The Specialization and Transformation of Constructive Existence Proofs *

Peter Madden
Department of Artificial Intelligence,
University of Edinburgh, 80 South Bridge,
Edinburgh EH1 1HN, Scotland.

Abstract

The transformation of constructive program synthesis proofs is discussed and compared with the more traditional approaches to program transformation. An example system for adapting programs to special situations by transforming constructive synthesis proofs has been reconstructed and is compared with the original implementation [Goad, 1980b, Goad, 1980a]. A brief account of more general proof transformation applications is also presented.

1 Introduction

A current dilemma emerging in the field of Computer Science, and hence also Artificial Intelligence, is that demands for quantity and complexity of software are outstripping the tools currently available. A solution to this problem is offered by the field of *automatic programming.* This can be broken down into three main, interelated, sub-fields:

- The *automatic generation (synthesis)* of programs from specifications (input-output relations).
- The *automatic verification* that a program meets its specification.
- The *automatic transformation* of inefficient programs into more efficient programs with the same specification.

So by tackling these issues, software reliability can be improved, provided that it is easier to write bug-free specifications than bug-free programs.

For several years the Mathematical Reasoning Group, MRG, in the Department of Artificial Intelligence, under the direction of Prof A Bundy, have undertaken a great deal of research into the field of automatic programming [Bundy *et* a/., 1982, Bundy *et* a/., 1988, Bundy, 1988]. The first two issues above have been successfully tackled within the Oyster proof refinement environment:[1] By using *logic programming* and *constructive logic,* the task of generating programs is treated as the task of proving a theorem. Hence knowledge of theorem proving, and in particular automatic proof guidance techniques, are used *(section £).*

The third issue, that of program transformation, is the latest to be tackled by the MRG and forms the main contextual subject of this paper: The automatic transformation of programs by transforming their synthesis proofs *(sections 2 and 4)-*

This research involved developing transformation techniques which increase the efficiency of the original program, the *source,* by transforming its synthesis proof into one, the *target,* which yields a computationally more efficient algorithm. This process is known as *program optimization section 4*

The same basic techniques where also used in a process known as *specialization* whereby the task (the input-output relation) of the source program is either altered or adapted to a special situation.

As an example of the proof transformation approach to program optimization we will concentrate on this specialization process since it represents a well circumscribed, and fully implemented, sub-domain of the proof transformation field *(section S).*

However, a brief account of the more general proof transformation approach to program optimization is provided to set the specialization research in context *(section 4)-* For a detailed account of the proof transformation research see [Madden, 1988c].

2 The Environment

Within Oyster, a program specification, comprising the desired input-output relationship, is represented by a statement in *constructive* logic, specifically, Martin-Lof type theory.

The Martin-Lof type theory is a constructive, higher order, typed logic. This is especially suitable for the task of program synthesis since executable code is built up as the proof is constructed such that all elements of the former have a one to one correspondence with elements of the latter (the vice-versa is *not* true, proofs contain additional information: *section 8).*

By finding a constructive proof of the program specification we can routinely extract an algorithm from the proof which satisfies the desired input-output relation. Hence *the techniques of theorem proving can be brought to bear on the program synthesis and transformation domains:*

If we represent the program specification as specification(input,output) and let the symbols V and 3 rep-

•This research is supported by SERC grant GR/D/44270 and a studentship to the author. Thanks to Alan S ma ill and Alan Bundy for their help.

[1] *Oyster* is the Edinburgh Prolog implementation of NuPRL; version "nu" of the *Proof Refinement Logic* system originally developed at Cornell [Horn, 1988, Constable *et* a/., 1986].

resent the quantifiers "forall" and "for some" (or "there exists at least one") then by finding a constructive proof of:

$\vdash \forall$ input $\exists$ output specification(input,output)
we can *extract* an algorithm alg :
$\vdash \forall$ input specification(input,alg(input)).
This algorithm is known as the *extract term*.

