mapping E_i is of the form $\{\mathrm{f1}_i \rightarrow e_i\}$, and the equivalent transformed program is:

```
f1_s(n) =
 if (n < 50) then
   1/10 (-(1/2 (1 - √5))^n)(-5 + √5) + (1/2 (1 + √5))^n (5 + √5)
 else if (n < 100) then -2 + c1 2^n - n
 else
   c2 - c3 (-1)^n - 0.375 (-1)^{2n} + 0.75 n - 0.25 (-1)^{2n} n + 0.25 (n)^2 ;
```

Experiments show that the runtime of the final transformed program `f1_s(n)` is around 1-2 microseconds, for values of n ranging between 1 and 1000. By contrast, the original program took more than 196 seconds to calculate `f1(75)`. The implementation has been done in C and we used the built-in `pow` function. For future research, we want to extend the class of the MRB programs to include mutual recursive functions as well.

3.3 Extension to recursive calls in conditionals (SRB-RT)

We have presented the strategies for removing recursion for two classes of programs, SRB and MRB. However, in practice, more complicated examples have recursive calls in conditionals or even as arguments to auxiliary functions. We extend the present framework by including recursive calls in conditionals for SRB programs. For the sake of simplicity, we limit the discussion to self recursive functions. Consider an SRB-RT function template, `srbrt`, which has one recursive call in the conditional `cond_rec_call`:

```
srbrt(x) = if c0(x) then b0(x)
           else if ...
           else if cond_rec_call_m then rec_call1_ℓ(x)
           else p(x);
```

where $p(x)$ is a function of the input x, and $\mathrm{cond\_rec\_call}_m$ contains one recursive call `srbrt(x-m)` of order $m \leq k$, and $m \in \mathbb{N}$. $\mathrm{rec\_call1}_\ell(x)$ has the same form as in SRB or MRB functions and has recursive calls of order $\ell \leq k$.

For simplicity we do not consider trivial cases in which programs have recursive conditionals which are either always `true` or always `false`. However, we are concerned with the cases in which the programs take multiple paths at runtime. We construct a path analysis for SRB-RT functions, and we keep track of a domain variable D which identifies those intervals that still have to be processed. D is an interval, or a union of intervals, and we initialize it to be the set of natural numbers $\mathbb{N}$ less all initial base cases. The iterative analysis terminates when $D = \emptyset$.

(Step 1) For the input $x = k$ (the order of the recurrence relation), substitute the result of evaluating the base case $b_{k-m}(k-m)$ into the recursive call in the conditional:

$$\mathrm{cond\_rec\_call}_m[b_{k-m}(k-m)/\mathrm{srbrt}(k-m)]$$

If the result is `true`, then we analyse the call $\mathrm{rec\_call1}_\ell(x)$, proceeding with step 2. If the result is `false`, then we analyse $p(x)$, proceeding with step 3.

(Step 2) Solve the recurrence relation for $\mathrm{rec\_call1}_\ell(x)$, where the BaseEq set is the set of corresponding base cases:

$$\{\mathrm{RecEq} \cup \mathrm{BaseEq}\}_{\overline{\mathrm{F}}} \xrightarrow{\mathrm{RS}} \xrightarrow{\mathrm{S}} e(x)$$

Substitute the $e(x-m)$ into the recursive call in $\mathrm{cond\_rec\_call}_m$, and solve the (in) equation:

$$\mathrm{cond\_rec\_call}_m[e(x-m)/\mathrm{srbrt}(x-m)]$$

The solution of this is a union of disjoint ordered intervals, and we identify the *least* such interval intersected with the domain D, here termed TT. The complementary set of intervals, TF, is $D \backslash \mathrm{TT}$. Thus we can conclude that, $\forall x \in \mathrm{TT}$, the branch taken is the `then` branch. Update the domain $D = \mathrm{TF}$, and if $D \neq \emptyset$ then go to step 3; otherwise the algorithm terminates.

