Recursive Program Optimization Through Inductive Synthesis Proof Transformation

Recursive Program Optimization Through Inductive Synthesis Proof Transformation*

Peter Madden
Max-Planck-Institut für Informatik
Im Stadtwald, D-66123 Saarbruecken, Germany
Email: madden@mpi-sb.mpg.de

Alan Bundy, **Alan Smaill**
Department of AI, University of Edinburgh,
80 South Bridge, Edinburgh EH1 1HN, Scotland.
Email: bundy/smaill@aisb.ed.ac.uk

Abstract

The research described in this paper involved developing transformation techniques which increase the efficiency of the noriginal program, the *source*, by transforming its synthesis proof into one, the *target*, which yields a computationally more efficient algorithm. We describe a working proof transformation system which, by exploiting the duality between mathematical induction and recursion, employs the novel strategy of optimizing recursive programs by transforming inductive proofs. We compare and contrast this approach with the more traditional approaches to program transformation, and highlight the benefits of proof transformation with regards to search, correctness, automatability and generality.

1 Introduction

As computer programs play an increasingly important role in all our lives so we must depend more and more on techniques, preferably automatic, for ensuring the high quality (*efficiency* and *reliability*) of computer programs. By *efficient* we mean that a program is designed to compute a task with minimum overhead and with maximum space and time efficiency. By *reliable* we mean that a program is ensured, or guaranteed in some sense, to compute the desired, or specified, task.

The most promising technique being developed for the automatic development of high quality software are *formal methods*, which are used to provide programs with, or prove that programs have, certain properties: a program may be proved to *terminate*; two programs may be proved equivalent; an inefficient program may be *transformed* into an equivalent efficient program; a program may be *verified* to satisfy some specification (i.e. a program is proved to compute the specified function/relation); and a program may be *synthesized* that satisfies some specification.

The research described herein addresses both the reliability and efficiency, as well as the automatability, aspects of developing high quality software using formal methods. We describe novel theorem proving techniques for automatic program optimization. The target program is a significant improvement on the source (the efficiency criteria), and is guaranteed to satisfy the desired program specification (the reliability criteria).

A further motivation behind exploiting proofs for the purposes of program transformation is that proofs will contain more information than the programs which they specify. Programs need contain no more information than that required for simple execution. Proofs, on the other hand, represent a *program design record* because they encapsulate the reasoning behind the program construction by making explicit the procedural commitments and decisions made by the synthesizer. This non-algorithmic information, which includes the relations between facts involved in the computation of the synthesized program, is ideal for controlling the transformations.

*This research was supported by SERC grant GR/F/71799, an SERC Studentship and an SERC Postdoctoral Fellowship to the first author, and an SERC Senior Fellowship to the second author. The authors would like to thank the many useful suggestions made by two anonymous JAR referees. An earlier, and much shorter, version of this paper appears in the [25].

Further motivations include the advantages of proof transformation, concerning search, control and correctness criteria, over the more traditional styles of program development. We address these in detail in §**??**. Further applications (potential and real) of this research are discussed in §**??**.

1.1 Contents

In §**??** we provide a background to proof transformation by discussing the duality between proofs and programs. In §**??** we describe properties of the OYSTER system, and of (synthesis) proof refinement in general. The duality between mathematical induction and recursion, in a constructive setting, is discussed in §**??**.

In §**??** we provide an overview of the central concepts pertaining to, and the properties of, the PTS: §**??** provides a high-level view of the PTS design; §**??** introduces one of our running examples, and illustrates how specific recursion schemas correlate with the induction schemas used for synthesis; in §**??** we introduce the *tupling* technique for removing redundant computation from recursive procedures. §**??** serves as a gentle introduction to program through proof transformation, and enables the reader to maintain a high-level picture when we come to the more detailed low-level expositions.

In §**??** we provide details concerning: the motivations for proof transformation; the abstraction of information from proofs for the purpose of optimization; how the PTS constructs the synthesis and verification components of an optimized proof; and the adaptation of the tupling technique to the proofs as programs paradigm.

In §**??** we explain, through detailed examples, the methodology of the PTS: §**??** illustrates linearizing exponential procedures through proof transformation; §**??** illustrates the removal of nested recursion schemas (i.e. loop removal); and §**??** briefly describes a more complex example. We also discuss the overall performance of the PTS, §**??** and §**??**.

In §**??** we compare the properties of the PTS with existing program transformation techniques and systems. We highlight the advantages of the former. In particular we compare our approach to the *unfold/fold* technique and the use of *dependency graph* analysis for *tupling program transformations*. We also discuss applications of the research, and some anticipated future directions for extending the PTS system.

Finally, in §**??**, we provide a concluding summary.

2 Background: The Duality Between Programs and Proofs

Constructive logic allows us to correlate computation with logical inference. This is because proofs of propositions in such a logic require us to construct objects, such as functions and sets, in a similar way that programs require that actual objects are constructed in the course of computing a procedure.[1] This duality is accounted for by the *Curry-Howard isomorphism* which draws a duality between the inference rules and the functional terms of the λ-calculus [**?**, **?**].

Such considerations allow us to correlate each proof of a proposition with a specific λ-term, λ-terms with programs, and the proposition with a specification of the program. Hence different constructive proofs of the same proposition correspond to different ways of computing a specific program specification. The reasoning for this can be set out as follows:

[1] Thus we cannot, for example, compute (or constructively prove) that there are an infinity of prime numbers by assuming the converse and deriving a contradiction, rather we must produce a program that computes them (or a proof that we can always construct another prime number greater than the ones known so far).

1. proofs of propositions correspond to terms of the appropriate *type*, such that,
2. the propositions are identified with the type of *their* proofs;
3. proofs are closely correlated with the terms of the λ-calculus;
4. so by 2 and 3: propositions are identified with the type of the λ-terms, and;
5. λ-terms can be equated with functional programs;
6. therefore, by 4 and 5, the propositions can be viewed as *types* of programs;
7. in other words, the propositions of the λ-calculus can be correlated with descriptions (specifications) of programs which specify *what* task is computed by the program, and;
8. the proofs of the propositions can be correlated with programs which determine *how* the task is computed;
9. hence, different proofs of the same proposition can be correlated with different programs for computing the task specified by that proposition.

Thus by controlling the form of the proof we can control the efficiency with which the constructed program computes the specified goal. Here in lies the key to transforming proofs that yield inefficient programs into proofs that yield efficient programs.

A program specification is represented, schematically, as

$$\vdash \forall inputs, \exists output.\ spec(input, output) \tag{1}$$

Existential proofs of such specifications must establish (constructively) how, for any input vector, an output can be constructed that satisfies the specification.[2] Thus any synthesized program is guaranteed correct with respect to the specification. Furthermore, by finding a constructive proof of (**??**) we can *extract* an algorithm, alg such that,

$$\vdash \forall input.\ spec(inputs, alg(input))$$

alg is known as the *extract term* (or *extract program*) of the constructive proof.

So, for example, suppose we wish to compute a value for the integer log to the base 2 of our input, then from a proof of the following specification:[3]

$$\vdash \forall\ input{:}integer,\ \exists\ output{:}integer.\ (\ 2^{output} \leq input\ \wedge\ input < 2^{output+1}\ )$$

we extract an algorithm alg which satisfies the following:

$$\vdash \forall\ input{:}integer.\ (\ 2^{alg(input)} \leq input\ \wedge\ input < 2^{alg(input)+1}\ )$$

and which does the required job. Proving that a given extract algorithm does satisfy the above is known as *verification*.

2.1 The OYSTER System

The OYSTER system is an implementation of a constructive type theory which is based on Martin-Löf type theory, [**?**]. OYSTER is written in Quintus Prolog, and run at the Prolog prompt level, so it is controlled by using Prolog predicates as commands. Proof tactics can be built as Prolog programs, incorporating OYSTER commands (which are simply Prolog predicates). An advantage of using Prolog as the meta-language for defining tactics is that the proof mechanisms can exploit the unification and back-tracking properties of Prolog.

The main benefit of using type theory is that, recalling the previous section, it nicely combines typing properties with the properties of constructivism, such that we can both correlate the propositions of the λ-calculus with specifications of programs and correlate the proofs of the propositions with how the specification is computed.

The main benefit of using a sequent calculus notation, as opposed to that of any of the numerous natural deduction systems, is that at any stage (node) during a proof development, all the dependencies (assumptions and hypotheses) required to complete that proof stage are explicitly presented within a *hypothesis list*. A

[2] Thus constructive logic *excludes* pure existence proofs where the existence of $output$ is proved but not identified.

