Entailment for Specification Refinement

Thomas Haynes, Rose Gamble, Leslie Knight and Roger Wainwright
Department of Mathematical & Computer Sciences
600 South College Avenue
The University of Tulsa
Tulsa, OK 74104–3189
e–mail: [haynes,gamble,knight,rogerw]@euler.mcs.utulsa.edu
*

Abstract

Specification refinement is part of formal program derivation, a method by which software is directly constructed from a provably correct specification. Because program derivation is an intensive manual exercise used for critical software systems, an automated approach would allow it to be viable for many other types of software systems. The goal of this research is to determine if genetic programming (GP) can be used to automate the specification refinement process. The initial steps toward this goal are to show that a well–known proof logic for program derivation can be encoded such that a GP–based system can infer sentences in the logic for proof of a particular sentence. The results are promising and indicate that GP can be useful in aiding program derivation.

1 Introduction

Specification refinement, a part of formal program derivation, is the process of transforming an abstract specification of a software system into a corresponding implementation that is provably correct [Chandy and Misra, 1988, Gamble *et al.*, 1994, Roman *et al.*, 1993]. The process involves the stepwise refinement of the specification in which each step preserves correctness. Thus, the final specification and implementation are provably correct by construction. Formal program derivation is largely a manual process requiring expert knowledge and intuition to determine which refinements to perform and then to prove that the refinement steps do not violate the original specification. However, there exist methodologies and techniques that can be formulated as expert rules of thumb. These methodologies and techniques can be difficult to formally state and thus put in an automated system. In addition, some refinements come by an inductive leap as the designer determines the details for the next specification and finds that those details preserve correctness. Such an inductive leap is also difficult to automate.

Currently, there are few systems that perform program derivation using rules of thumb. Theorem provers are one way to transform one specification to another [Lowry, 1991], but they require full domain knowledge of the possible transformations. Because of this lack of automation in program derivation, it is rarely used in large software engineering projects. With increased use, program derivation would lead to higher quality software because functional reliability could be guaranteed. Thus, there is need for a system that incorporates both theorem proving, expert knowledge, and inductive search to make program derivation a viable part of the software development process.

We investigate the current technology of genetic programming (GP) [Koza, 1992] to aid in the automated process of program derivation. We begin by using GP to search for the proof of entailment of a sentence in a knowledge base of theories from the UNITY [Chandy and Misra, 1988], which is a popular language for expressing the formal specification of concurrent programs and for using program derivation to develop concurrent implementations that are provably correct. These initial steps, of encoding the theories for use by the GP and showing proof of entailment, present a framework to establish the efficacy of GP technology for use in program derivation and for creating an automated system utilizing GP to correctly refine specifications. Because each refinement step must preserve the correctness of the original specification, automating the process requires a theorem prover to verify correct refinements.

The paper is divided into the following sections. The next section provides a brief overview of UNITY and pro-

*This is a preprint of the paper in the *Proceedings of the First Genetic Programming Conference*, pages 90-97.

gram derivation. Section 3 presents the design and implementation details for the GP system. Section 4 presents experimental results. In Section 5 we discuss the system design with respect to the results. Section 6 provides a discussion relevant to our initial goal. Section 7 concludes the paper.

2 The UNITY Language

The UNITY language [Chandy and Misra, 1988, Kaltenbach, 1994] includes specification constructs in which *safety* and *progress* assertions can be made about a program. The UNITY proof logic allows one to perform formal program derivation. A program developer specifies the problem with an abstract initial specification. The proof logic guides the programmer to refining that specification and proving that the refinement at each step is correct. When the specification is sufficiently detailed, a program is directly constructed that satisfies the specification. The program is then refined to execute on specific architectures in specific languages with its correctness guaranteed. Using UNITY, a designer can concentrate on creating the best program to solve a problem without worrying about the final architecture on which the program will be executed. Though there are many tricks to specification and program refinement, experience and intuition account for a large majority of the effort involved.