(Step 3) Substitute $p(x-m)$ into the recursive call in the conditional cond_rec_call_m, and solve the (in) equation:

$$\text{cond\_rec\_call}_m[p(x-m)/\text{srbrt}(x-m)]$$

The solution of this is again a union of disjoint ordered intervals, and we identify the *least* such interval intersected with the domain D, here termed FT. The complementary set of intervals, FF, is $D \backslash \text{FT}$. Thus we can conclude that, $\forall x \in \text{FF}$, the branch taken is the `else` branch. Update the domain $D = \text{FF}$, and if $D \neq \emptyset$ then go to step 2; otherwise the algorithm terminates.

```
f2( n ) = if (n = 0) then 1
          else if (f2(n − 1) > 10) then f2(n − 1) + 4
          else 2 ∗ n;
```

We demonstrate these steps using `f2`, an example of an SRB-RT program, first initializing the domain to be $D = \mathbb{N}\backslash\{0\}$.

(Step 1) For $n = 1$, substitute `f2(1-1)` with 1 in the inequality $(\text{f2}(0) > 10)$. The result is $(1 > 10) \rightarrow$ false, and we apply step 3.

(Step 3) Solving $2 * (n-1) > 10$ over $\mathbb{N}$ results in $n > 6$, thus $\text{FT} = n > 6$, and $\text{FF} = n \leq 6$. We now update the domain with FT, resulting in $D = n > 6$. We move on, considering the other branch, using step 2.

(Step 2) Use the $\xrightarrow{\text{RS}}$ and $\xrightarrow{\text{S}}$ rules on the `then` branch:

$$\begin{aligned}
P_{\text{f2SRB}} &\\
&\rightarrow \quad \{\text{RecEq} \cup \text{BaseEq}_{\text{then}}\}_{\text{f2SRB}_n} \\
&\rightarrow \quad \{\text{f2SRB}_n = \text{f2SRB}_{n-1} + 4, \text{f2SRB}_6 = 12\}_{\text{f2SRB}_n} \\
&\xrightarrow{\text{RS}} \quad \{\text{f2SRB} \longrightarrow 4(-3+n)\} \\
&\xrightarrow{\text{S}} \quad \{\text{f2SRB} \longrightarrow 4(-3+n)\}
\end{aligned}$$

Solving $4 * (-3+n-1) > 10$ over $\mathbb{N}$ results in $n \geq 7$, thus $\text{TT} = n \geq 7$, and $\text{TF} = \emptyset$. We now update the domain with TF, resulting in $D = \emptyset$, and our algorithm will terminate. The final transformed program is:

```
f2_s( n ) = if (n = 0) then 1
            else if n ≤ 6 then 2 ∗ n
            else 4 ∗ (−3 + n);
```

4. Recursive functions with multiple recursion parameters

We saw in the previous sections recurrences of one parameter which may be solved in a generic way. A challenging problem is the extension to recurrences with more than one parameter, corresponding to recursive functions with multiple recursion parameters. Consider this example function `fib2`:

```
fib2( m, n ) = if ((m ≤ 1) or (n ≤ 1)) then 1
               else fib2(m − 1, n − 1) + fib2(m − 2, n − 2);
```

Both of the parameters are recursion parameters, but it is possible to reduce it to a recurrence with a single parameter, and derive an equivalent non-recursive program:

```
fib2_s( m, n ) =
    1/2(1 + √5/5)((1+√5)/2)^min(m,n) + 1/2(1 − √5/5)((1−√5)/2)^min(m,n);
```

Transforming functions with multiple recursion arguments such as these has always been difficult. Some existing methods are available to transform such a program into one with linear complexity. Synchronisation analysis [9], in particular, classifies multiple recursion arguments as level-3 synchronisations, and guarantees that a linear-complexity program can be obtained via the multiple-recursion tupling technique.

The notation $\overline{n} = (n_1, \ldots, n_s)$ denotes an s-tuple of positive integers, and the notation $\overline{f} = (f_1, \ldots, f_s)$ an s-tuple of functions given by $\overline{f(\overline{n})} = (f_1(n_1), \ldots, f_s(n_s))$. We say that $\overline{f}$ is an s-tuple (or vector) of strictly monotonically decreasing functions iff $\forall i \in \{1, \ldots, s\}$ each f_i is a strictly monotonically decreasing function. f_i is a strictly decreasing function iff $f_i(a) > f_i(b)$, $\forall a < b$. Successive self-application of a function is denoted by $f^{(\ell)}$. $\overline{f(\overline{n})}$ is extended to $\overline{f^{(\ell)}(\overline{n})} = (f_1^{(\ell)}(n_1), \ldots, f_s^{(\ell)}(n_s))$.