[3] Typing is not, of course, restricted to integers. Types can be natural numbers, lists of natural numbers (or integers), sets, strings, trees and so forth. Throughout the course of this paper we shall often omit typing information so as to make formulae more readable. In general, only when it is not obvious, or when it is pertinent to the text, shall we explicitly label the types of objects.

sequent is of the form [HYPOTHESES] $\vdash$ [CONCLUSION], where, in the course of proving the conclusion, refinements may either act upon the hypotheses (so called $elim$ refinements) or act upon the conclusion (so called $intro$ refinements).

A major motivation behind the development of the OYSTER system is that the language uniformity of the logic programming environment allows for the construction of *meta-theorems* which express more general principles, concerning the object level theorem proving. This allows for the construction of programs, in Prolog, that manipulate proofs inside the system itself. One such function is the construction of *tactics* which combine the object-level rules of the system in various ways and apply them to proof (sub)goals. Within the context of the PTS, this allows for the construction of (meta-level) transformation tactics that operate upon the (object level) source proofs to produce target proofs from which optimized programs can be extracted.[4]

2.1.1 The Nature of OYSTER Synthesis through Proof Refinement

OYSTER proofs are *refinement proofs*, and are edited using a *refinement editor*. The OYSTER proof starts with the expression to be proved at the root of its proof tree, and constructs the tree back towards the leaves: the inference rules of the logic – *refinement rules* – are applied in reverse to a goal, to reduce, or *refine*, it to a set of sub-goals which, in turn, require proving in order to complete the overall proof. Thus, for example, if the user tells OYSTER to apply $\forall - introduction$ to a top-level goal statement, the system applies the rule in *reverse* – the effect of this is not to introduce, but to *remove* the topmost connective (since the proof tree is being developed backwards).

Any proof is *complete* when the proof tree has been sufficiently developed *backwards* such that all leaves are accounted for – i.e., when every leaf node can be proved without producing any further sub-goals. We refer to such proofs as being *goal-directed*. The refinement editor allows proof trees to be *traversed*, and refinement rules (or combinations thereof called proof tactics) to be applied to chosen nodes.

The end-nodes, or leaves, of a proof will always correspond either to axiomatic equalities, well-formedness goals or the discharge of assumptions (i.e. where each component of the goal conclusion matches with one of the proof hypotheses).

2.1.2 Program Extraction

The OYSTER extract programs consist of λ-calculus function terms, $\lambda(x, f_x)$ where f is some computed function and f_x the output when f is applied to input x. Since all type checking (well-foundedness checking) is done during the proof development then the extract terms need not, and do not, contain any typing information. At any stage during the development of a proof it is possible to automatically access the extract term of the proof constructed so far. Each construct in the extract term corresponds to a proof construct. As such, the extract term reflects the algorithmic ideas behind the proof of the theorem.

There is a built-in evaluator for type theoretic terms, which allows for the direct execution of OYSTER programs. Within type theory, each mathematical sentence, or proposition, is considered as a type, the elements of which are proofs of that sentence. A *type*, by definition, is a term which can be *inhabited* by other terms, or, equivalently, all types can have members. The existence of an extract term, corresponding to a particular proposition, is evidence that the proposition's type is inhabited, and this is equivalent to the proposition being constructively proved. All constructs of a completed proof that have an associated extract term of computational significance are collectively referred to as the *synthesis component* of the proof.

However, establishing that all the extract terms assembled from the synthesis component of a proof will indeed constitute a program that computes the specification embodied in the root node of a proof requires *verification*: the *verification component* of a proof is not used in executing the extract term, but ensures that the extract term satisfies the specification

Ideally, as with conventional computational description, the λ-calculus extract terms should only contain information about the function to be computed (whereas the proofs will contain additional information, such as verification steps, which is not concerned with simple execution). In practice, however, it is not so easy to (automatically) abstract away all the verification information from the extract.

2.2 The Induction-Recursion Duality

OYSTER provides primitive recursion schemas for the basic types: integers, natural numbers and lists. The recursion schemas enable one to define recursive functions through case analyses, where the cases are determined by the structure of the type; and apply induction as an inference (refinement) rule. to each form of

[4] The language uniformity property has also led to the development of an automatic proof planning system CLAM [**?**] (*cf.* §**??**).

induction employed in the proof there corresponds a dual form of recursion [**?**]. Such dualities offer the user a handle on the type, and efficiency, of recursive behaviour exhibited by the extracted algorithm. Thus applying inductive inference enables the synthesis of the dual recursion in the extract program (we return to this in more detail in §**??**.

2.2.1 Recursive Definitions

An important class of recursive definition is that which allows one to refer to (standard stepwise) recursion over the natural numbers. The term p_ind allows one to construct such definitions. For example, addition, $+$, over the natural numbers is defined as

$$x + y \stackrel{def}{=} p\_ind(x, y, [\sim, rec, s(rec)]),$$

which states that if x is 0 then $x + y = y$, otherwise if $(x - 1) + y = rec$ then $x + y = s(rec)$, where s is the successor function.

- The first argument, x, is the recursion argument.
- The second argument, y, is the (truth) value if the recursion argument is 0.
- The third argument, $[\sim, rec, s(rec)]$ is a triple and describes how to compute its value if it is of the form $s(x)$. The expression, rec, denotes the value of the function being defined when applied to $(x - 1)$. The expression $s(rec)$ denotes the value of the function being defined when applied to x. Thus rec and $s(rec)$ correspond, respectively, to the induction *hypothesis* and induction *conclusion*.[5]

Similarly, cv_ind, specified thus:

$$cv\_ind(x, [y, h, P(x)])$$

allows one to refer to *course of values* recursion over the natural numbers. x names the induction candidate (the argument over which the recursion is defined). The second argument, $[y, h, P(x)]$, is a triple which defines the *recursive case* for the function being defined. The first two elements are y and h where: y is any natural number *less than* the recursive argument (i.e. $y < x$). Hence, during the course of a proof, y can be instantiated to any desired value *less than* x. Furthermore, we can, depending on the function being defined, have multiple values for y (as long as each is less than x). This is, in effect, how cases can be introduced into a proof employing course of values induction (*cf.* §**??** *below*). h is the value of the function being defined when applied to y. The third element of the triple, $P(x)$, provides the *step case* value for the function in terms of the first two elements, y and h, of the triple. Hence the third element, $P(x)$, computes the output value for the function/program being defined/synthesized. So $P(x)$ is a conditional function which branches according to the value of y (where the restriction $y < x$ holds).

2.2.2 Primitive Schemas

Employing any of the induction schemas in a (synthesis) proof will induce the corresponding, or *dual*, recursion schema in the extract algorithm. So, for example, stepwise recursion over the natural numbers is synthesized by applying stepwise induction, conventionally represented thus (where s is the successor (constructor) function):

$$\frac{\vdash P(0) \qquad \forall y : nat.\, P(y) \vdash P(s(y))}{\vdash \forall x : nat.\, P(x)}.$$

This states that P holds of any natural number, x, iff one can establish that P holds of 0 (the base case), and that, assuming P holds of some natural number y, that P holds of $s(y)$ (the step case).

Terms of the form $a : P$ should be seen, in constructive terms, as denoting the existence of a proof of P along with a corresponding extraction term P. Depending on context, P may be a hypothesis or (part of) a goal conclusion. We refer to terms such as $s(y)$ as *induction terms* (i.e. those terms consisting of the induction constructor (or destructor) function applied to the induction variable). The proof extract construction resulting from an application of stepwise induction is the p_ind construct shown in previously in §**??**.

Stepwise induction on the naturals, along with stepwise induction on the integers and on lists, constitute the *primitive induction schemas*, and are built into the OYSTER system. Employing such induction as an inference rule will split the proof into the corresponding cases. Each case will have a corresponding proof

[5] In general the value of the p_ind function at $s(i)$ can be any function of i and of the value of the function at i. In our example the value depends only on the recursive value, and so the first argument of the triple is the anonymous variable $\sim$.

and extract component. The structure of the program extracted from the complete proof will mirror that of the (instantiated) dual induction schema. This is a general observation: to each induction schema there corresponds a dual recursion schema. Hence a reliable heuristic that applies to synthesis through inductive theorem proving is that the behaviour of the induction variable should mirror that of the recursive terms in the function's definition.