In UNITY, there is a progress property called leads-to, which is written as $p \mapsto q$. This assertion means that if p is true in some program state, then eventually the program will reach a state in which q will become true, regardless of the value of p when q becomes true. Typically, an initial specification will include a leads-to property that states that the initial state must lead to a goal state, as stated below in (P1).

$$Initial\_State \mapsto Goal\_State \quad \text{(P1)}$$

Because this specification may be abstract, refinement steps are made to add detail while preserving correctness. For example, we may divide the property into intermediate goals that correspond to the completion of a program task. In the properties below, we see that from an initial state the program should reach a state in which Task 1 will begin. From that state, eventually Task 2 will begin, which eventually leads to the goal state. Due to the transitivity of leads-to, (P2), (P3), and (P4) together form a refinement of (P1).

$$Initial\_State \mapsto Task\_1 \quad \text{(P2)}$$
$$Task\_1 \mapsto Task\_2 \quad \text{(P3)}$$
$$Task\_2 \mapsto Goal\_State \quad \text{(P4)}$$

Overall, the process of specification refinement includes a cycle of the following major steps: (1) create an abstract specification, (2) suggest refinements to the specification, and (3) prove the refinements preserve correctness. In the process of refining properties, the developer has to make an inductive leap to suggest possible refinements that specify a valid direction. Then the proof logic must be brought to bear on the new properties to show that they are a correct refinement. Though the process guarantees that correct programs will be developed, a must in safety-critical applications, it is a very difficult and tedious process. In this respect, automation is the best chance to incorporate formal program derivations into large software development.

In this paper, we determine the viability of using GP for the task of automating specification refinement. The initial portion of the research is to determine if step (3) above can be performed by the GP. Therefore, we prove that the entailment of a sentence α from a knowledge base of theories can be inferred automatically by a GP–based system [Koza, 1992]. In our example above, the sentence is (P1) and the knowledge base consists of (P2)-(P4). The inference rule representing the transitivity of leads-to shows that (P2)-(P4) entails (P1). The results of our initial research suggests that the possibility exists, with extensions to the current GP technology, that step (2) can be automated as well.

2.1 Introduction to GP

Genetic programming is an extension of genetic algorithms for generating computer programs [Koza, 1992]. By applying the rules of natural selection, the GP will evolve a program to solve a problem. Given a set of functions and terminals, the GP will generate a random population of programs. The members of the population, chromosomes, are represented by parse trees, where the leaves are terminals (constants or state data) and the branches are functions. New and possibly better programs are created by combining or modifying the subtrees through crossover and mutation. The "goodness" of a program, also referred to as its fitness, is determined by how well the program solves the problem.

Two properties restrict or govern the GP method. The first is closure – every function must handle all terminals and output produced by other functions. Closure ensures the creation of valid programs across genetic operations. The second is sufficiency – the designer must supply enough functions and terminals to solve the problem.

In traditional GP, all of the terminal and function set members must be of the same type. Closure ensures that all inputs to functions can be handled. Montana [Montana, 1995] introduced strongly typed genetic programming (STGP), in which the variables, constants, arguments, and returned values can be of any type. The only restriction is that the data type for each element be specified beforehand.

3 Proving Entailment of UNITY Theorems

Our intent is to encode sentences using UNITY operators to form a knowledge base KB, and a target sentence α, and, then, to use propositional inference rules from the UNITY proof logic to show that the knowledge base entails that sentence, $KB \models_i \alpha$. We employ a GP package, which utilizes strong typing [Montana, 1995], to apply the inference rules. The main domain specific tasks that a GP researcher must tackle are the determination of what comprises both the terminal and function sets and what is the fitness criteria for a chromosome. The terminal set shown in Table 1 is comprised of enough predicate variables to solve the problems. Both a subset of the theorems presented in Chandy and Misra [Chandy and Misra, 1988] and some additional simple logic rules were chosen as the propositional inference rules used to prove entailment. The subset and the simple rules combine to form the function set as shown in Table 2. Both the terminal and function set provide sufficient functionality and representation to prove the entailment of sentences from the provided KB.

Terminal	Type	Purpose
P	Boolean	The predicate P.
Q	Boolean	The predicate Q.
R	Boolean	The predicate R.
S	Boolean	The predicate S.
T	Boolean	The predicate T.

