# Run Transferable Libraries — Learning Functional Bias in Problem Domains

Maarten Keijzer[1], Conor Ryan[2], and Mike Cattolico[3]

[1] Prognosys, Utrecht, The Netherlands mkeijzer@xs4all.nl
[2] University of Limerick, Ireland conor.ryan@ul.ie
[3] Tiger Mountain Scientific Inc., Kirkland, WA USA mike@tigerscience.com

**Abstract.** This paper introduces the notion of Run Transferable Libraries, a mechanism to pass knowledge acquired in one GP run to another. We demonstrate that a system using these libraries can solve a selection of standard benchmarks considerably more quickly than GP with ADFs by building knowledge about a problem. Further, we demonstrate that a GP system with these libraries can scale much better than a standard ADF GP system when trained initially on simpler versions of difficult problems.

## 1 Introduction

This paper introduces the concept of Run Transferable Libraries (RTLs) that overturns the traditional notion of independent runs in GP. Instead of executing a large number of independent runs, in each of which GP attempts to solve a problem or learn a target function from scratch, a system using RTL accumulates information from run to run, learning more about the problem area each time it gets applied. The method is partly inspired by the use of libraries in conventional programming, where they are used to capture common functionality for a certain *domain* of functionality. Similarly, an RTL enabled search method uses libraries that provide functionality that is valid in *problem domains*. However, unlike standard programming libraries which tend to be static, RTL enabled search methods evolve their libraries over time and update them based on past experience. It is thought that by using Run Transferable Libraries it is possible for a search strategy such as GP to become *competent* in solving more difficult problems than can be tackled currently.

This approach can have major implications for the scalability of a system. Not only is it possible for an RTL to learn from run to run on a single problem instance, becoming more adept at solving the problem at hand, it can also be trained on "simple" problem instances, in order to tackle more difficult problems later on. A third use of RTLs is to train a library to become competent in a problem domain where initial runs are unsuccessful, tackling same complexity problems with more and more success as the RTL enabled system becomes more experienced. In effect, by training a library on different problem instances, we hope to learn the underlying bias that makes the set of problems a coherent

K. Deb et al. (Eds.): GECCO 2004, LNCS 3103, pp. 531–542, 2004.

whole (a problem domain). The bias that an RTL method tries to learn is of a functional nature, a set of functions that help in solving related problems faster. It is hypothesised that to do this an RTL approach needs to have:

- the ability to learn to tackle a difficult problem by using information gathered from previous, possibly unsuccessful attempts;
- the ability to apply information learned from a simple instance of a problem set to a more difficult instance;
- the ability to learn how to efficiently solve problems taken from the same problem domain.

We demonstrate that the current implementation of RTL, our system *Mnemosyne*[1], can, even when applied to a *single* instance of problems such as 4- and 5- parity and a small version of the Lawnmower problem, learn about the problem so quickly that it outperforms GP (with ADFs enabled) in terms of computational effort, especially on larger versions of the problem. It is also demonstrated that we can *unbias* an initial function set used for logical functions by training on randomly generated instances of logical functions. When applying a library trained in such a way to the parity problem, it shows a decrease in computational effort, albeit not so large as when the library is trained on parity problems themselves.

## 2 Scaling and Problem Domains in GP

Traditionally, AI has been divided into two fundamentally different approaches, *strong* and *weak*. Strong AI usually involves capturing as much prior knowledge as possible about a problem before attempting to solve it, while Weak AI is concerned with the application of more general methods that can be used across a wide spread of problems. This is possible because Weak AI methods do not require prior knowledge of a problem.

GP (and Evolutionary Computation in general) has been described as a *strong-weak* method [2] because of the way in which populations capture pertinent information about a problem and recombine it to produce more useful solutions. The typical dynamic for an EC run is that the population starts out with very poor performing individuals, and slowly accumulates knowledge about the problem until a satisfactory solution is found.

Most applications of EC involve independent runs; that is, all the knowledge accumulated by each run is discarded once it terminates. Furthermore, when one attempts to apply GP to a more difficult problem, even a more difficult instance of a problem already solved, all the effort spent in previous runs is lost, and the system must start from scratch again, which has lead to concerns about the scaling abilities of GP.