Oyster reasons backwards from the theorem to be proved using a sequent calculus notation, which includes rules of inference for mathematical induction. The search for a proof must be guided either by a human user or by a Prolog program called a *tactic.* The process of applying the tactics to a proof specification will yield sub-goals to which further tactics may be applied. The proof terminates, or is *complete,* when the application of tactics produces no further sub-goals. This process is known as *proof refinement,* the rules which apply the tactics are the *refinement rules* [Constable, 1982].

The Oyster system has an open ended variety of such program synthesizing tactics. These tactics control the application of the object-level knowledge, such as rewrite rules, axioms and definitions. The idea is that they embody heuristic knowledge about theorem proving in constructive logic. As such, the program synthesizing tactics perform *meta-level inference.*

2.1 Transforming Proofs

The field of *program transformation* is not a new one. Considerable research has been done on the transformation of programming languages of one form or another, for example: Darlington and Burstall have designed a research tool for the development of functional program transformation methodologies [Burstall and Darlington, 1977, Darlington, 1981]. Tamaki and Sato have carried out similar research except they are concerned with the transformation of logic programs [Tamaki and T.Sato, 1983]. Grant and Zang have developed heuristics for the automatic transformation of Prolog programs [Grant and Zhang, 1988].

However, all these prior approaches are concerned with the direct transformation of executable code and *not* with the transformation of constructive proof structures. The latter approach has considerable advantages:

- Intuitively, a proof will contain more information than the program which it specifies since a program need contain no more information than that required for execution. A proof, on the other hand, will contain the *thinking behind the program design.*
- Furthermore, the *logical structure* of a proof is better understood than that of a program. See, for example, [Kreisel, 1968] . Indeed, much of the proving power of the Oyster system is due to the incorporation of extensive knowledge of the structure of inductive proofs, which are responsible for synthesizing recursive programs (see *section 5* and [Boyer and Moore, 1979]).
- The exploitation of proof theory for the synthesis of programs is a fairly well established field [Kowalski, 1979, Kreisel, 1985] and the proof transformation approach is not limited to the Oyster system (cf. [Goad, 1980b]). However, this particular environment does combine the important features that both synthesis and target languages share the same formal language *and* that this language, Martin-Lof type theory, is *constructive* in nature. These features mean that synthesis and, specifically, transformation can be treated uniformly, henceforth referred to as the *proof-program uniformity:* For each transformation operation performed on a synthesis proof there will be a one-to-one corresponding transformation in the target program language.
- The *proof-program uniformity* prevents the proliferation of complexity that occurs when the specification and implementation languages are different.
- Unlike the proof refinement environment, usual program transformations do not have a specification present, so transformations have to be restricted to those that preserve input/output behaviour. Proof transformation is not so restricted.

A more thorough comparison between, on the one hand, the traditional approaches to program transformation and, on the other, the transformation of constructive synthesis proofs can be found in [Madden, 1988b].

3 Adapting Algorithms

One researcher who is concerned with transforming proof structures as opposed to transforming more conventional descriptions of algorithms (be they functional or in some logic programming formalism) is C.A.Goad [Goad, 1980b]. Information contained in proofs, which goes beyond that needed for simple execution, is exploited in the adaptation of algorithms to special situations (for example, adapting a sorting algorithm or a bin-packing algorithm to operate, with maximum efficiency, on input lists of a specific length). Goad's system has been successfully reconstructed, and extended, in the *Oyster proof refinement* environment and subjected to test on a number of examples [Madden, 1988c].

The main feature of this *specialization system* is an open-ended set of *pruning* transformation operators which are guaranteed to reduce the size of the source proof and corresponding source algorithm. The pruning mechanism, which need not be limited to specialization, is designed to remove those branches from the proof tree which result in redundant computation.

3.1 Specialization and Pruning.

The reconstructed specialization/pruning processes, and their explanation, differ from Goad's in that information gleaned from the constructive existence proof is exploited, during transformation, on that proof itself. This is as opposed to transforming the algorithm extracted from the proof.