Standard stepwise induction is sometimes referred to as $+1$ succesor induction, or $(+1)s$ induction for short. This is to distinguish it from any number of $(+n)s$ inductions where n applications of the induction constructor function are applied, in the conclusion, to the induction variable. §**??** illustrates a $(+2)s$ stepwise schema.

2.2.3 Non-Primitive Schemas

More sophisticated induction schemas can be established by performing higher order proofs that appeal to the primitive schemas in order to justify the sophisticated scheme. An example of a non-primitive scheme is course of values induction.[6] As with the primitive schemas, course of values recursion over the natural numbers is synthesized by applying course of values induction. This is done by employing the following *general induction*:

$$\frac{\forall z : nat,\ \forall y : nat.\ ((y < z) \rightarrow P(y)) \vdash P(z)}{\vdash \forall x : nat.\ P(x)}.$$

This states that P holds of any natural number, x, iff one can establish that A holds of any natural number, z, assuming that P holds of any natural number, y, less than z. If two, or more, different values of y are appealed to then the induction becomes course of values.

Employing course of values induction as an inference rule does not automatically split the proof into a separate base and step case. Rather, the resulting subgoal represents the original proof tree with the induction hypothesis, $(y < z) \rightarrow P(y)$, entered into the proof as a new assumption (which tacitly includes the assumption that the hypothesis itself has a proof). The onus for splitting the proof into various cases, as defined by the function being synthesized, then lies with the user.

The proof extract construction resulting from an application of course of values induction is the cv_ind construct shown in §**??**.

3 Optimization of Recursive Algorithms By Transforming Inductive Proofs: an Overview

Rather than enter directly into the technicalities of program through proof transformation, we shall first provide an overview of the main concepts involved. In §**??** we provide a high-level description of the proof transformation system. In §**??** we give a brief introduction to program synthesis by theorem proving. We illustrate how the efficiency of (recursive) program is dependent on the nature of the induction scheme employed and on the subsequent proof commitments. Finally, in §**??**, we introduce the reader to *tupling*.

3.1 The PTS: Inductive Proof Transformation

Boyer and Moore have done extensive work on heuristics for inductive proofs [**?**, **?**]. Relationships between induction and recursion have been generalized such that most recursive structures have a corresponding induction schema which can be employed to synthesize programs exhibiting the desired recursive behaviour [**?**].

The computational efficiency of a recursive algorithm is directly related to the *form* of the recursion. The way in which an algorithm recurses on its input can be *controlled* by the way in which mathematical induction is employed in the algorithm's synthesis. This provides the theoretical under-pinning of the transformation system: recursive programs are optimized by transforming the induction schema employed within the corresponding synthesis proofs.

Fig. **??** schematically depicts the source to target meta-level transformation. Program optimization through proof transformation consists in the automatic transformation of a source induction proof to a target proof whose induction schema has a more efficient associated complexity. The pre- and post-conditions of the

[6] Other non-primitive examples include *divide_and_conquer* induction and induction based on the construction of numbers as products of primes.

transformation correspond to the induction schema, and the recursive data-type, of the source and target proofs. The input consists of a complete source inductive synthesis proof. This is depicted on the left hand side of the diagram. The triangle labeled *proof tree* depicts the tree shape of the refinement proof (recall that the proof, or refinement, tree is constructed backwards from the specification toward the leaves). The source proof yields a complex source algorithm, *exp*, which recurses with *exponential* behaviour due to the fact that a particular induction – course of values – is employed during the synthesis. The term *extract* represents the automatic program extraction process.

The target proof is represented on the right hand side and is constructed completely automatically, by the PTS, from the source through the application of operators which map and then transform portions of the source proof. In particular, the source course of values induction is transformed into the more efficient stepwise target induction, thus yielding a target extract algorithm that recurses on its data-structure in more efficient *linear* fashion.

The PTS controls the transformations by exploiting extra information contained in proofs which is extraneous to that required for the simple execution of straightforward programs: a description of the task being performed; a verification of the method, and; an account of the dependencies between facts involved in the computation.

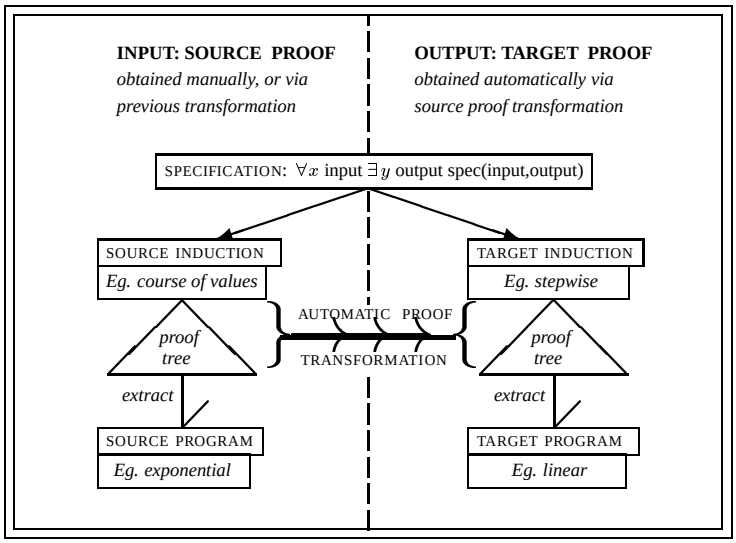


Figure 1: Recursive program optimization through induction schema transformation.

With reference to fig.1, and recalling §**??**, the demands for efficiency of programs are succinctly expressed by quoting from [**?**] (*italics* added by the author):

> The first criterion on which a program is judged is the correctness with respect to its specification. The second criterion is the efficiency of the program with respect to other programs *satisfying the same specification*, which is reflected by time and space complexity of the program.

Efficient programs obtained through proof transformation satisfy both these criteria: the target program necessarily satisfies the specification from which it was constructed; both source and target programs are derived from the same specification, and; the recursive procedure traced by the target program will be more efficient than that of the source.

3.2 Proof Construction and the Induction-Recursion Duality

We can construct at least two proofs, within OYSTER, from which two alternative recursive algorithms can be extracted, each of which computes the Fibonacci function. The difference between the two syntheses is that each employs a different induction schemata: course of values induction will induce course of values recursion in the Fibonacci extract algorithm and stepwise induction will induce *stepwise* recursion.

3.2.1 Course of Values 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.[7] Using a standard functional notation, the Fibonacci function is usually defined by the following course of values definition:

- **source definition:**

$$\begin{aligned} fib(0) &= 1; && (2)\\ fib(1) &= 1; && (3)\\ fib(n+2) &= fib(n+1) + fib(n). && (4) \end{aligned}$$

We can give a formal specification for a program that computes the above definition as follows:

$$\forall input,\ \exists output.\ fib(input) = output \tag{5}$$

where fib is defined through three lemmata corresponding to the three branches, (**??**), (**??**) and (**??**), of the above course of values definition. Note that (**??**) is an instance of the specification schema, (**??**), given in §**??**.

The most natural way to synthesize a procedure for computing the Fibonacci numbers is to employ the course of values induction to (**??**). This is because it directly mirrors the course of values recursion exhibited by the standard Fibonacci definition. The induction schema of §**??** becomes instantiated as follows:

$$\frac{H:\ (\forall z, \forall y.((y < z) \rightarrow \exists n'.fib(y) = n') \vdash \exists n''.fib(z) = n''}{C: \vdash \forall x, \exists n.fib(x) = n}.$$

The proof of the induction conclusion, C, requires identifying an existential *witness* for n. That is, an instantiation for n must be provided that makes C true. Since this is a course of values proof, $fib(x)$ is constructed as a conditional, branching according to the value of y: first with a value for y of $x-1$, and subsequently with a value of $x-2$. The resulting constructs for $fib(x-1)$ and $fib(x-2)$ appear as two new hypotheses. These are then added to obtain a witness for n, i.e. $\vdash \forall x, fib(x) = fib(x-1) + fib(x-2)$.

Fig. **??**(a) depicts the computational trees for $fib(5)$ using course of values induction. Note especially the redundant (repeated) nodes in the tree for course of values induction. In order to calculate $fib(n)$ one must first calculate $fib(n-1)$ 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. Such a procedure is termed *tree recursive* since it resembles a tree where the branches split into two at each level.