We say that a recurrence relation is in *s-tuple normal form* iff it has the form:

$$X_{\overline{n}} = F(X_{\overline{f(\overline{n})}}, X_{\overline{f^{(2)}(\overline{n})}}, \ldots, X_{\overline{f^{(\ell)}(\overline{n})}}) \tag{4}$$

where F is an expression of one of the kinds presented in previous sections.

THEOREM 1. *Consider an s-tuple normal form recurrence of type (4) such that $\overline{f}$ is a vector of strictly monotonically decreasing functions, and F is solvable in one parameter. Then, there exists a solution of (4) which can be expressed in a closed form.*

PROOF. The key idea to solve the s-tuple recurrence of type (4) is to reduce it to a recurrence with one parameter. We substitute the s-tuple $(n_1, \ldots n_s)$ with t, and extend this for all arguments, so $(f_1^{(1)}(n_1), \ldots, f_s^{(1)}(n_s))$ becomes $t-1$ and so on until $(f_1^{(k)}(n_1), \ldots, f_s^{(k)}(n_s))$ becomes $t-k$, for any positive integer k. By doing this substitution, the recurrence relation (4) can be reduced to $Y_t = F(Y_{t-1}, \ldots, Y_{t-\ell})$. According to the hypothesis, F has a solution which can be expressed in a closed form. Although this conversion was relatively easy, coming back from the generic solution of Y to a generic solution of X is not simple at all. Denote by $\text{sol}_1(n_1), \ldots, \text{sol}_s(n_s)$ the solutions of equations $f_1^{(k)}(n_1) = 0$ up to $f_s^{(k)}(n_s) = 0$ solved in variable k, respectively. Since $f_1, \ldots, f_s$ are strictly monotonically decreasing functions, the values $f_1^{(k)}(n_1), \ldots, f_s^{(k)}(n_s)$ are decreasing to 0. Let us consider k such that $t-k = 0$. Then, there exists $i \in \{1, \ldots, s\}$ such that $f_i^{(k)}(n_i) = 0$ and

$$\begin{aligned}
&f_1^{(k)}(n_1) \geq 0, \ldots, f_{i-1}^{(k)}(n_{i-1}) \geq 0, \\
&f_{i+1}^{(k)}(n_{i+1}) \geq 0, \ldots, f_s^{(k)}(n_s) \geq 0
\end{aligned}$$

The solution of $f_i^{(k)}(n_i) = 0$ is $k = \text{sol}_i(n_i)$. We prove now that

$$\begin{aligned}
&k \leq \text{sol}_1(n_1), \ldots, k \leq sol_{i-1}(n_{i-1}), \text{ and} \\
&k \leq sol_{i+1}(n_{i+1}), \ldots, k \leq sol_s(n_s)
\end{aligned}$$

Suppose, by *reductio ad absurdum*, that $\text{sol}_1(n_1) < k$. According to the monotony of f_1, $f_1^{(\text{sol}_1(n_1))}(n_1) > f_1^{(k)}(n_1)$. This implies $0 > f_1^{(k)}(n_1) \geq 0$, which is a contradiction. So, $k \leq \text{sol}_1(n_1)$. Similarly, all the other inequalities can be proved. Let us denote by $F_{a,b}(\overline{n})$ the expression obtained from $f_b^{(k)}(n_b)$ by replacing k with $\text{sol}_a(n_a)$, for all $b \in \{1, \ldots, s\}$. Then, we take $t = k = \text{sol}_i(n_i)$ and

$$Y_0 = X_{F_{i,1}(\overline{n}), \ldots, F_{i,i-1}(\overline{n}), 0, F_{i,i+1}(\overline{n}), \ldots, F_{i,s}(\overline{n})}$$

So, the generic solution of F can be done by providing t as the minimum of the solution set $\text{sol}_1(n_1), \ldots, \text{sol}_s(n_s)$. □

Consider the example function `fib1` given in Section 1. The corresponding recurrence is $X_{m,n} = X_{m-1,n-2} + X_{m-2,n-4}$, and as in Theorem 1, we do the substitution of vector (m,n) by t. Therefore, the vector $(m-1, n-2)$ is replaced by $t-1$, and so on until $(m-k, n-2\cdot k)$ is replaced by $t-k$. This implies that we have an *equivalent* one parameter recurrence relation $Y_t = Y_{t-1} + Y_{t-2}$.