Another approach to the scaling issue is to exploit the inherent modularity present in many problems, through the use of Automatically Defined Functions

[1] Mnemosyne is the mother of the Muses. She is the personification of Memory.

(ADFs) [5] or some similar structures that permit GP to decompose the problem. The following are examples are some of the more successful attempts to improve the scaling abilities of GP using one of these methods.

**ADFs.** Automatically Defined Functions were introduced by Koza [5] ostensibly as a method for decomposing problems. ADFs are essentially local functions that individuals can call, which are also subject to evolution.

Prior to running ADF GP, one typically chooses the number of ADFs each individual has, as well as their arity. Thus, each individual is made up of a *result producing branch* (RPB) which is essentially a main program, and one or more ADFs, each of which can be called by the RPBs.

Koza showed that, on decomposable problems, using ADFs appropriate for the problem significantly improved the performance of GP. Indeed, while the choice of number and structure of the ADFs effects the performance of the system, Koza showed that, on these problems, using *any* kind of ADF yielded better performance than without. Furthermore, it is possible to evolve the number and arity of ADFs, although this can lead to considerably longer evolution times.

**GLib.** Another approach to the exploitation of modularity is Angeline's [1] Genetic Library Builder (GLiB). Unlike ADFs, GLiB relies upon *subroutines* as its modularity mechanism. Subroutines in GLiB are referred to as *modules.*

Modules are created dynamically during a run, through the use of a *compression* mutation operator. Compression is probabilistically applied to a sub-tree of an individual, and compresses it into a module, which is a node with the same functionality. This has the dual effect of increasing the expressiveness of the language and decreasing the average size of individuals in the population.

A newly created module can then be passed onto later generations in the same way any other node is. The extent to which it is used by descendants depends on a combination of the utility of the module and individuals' successful use of it. Unlike ADFs, however, due to the extraction process, GLiB modules cannot reuse arguments in the modules. It has been argued that this impedes performance in GLiB [4].

GLiB also has an *expansion* mutation operator, which expands a previously compressed node. This was introduced to combat the loss of diversity that the system experienced due to the presence of the modules.

**ARL.** The Adaptive Representation through Learning(ARL) [8] system is a combination of the GLiB and ADFs. Unlike GLiB, ARL selected individuals to compress based on how well their fitness improved upon that of their parents.

ARL was shown to perform well relative to the other methods, but has been criticised [7] for requiring many control parameters, and has yet to find wide acceptance in the field.

**Subtree Encapsulation.** Each of the above schemes is concerned with producing modules and functions in parallel with the individuals that will use them. Once a run is finished, all the effort involved in producing these modules must be repeated for the next run. A different approach was taken by [7] with their *Subtree Encapsulation* method. Instead of producing modules on the fly, they harvest completed runs for useful modules which are then subsequently used to augment the terminal set of another run.

To achieve this, a database of all subtrees generated during a run is maintained. The database maintains a count of the usage of each tree, which is then used to decide which subtrees should be added to the terminal set; the most frequently used ones being encapsulated as atoms.

It was shown that their method outperformed standard GP both in terms of the speed to good solutions and to the overall quality of solution generated. Perhaps somewhat surprisingly, they also showed that simply *randomly* choosing the subtrees gave them better performance than the usage-based approach.

**Sequential Runs.** There have been several attempts to pass information from run to run, although these have all been with a view to solving a *single* problem, rather than training a system that can be applied to successively more difficult problems. The most well known of these techniques is to simply seed a population with the *best-of-run* individual from the previous run. This has been used with mixed success by many researchers, as some have found that this can lead to very premature convergence on that individual.

Beasley described the use of *derating functions* for the discovery of all peaks in a multi-modal function [3]. His system attempted to discover one or two peaks with each run, and to then modify the fitness function of subsequent runs to penalise individuals that revisited that part of the search space. He discovered that his algorithms substantially outperformed standard niching techniques on these problems, mainly because they made the problem easier. Each run was presented with fewer and fewer peaks until they were all discovered.