Fig. **??**(a) can also be regarded as a *dependency graph*, DG, for the course of values recursive procedure since it is a representation of a particular function call's evaluation tree which shows the calling structure of the subsidiary recursive calls. Strictly speaking, fig. **??**(a) is a *grounded* DG, since it is constructed using grounded function calls. A *symbolic* DG, on the other hand, is based on symbolic function calls and is potentially infinite in size. The reader may wish to look ahead to fig.13, §**??**, which shows a portion of the symbolic DG for $fibn$.

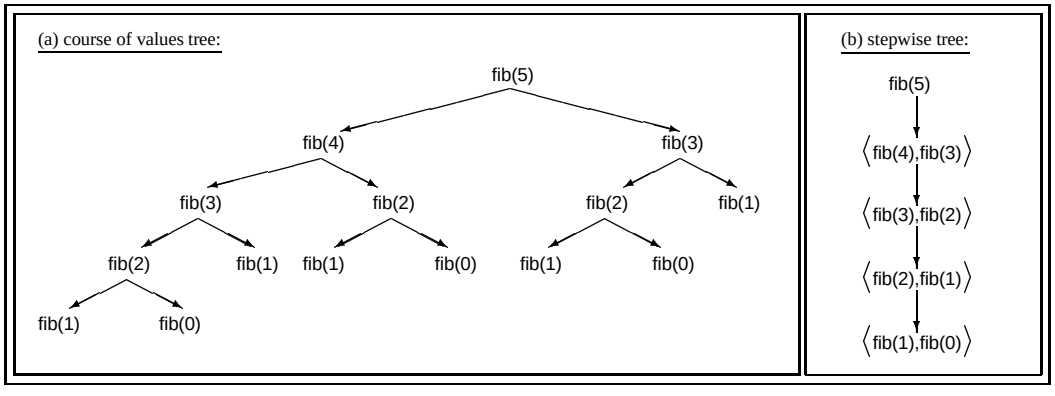


Figure 2: Computational tree for $fib(5)$ induced by (a) course of values induction, and, (b), stepwise induction

[7] Representations of the completed proofs are displayed, and examined, in §**??**.

3.2.2 Stepwise Induction and Tupling

Alternatively, we can also employ stepwise induction over the naturals to synthesize a program that computes the *same* specification, (**??**), as the previous course of values extract. This is achieved by employing tuple constructs, at the stepwise induction cases, in order to evaluate the Fibonacci numbers. Tupling removes redundancy by grouping together, or *merging*, potentially re-usable function calls – repeated computation – that appear in the tree recursive process generated by the course of values definition (*cf.* fig.2). The result of tupling in this case is *linearization*: the production of a stepwise recursive algorithm which computes the Fibonacci function, fib, through an auxiliary linear process g.

- **target definition:**

$$\begin{array}{rcl} fib(n) & = & m \quad where\ \langle \_, m \rangle = g(n); \\ g(0) & = & \langle 1, 1 \rangle; \\ g(n+1) & = & \langle u1 + u2, u1 \rangle\ where\ \langle u1, u2 \rangle = g(n). \end{array}$$

The auxiliary function $g(n)$ is constructed in terms of $g(n-1)$, where the first argument in both cases takes the "combined values" form (in effect, the tupling combines the values of the two step cases of the less efficient course of values definition). The linear trace for computing $fib(5)$ through the auxiliary procedural call $g(5)$, is depicted in fig. **??**(b): the angled brackets in the stepwise sequence symbolize *tuple formation* in that the output of each recursive pass is some function of the arguments within the brackets. The function g is defined in terms of a tuple that consists of two components, each of which are made up from subsidiary calls to fib: the first corresponds to the sum of $fib(n-1)$ and $fib(n-2)$, i.e. $fib(n)$. The second tuple component corresponds to the first argument of the first tuple component, $fib(n-1)$. The tuple functional applies the addition function to the first and second arguments. So the goal $g(n)$ is ultimately satisfied by defining it in terms of the known course of values definition, i.e:

$$g(n) \quad = \quad \langle (fib(n-1) + fib(n-2)), fib(n-1) \rangle. \tag{6}$$

Note that the first tuple component is equivalent to the body of the recursive step of the course of values definition. Note also that 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*, with a branching rate of 1, and hence the resulting algorithm requires far less computational effort in computing $fib(n)$ than that synthesized by employing course of values induction.

Regarding synthesizing a program to compute the stepwise procedure, the first step is to apply stepwise induction to (**??**). This yields the following (instantiated) schema:

$$\frac{\vdash\ \exists t^0.\ g(0) = t^0 \qquad \forall y, \exists t'.\ g(y) = t'\ \vdash\ \exists t''.\ g(s(y)) = t''}{\vdash \forall x, \exists t.\ g(x) = t}.$$

As with the course of values proof, the proof requires establishing witnesses for the existential quantifiers. In this case we are required to find existential witnesses for t; the tuple through which g is defined. At the base case of the induction, we simply employ symbolic evaluation using the terminating branches, (**??**) and (**??**), of the source definition in order to provide a witness, $\langle 1, 1 \rangle$, for t^0. A witness at the induction step case is provided by a process of unfolding the induction conclusion with the source definitional equations (notably (**??**)) until a match is found with the induction hypothesis (i.e. the body of (**??**)). This enables the unification of conclusion and hypothesis (**??**) there by providing a witness for t in terms of g (thus introducing recursion into the program).

Greater detail concerning both the above proofs is provided in §**??**, where we describe how such stepwise proofs are automatically constructed from source course of values proofs.

3.3 Background to the Tupling Technique

The PTS operates by using information in the source course of values proof to guide the automatic construction of the target stepwise proof. This research offers the first instance of the tupling technique being employed within the context of proof transformation (as opposed to the direct transformation of programs).

Existing systems that automate tupling transformations, within the context of program transformation, depend on an analysis of such graphs so as to obtain dependency information which guides subsequent transformation [**?**]. In §**??** we illustrate how, within the context of proof transformation, such dependency information can be read directly from the source proof thus circumventing the need for DG construction and analysis.

Tupling, originally developed as an optimization technique in [**?**], is a form of *tabulation*, albeit constructed in real-time, since the tuple represents a record of previous recursive calls. Tupling is an important means of linearizing exponential procedures. It works by grouping together, in a single recursive tuple function, the separate recursive expressions in the source procedure. The main advantage of tupling over the most general kind of table for redundant computation, *memo-tables* [**?**], is that we store only the subsidiary calls of a specified function, rather than calls from the whole program. In the case of memo-tables there is a heavy storage requirement as entries inserted during function execution, are not usually removed even if they are no longer required.[8]

Existing program transformation systems reported within the literature also employ the tupling technique in order to remove redundancy from recursive procedures (e.g. [**?**], [**?**, **?**], and later in[**?**]). However, these systems do not operate within the proofs as programs framework. The general strategy of program transformation employed by these systems originates from [**?**] and is referred to as the *unfold/fold* strategy. This strategy basically consists in defining the target program in terms of the source, and then, by a process of re-writing recursive definitions, deriving a recursive definition for the target program which is independent of the source definition. This general strategy has since been incorporated, in a variety of guises and applications, in many program transformation systems. The three most problematic steps in the unfold/fold strategy, regarding search, control and automation, are:

- the so called *eureka* step: obtaining the initial definition of the target in terms of the source ((**??**) in our example of §**??**);
- the control problems associated with when to apply the re-writing step(s) which eliminate any reference to the source definition from the target recursive step, and;
- the principled application of lemmas (or laws) often required to propagate the program derivations.

We shall return to a more detailed exposition of this related work in §**??** in order to explain how proof transformation offers a promising means of overcoming these problems. We shall compare the work described in this paper with that reported in [**?**]: a recent systemization and extension of the earlier transformation strategies discussed in [**?**].

4 Proof Transformation Strategy

The PTS is tuned to recognize the key positions within inductive proofs that have a decisive effect on the recursive behaviour of the extract algorithm. These key positions correspond to the application of an induction rule, the constructive type of the objects required to witness the induction cases, the actual proof constructs introduced to witness the induction cases, and finally the definitions chosen to complete the verification component of the proof.

Although the transformations involve using the source proof to *guide* the new construction of a target proof by mapping, and then transforming, portions of the former, the source proof, and extract, is itself preserved. This is an intentional design factor since, for some applications, it may prove desirable to have access to both the source and target proofs at the termination point of the transformation.