Name	Type and improvement	Inputs	F_rec (mS)	F_rs (μS)	F_s (μS)
U(n) (Domino Puzzle)	SRB $\mathcal{O}(e^n) \rightarrow \mathcal{O}(\log n)$	10	0.003	1.993	1.755
		323	0.648	2.576	2.035
cheb(n,a) (Chebyshev)	SRB $\mathcal{O}(e^n) \rightarrow \mathcal{O}(\log n)$	15, 1	0.033	1.866	1.866
		25, 1	4.052	1.883	1.883
f1(n)	MRB $\mathcal{O}(e^n) \rightarrow \mathcal{O}(\log n)$	45	29,465.390	1.960	1.667
		75	196,457.510	1.591	1.531
fib2(m,n)	MULT-SRB $\mathcal{O}(e^{\min(m,n)}) \rightarrow \mathcal{O}(\log \min(m,n))$	20, 30	0.026	1.727	1.630
		40, 60	33.690	1.785	1.723
f2(n)	SRB-RT $\mathcal{O}(e^n) \rightarrow \mathcal{O}(1)$	10	0.004	1.379	1.366
		25	103.561	1.389	1.376

Table 2. Table showing relative times for recursive and transformed programs. `F_rec` is the original program, `F_rs` is the program transformed using the $\xrightarrow{\text{RS}}$ rule, and `F_s` is the program transformed further using the $\xrightarrow{\text{S}}$ rule. Note that different columns have different time units.

This can be solved, giving

$$\begin{aligned} Y_t \quad \rightarrow \quad & \frac{1}{2}((1-\frac{\sqrt{5}}{5})\cdot Y_0 + \frac{2\sqrt{5}}{5}\cdot Y_1)(\frac{1+\sqrt{5}}{2})^t \\ & + \frac{1}{2}((1+\frac{\sqrt{5}}{5})\cdot Y_0 - \frac{2\sqrt{5}}{5}\cdot Y_1)(\frac{1-\sqrt{5}}{2})^t \end{aligned}$$

In order to get the closed form for the recurrence $X_{m,n}$, we apply Theorem 1, by replacing $t = min(m, \frac{n}{2})$, and

$$\begin{aligned} Y_0 = X_{max(m-\frac{n}{2},0),max(0,n-2\cdot m)} &= 1 \\ Y_1 = X_{max(m-\frac{n}{2},1),max(1,n-2\cdot m)} &= 1 \end{aligned}$$

The equivalent transformed program is:

```
fib1_s(m,n) =
  1/2(1 + √5/5)((1+√5)/2)^min(m,n/2) + 1/2(1 − √5/5)((1−√5)/2)^min(m,n/2);
```

5. Prototype and results

Our automated prototype system, `APTSR1` (which stands for Automatic Program Transformer by Solving Recurrences), is an initial attempt to implement the technique, using the `MATHEMATICA` computer algebra system [25], as an embedded engine to automatically solve recurrences.

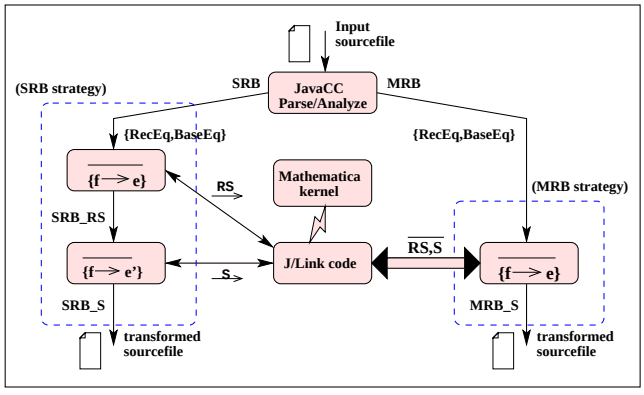

Figure 1. System architecture of `APTSR1`.