Table 1: Terminal Types

The members of the function set map trees from GP space into trees in propositional logic space. A simple heuristic in the logical inference process is to only apply inferences when there is a match between the "arguments" of the hypothesis and sentences in the KB [Chandy and Misra, 1988, Russell and Norvig, 1995]. Strong typing enables the GP system to guide the pattern matching rule. The restriction of only considering valid child nodes has a side effect of reducing the size of the possible search space [Haynes *et al.*, 1995, Montana, 1995]. A corollary of this is that STGP chromosomes are more representational than vanilla GP chromosomes, and hence tend to be smaller.

3.1 Fitness Evaluation

Each chromosome represents a propositional logical inference from the KB and is evaluated to determine its fitness. The typical GP evaluation function evaluates the parse tree in a preorder traversal, since short–circuiting can occur inside macro functions. The evaluation function for the inference engine has a postorder traversal, which is due to the mapping of GP space into propositional logic space. Since each child subtree must be known before a function can be applied, there is no short–circuiting of arguments.

The fitness function evaluates the soundness of an inference from the KB. To aid this process, two additional knowledge bases are maintained: a newly derived sentence, KB_n, and a derived sentence, KB_d. We also define KB_o as the given sentences and $KB = KB_o + KB_d$. The KB_n is comprised of the newly derived sentences for the current generation. The KB_d begins as an empty list. After each generation, the sentences from the KB_n are moved to the KB_d. The KB_d allows us to track sentences derived by the inference engine.

The fitness function maps the chromosome from a parse tree to a series of operations of inference rules in the propositional logic space. The resultant series of rules can be represented as a tree structure, which we will call a proof tree. The fitness evaluation has different categories to allow for the evolution of better answers. The fitness evaluation is as follows:

- N points for true sentences in the proof tree that are from the KB_o.
- M points for true sentences in the proof tree which are in the KB_d.
- $4 * (M + N)$ points if the proof of $KB \models_i \alpha$ is at the root of the proof tree.
- $2 * (M + N)$ if the proof tree contains the proof of $KB \models_i \alpha$.
- $M + N$ points for the final proof tree being true.

The KB_o is separated from the KB_d to bias the inference engine to build a proof from the sentences in the KB_o. The KB_n is maintained separately from the KB_d to allow a reward for the discovery of new sentences (Note that KB_n does not contribute anything during fitness evaluation.). We are concerned with both fairness of derivation of a sentence and in detecting random generation of a sentence versus derivation from the KB_o. We enforce fairness in that a chromosome which derives a sentence without proving its validity will not be rewarded either before or after another chromosome in the current generation has derived the sentence while proving its validity. Chromosomes in later generations may treat any appearance of a sentence derived in an earlier generation as valid.

These separations of sentences are symptomatic of a basic difficulty of composing fitness functions for genetic programs: how are good sub–structures identified? For example, if the fitness function simply determines if the mapping of the chromosome to an inference procedure results in the target sentence α, then with a randomly generated initial population, it is very unlikely that any chromosome derives α. Thus additional terms are needed in the fitness function to differentially evaluate

Function	Return	Arguments	Equivalent UNITY Statement
And	And	Generic A and Generic B	$p \wedge q$
AntiReflexivity	Unless	Generic A	$\frac{p}{p \ \ unless \ \ \neg p}$
FDisjunction	LeadsTo	LeadsTo A and LeadsTo B	$\frac{p \mapsto q, \ p' \mapsto q'}{p \vee p' \mapsto q \vee q'}$
Implication	Implies	LeadsTo A	$\frac{p \Rightarrow q}{p \mapsto q}$
Implies	Implies	Generic A and Generic B	$p \Rightarrow q$
LawOfExpansion1	Generic	Or A	$\frac{p \wedge q \ \ \vee \ \ p \wedge \neg q}{p}$
LawOfExpansion2	Or	Generic A and Generic B	$\frac{p, \ q}{p \wedge q \ \ \vee \ \ p \wedge \neg q}$
LeadsTo	LeadsTo	Generic A and Generic B	$p \mapsto q$
Not	Not	Generic A	$\neg p$
Or	Or	Generic A and Generic B	$p \vee q$
PSP	LeadsTo	LeadsTo A and Unless B	$\frac{p \mapsto q \ , \ r \ \ unless \ \ b}{p \wedge r \mapsto (q \wedge r) \vee b}$
Reflexivity	Unless	Generic A	$\frac{p}{p \ \ unless \ \ p}$
SimpleLogic1	Implies	Generic A and Generic B	$\frac{p \ , \ q}{p \wedge q \ \Rightarrow \ p}$
Transitivity	LeadsTo	LeadsTo A and LeadsTo B	$\frac{p \mapsto q \ , \ q \mapsto q}{p \mapsto r}$
Unless	Unless	Generic A and Generic B	$p \ \ unless \ \ q$