4.1 Abstracting Salient Features of the Proof

Proof trees are internally represented within OYSTER as quite complex Prolog data-structures.[9] However, these OYSTER data-structures, and the corresponding proofs, contain large amounts of information which is irrelevant to *both* execution and the tupling transformations. Hence inefficiency would result from this additional information being subject to extensive manipulation in the course of the transformations. To avoid computational effort being expended on attempting to access individual semantic units the PTS processes, by abstraction, the OYSTER internal proof representations into more accessible list structures called *rule-trees*. A typical rule-tree will either explicitly contain, or contain labels which allow for the direct accessing of, the following information:

- *Some* of the assumptions (hypotheses) made during the proof.

[8] However, memo-tables do have the advantage of being more general in their range of function applications.

[9] Within the pre-processed OYSTER representation there are many Prolog variables hanging on to the various (sub)lists and it is generally hard to follow what parts of information form semantic units.

- The branching structure of the proof.
- The rules applied along with any corresponding arguments.
- An account of the dependencies between facts in the proof:
  - dependency information concerning inter-relations between (sub)goals; and
  - dependency information concerning inter-relations between (sub)goals and assumptions (hypotheses).

So, recalling the Curry-Howard isomorphism, §**??**, the rule-trees contain an account of the dependencies between facts involved in the computation of the λ-function constructed by the corresponding proof.

Each rule entry consists of a refinement rule such that a rule-tree corresponds (schematically) to:

$$apply(Rule_1)\ then\ [apply(Rule_2)\ then[\ ...\ apply(Rule_n)]\ ...\ ]],$$

and as such is akin both to a proof plan which combines a number of proof tactics and/or rules into a large tactic such that a complete proof can be (re)produced from the plan, and to a skeleton of a proof in which the inference rules of the proof are recorded, but not the formulae to which they are applied. A source rule-tree contains all the information required to reproduce faithfully the source proof from which it is abstracted. Similarly, at the termination point of a transformation, the target rule-tree contains all the information required to produce the complete target proof (indeed, once constructed, target rule-trees are automatically applied as large tactics to the specification goal there by producing a complete target proof).

The fact that proofs are transformed indirectly via the transformation of the rule_tree proof tactics (or proof-plans) is not a necessary feature of the proof transformations but is rather employed for purposes relating to the efficiency of the actual transformation process. We only mention them here to establish that the internal proof representations of the PTS have no effect other than to increase the efficiency of the transformation process. In this paper we are primarily concerned with how information in the source proof is used to construct the target proof, and not with implementational detail. Hence, unless directly relevant, we shall in subsequent sections describe the proof transformation process as passing directly from source proof to target proof without the intermediate creation of the rule_tree abstractions.

4.2 Tactic Transformation: Conditionally Guided Proof Modification

The PTS transformations are, then, akin to meta-level tactic transformations guided in part by whether or not certain syntactic properties are true of the source proofs. Such syntactic properties function as transformation tactic pre-conditions. We can also predict the probable outcome of the application of a transformation tactic in terms of syntactic properties of the target proof. A source to target transformation will be deemed successful if the target proof satisfies the post-conditions.[10]

We can give fairly high-level pre- and post-conditions for the induction schema transformations. For example, transformations from an exponential procedure to a linear procedure include, amongst their pre-conditions, that the dominant induction in the proof is a course of values induction (i.e the proof must contain a cv_ind construct). Amongst the post-conditions will be the presence of a stepwise construct in the target proof. In §**??** we provide further pre- and post-conditions specific to the proof tupling transformations.

Similarly, transformations from a linear procedure to a logarithmic procedure have as a pre-condition that the dominant induction in the source proof is a stepwise schema. The target must then satisfy the post-condition of having a *divide and conquer* induction. We do not cover logarithmic transformations in this paper. A theoretical description of such transformations is given in [**?**], and we discuss systemizing such transformations in [**?**].

4.3 Efficiency, Correctness and Automation

The presence of a program specification both provides a termination condition and guarantees that all proofs tranformed by the PTS yield programs that are correct with respect to that specification (*cf.* fig.1). Traditional program transformation systems have no such formal specification and this this means there is no immediate means of checking that the target program meets the desired operational criteria. By proving that the target program satisfies the original specification, we avoid the need to establish that any re-write rules used are

[10] If the source proof satisfies the pre-conditions then only in exceptional cases will a complete target proof be produced which violates the post-conditions.

in themselves correctness (equivalence) preserving. This will, as a general rule, require as much effort as providing an explicit proof of correctness for the source to target transformations. For example, many of the systems that employ the *unfold/fold* strategy re-write the recursive step(s) of a source program through the application of various *equality* lemmas, each of which needs to be proved (by induction) if the source to target transformation is to preserve equivalence [**?**, **?**].

Furthermore, there is no guarantee that unfold/fold style derivations will actually lead to any optimization, where as proof transformations replace an induction yielding an inefficient recursion schema with one that yields a (more) optimal schema. Thus target programs are guaranteed to compute the input-output relation specified originally for the source, and to do so more efficiently.

Regarding automation, the proofs contain sufficient information to allow the source to target proof transformations to proceed without *any* user interaction. In other words, in forming proofs from source proofs, the PTS abstracts precisely that information which allows for the automatic construction of the target proof.

4.4 Synthesis and Verification

The *synthesis component* of the transformation process is concerned with the formation of the target tuple, the replacement of the source induction by a target induction with a more efficient induction rule (e.g., applying stepwise in place of course of values induction) and/or merging a nested induction structure in the source into a single induction in the target, and the subsequent witnessing of the target induction cases. The *verification component* is concerned with performing specific sequences of unfolding operations at the instantiated induction step using *both* source and target equations. Symbolic evaluation and well-formedness tactics are also usually applied at the induction cases.

We categorize the proof constructs mapped and/or transformed from the source proof according to which component of the proof is being transformed. The synthesis component will involve abstracting, and then transforming, (sub) structures from the source in order to:

(i) construct the target tuple;

(ii) determine the nature, and number, of elimination rule applications; and perhaps most importantly;

(iii) witness the existential quantifier at the target induction cases by mapping across structures from the source induction cases.

The connection between (ii) and (iii) is that the elimination rules employed within the proof, particularly those used in order to supply the source induction witnesses, provide an account of the inter-relations between (sub)goals and hypotheses. This dependency information is then used to supply witnesses for the target induction.

The verification component will involve abstracting, and then transforming, all those source proof branches associated with:

- tactics for controlling unfolding;
- well-formedness goals; (such as the applications of type-checking rules); and
- the application of lemmas – lemmas used for the satisfaction of the source induction cases are mapped across and, after some simple transformations, used by the unfolding tactics in order to satisfy matching target sub-goals.

Both synthesis and verification involve:

(1) the fairly extensive mapping, and subsequent transformation, of constructs from the source proof; combined with

(2) heuristic theorem proving strategies; and

(3) transformation techniques such as *tupling*.

With regard to (1), by matching target sub-goals with source sub-goals, the PTS determines to what extent it needs to patch the corresponding source proof branches in order to apply them successfully to the target sub-goals.

4.5 Tuple Construction

As well as the more general pre- and post-conditions for optimizing recursive programs through transforming the source induction, §**??**, we also give lower-level pre- and post-conditions which are specific to the tactic based proof tupling transformations:

1. *Pre-condition*: There exist two or more induction terms, $f'(n), ..., f'(n-i)$, which share some *common induction variable(s)* in a function definition (where $i \geq 2$).

2'. *Post-condition*: There must be present(constructed) a fixed sized tuple - the *eureka tuple* - within which common subsidiary function calls arising from the unfoldings of each of $f'(n), ..., f'(n-i)$ are merged, thus forming a recursive function without the original redundancy.

Note that condition 1 is, in effect, a defining condition of course of values induction. This means that any proof employing one, or more, course of values induction schemes will generally be a good candidate for optimization by tupling.

We shall refer to the tuple size, or the number of subsidiary calls tabulated within the tuple, as Φ. In general, i will provide an accurate, and the best, value for Φ. Regarding the induction step of the course of values schema,

$$\forall x, \forall y.\ ((y < x) \rightarrow P(y)) \vdash P(x),$$

the system evaluates the best tuple size by observing the source course of values schema and determining the number of times the induction hypothesis is invoked for different values of the induction parameter y. In other words, from a reading of how many distinct eliminations are performed on the induction hypothesis of the source, the system can automatically calculate the best value for Φ. The contents of the tuple are then those recursive calls corresponding to the Φ separate invocations of y.