A similar approach was taken by Streeter et all [9] with a system that iteratively refined a circuit across several runs. This work was concerned with generating an electric circuit which produces an output that is an approximation to the *error function* of an existing circuit, which may or may not have been generated using GP. The generated circuit can be added to the existing one to produce a new one which performs its task with greater accuracy. These iterations can be repeated as many times as necessary, or until the benefit of the improvements gained is outweighed by the cost of an extra run.

## 3 Mnemosyne

The Mnemosyne system implements RTL by keeping a library that is divided into segments, each segment containing functions of a certain arity. A segment has a function and terminal set defined for it. Typically the function set for a segment contains the primitive functions defined for the problem (domain), while

```
InitLibrary;
for each library iteration do
    RunGP;
    UpdateLibrary;
end
```

**Algorithm 1:** Overall Algorithm

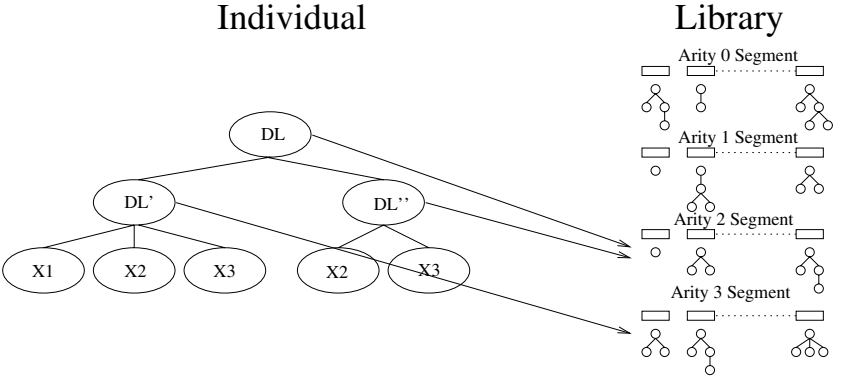


**Fig. 1.** An overview of Mnemosyne. Each DL-node consists of a floating point label which will be used to is used to look up the appropriate library segment, according to the arity of the function.

the terminal set contains the arguments for the function. Functions in segments have a floating point value associated with them, these are called *tags*. The tags are the main means for *linking* to library elements. Therefore the library elements are referred to as Tag Addressable Functions (TAFs).

A general overview of the Mnemosyne training algorithm is provided in Algorithm 1. It defines an individual GP run (**RunGP**) as a basic sub-routine that is used to collect statistics about the use of library elements. After each individual run, the library is updated using these statistics. Details on the update mechanism are provided in Section 3.1.

Individual runs use the library by virtue of the presence of a special function in the function set: a Dynamically Linked Node (DL-node). Such a node is fully specified by an arity and a *tag* value. The arity defines which segment the node links to, while the tag value defines which particular function is used, by searching for the nearest tag value in the segment. An overview of the contents of the library and how an individual links to the library is depicted in Figure 1. In the experiments below, the function set in the individual runs is limited to DL-nodes only, this in order to force the run to use the library and identify important elements, and to avoid using the bias inherently present in this set of primitive functions.

At the initialization phase of the individual runs, the DL-nodes are initialized with a uniformly chosen arity, and a tag value that is chosen uniformly from the

available tags. Thus, each TAF has an equal chance of being present in the initial population. During a run, a tag-mutation operator is used, which will mutate the tags in the DL-nodes, leading to the node being linked to (possibly) a different TAF. This mutation operator operates in conjunction with a regular subtree crossover and reproduction operator.

