01 Mar 1993

Finding Fixed Point Combinators using Prolog

Richard Rankin

Ralph W. Wilkerson
Missouri University of Science and Technology, ralphw@mst.edu

Recommended Citation

R. Rankin and R. W. Wilkerson, "Finding Fixed Point Combinators using Prolog," *Proceedings of the ACM Symposium on Applied Computing*, pp. 604 - 608, Association for Computing Machinery, Mar 1993.
The definitive version is available at https://doi.org/10.1145/162754.168691

FINDING FIXED POINT COMBINATORS USING PROLOG

Richard Rankin
Ralph Wilkerson

Computer Science Department, University of Missouri-Rolla

Abstract

A powerful new strategy, called the kernel method, has been developed by Larry Wos and William McCune at Argonne National Laboratories, to study various fixed point properties within certain classes of applicative systems. We present a very simple Prolog reasoning system, named JIST, which incorporates both stages of the kernel method into a single unified program. Furthermore, the prolog tool has been extended to run within a distributed environment using the Linda protocol.

Introduction

Through a combination of his kernel strategy and the automated reasoning program, OTTER, Wos has been extremely successful in addressing various fixed point properties of fragments of combinatory logic [W88]. Researchers in combinatory logic regard the search for such fixed point combinators as difficult problems. Thus, the incredible success of Wos in developing a strategy for finding these is regarded as a real achievement for automated reasoning.

The kernel strategy, effective as it is, does not completely solve the problem by any means, but it is the first such tool that, under certain conditions, can effectively find such combinators in a relatively short amount of time. There are numerous unanswered questions regarding this strategy, as well, foremost of which is the question as to whether the kernel method is complete.

The kernel strategy is a two stage process which first attempts to establish the existence of the reducible weak fixed point property (which implies the weak fixed point property) and, assuming the success of this first stage, attempts to establish the existence of the strong fixed point property. The entire strategy is based upon proper manipulation of the elements of the basis B of the fragment A.

The kernel method has been implemented in the general purpose theorem prover, OTTER. However, there are particular aspects of the OTTER implementation which can be quite cumbersome. In particular, the kernel method is essentially a two-stage process, requiring, at a minimum, some human intervention to prepare the output from stage 1 to be input into stage 2. This is in large part due to the peculiarities of OTTER. An additional restriction, though not part of the kernel strategy is the restriction strategy, called the 1's Rule, which is used in stage 2 to restrict the application of the inference rule paramodulation. In an article of this length, it is impossible to discuss in detail all the nuances of combinatory logic or automated reasoning, and refer the reader to [W88]. However, we will provide the basic notions required for this discussion.

Basics of Combinatory Logic and the Fixed Point Properties

Combinatory logic is concerned with the abstract idea of applying one function to another, and it can be shown that any computable function can be expressed within its framework [B81]. Formally, combinatory logic is an equational system which has the S and K combinators as a basis:

$((Sx)y)z) = (xz)(yz)$ and $(Kx)y = x$.

These two combinators, S and K, generate all of combinatory logic and specifically, what interests logicians is the study of subsets of this logic, called fragments. Of particular interest is whether these fragments satisfy the weak or strong fixed point property.

Definition: Let B be a basis for a fragment A, then we say A satisfies the weak fixed point property if and only if, for

ACM-SAC '93/2/93/IN, USA
© 1993 ACM 0-89791-568-2/93/0002/0604...$1.50

all combinators x, there exists a combinator y such that y=xy, where y is expressed solely in terms of the combinators in B and the combinator x.

Definition: Let B be a basis for fragment A. Then we say A satisfies the strong fixed point property if and only if there exists a combinator y such that, for all combinators x, yx=x(yx),
where y is expressed solely in terms of combinators from B. We call such a y a fixed point combinator.

Before continuing, here are a few simple examples. Consider the fragment whose basis consists of the single combinator L,
Lxy = x(yy) .

Then Ly(Lx) = y(Lx(Lx)) which is clearly of the form y = xy, and thus the weak fixed point property holds in this case. For the strong fixed point property, consider the fragment whose basis consists of the single combinator U, Uxy = y(xxy). Then the combinator UU is a strong fixed point combinator since UUy = y(UUy) for all y.

The kernel method is implemented using the inference rules of paramodulation, which generalizes equality substitution, and binary resolution on a clausal representation of the combinators. See [M90] for a discussion of this system. As an illustration of paramodulation, consider the paramodulation from the right hand side of Bxyz=x(yz) into the term Bu of Buvw = u(vw). Here the most general unifier {B/x, yz/u} is obtained to construct the new equality BByzvw = yz(vw) in a single inference step. This form of equality reasoning, however, must be carefully restricted or else enormous numbers of useless equality clauses can be generated by such a reasoning program.