Table 2: Function Types

these "bad" derivations. The key aspect of the family of genetic algorithms is that even though a particular structure is not effective, it may contain useful sub–parts which when combined with other useful sub–parts, will produce a highly effective structure [Koza, 1992]. The fitness function should be designed such that useful sub–structures are assigned due credit.

A potential negative consequence of assigning credit for sub–structures, *in this domain*, is that a good sub–structure might appear many times in a chromosome, and thus, falsely inflate the total fitness of that chromosome. Note that in typical GP research, this is seen as a benefit as it is the foundation of the schema theorem [Haynes, 1996]. However, we want the inference engine to derive new valid sentences, not to re–derive the same sentence over and over again. The fitness function for the inference engine has to carefully balance these two concerns: allow multiple occurrences of a sub–structure and spend time re–deriving the same sentence or not allow multiple occurrences and have the population prematurely converge.

Finally, it should be pointed out that by collecting valid sentences in KB_d, we are forming a collective memory of previous successful entailments. Given two populations composed of exactly the same individual chromosomes, the fitness evaluation can be different depending on the chromosomes of all previous generations.

3.2 Mapping from Chromosome to Proof Tree

The terminal set represents five predicates, which are sufficient in number to solve the problems we will consider. The function set does not verify the validity of the predicates, but rather it manipulates proof trees in the predicate space to determine the validity of sentences composed of the atoms the predicates represent. These trees are constructed from the leaves up to the root because the chromosomes, which are trees, are evaluated from the leaves to the root. The task of the fitness function is to apply functions in the GP space to the corresponding proof trees represented by the child nodes of the function node.

As a terminal is encountered, it is added as a single node tree to the tree manager, which is a list containing all the proof trees for a chromosome. If the terminal is a valid sentence by itself, i.e. it is in either KB_o or KB_d, then its validity is marked as *TRUE*, else its validity is marked as *DONT_CARE*. After all of the child nodes of a function node have been processed, the proof trees of the children are then combined into the proof tree of the current node by applying the associated UNITY operator for that function node. Its validity is determined by the validity of the proof trees associated with the child nodes, and in some cases on a characteristic property of the UNITY operator itself.

As an example, we will build a proof tree for the problem of showing

$$\frac{p \ \mapsto \ q}{p \wedge \neg q \ \mapsto \ q} \quad or \quad \frac{p \ \mapsto \ q}{p \wedge \neg q \ \mapsto \ q \wedge \neg q \ \vee \ q}.$$

Since we do not provide a function to rewrite $q \wedge \neg q \ \vee \ q$ as q, we have logically rewritten the problem to prove that $p \wedge \neg q \ \mapsto \ q \wedge \neg q \ \vee \ q$. While adding an additional UNITY operator will help, simple editing of the output will not: the problem has to be restated in order for the sentence to be resolved. An evaluation of the optimally encoded chromosome, **PSP**(**LeadsTo**(P,Q), **Anti-**