There are three distinct stages to the transformation of the proof in its adaption to a special situation:

1. *Specialization* amounts to the *partial evaluation of a constructive existence proof.* Specialization alone, i.e. with no subsequent pruning, can increase the efficiency of the resulting algorithm, especially where induction schemata are involved *(seesection 4.8.*
2. Following specialization, the first stage of *pruning* may be attempted: *normalization.* Normalization is designed to,

   (a) remove certain branches from the *specialized proof tree* which will never be satisfied under the particular constraints (instantiations) of the desired adaption (partial evaluation),

(b) set up the proof for the second stage of pruning:

3. *Dependency pruning* is designed to remove those branches from the *specialized and normalized* proof tree which result in redundant computation. This pruning is guided by a kind of dependency information which does not appear in ordinary programs [Kreisel, 1985]. Such redundancies will also not be present, or only implicitly/potentially so, in the original proof.

A few additional points are worth mentioning at the outset:

- The partial evaluation can be done on an incomplete proof with unproved lemmas without compromising the computational usefulness of the proof as a whole.
- Although specialization followed by pruning is not *guaranteed* to increase efficiency, it will do so most of the time simply because its purpose is to tailor algorithms (or proofs) to a specific task, or rather to a specific class of input. Pruning is, however, guaranteed to reduce the size of the algorithm.
- Pruning is guaranteed to preserve the validity of an algorithm for the specification embodied in the root node of the proof describing the algorithm. That is, given the constructive proof and a partial evaluation, pruning is guaranteed to prune *only* the computationally redundant parts of the proof tree without effecting the input/output relation which specifies that proof.
- Conventional computational descriptions (such as the conditional form or some logic programming description) are *not* subject to the pruning transformations. This is because any valid transformations on conventional descriptions must preserve extensional meaning since they only contain information about the function to be computed. A nice example of the benefits of proof transformation as opposed to program transformation.

3.1.1 Explanation by example

To illustrate specialization and pruning, Goad uses the following algorithm for computing an upper bound for both the sum and product of two positive rational numbers x and y;

- **(1) $u(x, y) = x \leq 1$ then $(y + 1)$ else $(if\, y \leq 1$ then $(x + 1)$ else $2xy)$**

The algorithm specified above is in its *conditional* form and as such may only be slightly simplified as a result of partial evaluation; suppose the value 0 is supplied for y, this results in the "specialized" conditional;

- **(2) $u(x, 0) = x \leq 1$ then $(0 + 1)$ else $(if\, 0 \leq 1$ then $(x + 1)$ else $2x0)$**

which, upon evaluation, reduces to;

- **(3) $u(x, 0) = x \leq 1$ then 1 else $(x + 1)$**

This simplification corresponds to the first stage of pruning; *normalization*.

The formalization of the upper bound algorithm as a *constructive existence proof*, its subsequent specialization (partial evaluation) and the application of normalization pruning then allows $u(x, 0)$ to be *automatically* simplified, by the use of dependency pruning, to the expression;

- (4) $(x + 1)$

This is because the constructive existence proof will contain a case analysis whereby the case split is dependent on the size of x. Now, the fact that (x + 1) is an upper bound for both (x + 0) and 0 x x does not depend on x being less than one. This dependency information is contained in the proof and, via partial evaluation and pruning, allows the removal of the "computationally redundant" case split according to the size of x. Note that (3) and (4) are different functions (eg different input/output behaviour is observed for x = 0.5). However, as far as the partial evaluation (specialization) is concerned, in this case *y* being set to 0, subsequent *normalization* will preserve input/output behaviour. In other words whilst normalization will transform the algorithm, reducing its size, it is guaranteed to preserve the input/output behaviour. Dependency pruning, on the other hand, may change *both* the algorithm and the function.

This is a major advantage of the Oyster transformation system since usual program transformation do not have a specification present, so transformations have to be restricted to those that preserve input/output behaviour.