A quick and simple heuristic for constructing the explicit target tuple definition is simply to form the target tuple structure by a direct 1-1 mapping of the function calls in the body of the source definition recursive step. This is not, of course, guaranteed to produce the best tuple, but it will not produce a target program any less efficient than the source. The system will not produce an erroneous target program by employing this heuristic, despite the fact that there are examples where an erroneous tuple would be produced by mapping the source recursive step.[11] This is simply because the target specification, identical to that of the source, cannot be satisfied by a proof employing an erroneous tuple function.

Functions which are constructed using schemas other than course of values induction can also satisfy condition 1 in an implicit sense. For example, a function, f_{+2}, synthesized using $(+2)s$ stepwise induction may well be a candidate for proof tupling since an invocation of $f_{+2}(s(s(n)))$ will require *two* subsidiary calls on $f_{+2}(s(n))$ and $f_{+2}(n)$. We formally display the $(+2)s$ schema and provide an example of proof tupling on an instance of f_{+2} in §**??**.

Regarding the transformation of nested inductions consider the following schematic definition:

$$f(n) = f_1(n) + f_2(n-1),$$

It may be the case that upon unfolding either, or each of, f_1 and f_2, two or more induction terms, $f_j(n), ..., f_j(n-i)$, which share some common induction variable(s) are exhibited. This is the case with auxiliary recursive functions wherein the redundancy is not immediately obvious since it occurs amongst the auxiliary recursive calls (viz. the computation of the function(s), in the body of the definition, which are not self-recursive). Such "auxiliary redundancy" manifests itself in the source proof in the form of a nested induction. The task of proof tupling on such nested induction structures is to "merge" the computation associated with the innermost induction with that of the outermost induction. Hence the explicit definition for the target tuple is determined by calculating the value of Φ, and the recursive calls to be tabulated, for the outer and (each of the) nested inductions and then simply combining the results. We shall illustrate by example the optimization of these kinds of inductively synthesized functions in §**??**.

It is worth noting that, in practice, tuples are represented in the OYSTER proofs by conjunctions of function calls. That is, the program extraction process sets up a correspondence between conjunction proof constructs and tuple program constructs. This approach has certain advantages to which we shall return in §**??**.

[11] For example, we would need to use the more rigorous approach to determine the tuple definition for a variant of Fibonacci with the following recursive step:

$$fib_3(n) \ = \ fib_3(n-1) + fib_3(n-3)$$

The quick heuristic would erroneously produce a tuple of size 2, i.e. $\langle fib_3(n-1), fib_3(n-3) \rangle$, whereas an analysis of a source course of values proof for fib_3 would reveal that 3 distinct invocations of (eliminations on) the induction hypothesis are required. Thus the correct tuple should be $\langle fib_3(n-1), fib_3(n-2), fib_3(n-3) \rangle$.

Henceforth, we shall distinguish proof transformations which employ a tupling technique from program tupling transformations by referring to the former as *proof tupling* and the latter as *program tupling*.

5 Proof Transformation: Examples

The proof transformations performed by the PTS can be broadly categorized in two ways:

1. **Transformation of induction schemas:** The source induction schema is replaced by a different, but logically equivalent, target induction schema.[12]
2. **Transformation of nested inductions**, or **Loop Removal:** A nested application of induction in the source is "merged" with the outermost induction to produce a target proof with a single induction. We may also refer to such transformations as *loop removal* (since a recursion loop is removed from the source).

Both 1 and 2 are automatic and involve essentially the same strategy: the system cuts in an extra goal, G into the "simple" proof of the program specification, S, thus yielding two subgoals: the first being the original goal S, with G as an additional hypothesis, and the second being G itself. The proof of (sub)goal G is then responsible for synthesizing the more efficient computation of the input-output relation specified in S. In both cases the need to treat the identification of G as a eureka step is removed by exploiting the structure of the source proof. Furthermore, the source proof provides the information required to witness the induction step of the target proof (and thereby build recursion into the target program).

In this section we provide detailed analyses of three examples of proof transformation which involve tupling. The first corresponds to linearization by the transformation of course of values induction schemas. The second corresponds to the transformation of nested inductions. The third example involves both the transformation of a source induction scheme and the merging of nested inductions. As well as combining aspects of the first two examples, it also illustrates the transformation of a different induction schema, $+2$ succesor induction, than that in the first example.

The reader should bear in mind throughout this section that we regard the construction of source proofs as given (i.e. either as output from a previous transformation, or from an interactive synthesis session within the OYSTER system). The construction of target proofs, on the other hand, is automatic given the source proofs as input. Thus although a comprehensive explanation requires us to provide a step by step description of the target proof constructions, the process is fully automated regarding the PTS.

5.1 Example 1: Linearization

Remaining with the Fibonacci example, we provide representative figures for synthesizing Fibonacci the source course of values proof, fig.3(a), and for synthesizing the target *stepwise* proof, fig.3(b). Taken as a whole, fig.3 depicts the correctness guaranteed transformation of a course of values proof to a stepwise proof. For the sake of clarity, we omit some of the type checking, substitution and elimination rules (such omissions being indicated by a broken vertical arrow). We shall have course to often refer back to fig.3 throughout the text. Thus to aid clarity we adopt the naming convention that symbols appearing in the text in calligraphic font refer to either the correspondingly named formulae, proof branches, or the arcs depicting proof mappings, of fig.3.[13] For example, we use the arcs, $\mathcal{M}1$ to $\mathcal{M}8$, that pass from fig.3(a) to fig.3(b) to depict those (sub)structures of the source proof which are used to develop the target proof. These "mappings" will be explained in §**??**. We shall first describe the nature of the source proof (i.e. fig.3(a)). The nature of the target proof construction, fig.3(b), will become evident when we discuss the transformation of the source (§**??**).

5.1.1 The Source (Course of Values) Proof

The specification, $\mathcal{FIB}$, for a program that computes the Fibonacci numbers, is shown below:

$$\mathcal{FIB}\colon \forall x,\ \exists y.\ fib(x) \ = \ y, \tag{7}$$

fib is defined through the use of three proved, and subsequently stored, lemmas corresponding to the three cases of the course of values definition (§**??**):

[12] By logically equivalent induction schemas we mean that the associated induction theorems are inter-derivable. This guarantees that any two proofs satisfying the same complete specification but differing only in which of the two schemas are employed are *functionally equivalent*.

[13] The same convention is adopted regarding the later examples and their corresponding proof figures.

lemma 1: $fib(0) \quad = \; s(0)$;
lemma 2: $fib(s(0)) \; = \; s(0)$;
lemma 3: $\forall x, \exists y_1, \exists y_2.\; x \neq 0 \wedge x \neq s(0) \wedge \; fib(p(x)) = y_1 \wedge fib(p(p(x))) = y_2 \; \rightarrow \; fib(x) = y_1 + y_2$,

where p is the $predecessor$ function defined by induction over the naturals such that $fib(x-1) \equiv fib(p(x))$ and $fib(x-2) \equiv fib(p(p(x)))$.[14] The p operator is usefully employed as a destructor function of a function's data-structure (as opposed to using the canonical successor function, s, to build constructor definitions). The reason for specifying Fibonacci indirectly, through the use of proved lemmas, is so that the proof specification, (**??**), does not constrain the dominant induction of the proof to course of values (since in the case of the target proof we will wish to construct a stepwise proof of (**??**)).

Lemmas 1 and 2 define the base cases of the Fibonacci definition. Lemma 3 defines the recursive case and is naturally a course of values definition: values are given for inputs 0 and $s(0)$, and $fib(x)$ requires appealing to a pair of output values obtained when the input is less than x, specifically, $fib(p(x))$ and $fib(p(p(x)))$. A ramification of the induction-recursion duality is that the behaviour of the induction variable should mirror that of the recursive terms in the function definitions [**?**]. Hence (**??**), or $\mathcal{FIB}$, is most naturally proved by course of values induction. The proof requires an initial application of the $\forall - intro$ refinement. This has the effect of removing the universal quantifier.[15] This is followed by applying course of values induction on x (denoted, in fig.3(a) by CV induction(x)). The cases of the induction schema are then satisfied by setting up a nested case analysis structure by performing two case-split refinements, where the second case-split is nested

[14] Depending on context, we shall subsequently use the postfix notation, e.g. $(x-1)$, interchangeably with the prefix notation, e.g. $p(x)$ (similarly for, e.g., $x+1$ and $s(x)$).