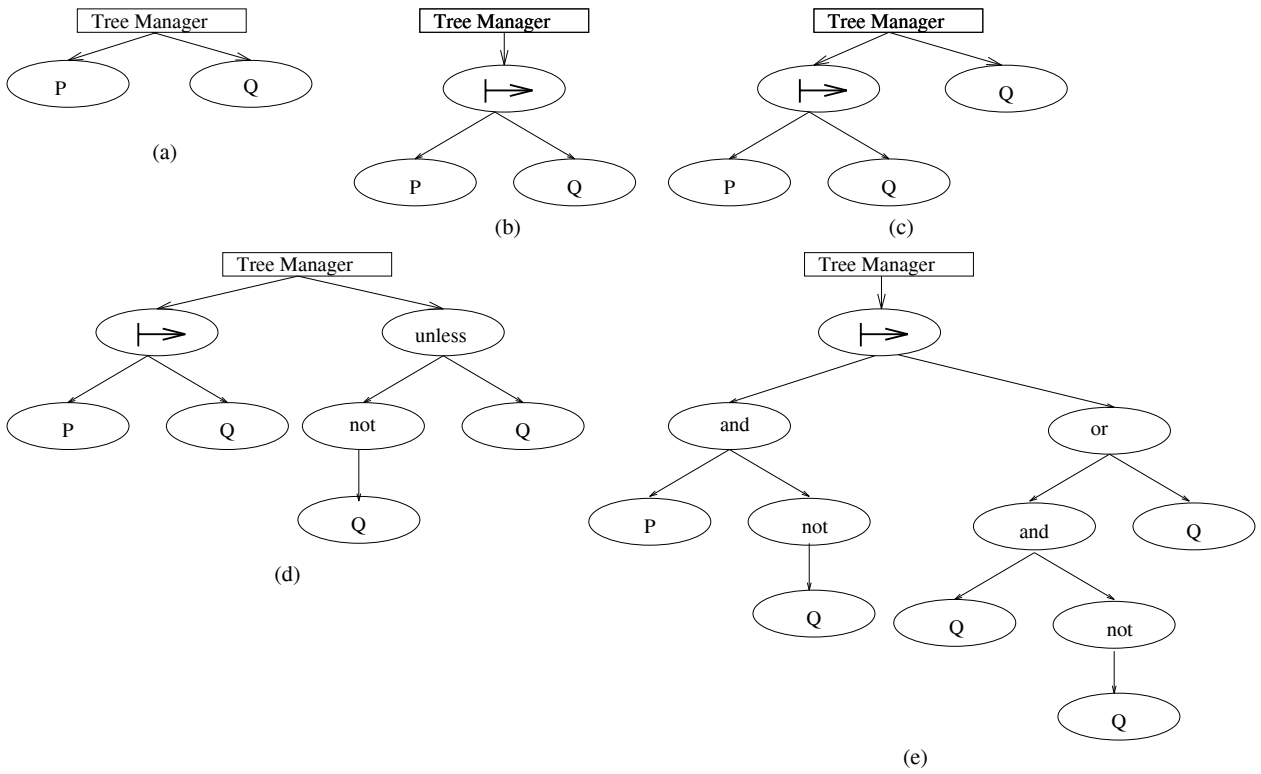

Figure 1: Effect of transformation of S–expression to proof tree. (a) Addition of terminals to Proof Tree Space. (b) Application of **LeadsTo** to Terminals. (c) Another terminal is added to the tree manager. (d) Application of **AntiReflexivity**, an unary operator. (e) The final result in Proof Tree Space.

Reflexivity(Q), is shown in Figure 1.

4 Experimental Results

We present one problem that is illustrative of the workings of the inference engine:

$$\frac{p \mapsto q \quad , \quad q \mapsto r \quad , \quad r \mapsto s}{p \mapsto s}.$$

This problem can be solved using two applications of transitivity. With population sizes of 500 and 1000, the solution appears after the randomly generated population performs crossover and mutation. With a population size of 100, the solution chromosome does not appear without some work by the STGP system. We expected that a larger population size would be needed. However, we are using an STGP system, not a vanilla GP system. The majority of the work is being done by the STGP algorithm, which generates more representational chromosomes, i.e. it reduces the search space [Haynes *et al.*, 1995, Montana, 1995]. If we add another level of transitivity, $S \mapsto T$, then even with a population size of 2000, the solution does not appear in the early generations. Since strong typing is needed to ensure legitimate proof trees are generated, see Section 5, we are not able to test an untyped system. However, we can generate an initial population, and we find that even for population sizes of 16,000 useful sub–trees do not appear in that population. For a typed system, building blocks do appear immediately. For the original problem, with two applications of transitivity, the solution appears within these 16,000 chromosomes, but it does not for the modified problem, i.e. an additional level of transitivity.