For practical purposes, Dependency pruning should hence be employed where the desired output is more qualitative than quantitative. By a qualitative output we mean some specific result which may be achieved regardless of the algorithms behaviour such as *bin-packing* or *sorting.* For ease of explanation we do, in fact, prune the upper-bound algorithm, and consequently alter the function. [2]

The representation of the above 4 conditional expressions as constructive existence proofs allows our reconstruction to do the corresponding transformations,

l)=>specialization=>(2)=>normalization=>
3)=>*dependency pruning*=>(4),
completely automatically.

3.1.2 Relations between the source and target specifications

To be safe, specialized proofs should be sound where as pruned proofs need not be since we are concerned only with computational output. However the source proof, the specialized target, the normalized target and the fully pruned target proof should all satisfy that input-output relation formalised in the source specification: The source specification will be an over generalization as far as the target is concerned. The converse does not apply; the source proof will *not* satisfy the specialized, normalized *or* fully pruned target proof specifications.

The input-output specification for many synthesis proofs may well be very much *under specified]* for example, it may simply state that from an input list of integers the output will be an integer. Such a specification is generally employed in Oyster when we wish to do a *program synthesis,* with the onus on the *user* to *construct* the proof, as opposed to program verification. At

[3]The upper-bound algorithm is a convenient example which has neatly nested case analysis which allows us to demonstrate the full specialization and pruning transformation procedures.

such a level of generality there is ample scope for specialization but the subsequent application of pruning, if possible, would imply that the prior synthesis had not been done properly; one would not expect to build in much redundancy when synthesizing an algorithm from a very general specification.

However, if the specification is far more concrete, in particular, if it states certain conditions on the input and output (and not merely of what type they are) then we may expect that upon specialization, partial evaluation, certain of these conditions will never, or need never, be satisfied (be redundant). Hence pruning becomes a more viable option where proofs satisfying, more or less, complete specifications are concerned.

3.2 Different approaches of the reconstruction

There is far less structure required for Goad's natural deduction proofs than for the Oyster sequent calculus counterparts. For example, if we have a case split in Goad's system then we do *not* have to provide a decision procedure for the case in order to complete the proof. This does not effect the fact that the *Prawitz natural deduction system,* employed by Goad, *is* rich enough in content to represent the dependency information. However, *unless* the Prawitz system is *explicitly given a decision procedure in the actual proof specification* then the proof itself *cannot* be transformed in order to adapt the corresponding algorithm. More generally, Goad is prevented from performing transformations on the proof itself because the proof lacks in "computational content". What he is therefore forced to do is transform the extract-term.

The greater "computational content* of Oyster proofs means that we transform the proof rather than the extract. This approach has certain advantages:

3.2.1 Differences with respect to specialization:

- Emphasis is placed on *specialization,* as well as any subsequent pruning, as a means to increase the efficiency of an algorithm (in particular, any recursive behaviour). The specialization process when applied to Oyster proofs containing induction schemata, greatly decreases the computational effort required to unfold that schema.
- Specialization is particularly suited to those instances of Oyster proof synthesis where top level goals are *under specified* (i.e have little computational content, expressing only the input/output relation between types). This difference is as a consequence of the differences between the environments discussed in *section 4.2.*
- Furthermore, and of particular importance within such applications as specialization and partial evaluation, the proof environment entails that transformations are not restricted to preserving input/output behaviour because proof specifications themselves can undergo transformation.
- In Goad's system information gained by the specialization of the proof is used to guide the pruning of the *proof extract.* As far as the reconstruction is concerned, normalization can be performed on both the proof or extract and dependency pruning is performed on the proof. This approach has considerable advantages (see next *section 4.3.1).*

3.2.2 Differences with respect to pruning

Normalization

As alluded to above, Goad in fact deploys his pruning on the extract term of his proof. What Goad is not doing is transforming the proof tree. This does not make the "proofs over programs" approach to transformation a red herring in Goad's paper since it is only due to the information contained in the (partially evaluated) proof that the dependency information is made explicit. So what Goad is doing is using information in the proof to transform the extract as opposed to our approach of transforming the proof itself.

Dependency pruning

Goad also applies the dependency pruning to the extract terms of his proofs and what was said above applies here to.