### 3.1 Updating the Library

TAFs consist of a tuple $\langle t, w, v, f \rangle$, where $t$ denotes the tree (usually a function with one or more arguments), $w$ the multiple run worth, $v$ the floating point tag value and $f$ the per run usage (frequency). A library consists of multiple TAFs, and is usually sorted on the tag value for quick $O(logN)$ lookup. The library is initialized with randomly generated TAFs using a library specific function and terminal set using the ramped-half-and-half method. Initially, after loading or creating the library, the per-run frequency count $f$ is set to 0. During the run, whenever TAF $i$ is linked to a tree of size $s$, the frequency is updated by:

$$f_i \leftarrow f_i + 1/s$$

Normalisation by the size of the calling tree is performed to avoid a situation where the larger individuals in the population have the deciding vote on which tafs will propagate. After the run, the frequencies are normalised to sum to one, and added to the long-term worth of the library element:

$$w_i \leftarrow (w_i + f_i / \sum_j f_j) \times d$$

The library update mechanism attempts to ensure that the frequency of occurrence of library elements is proportional to their long-term usage information. First the mean usage $\bar{w}$ for library elements is calculated. Then, each library element for which $w_i \geq 2\bar{w}$, is selected for reproduction/variation. To keep a constant library size, an individual $j$ to replace is chosen that has the lowest worth $w$. The chosen TAF $i$ will either be reproduced (50% of the time) or crossed over with a randomly chosen TAF and copied over individual $j$. Subsequently, the $w$ values of both $i$ and $j$ will be shared between the parent and the offspring, keeping the sum of $w$ constant.

$$\begin{aligned} w_j &\leftarrow \frac{w_j + w_i}{2} \\ w_i &\leftarrow w_j \end{aligned}$$

And the tag value of the offspring is mutated by adding a normal distributed random number

## 4 Experiments

We tested the system on two classes of problems often associated with ADFs, namely the Boolean *even n-parity* and the *Lawnmower* problems. The even-n-parity problem takes $n$ Boolean arguments and returns **true** if an even number of the arguments are **true**, and otherwise returns **false**. The Mnemosyne system was implemented using ECJ [6].

The difficulty of these parity problems is easily scaled by increasing the number of inputs, and have proved a very tough benchmark for standard GP. Koza used these problems to illustrate the utility of employing ADFs, and showed that ADF GP could solve a selection of them with far less computational cost and a much smaller population (4,000 vs. 16,000). Koza examined problems up to and including the even-11-parity problem, and reported that the sheer time involved in running these experiments prohibited him from going any further, as the number of test cases increase exponentially with the number of inputs.

The Lawnmower problem was specifically designed as a test bed for ADFs [5]. It is a difficult enough problem to exhibit hierarchical decompositions and exploitable regularities, the characteristics upon which ADFs thrive. The Lawnmower problem is also such that it can be scaled with much more granularity than the Boolean even-k-parity problems.

In the Lawnmower problem, the aim is to produce a program that controls the movement of a lawn mower such that the mower cuts all the grass on a particular lawn. The lawn is laid out as a toroidal grid, and the lawn mower can occupy one grid location at a time. Koza examined problems from 32 up to 96 squares and demonstrated that, while ADFs aren't necessary to solve the problem, they do so much more quickly, and scale much better.

To tackle these problems, the usual syntactic distinction between terminals (leaf nodes) and functions (internal nodes) is abandoned and a more functional distinction of variables and functions is used: variables are considered to be problem dependent entities that are not transferable between different problem instances, while functions are a set of operations that are valid throughout a problem domain. This distinction allows for functions of arity 0 (e.g., sensors) to be considered different entities from variables (also of arity 0) as found in the parity problems. In the experiments, functions will be placed in the library, while variables (that are not supposed to be transferable) are used in the individual GP runs. The library will only use functions, while the evolving population will be constrained to use the library and the variables.

All experiments were done with the same set of parameters: libraries and individual runs are initialised using the ramped-half-and-half method (ramp varying between 2 and 6), population and library segment size is set to 500, the individual runs run for 50 generations, the decay factor in the library is set to 0.9, crossover probability in the library equals 0.5 (meaning that half of the variation events inside the library are straight reproduction events), the individual runs use 10% reproduction, 45% crossover and 45% tag mutation. The parameters have been set using a few initial runs, and are possibly not very optimal.