A difficulty is shown in Figure 2, which graphs the best chromosome's fitness and average fitness per generation. The average fitness is tracking the fitness of the best chromosome. This is a linear curve, which is not typical of the curves one gets from most GP problems. (This linearity is similar to that artificially induced in the research of McPhee and Miller [McPhee and Miller, 1995] into introns.) The system is exhibiting symptoms caused by an ever growing maximum fitness. As new true sentences are derived, they are added to the KB_d to be matched against later. For example, $\mathbf{AR}(Q)$ will slowly progress to $\mathbf{AR}(\mathbf{AR}(\mathbf{AR}(\ldots(\mathbf{AR}(Q))\ldots)))$. This growth of true, yet meaningless, sentences will continue until restricted by the solution space as a function of the tree size. As the KB_d grows, the chromosomes can exploit it, resulting in ever increasing fitness scores. However, these scores are not indicative of the chromosomes converging to a good solution. As shown in Figure 2, the fitness dramatically jumps once the solution appears within a subtree. Also, within 5 generations, the

solution rises to the top of the tree. This supports our hypothesis that finding the sentence α inside the chromosome is a good building block which will allow the inference engine to find the final solution.

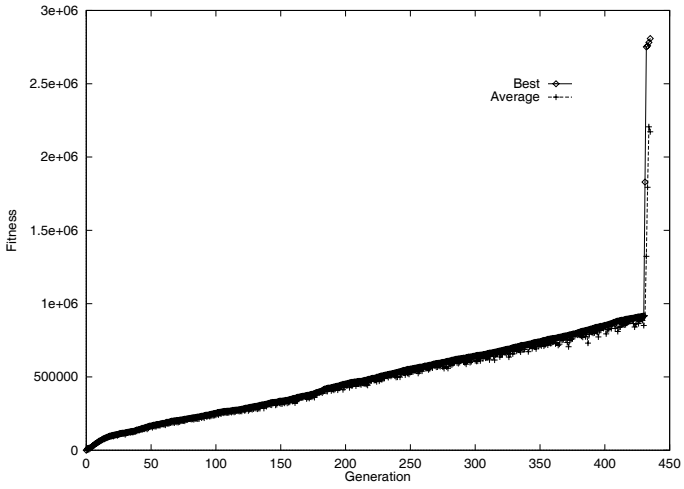

Figure 2: Best and Average Fitness for double transitivity problem.

The S–expression in Figure 3 corresponds to the chromosome which scored the best fitness in generation 431 (the first generation to derive the proof). The line marked $\star$ denotes a **Transitivity** which is applied on two **LeadsTo**: **TR**(**LeadsTo**(P, Q),**LeadsTo**(Q,S)). From the form of this derivation, it can be seen that the sentence: **TR**(**LeadsTo**(Q,R),**LeadsTo**(R,S)) must have been arrived at earlier, and the sentence **LeadsTo**(Q,S) must have been added to KB_d. Thus the chromosome can back up its claim that the sentence **LeadsTo**(Q,S) is true by referring to the proof which placed this sentence in KB_d. Finally, in generation 436, a chromosome corresponding to the $\star$ line in Figure 3 is generated. Our ultimate goal of a chromosome succinctly proving the entailment of the sentence has been achieved.

```
FD( FD( PSP(
⋆               Trans( LeadsTo( P, Q ), LeadsTo( Q, S ) ),
                Reflex( Unless( Q, R ) ) ),
          PSP( PSP(  LeadsTo( R, P ), Unless( S, R ) ),
                AR( P ) ) ),
     FD( PSP( PSP(  LeadsTo( P, P ), Unless( P, P ) ) ,
                AR(  Implies( P, Q ) ) ),
          PSP( PSP(  LeadsTo( P, P ), Unless( P, P )),
                AR(  Implies( S, R ) ) ) ) )
```

Figure 3: Generation 431's best S–expression.

5 Design Issues

Our method of credit apportionment presents several unresolved difficulties. It is difficult to differentiate between derived and randomly generated true sentences. We would like to be able to reward discovery of a true sentence (actual learning) rather than (or more than) random generation of a true sentence. Also, as the KB_d grows, the maximum fitness also grows, which makes it difficult to compare the members across generations.