It should be pointed out that paramodulation is normally restricted from paramodulating 'from' or 'into' terms which are variables. The reason for this is that such a paramodulation will always succeed, resulting in an overwhelming number of conclusions being inferred. There will be occasions where such an option may be needed, but it should be used with great care.

The Kernel Method in Otter

As noted above, the kernel method is a two stage method. From a review of the literature, it seems that the only known automated method using the kernel method was the implementation done using OTTER.

The OTTER kernel method requires two distinct program to be run. The stage 1 program uses paramodulation to expand inequality clauses. The original inequality clause is derived by the negation of the conclusion that a kernel exists. As this clause, and descendant clauses are reduced, the goal is to find a contradiction between one or more of the inequalities, and the reflexive property. The output of this stage is visually inspected to determine if such clauses have been constructed.

Stage 2 in OTTER requires that the candidate clauses from stage 1 be transferred to the stage 2 program. The stage 2 program then uses paramodulation as an expansion rule, in an attempt to find a clause unifiable with (Θ f) such that f does not occur in Θ. This establishes the strong fixed point property, with Θ being the strong fixed point combinator. Again, the output must be visually inspected after the program runs to determine if any such Θs have been found.

Suppose K is a kernel found in stage 1, i.e. K =xK for all x. Now, if there exists a combinator Θ such that Θx = K and K = xK, then it follows that x(Θx) = x(xK) = xK = K = Θx. That is, Θ is a strong fixed point combinator. Thus we want to attempt to expand the kernel K into a term Θx such that Θ contains no occurrences of x.

Implementation in Prolog

There are various reasons which make the implementation of the kernel method in prolog an appealing option. Besides the usual
benefits of prolog's compact code and ability to do pattern matching with relative ease, a prolog implementation would allow the researcher to easily experiment with different search techniques. Furthermore, since Sicstus Prolog version 2.1 was used for the development of this project, it is possible to take advantage of the Linda protocols to implement distributed processing across a workstation network. This proved to be invaluable in developing a high performance implementation of the kernel method since stage 1 and stage 2 could effectively be run as distinct simultaneous processes.

The prolog implementation, called JIST, maintains, conceptually, the two stage approach. On single cpu systems, however, any kernels found are passed to the stage 2 portion of the program for immediate processing. On a multiple cpu system, one instance of stage 1 exists, supplying kernels to many instances of stage 2 programs.

The first problem which needed to be addressed was prolog's unsound unification algorithm, which results from the lack of an occurs check in most prolog implementations (including Sicstus Prolog). This problem was efficiently overcome by utilizing the sound unification code from Stickel's Prolog Technology Theorem Prover (PTTP) [S88]. The implementation of sound unification, while quite long in prolog terms, is relatively efficient and avoids the costly occurs check. Furthermore, one 'procedure' from Stickel's unification code , not_occurs_in, was utilized in stage 2 of the kernel method as well.

Another aspect of prolog which needed to be addressed was its use of unbounded depth-first search. This causes prolog to be incomplete, and,as such, could cause the theorem prover to search too deeply and increase memory requirements excessively. Instead, the theorem prover

employs a variant of the well known bounded search technique called depth-first search with iterative deepening [REF], which is a breadth-first search sacrificing some speed for more efficient memory utilization.

Combinators were represented in prolog by use of list structures. This was done to provide for the simple addition of combinators to the database, and was felt to involve a minimum of prolog experience on the part of the potential user. In particular, a combinator is an equality clause such as Bxyz = x(yz) for the B combinator. It is customary to assume left associativity for the combinators unless the parenthetical groupings indicate otherwise. In other words, the B combinator is short for the equality a(a(a(B,x)y),z) = a(x,a(y,z)) where 'a' is the function which in combinatory logic means that one combinator is applied to another. This is perhaps, the more common notation, as opposed to Wos' notation of Bxyz=x(yz). In prolog, this same equality clause would be represented in list notation and the predicate equal: equal([a,a,[a,b,X],Y],Z], [a,X,[a,Y,Z]]). with the X,Y and Z being prolog variables, and 'a' and 'b' being constants. Note that logical variables in prolog are generally uppercase, while logicians normally indicate logical variables with lower case letters.