Fortunately for Goad, within the system he describes the uninstantiated variable of a redundant case split *does* appear in the extract hence signaling the appearance of redundant *code.*

The Oyster extraction process is a bit smarter; it hides anything which does not require instantiating in order to compute the (partially evaluated) output. This makes pruning the extract somewhat difficult in practice but not in principle (if Oyster were *really* smart it would prune what it hides and complete the proof accordingly).

In the reconstruction it is the *proof tree* which is *both* automatically specialized, normalized and pruned.

Why prune the *proof* rather than the *extract?*

There are three main reasons why it is preferable to use the dependency information contained in the specialized (and normalized) proof to transform that proof itself rather than the extract.

The *first* reason is methodological; Rather than extract an algorithm at each stage of the transformation process, it is computationally more efficient to have the transformations operate on the proof and then, once completed, extract an algorithm from the transformed proof. Indeed, this approach allows specialization and normalization to be done concurrently.

The *second* reason is that by always performing transformations on the proof we can, at each stage of the transformation, check to see that we still have a viable algorithm. If at each stage the resultant proof can be marked and copied then we at least know that we have a sound algorithm which preserves the desired input/output relation, up to, and including, that stage.

The *third* point is of particular importance if we wish to extend the transformation system beyond specialization. The whole idea of working with proofs is that they contain information which goes beyond that required for simple execution. The dependency information utilized for pruning is one such example. However if, as with Goad's system, such information is used to guide transformations on the *extract* then whatever stage of the overall process we are in we still have the *same source proof.* This means that by the time dependency pruning has been completed the proof will *not* be a faithful formalization of the target algorithm. If, however, it is the proof that is transformed then the target algorithm just is the extract for the transformed proof. So the transformed proof will faithfully represent the target algorithm and can be exploited further if desired.

The specialization system has been successfully subjected to test on numerous example proofs, some of which are very lengthy and complex: for example, the adaption of *sorting programs* to special situations, such as the number of elements to be sorted, by the specialization, and pruning, of synthesis proofs which yield sorting algorithms. These proofs contain nested induction schemata and consequently require recursive calls on the whole specialization process. They do, however, present a preliminary indication of the frequency with which prunable redundancies occur in the "real" computational world

A full account of these examples can be found in [Madden, 1988c].

4 Transformation of Recursive Algorithms

The computational efficiency of a recursive algorithm is directly related to the *form* of recursion, and the way in which an algorithm recurses on its input can be *controlled* by the way in which mathematical induction is employed in the algorithms synthesis.

The clues that the recursive argument position and structure of a function give us as to the best induction schema and induction variable to use have been incorporated, in the form of heuristics, into the Oyster system. Boyer and Moore have done extensive work on heuristics for inductive proofs [Boyer and Moore, 1979]. The relationships between induction and recursion which they established have been generalized such that most recursive structures have a corresponding induction schema which can be employed to synthesise programs exhibiting the desired recursive behaviour [Aubin, 1975, Stevens, 1987].

There is ample in the literature concerning recursive structures and there relative efficiency. For a general and well established account see [Peter, 1967]. The relative efficiency of recursive structures synthesized with the Oyster system has been investigated in [Madden, 1987].

We can now see how the proof refinement environment assists the automatic optimization: A proof from which a program is derived will contain more information than that required for simple execution. Using notions from analogy and past work on synthesizing recursive algorithms, my thesis research involved systematically relating proofs in such a way that more efficient programs can be obtained by transforming the associated proofs. The crucial element in the transformation is that *recursive programs are optimized by transforming the induction schema employed within the corresponding synthesis proofs.*

For ease of explanation we will consider a very simple example: the transformation of an algorithm which computes the *fibonacci function.*

At least two alternative induction schemata can be employed when synthesizing the *fibonacci program: course of values* induction and *stepwise* induction.