It also became apparent that while the system learned that **AR**(Q) was true, it could not learn that **Unless**(Q,$\neg Q$) was just as true. We altered our code to detect when the functions **Unless**, **LeadsTo**, **Implies**, and **Implication** were being applied to the same subtree. If they were, then these derived sentences were added to the KB_n. In addition, **Unless** can detect if it is being applied to a subtree, and the negation of that subtree.

This inference engine also raises some interesting concerns about the standard manner in which GP systems are constructed. The traditional GP system is designed with only closure properties to ensure that the function set can handle any input. This simple restriction will not work with our system. For example, if the interior grandchild nodes of a **Transitivity** node are not the roots of identical subtrees, then the evaluation and the side effects of the node are meaningless and undefined. What proof tree should be passed up the GP S-expression? Our system handles this by carrying out the transitivity with the two exterior grandchild subtrees, and by marking the validity of the resultant proof tree as *FALSE*. The subtler problem is what to do if the two proof trees manipulated by the **Transitivity** node do not have **LeadsTo** nodes in their roots? This problem also points out that there must be function nodes underneath the **Transitivity** node.

Our system handles this class of problems by enforcing strong typing [Montana, 1995] for each function and its children. Tables 1 and 2 show the types used for the terminal and function sets. By strictly enforcing type instantiations across both the random generation of chromosomes and crossover, the whole issue of correct proof trees can be avoided. Unfortunately the type system does not allow for context sensitive typing. All of the supported functions use "single" levels of typing. The law–of–expansion, which was used in several problems from Chandy and Misra [Chandy and Misra, 1988], presents a problem in which the grandchildren of the **LawOfExpansion1** node must honor a specific function, i.e. **Or**, when in the context of this node. This requires the GP to be context sensitive when applying crossover or mutation to a parse tree. For example, the child nodes of **Transitivity** are of type `leads-to` and could conceivably be recombined with other nodes with type `leads-to` children. The law-of-expansion has a type `Or` with two children of type `And`, which under any other circumstance could be worked on independently, but in this instance must be

manipulated together. This problem restricts the context in which functions of type `And` and `Or` can be used. The type hierarchy presented in Haynes *et. al.* [Haynes *et al.*, 1996] solves this problem.

5.1 An Alternative Approach

The proof of entailment of a sentence in a knowledge base is achieved by the GP derived chromosomes beginning to sense patterns in the given sentences. The current fitness evaluation "encourages" the derivation of linear subtrees: find a valid sentence, and make it longer. An alternative approach is to generate a KB_s of both simple true sentences of the form **Unless**(Q,Q) and the KB_o to form a fixed size table which can be enumerated. The terminal set would become instances of this enumerated KB_s. The function set would then manipulate these sentences to form the desired solution. The KB_d and KB_n would not be utilized.

We want to reward the discovery of a valid sentence, and its subsequent usage in future generations. It should both be combined with other functions and valid sentences to derive the desired final sentence, and start appearing in as many chromosomes as possible. The current implementation does not reward the discovery of how to prove sentences. For example, if one chromosome discovers that the subtree **TR**(**LeadsTo**(P,Q),**LeadsTo**(Q,R)) is *TRUE* because either the two **LeadsTo** subtrees are given or have been proved earlier, then the entire subtree is not highly valued. Instead, the subtree **LeadsTo**(P,R) is much more compact and can appear more often in a chromosome which leads to a higher fitness value. Smaller building blocks are less likely to be torn apart.

By not adding to the KB_d, and by forcing the chromosomes to operate on a fixed set of true sentences, we can free ourselves from this problem. Also, we will get rid of the ever increasing maximum fitness, and the inability to examine fitness graphs to determine the viability and progress of the generations.

6 Discussion

As mentioned earlier, we want to take the lessons learned from the inference engine to create a GP system that will refine a UNITY program. The difficulties we foresee in using GP to refine a UNITY program are:

- Encoding the UNITY program in a manner that will permit the GP to work with it effectively.
- Translating the presented heuristics for refining a program into information the GP can use.
- Evaluating the fitness of the new program with respect to its functionality as compared to the old program.
- Evaluating the fitness of the new program with respect to the amount of parallelism introduced.