Recall that stage 1 of the kernel method searches for a combinator y such that y = xy for all x. Such a y is called a kernel of the given fragment. This property is asserted by the use of the clause not_equal([a,f,X],X) which is the skolemized form of the negation of the weak fixed point property. This, in essence, allows for the goal of the program to be the construction of a proof by contradiction.

Stage 1 works in searching for kernels by obtaining an equality clause representing a combinator in the fragment under consideration. Then, an attempt is made to find a unifiable subterm in the lefthand side of the not_equal clause, and the right hand side of the equality clause. A new not_equal clause is generated using the equivalent of paramodulation, and is asserted into the database. A check for a contradiction is then performed on the new clause. The contradiction will occur as a violation of the reflexivity property which is required for paramodulation. If a contradiction has occurred, then the appropriate kernel is output. This is a valid kernel due to the fact that the contradiction is arrived at from the proof by contradiction using the denial of the weak fixed point property. A sample output clause from stage 1 is shown below.

After a contradiction is found, the program backtracks, and attempts to find additional kernels at the same level. A new kernel could conceivably come from a different subterm of the same clauses just used, or from other clauses at the current level of the database. This process continues until a maximum depth bound has been exceeded. When all the processing has been completed for a certain level, the program proceeds one level deeper into the search tree.

Sample stage 1 output, fragment b,m:

kernel([a,m,[a,[a,b,f],m]],0,0).

It should be pointed out that the kernels of stage 1 are usually fairly easy to find, provided they exist at all. In fact, Wos speculates, and we concur, that the fact that no kernels are quickly found may, in fact be an argument against their existence. This speculation, however, has not been substantiated to date.

Kernels are placed into a file as they are located. It is this file which supplies the input data for stage 2, and allows for a simple conversion to running the theorem prover on a distributed network. When using the Linda protocols, the file is replaced with a 'blackboard' processor which provides multiple copies of stage 2 with the kernels as they become available. The stage 2 processes, in either version, search for strong fixed point combinators.

Stage 2 also uses paramodulation as its inference rule. In this case, though, we paramodulate into terms on the left hand sides of the kernels obtained from stage 1, using the equality clauses representing the elements of the fragment under consideration. Paramodulation is, therefore, being used as an expansion rule, and not a reduction rule as in stage 1.

Stage 2, in fact, attempts to unify newly generated potential strong fixed point clauses with the list structure [a, THETA, X]. If this is possible, a secondary check is made to assure that : not_occurs_in(X,THETA). When these cases both hold, a strong fixed point has been found. When such a fixed point combinator is found, it is written to a file, and attempts are made to find more strong fixed points through backtracking, using the same clause. When this stage is completed, the next kernel is input from either the file or the blackboard, and processing continues. Sample output from stage 2 is shown below.

Sample output from stage 2:
Process: 2
Combinators Available:
comb([a,[a,b,X],Y],Z],[a,X,[a,Y,Z]])
comb([a,m,X],[a,X,X])
comb([a,[a,l,X,Y],[a,X,[a,Y,Y]])

CANDIDATE:
kernel([a,[a,[a,[l,[a,b,f]],[a,l,[a,b,f]]],X],0,0)
FOUND Theta: ((b((b(lm))l))b)
found after 6.70399 of CPU time
clause: 344
FOUND Theta: ((b((b((bm)m))l))b)
found after 31.796 of CPU time
clause 1375

It should be noted that this theorem prover does not restrict paramodulation by use of the 1's rule. In OTTER, this special restriction strategy is used to force the into term to include the first symbol of the argument which contains it.

While this helps in some cases, it is by no means a guarantee of success. The theorem prover discussed here maintains paramodulation position vectors with a slightly different notation, but does not use them to restrict paramodulation. Using the depth-first with iterative deepening variant, and a position vector, we can not only easily implement the 1's rule, it can be extended or altered to any positional strategy desired.

Advantages of JIST

There are several distinct advantages to the system presented here as compared to the OTTER system presented by Wos and McCune. The JIST system was designed to be run under a 'standard' prolog implementation. For researchers willing to run the program, 'as is' , only the list representations of the combinators need be adjusted. It is not necessary for the user to search through the output of stage 1, as it is when using OTTER, to locate the kernels to use for stage 2.

JIST processes clauses one level at a time. This technique allows the system to locate kernels and fixed points lower in the search tree. As each level is processed, it is discarded, leaving only the next level in memory. This allows an arbitrary depth of the search tree to be completely processed without a depth first search which could miss shorter fixed points. (Although when paramodulation into variables is allowed, the depth possible to process in a finite time is small). Paramodulation into variables is allowed merely be setting a flag in the prolog database. The depth of search is also set by a single prolog flag.