Figure 1 shows the software architecture of the tool, linking program analysis, classification, and the `MATHEMATICA` kernel to solve the recurrences for both SRB and MRB forms, leading to a transformed source file. `JavaCC` is a parser generator, and in `APTSR1` it is used to automatically parse the input sources, and then extract the recursive equations and base cases. After classification, the code uses `J/Link` (a Java API from Wolfram Research) to connect with the kernel.

Input n	fib	fib3[17]	fib_s
20	0.910	0.00154	0.00185
40	13, 773	0.00321	0.00186
60	> 200, 000	0.00480	0.00187
80	> 200, 000	0.00643	0.00187
100	> 200, 000	0.00805	0.00189
120	> 200, 000	0.00972	0.00190
140	> 200, 000	0.01139	0.00192
n	$\sim \mathcal{O}(e^n)$	$\sim \mathcal{O}(n)$	$\sim \mathcal{O}(1)$

Table 3. Runtime (in mS) of Fibonacci code in C

A clear demonstration of the improvement is seen in Table 2, which compares the runtimes of the recursive and the automatically transformed versions of the programs mentioned in this paper. The programming language in each case is C, and we used the same platform for each test: a lightly loaded *linux* cluster. The table shows significant improvements in the runtime of the algorithms, reflecting the reduction from exponential complexity achieved by our transformation. Note for example the change from (say) U(10) to U(323), where the recursive program takes 200 times longer, whereas the closed-form version shows only a marginal increase in time.

We also compare (in Table 3) our implementation `fib_s(n)` with that done in [17], an iterative Fibonacci function obtained by the incrementalization method. Our (near) constant time implementation increases more slowly as n increases. The function `fib_s` is:

```
fib_s(n) =
  1/10(−(1/2(1 − √5))^n)(−5 + √5) + (1/2(1 + √5))^n (5 + √5);
```

The function (`fib3`) from [17] is repeated here for comparison:

```
if (n == 0) then { i = 0; a = 1; }
else if (n == 1) then { i = 1; a = 1; }
else { i = 2; a = 2; b = 1; };
while (i != n) {
   i = i + 1; a' = a; b' = b; a = a' + b'; b = a';
}
return a;
```

6. Conclusion

The contribution of this paper is two-fold. Firstly, we describe a new framework for transforming programs by removing recursion entirely, resulting in new programs without recursion, normally

called *closed forms*. Consequently, new programs run in linear or logarithmic times, if not less. Secondly, (perhaps the most challenging problem) we extend the framework to handle recurrences corresponding to recursive functions with multiple recursive parameters.

There are many program transformation strategies which transform recursive programs of exponential complexity into new programs of linear complexity [1, 2, 3, 4, 6, 7, 13, 14, 18, 19, 21, 24]. Futamura's method for removing recursion from linear recursive programs was recently further extended to deal with recursive programs with one descent function [15]. Our framework differs from these strategies in that we discover the closed forms of these recursive programs. Our technique exploits symbolic computation during transformation. More notably, the transformation is *not* syntax-directed, and relies on recurrence analysis to derive the desired solutions. We are not aware of any systematic syntax-directed techniques of transforming programs with exponential complexity into ones with logarithmic complexity. An ad-hoc method can be found in a footnote of the first-year undergraduate textbook *Structure and Interpretation of Computer Programs*.

Only a few existing program transformation strategies, such as [9, 10, 18] can handle recursive programs with multiple recursion parameters. In this paper, we presented a novel technique for converting a class of such programs into linear recurrences, and recovering the desired solutions from those of their linear recurrences. While the class of programs corresponds to the notion of *level-3 synchronization* from [10], the final transformed programs, of logarithmic complexity, are definitely more efficient than those produced by the techniques described in that work. We intend to extend the class of recursion programs with multiple recursion parameters to include different levels of synchronization, as defined in [10].

Our transformations are defined in as generic a manner as possible, and so are reusable across a wide range of languages. Our future research will focus on providing the user with options for different degrees of automation. For example, given an MRB program P supporting an interface which provides a contract (expressed as a set of input constraints $\{C_i\}$, ie., pre-conditions) to the user, P can be *selectively specialized* based on the contract. That is, only those branches in P which follow from a satisfiable $\{C_i\}$ are subject to the application of MRB transformation, while the other branches are eliminated. This will provide an opportunity for synergy between the technique of partial evaluation and MRB transformation.