[15] Recall, §**??**, that a feature of the goal-directed proofs is that introduction ($intro$) rules have the *quantifier stripping* effect usually associated with elimination rules in forwards proof systems. Conversely, elimination ($elim$) rules have the effect of introducing an existential instantiation in the hypotheses of sequents.

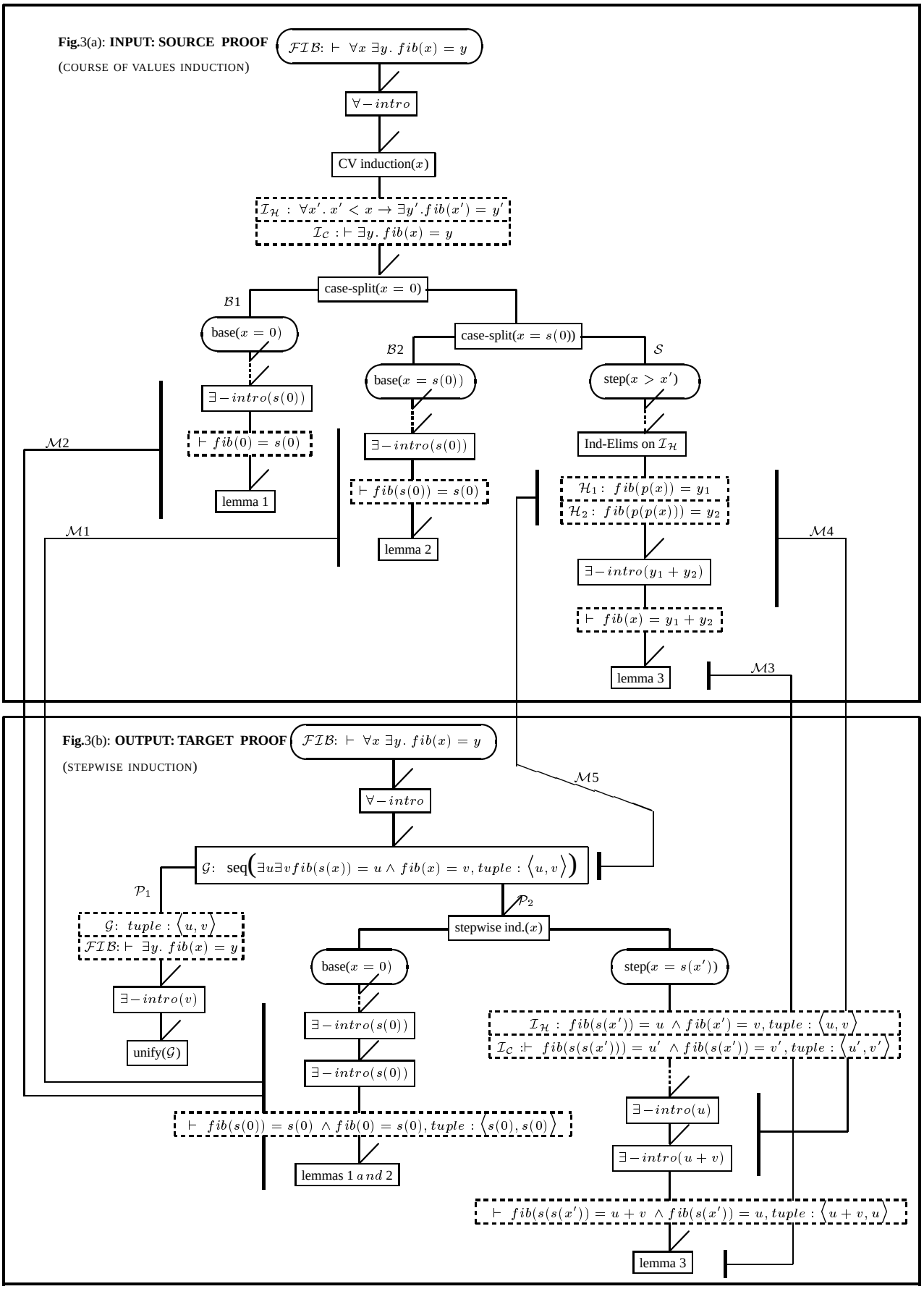


Figure 3: Schematic Representation of Source to Target Proof Mappings for Fibonacci

within the first. The outermost case split corresponds to $x = 0 \vee x \neq 0$, and the innermost case to split to $x = s(0) \vee x \neq s(0)$. By having the case splits nested in this way, we cover all the conditions specified in the course of values definition. By using the $\exists - intro(\omega)$ rule, a suitable witness, ω, is introduced at each case, and then verification is performed by appealing to (unfolding with) the relevant lemma (with various well-formedness goals being satisfied along the way). Within the dashed-boxes (fig.3) we have included key hypotheses and (sub)goals (conclusions): the application of course of values induction yields the induction hypothesis, $\mathcal{I}_{\mathcal{H}}$,

$$\mathcal{I}_{\mathcal{H}}\colon\ \forall x'.\, x' < x \rightarrow \exists y' fib(x') = y',$$

and the induction conclusion, $\mathcal{I}_{\mathcal{C}}$,

$$\mathcal{I}_{\mathcal{C}} : \vdash \exists y.\, fib(x) = y.$$

At the two base cases, $\mathcal{B}1$ and $\mathcal{B}2$, we provide in both cases, a witness of $s(0)$.

The goal at the induction step case, $\mathcal{S}$, is to reduce the induction conclusion, $\mathcal{I}_{\mathcal{C}}$, to terms which can be unified with those in the induction hypothesis, there by providing a witness for the existential variable – y in the case of $\mathcal{I}_{\mathcal{C}}$ – which introduces recursion into the step branch of the function. This is achieved by:

- eliminating on the induction hypothesis, $\mathcal{I}_{\mathcal{H}}$, twice: first with a value for x' of $p(x)$, and subsequently with a value of $p(p(x))$.[16] In fig.3(a) this is depicted by the term "Ind-Elims on $\mathcal{I}_{\mathcal{H}}$". The constructs resulting from the eliminations appear as two new hypotheses, $\mathcal{H}_1$ and $\mathcal{H}_2$, which provide outputs for $fib(p(x))$ and $fib(p(p(x)))$, named y_1 and y_2 respectively; and,
- recursion is then built into the function being constructed by using $\mathcal{H}_1$ and $\mathcal{H}_2$ as unifiers, or *fertilizers*, to provide a witness for the step case $fib(x)$, namely $y_1 + y_2$. This completes the recursive branch synthesis (since $y_1 + y_2 \ \equiv\ fib(p(x)) + fib(p(p(x)))$).

Thus, to witness a value for the induction step we appeal, twice, to the induction hypothesis $\mathcal{I}_{\mathcal{H}}$. These eliminations on the induction hypotheses, and the fact that they are explicitly recorded in the sequent hypothesis lists, will be seen to be crucial for the automatic construction of the target induction (§**??**).

Upon completion of the synthesis component of the target proof, verification is performed by appealing to the stored lemmas: lemmas 1 and 2 for the base cases, and lemma 3 for the step case.

The unification of the induction conclusion with the hypothesis is called *fertilization*. Formulae are "unpacked" - or *unfolded* - by replacing terms by suitably instantiated definitions. Fertilization is facilitated by the fact that the induction conclusion is structurally very similar to the induction hypothesis except for those function symbols which surround the induction variable in the conclusion. An implemented rewriting technique known as *rippling* exploits this property of inductive proof by proliferating the process of unfolding such that recursive terms are gradually removed from the recursive branches until a match – fertilization – can be found with the induction hypothesis [**?**]. In §**??** we say a little more concerning the general inductive proof strategy and how this has positive ramifications regarding the completeness of the proof transformation system.

5.1.2 The Source (Course of Values) Extract Program

The complete extract program results from the combination of all the separate proof branch constructions appearing at the proof branch leaves of the first base case, second base case, and step case respectively. We indicate, in fig.4, the input/output associated with each case computation in the λ-calculus representation of the complete extract program (*cf.* §**??** for an explanation of the cv_ind proof construct). The program construction associated with a case analysis is of the form $eq(x, y, P, Q)$, which specifies the required decision procedure: if $x = y$ then P, otherwise Q.