To compare the results of the library against results taken from the literature, the concept of *conditional effort* is introduced. The calculation of the conditional effort is the same as the normal effort, the difference being that it is conditional on the library iterations that have already been performed. After training a library, it will be tested on a new set of problems; and only this new set of problems is used to calculate the effort. To recognise the effort that has already been put in the library, the number of iterations the library has been trained is reported as well. This was necessary as the normal *number of individuals processed* statistic has not enough granularity to make distinctions between solving problems of varying difficulty. Also, in practice, we view the creation of a library not as an event that needs to be done for every problem we encounter, but advocate re-use of libraries to tackle new problems. Therefore, we view the effort to create the library to be on the same footing as changing other parameters, particularly tuning function sets to a particular domain.

### 4.1 Boolean Even-n-parity Experiments

The central thesis of this paper is that RTLs permit GP runs to transfer knowledge in the form of functions, not only from run to run, but also to transfer knowledge across increasingly difficult problems in the same domain. Thus, our baseline experiments involve comparing Mnemosyne to ADF GP on a single problem instance. A second set of experiments was designed to test the ability of the system to transfer knowledge from run to run. To this end, an RTL was generated ("trained") on one instance of the problem, and subsequently applied to problems of increasing difficulty. Two sets of experiments were carried out in this case; one for an RTL initially trained on the even-4-parity problem, and subsequently applied to 5- onto 10-parity, and another in which the RTL was initially trained on the even-5-parity problem, and subsequently applied to 6- to 10-parity problems.

For the boolean parity problems, the function set {AND,OR,NAND,NOR} is used. It is known that this function set is very biased *against* parity problems. Overcoming this bias and converging to the optimal set of either XOR or EQ, combined with negation could be a winning strategy. The library consists of 3 segments, for functions of arities 1,2 and 3. The primitive function set that is used inside the library consists of the aforementioned four primitive logical functions. The solutions evolving in the separate runs use the DL-nodes that refer to the elements of the library in the function set; the terminal set consists of the problem-dependent variables. In this setup, it is impossible to encode solutions for individual problems in the library. The library needs to evolve in such a way that it helps the individual runs to solve the problems.

### 4.2 Lawnmower Problems

A similar experimental set up was used for the Lawnmower problem. An initial library was trained on an 8x8 lawn, and then tested on progressively larger lawns, up to 8x12, the largest size reported on by Koza in [5]. The library consisted

of the the standard functions for this problem, that is { LEFT,MOW,V8A, PROGN2, FROG } while the evolving population was made up of DL-nodes. Also for this problem, the library consists of three segments, here of arity 0,1, and 2. The library and GP parameters were the same as for the parity problem.

## 5 Results

Figure 2 shows the results for training the library on the parity 4 and 5 problems and subsequently testing them on higher parities. Fifty independent libraries were trained on the lower parity problems for 95 iterations. The end-of-run libraries thus consisted of feedback obtained from 95 *dependent* runs, i.e., runs that use updated libraries. The libraries produced after every fifth iteration were saved and tested against the higher parity problems. Libraries with 0 iterations are randomly initialised. Thus, a test-run at library iteration 15, tests a library that was trained using 15 different runs of the smaller parity problem. Figure 2 shows a clear improvement, where the conditional effort of solving the parity problems decreases with library iterations. It is also clear that the library is capable of generalising: libraries trained on parity 4 have a better expected performance on the higher parities than untrained libraries. When comparing the libraries trained on parity 4 with those trained on 5, it is interesting to note that the libraries trained on the simpler problem apparently generalise better. It seems that the simpler problem allows the library to focus on the characteristics of this problem domain better. After 95 iterations of training on parity 4, applying this library on a parity 10 problem apparently seems to outperform standard GP using ADFs on parity 4 itself! Considering that parity 4 problems consist of a factor 64 less fitness cases than parity 10, it seems that the Mnemosyne RTL approach provides good scaling characteristics on this type of problem.

A cursory inspection of the content of the libraries for the segments for arity 2 shows that the library tends to converge to a majority of either the XOR or the EQ function, accompanied with low but persistent numbers of auxiliary functions. The library thus learned to remove the initial bias induced by the four primitives, and induced a bias more appropriate to this type of problem.