To employ *course of values* induction in the synthesis of an algorithm which takes as input n requires appealing to all, or a subset of, the output values obtained when the input is any value less than n. We will not show the proofs since they are very lengthy but the *extracted algorithm* can be represented thus:

1: *source*:

$$fib(0) \Longleftarrow 1$$
$$fib(1) \Longleftarrow 1$$
$$fib(n+2) \Longleftarrow fib(n+1) + fib(n).$$

By employing *course of values* induction we obtain an algorithm such that in order to calculate *fib(n)* one must first calculate *fib(n* - l) and *fib(n* - 2). Each of these sub-goals leads to another two recursive calls on *fib* and so on. In short the computational tree is exponential where the number of recursive calls on *fib* approaches 2^n.

However, we can, with relative ease, transform the course of values inductive proof into a target proof that, in effect, employs stepwise induction *without having to do the synthesis from scratch.* When we are dealing with integers as input/output values then *stepwise* induction simply corresponds to the standard mathematical induction on integers.

Although the automatic Oyster proof transformation system operates on *proofs,* as opposed to *programs,* the rationale employed by the system falls within the *fold/unfold* framework: We first construct a function, say *g,* which combines the values of the two step cases of the less efficient course of values definition (the function *maketuple(x,y)* can loosely be interpreted as a variadic which simulates the action of tuples).

By folding the base and step cases of this new function definition with the original source equations, or unfolded versions thereof, we end up with a proof that yields a function definition which corresponding to the following;

2: *Target*:

$$g(0) = maketuple(\langle 1, 1\rangle)$$
$$g(n+1) \Longleftarrow maketuple(u1 + u2, u1) \quad where$$
$$maketuple(u1, u2) = g(n)$$

In this case there is no recourse to the original *fib* definition and *g(n)* requires only n recursive calls *(stepping down* to the base case g(0)). In other words, the computational tree resulting from stepwise induction is *linear* and hence the resulting algorithm requires far less computational effort in computing *fib(n)* than that synthesized by employing *course of values* induction.

4.1 Automation: Generating New Function/Predicates and Their Definitions

In general, the transformation of the *source* into the *target* is controlled by "collapsing" the less efficient induction schema into the more efficient one whilst making repeated checks that the *type* information and resulting syntax are correct.[3]

The process of providing new functions/predicates, along with definitions, is known in the transformation field as the *Eureka step.* Nearly all program transformation systems rely on generating such new procedures so that the resulting function/predicate can be *folded* with equations from the current equation set, thus introducing recursion into the target program [Burstall and Darlington, 1977, Darlington, 1981, Tamaki and T.Sato, 1983, Grant and Zhang, 1988, Bruynooghe *et a/.*, 1988,

[d] *Type* errors occur when a (sub)goal of the proof is of the wrong constructive type.

Madden, 1988a, Madden, 1989]. Such a process is notoriously hard to automate and practically all such *unfold/fold* systems rely on user interaction. Even those that have achieved limited success in automating the *Eureka step* rely heavily on some form of user-provided control program.

The reason for this difficulty is due to a trade of between the degree of automation in any particular application (e.g introduction of tail recursion, avoiding redundant computation, pruning mechanisms, partial evaluation) and the siie of the class of *unfold/fold* transformations one wishes the system to encompass.

Within the Oyster proof transformation environment, the *Eureka* step has been *completely* automated for relatively simple functions, such as fibonacci, the exponent function and some simple sorting algorithms, by working out from the *source* proof induction step exactly what the auxilliary function, or tuple, is comprised of. This task is also neatly combined with the procedures for accessing and mapping components of the *source* proof.

This explanation is very much a simplification concerning the workings of, and programs transformed by, the proof transformation system.

A far more thorough and detailed account, especially of the flexibility of control that the induction schemas give us over the resulting recursive structures, can be found in [Madden, 1988a).

References

[Aubin, 1975] A. Aubin. Some generalization heuristics in proofs by induction. *Actes de Colloque Construction: Amelioration et Verification de Programmes, Institut de recherche d'informatiquc et d'automatiquc,* 1975.

[Boyer and Moore, 1979] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979. ACM monograph series.