The main attributes of this system, however, involve the amount of search space which can be traversed, as well as the removal of human intervention. In the single processor system, kernels found in stage 1 are automatically processed by stage 2. No human inspection of the output is required. Using OTTER, even the output of stage 2 must be visually inspected. OTTER fixed points located are tagged with a 'unit conflict' message, but still must be located by hand. In JIST, the fixed points are placed in a file as they are found. No human inspection or discrimination is required at all, allowing persons unfamiliar with the system to use it without prior training beyond loading the software and selecting the fragment of interest.

In the multi-processor system, a single cpu is dedicated to the blackboard. Another cpu finds kernels, and sends them one at a time to the blackboard. As stage 2 processors become available, the blackboard sends each kernel to a stage 2 processor, in the same sequence that such kernels were received. Each stage 2 processor then processes a kernel, and saves any fixed points found into a file. The number of stage 2 processors which can be supported is fixed only by any internal capacity limits from Sicstus Prolog. When the search based upon a particular kernel is completed, again, controlled by a user-established depth limit, the stage 2 processor clears its database and requests a new kernel to process. Since this is basically an infinite loop on both ends of the system, the system is terminated by hand when desired.

Performance

There are no standards by which to measure the performance of a specialized theorem prover such as JIST. Such terms as 'fast' or 'efficient' are meaningless. There can also be no direct comparisons of output against the only other known implementation in OTTER as the search techniques employed are different, and the results cannot be directly correlated.

For illustrative purposes, however, we present some information on the number and speed with which strong fixed point combinators (SFPC) can be found. All programs were run on NeXT workstations under a Novell network using the multiprocessor system. The stage 1 programs and the server (blackboard) program were run on the same cpu.

Using the combinators B, L, M, the first SFPC was found after 0.047 seconds. Using 6 instances of stage 2, after approximately 4 cpu minutes, 47 SFPC were located. Using the combinators B, M, W and eight instances of stage 2, the first SFPC was found after 0.11 cpu seconds. Within 1 cpu minute, 89 SFPC were found.

Using the combinators W, M, Q, L, a single processor from the multi-processor run found 100 SFPC in 6.51 cpu seconds. Using the combinators B, M, C, a single processor from the multi-processor run found 18 SFPC in 1.546 cpu seconds.

The system was checked against the output of OTTER, and was tested using the 'sage birds' listed in [S84]. Sage birds are Smullyans name for SFPC. All 11 sage birds were verified as being SFPC using JIST. In the case of two sage birds, however, due to the different search method employed in JIST, variations were found first for the fragments: B,M,W and W,S,B,W. The sage bird UU was found most quickly, with 0.0 seconds required. The is due to the feature of JIST which checks the possibility that the kernel may itself be a SFPC before further processing.

Conclusion

A prolog implementation of a Theorem Prover for the construction of strong fixed point combinators has been presented. This implementation is based upon the kernel method for discovering strong fixed point combinators for a fragment of combinatory logic. Logicians consider the search for such combinators as a difficult problem, and, because of the kernel method, this problem has successfully yielded to attempts at automating the process.

JIST, a theorem prover implemented in prolog, advances the kernel method in several ways. It attempts to minimize the amount of knowledge regarding the theorem prover itself required of the user. It removes the human intervention necessary in the previous implementation of the kernel

method by automatically locating, processing, and reporting both weak and strong fixed point combinators. It provides for a breadth first search for combinators, allowing combinators lower in the search tree to be discovered and is available in both serial and parallel versions for Sicstus Prolog.

Author Contacts

The authors may be contacted by mail at: Computer Science Department, University of Missouri - Rolla, MCS 325, Rolla, MO 65401. E-mail addresses are: rrankin@cs.umr.edu and ralphw@cs.umr.edu.

References

[B81] Barendregt, H. P., The Lambda Calculus, Its Syntax and Semantics, North-Holland, Amsterdam, 1981.

[M90] McCune, W., Otter 2.0 User's Guide, Technical Report ANL-9019, Argonne National Laboratory, Argonne, Illinois, 1990.

[S85] Smullyan, R., To Mock a Mockingbird, Knopf, New York, 1985.

[S88] Stickel, Mark E., "A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler", Journal of Automated Reasoning, 4(4), pp. 353 - 380, 1988.

[W88] Wos, L., and W. McCune, Searching for Fixed Point Combinators by Using Automated Theorem Proving : A Preliminary Report, Technical Report ANL-88-10, Argonne National Laboratory, Argonne, Illinois, 1988.