Figure 3 shows the computational effort needed to solve Lawnmower problems of increasing difficulty. Also here it can be seen that the computational effort decreases after longer periods of training, while the more difficult (larger) problems benefit from this increased training effort. Again, the benchmarks are easily surpassed. The second graph in Figure 3 shows a comparison of effort between a library *trained* on a lawn of 64 cells, and *tested* on a lawn of 256 cells, against a library *trained* on such a lawn of 256 cells. Although there is a fair amount of variability in the graph (this is caused by using 50 runs for calculating the effort), it is clear that at the very least the library that is trained on a smaller lawn does not have a worse performance than the library trained on the bigger lawn. Starting out with training on small and fast instances of a problem, seems to be helpful in solving the more difficult problem.

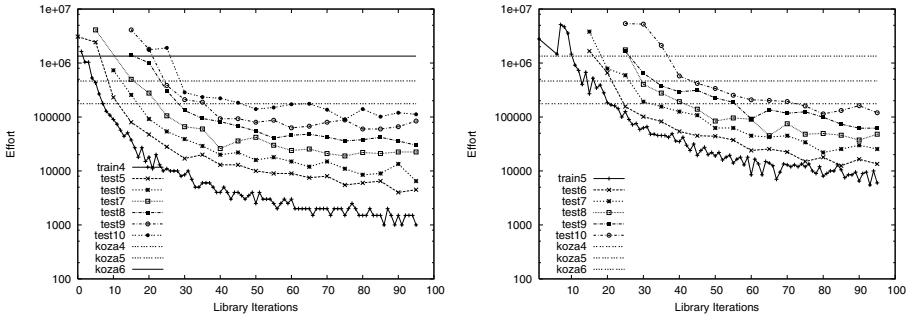


**Fig. 2.** Number of individuals needed to be processed to obtain a solution with 99% probability on the parity problems. Training was done using 50 independently trained libraries on 95 iterations, while testing was done on each library taken from every fifth iteration. The graphs show the results for libraries trained on parity 4 and 5. Also shown are the computational efforts as published by Koza using ADFs [5]. Runs that did not produce any solutions are not plotted. Note the log scale.

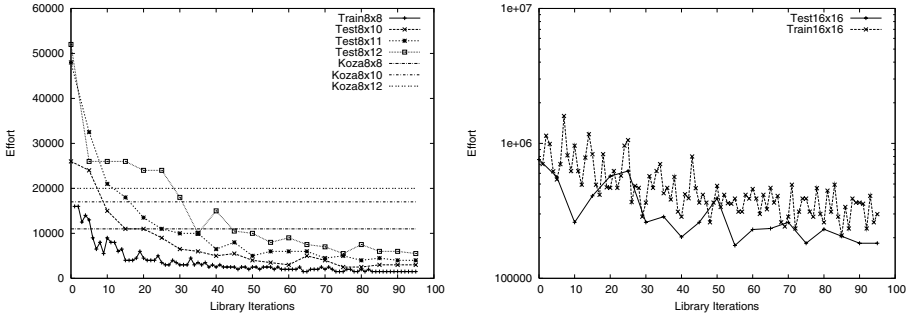


**Fig. 3.** Number of individuals needed to be processed to obtain a solution with 99% probability on the parity problems. Training was done using 50 independently trained libraries on 95 iterations, while testing was done on each library taken from every fifth iteration. The graphs show the results for libraries trained on lawns of size 8x8. Also shown are the computational efforts as published by Koza using ADFs [5].

Finally, a set of runs are performed where libraries are trained on random boolean problems of 5 inputs and an equal number of ones and zeros in the target output. The libraries were subsequently tested on the parity problems from 5 upwards to 10 inputs. The problem domain thus consists of a large subset of *all* logical problems, and the best the library should be able to do is to find an as *unbiased* set of functions as possible. Surprisingly, even in this setting, the library approach was able to get good performance: the conditional effort for parity 5 after training is 21,000 individuals; for parity 6, 42,000; parity 7, 111,000; parity 8, 240,000; parity 9, 388,000; and, finally, parity 10 1,056,000. Also here the effort is significantly lower than when trying to overcome the initial bias from scratch as is done using ADFs.