[Bruynooghe *et al,* 1988] M. Bruynooghe, L. De Raedt, and D. De Schreye. Explanation based transformation. *Research Report, dept. of Computer Science, tholieke Universiteit Leuven,* 1988.

[Bundy *et ai,* 1982] A. Bundy, L. Byrd, R. O'Keefe, B. Silver, and L. Sterling. Solving symbolic equations with press. In J. Calmet, editor, *Computer Algebra, Lecture Notes in Computer Science No. 144,* pages 109-116. Springer, 1982.

[Bundy *et al,* 1988] A. Bundy, D. Sannella, F. Giunchiglia, F. Van Harmelen, J. Hesketh, P. Madden, A. S ma ill, A. Stevens, and L. Wallen. Proving properties of logic programs: A progress report. In *1988 Alvey Conference,* pages 131-133, 1988.

[Bundy, 1988] A. Bundy. The Use of Explicit Plans to Guide Inductive Proofs. In R. Luck and R. Overbeek, editors, *CADE9.* Springer-Verlag, 1988.

[Burstall and Darlington, 1977] R. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of Assoc. Compt. Mach.,* 21(l):44-67, 1977.

[Constable *et al.,* 1986] R.L. Constable, S.F. Allen, H.M. Bromley, et al. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice Hall, 1986.

[Constable, 1982] R.L. Constable. Programs as proofs. Technical Report TR 82-532, Dept. of Computer Science, Cornell University, 1982.

[Darlington, 198l] J. Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence,* 16:1-46, 1981.

[Goad, 1980a] C. A. Goad. Computational uses of the manipulation offormal proofs. Technical report, Stanford University, 1980. STAN-CS-80-819.

[Goad, 1980b] C. A. Goad. Proofs as descriptions of computation. In *Lecture Notes in Computer Science.* Academic Press, 1980.

[Grant and Zhang, 1988] P. W. Grant and J. Zhang. Heuristics for the automatic transformation of Prolog programs. In *UK IT 88 Conference Publication,* pages 135-139. UK IT 88 Conference (sponsors: SERC, DTI, MOD), 1988.

[Horn, 1988] C. Horn. The NurPRL proof development system. Research Paper 213, Dept. of Artificial Intelligence, University of Edinburgh, 1988.

[Kowalski, 1979] R. Kowalski *Logic for Problem Solving.* Artificial Intelligence Series. North Holland, 1979.

[Kreisel, 1968] G. KreiseL A survey of proof theory. J. *of Symbolic Logic,* 33:321-328, 1968.

[Kreisel, 1985] G. KreiseL Proof theory and the synthesis of programs: Potentials and limitations. In B.Buchberger, editor, *EUROCAL '85.* Springer, 1985.

[Madden, 1987] P. Madden. An Oyster synthesis of several sorting algorithms. Research Paper 356, Dept. of Artificial Intelligence, University of Edinburgh, 1987.

[Madden, 1988a] P. Madden. Automatic program optimization via the transformation of Nuprl synthesis proofs. In *UK IT 88 Conference Publication,* pages 139-143, 1988.

[Madden, 1988b] P. Madden. The Lops approach to program synthesis; a comparison with NuPRL. Research Paper 397, Dept. of Artificial Intelligence, University of Edinburgh, 1988.

[Madden, 1988c] P. Madden. The specialization of constructive existence proofs. Research Paper 406, Dept. of Artificial Intelligence, University of Edinburgh, 1988.

[Madden, 1989] P. Madden. The automatic transformation of Oyster inductive proofs. Research Paper 412, Dept. of Artificial Intelligence, University of Edinburgh, 1989.

[Peter, 1967] R. Peter. *Recursive Functions.* Academic Press, 1967.

[Stevens, 1987] A. Stevens. A rational reconstruction of Boyer and Moore's technique for constructing induction formulas. In *The proceedings of ECAI-88,* pages 565-570, 1987.

[Tamaki and T.Sato, 1983] H. Tamaki and T.Sato. A transformation system for logic programs which preserves equivalence. Technical Report ICOT Research Center Technical Report, ICOT, 1983.