These difficulties are all related. Since a UNITY program must first be developed by hand, we know that a functional program exists. We also know that heuristic strategies exist for refining the initial program that can be executed by a human in a short period of time.

Olsson has conducted research into utilizing evolutionary techniques for synthesizing programs from specifications [Olsson, 1995]. His work differs significantly from ours in that he uses input/output pairs to evaluate the fitness of programs and his programs are in Standard ML. The input/output pairs are sample input with desired output. Our fitness cases are based solely on the application of UNITY operators which preserve correctness. Thus our transformation operators are bound up in the fitness set and not in the evaluation function. Both techniques are similar in that they only handle small programs.

Olsson also does not believe that GP is powerful enough to do program refinement. Since the appearance of his paper, many of the drawbacks he has highlighted have been or are being addressed and evaluated. Specifically GP systems can handle recursion [Brave, 1996], iteration [Spector and Alpern, 1995], and the automatic creation of functions not foreseen by the programmer [Koza, 1995].

7 Conclusion

This paper demonstrates that genetic programming is a viable method for proving the entailment of a sentence from a knowledge base of UNITY theorems. While the current implementation spends an inordinate amount of time proving trivially true sentences, it can find the correct solution. This shortcoming also applies to more traditional theorem provers. It is also shown, in Section 5, that strong typing greatly aids in the implementation of the GP Theorem Prover. Some further work in this area is highlighted by the **Law of Expansion** function. Note that DeMorgan's Laws will suffer this same problem. Finally, we are at work on an alternative evaluation function which will not suffer from the problem of ever increasing maximum fitness.

Acknowledgments

We wish to thank the anonymous referees for their insightful comments.

References

[Brave, 1996] Scott Brave. Using genetic programming to evolve recursive programs for tree search. In P. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*. MIT Press, 1996.

[Chandy and Misra, 1988] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[Gamble *et al.*, 1994] R. F. Gamble, G.-C. Roman, W. E. Ball, and H. C. Cunningham. Applying formal verification techniques to rule-based programs. *International Journal of Expert Systems*, 7(3):203–238, 1994.

[Haynes *et al.*, 1995] Thomas Haynes, Roger Wainwright, Sandip Sen, and Dale Schoenefeld. Strongly typed genetic programming in evolving cooperation strategies. In L. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 271–278. Morgan Kaufmann, 1995.

[Haynes *et al.*, 1996] Thomas Haynes, Dale Schoenefeld, and Roger Wainwright. Type inheritance in strongly typed genetic programming. In Kenneth E. Kinnear, Jr. and Peter J. Angeline, editors, *Advances in Genetic Programming 2*, chapter 18. MIT Press, 1996.

[Haynes, 1996] Thomas Haynes. Duplication of coding segments in genetic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR, August 1996.

[Kaltenbach, 1994] Markus Kaltenbach. Model checking for UNITY. Technical Report TR94-31, Department of Computer Sciences, The University of Texas at Austin, May 9, 1994.

[Koza, 1992] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[Koza, 1995] John R. Koza. Two ways of discovering the size and shape of a computer program to solve a problem. In L. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 287–294. Morgan Kaufmann, 1995.

[Lowry, 1991] Michael R. Lowry. Automating the design of local search algorithms. In Michael R. Lowry and Robert D. McCartney, editors, *Automating Software Design*, pages 515–546. AAAI Press, 1991.

[McPhee and Miller, 1995] Nicholas Freitag McPhee and Justin Darwin Miller. Accurate replication in genetic programming. In L. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 303–309. Morgan Kaufmann, 1995.

[Montana, 1995] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[Olsson, 1995] Roland Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1), March 1995.

[Roman *et al.*, 1993] G.-C. Roman, R. F. Gamble, and W. E. Ball. Formal derivation of rule-based programs. *IEEE Transactions on Software Engineering*, 19(3):277–296, March 1993.

[Russell and Norvig, 1995] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[Spector and Alpern, 1995] Lee Spector and Adam Alpern. Induction and recapitulation of deep musical structure. In *Proceedings of Internatinal Joint Conference on Artificial Intelligence, IJCAI'95 Workshop on Music and AI*, 1995.