The results strongly suggest that the concept of Run Transferable Libraries is sound: it is possible to capture significant functionality from one or more training problems in a library and use it to solve related and/or more difficult problems more easily. The results also show that the approach of iteratively re-solving the problem significantly improves GP's ability to scale on these problems. On both problems there is an indication that solving simple 'toy' versions of a more difficult problem helps better in solving the larger problem than to try more difficult versions. It seems that the feedback received in these simpler versions leads more directly to the induction of the right functional bias.

## 6 Future Work

The current research is still in an early stage, yet a myriad of questions and possible avenues present themselves. What learning regime do we need to use to obtain a library that is optimal for a certain problem domain? Does the system reach a stage where it overfits, i.e., when is a library too focused on a problem instance that the library is no longer transferable to different problems and, if so, when does this happen? How can this be overcome? Does the locality enforced in Mnemosyne help in evolving a library? Does it make sense to cross/mutate libraries?

It is envisioned that there is ample opportunity for using Run Transferable Libraries to solve problems: in most circumstances, a practitioner of GP is interested in solving a set of related, yet different problems. By training on simpler instances and learning the underlying functional bias of the problem domain, it is expected that the performance of the basic search can be enhanced. As an example, consider the subject of image classification: although every problem instance will need to find an appropriate way to classify the particular set of images at hand, detecting particular features in an image is a process that transcends problem instances. By evolving feature detectors on various problems using an RTL, a general purpose image classification program could be build up and used for new problem instances.

The definition of Run Transferable Libraries thus presents a possibility to give an operational (yet circular) definition of a problem domain: *A problem domain is a set of problems such that the necessary abstractions to tackle problem instances can be encoded in the form of a library of functions.* The circle is completed by noting that an RTL is defined as a system that helps in tackling problem domains.

## 7 Conclusion

The concept of Run Transferable Libraries is introduced and its worth is tested using a concrete implementation on two well-known benchmark functions for testing functional abstraction. It is shown that transferring an evolving library of functions from one run to the next helps not only in solving the same problem faster, but enables scaling in at least two ways: the ability to solve difficult

problems faster by first solving simpler but related problems; and the ability to solve similar problems faster. It is hypothesised that a library is a good method to learn and encode the *functional bias* of a problem domain, i.e., that set of functionality that helps in solving problem instances taken from that domain faster.

**Acknowledgements.** The authors would like to thank Clearwire Corporation for the use of their servers for some of the runs.

## References

1. P. J. Angeline and J. B. Pollack. The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, Bloomington, Indiana, USA, 1992. Lawrence Erlbaum.
2. Peter John Angeline. *Evolutionary Algorithms and Emergent Intelligence.* PhD thesis, Ohio State University, 1993.
3. David Beasley, David R. Bull, and Ralph R. Martin. A sequential niche technique for multimodal function optimization. *Evolutionary Computation*, 1(2):101–125, 1993.
4. Kenneth E. Kinnear, Jr. Alternatives in automatic function definition: A comparison of performance. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 6, pages 119–141. MIT Press, 1994.
5. John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, May 1994.
6. Sean Luke. ECJ 10 : An Evolutionary Computation research system in Java. Available at http://cs.gmu.edu/ eclab/projects/ecj/, 2003.
7. Simon C. Roberts, Daniel Howard, and John R. Koza. Evolving modules in genetic programming by subtree encapsulation. In Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 160–175, Lake Como, Italy, 18-20 April 2001. Springer-Verlag.
8. Justinian P. Rosca and Dana H. Ballard. Discovery of subroutines in genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 9, pages 177–202. MIT Press, Cambridge, MA, USA, 1996.
9. Matthew J. Streeter, Martin A. Keane, and John R. Koza. Iterative refinement of computational circuits using genetic programming. In W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 877–884, New York, 9-13 July 2002. Morgan Kaufmann Publishers.