Acknowledgments

We would like to thank the anonymous referees for their valuable comments. This research is partially supported by the NUS research grant R-252-000-138-112.

References

[1] M. Abadi, B. W. Lampson, and J-J Levy. Analysis and Caching of Dependencies. In *International Conference on Functional Programming*, pages 83–91, 1996.

[2] R. S. Bird. Tabulation Techniques for Recursive Programs. *ACM Comput. Surv.*, 12(4):403–417, 1980.

[3] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[4] W.N. Chin. *Automatic Methods for Program Transformation*. PhD thesis, Imperial College, University of London, 1990.

[5] W.N. Chin. Towards an automated tupling strategy. In *PEPM '93: Proceedings of the 1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 119–132, New York, NY, USA, 1993. ACM Press.

[6] W.N. Chin and M. Hagiya. A Transformation Method for Dynamic-sized Tabulation. *Acta Informatica*, 32:93–115, 1995.

[7] W.N Chin and M. Hagiya. A Bounds Inference Method for Vector-Based Memoisation. In *ICFP*, pages 176–187, 1997.

[8] W.N. Chin and S.C. Khoo. Tupling Functions with Multiple Recursion Parameters. In *WSA '93: Proceedings of the Third International Workshop on Static Analysis*, pages 124–140, London, UK, 1993. Springer-Verlag.

[9] W.N. Chin, S.C. Khoo, and T.W Lee. Synchronisation Analysis to Stop Tupling. *Lecture Notes in Computer Science*, 1381:75–89, 1998.

[10] W.N. Chin, S.C. Khoo, and N.Jones. Redundant Call Elimination via Tupling. *Fundamenta Infomaticae*, 69(1-2), 2006.

[11] A. Fournier and J. Buchanan. Chebyshev Polynomials for Boxing and Intersections of Parametric Curves and Surfaces. *Computer Graphics Forum (Eurographics '94 Conference Issue)*, 13(3):127–142, 1994.

[12] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: a Foundation for Computer Science*.

[13] Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling Calculation Eliminates Multiple Data Traversals. In *Proceedings 2nd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'97, Amsterdam, The Netherlands, 9–11 June 1997*, volume 32(8), pages 164–175. ACM Press, New York, 1996.

[14] J. Hughes. Lazy memo-functions. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 129–146, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[15] Y. Ichikawa, Z. Konishi, and Y. Futamura. Recursion Removal from Recursive Programs with Only One Descent Function. *IEICE Transactions*, 88-D(2):187–196, 2005.

[16] C. Julien, J. Payton, and G-C. Roman. Adaptive Access Control in Coordination-Based Mobile Agent Systems. In *SELMAS*, pages 254–271, 2004.

[17] Y. A. Liu and S. D. Stoller. From Recursion to Iteration: What are the Optimizations? In *PEPM '00: Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 73–82, New York, NY, USA, 1999. ACM Press.

[18] Y. A. Liu, S. D. Stoller, and T Teitelbaum. Static Caching for Incremental Computation. *ACM Transactions on Programming Languages and Systems*, 20(3):546–585, 1 May 1998.

[19] D. Michie. 'Memo' Functions and Machine Learning. *Nature*, 218:19–22, 1968.

[20] Marko Petkovšek. Hypergeometric Solutions of Linear Recurrences with Polynomial Coeffi cients. *J. Symb. Comput.*, 14(2-3):243–264, 1992.

[21] A. Pettorossi. Transformation of Programs and Use of Tupling Strategy. In *Informatica 77*, pages 1–6, 1977.

[22] A. Pettorossi and M. Proietti. Future Directions in Program Transformation. *ACM Computing Surveys*, 28(4es):171–171, 1996.

[23] A. Pettorossi and M. Proietti. Program derivation via list introduction. In *Proceedings of the IFIP TC 2 WG 2.1 international workshop on Algorithmic languages and calculi*, pages 296–323, London, UK, UK, 1997. Chapman & Hall, Ltd.

[24] W. Pugh and T. Teitelbaum. Incremental Computation via Function Caching. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 315–328, New York, NY, USA, 1989. ACM Press.

[25] Wolfram Research. `http://www.wolfram.com/`.

[26] The MPFR team. `http://www.mpfr.org/`.