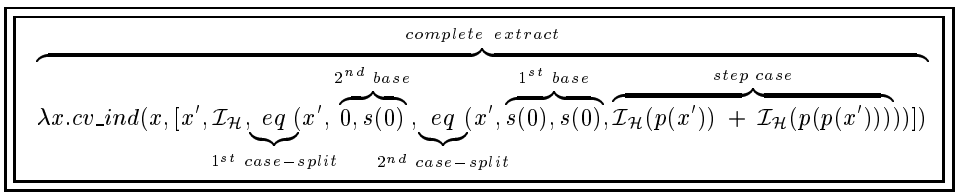


Figure 4: The course of values extract for $\mathcal{FIB}$

[16] Following these eliminations, the proof also requires us to establish that both $p(x) < x$ and $p(p(x)) < x$ are true.

The λ-calculus functional program extracted from the course of values inductive proof will compute the Fibonacci numbers according to the course of values definition (corresponding to the three lemmas). The proof reflects the same inefficiency generated by the extract program. This could not be otherwise since the procedural commitments and/or decisions made during the synthesis determine the nature of the recursive process generated by the synthesized (extract) program. The extract program dictates that in order to compute the step branch of the recursion the induction hypothesis, $\mathcal{I}_{\mathcal{H}}$, is evoked twice. This means that the recursive process generated by the extract program will be exponential (i.e. the tree recursive represented by the dependency graph of fig. **??**(a), §**??**).

It is clear, therefore, that there is a one-to-one correspondence between terms in the extract and terms in the proof from which it was extracted. However, it should also now be clear that the correspondence is not bi-directional: the course of values proof contained many steps which are not reflected in the extract program. Notably, due to the absence of anything resembling a hypothesis list, the extract program does not contain a record of the dependencies between facts involved the computation. Nor does it contain a complete representation of the verification component(s) of the proof (required for establishing the correctness of the computation). This provides a graphic illustration of how proofs contain information which is extraneous to that required for simple execution, but valuable for understanding the program design.

5.1.3 The Target Proof Construction: Exploiting Source Dependency Information

Regarding fig.3(a), if one looks at the source proof branch corresponding to the step case of the course of values induction then we can represent the proof nodes constituting the synthesis part of this branch as in fig.5 below.

induction hypotheses:	$\mathcal{I}_{\mathcal{H}}: \ \forall x'.\, x' < x \ \rightarrow \ \exists y'.\, fib(x') = y'$
refinement:	Ind-Elims on $\mathcal{I}_{\mathcal{H}}$
resultant hypotheses:	$\mathcal{H}_1: \ p(x) < x \ \rightarrow \ \exists y_1.\, fib(x-1) \ = \ y_1$
	$\mathcal{H}_2: \ p(p(x)) < x \ \rightarrow \ \exists y_2.\, fib(p(p(x))) \ = \ y_2$
induction conclusion:	$\mathcal{I}_{\mathcal{C}}: \ \vdash \ \exists y.\, fib(x) \ = \ y$
witnessing refinement:	$\boxed{\exists - intro(y_1 + y_2)}$
fertilized conclusion:	$\vdash \ fib(x) \ = \ y_1 + y_2$

Figure 5: Elimination and Witnessing Steps of Source (Course of Values) Proof

Fig.5 represents the elimination and subsequent witnessing step required for the fertilization of the induction conclusion with the hypothesis. From this information, the PTS can extablish that the source proof satisfies the pre-condition for proof tupling (§**??**): that there are two or more induction terms which share the common variable, x, at the induction step of the proof construction.

In order to identify the (eureka) tuple, the PTS records the maximum difference between the induction term in the induction conclusion and the *smallest* of the subsidiary calls used to witness a value for y.[17] Since $p(p(x)) < p(x)$, and the induction term x is greater than $p(p(x))$ by 2 then the required tuple size, Φ, is 2. So in order to calculate $fib(x)$, for any x, the PTS must "store", or tabulate, 2 subsidiary calls: $fib(p(x))$ and $fib(p(p(x)))$. Thus, in order to determine the size and contents of a target (tuple) definition, the PTS observes: how many times the hypothesis, $\mathcal{I}_{\mathcal{H}}$, is evoked in order to provide a witness at the induction conclusion $\mathcal{I}_{\mathcal{C}}$, and; the greatest number of applications, 2 in the case of Fibonacci, of the induction constructor/destructor function the proof employs when eliminating on the induction hypothesis in order to synthesize constructs for the induction witnesses. This procedure completely identifies an explicit definition, $\mathcal{G}$, for the auxiliary recursive procedure through which Fibonacci can be defined:

$$\mathcal{G}: \ \mathsf{seq}\big(\exists u, \exists v.\, fib(s(x)) = u \ \wedge \ fib(x) = v,\ tuple: \langle u, v \rangle\big)$$

Hence, by having access to the OYSTER internal proof representations of the source elimination and witnessing steps, the PTS has all the information needed for the automatic generation of the target tuple definition (depicted by $\mathcal{M}5$ of fig.3). Following the mapping across of the initial portions of the source proof — the specification and the $\forall$–$intro$ applications — $\mathcal{G}$ is cut into the target proof as a new fact. In effect, $\mathcal{G}$ is a nested specification goal that states the existence of a tuple of two components (i.e., $\Phi = 2$). Such new facts are cut into proofs,

[17] By smallest we mean that subsidiary call which has the greatest (least) number of applications of the induction destructor (constructor) function applied to the induction variable.

as a new sub-goal, by a generalized version of the *sequence*, or seq, rule. The generalized seq rule allows one to cut in, or sequence, a new fact into a proof by introducing a new node in the proof tree with two subgoals where: the first subgoal represents the original proof tree with the new fact as an additional hypothesis (which in constructive terms amounts to an additional hypothesis that there is a proof of the new fact), and; the second subgoal is responsible for constructing a proof of the new fact. So, sequencing $\mathcal{G}$ into the developing target proof produces the corresponding two sub-goals:

- the first (sub)goal, at proof branch $\mathcal{P}_1$ in fig.3(b), will be the original $\mathcal{FIB}$goal, with the universal quantifier removed, and with $\mathcal{G}$ as an additional hypotheses, and
- the second (sub)goal, branch $\mathcal{P}_2$, will require proving $\mathcal{G}$ itself.

Stepwise induction is applied at the second subgoal in order to prove the sequenced in goal $\mathcal{G}$ (this is denoted, in fig.3(b), by "stepwise ind.(x') to $\mathcal{G}$": i.e. stepwise induction on x' is applied to $\mathcal{G}$). At the base case an $\wedge - intro$ rule is applied which has the effect of decomposing the goal into the separate tuple components. Such decomposition of the tuple will always be controlled by the tuple size, Φ. The PTS then maps across the base case witnesses, 0 and $s(0)$, from the source proof in order to witness a base case value for each of the tuple constituents u and v ($\mathcal{M}1$ and $\mathcal{M}2$, fig.3). The base case is then verified by mapping across and applying the source base case lemmas and well-formedness tactics.

At the induction step we have the following goal to prove (of the form *hypothesis* $\vdash$ *conclusion*):

$$\exists u, \exists v. fib(s(x')) = u \ \wedge fib(x') = v \ \vdash \ \exists u', \exists v'. \ fib(s(s(x'))) = u' \ \wedge fib(s(x')) = v',$$

i.e., regarding figs.3(b) and 5, the PTS must establish that $\mathcal{I}_{\mathcal{H}} \vdash \mathcal{I}_{\mathcal{C}}$. The PTS must then provide witnesses for u' and v' in the conclusion. Furthermore, it must do so in terms of u and v in the hypothesis. This will both introduce recursion into the target function and eliminate all reference to the source fib function from the target definition. An application of $\wedge - intro$ splits the induction conclusion, into separate conjuncts producing two new sub-goals (the number of applications of $intro$ being determined by Φ):

$$\vdash \quad \exists u'. \ fib(s(s(x'))) \quad = \ u' \tag{8}$$

$$\vdash \quad \exists v'. fib(s(x')) \quad = \ v' \tag{9}$$

A witness for u', in (**??**), is required which is equal to $fib(s(s(x')))$: since, in this example, $\Phi = 2$, then a value for u' is obtained by appealing to those two subsidiary calls which take recursive arguments that differ from $s(s(x))$ by, respectively, 1 and 2 applications of the successor function s, i.e. $fib(s(x))$ and $fib(s(s(x)))$.[18] These subsidiary calls are precisely those labelled u and v in the induction hypothesis. However, to avoid all charges of eurekas, the PTS must automatically determine what function to apply to u and v in order to construct the witness for u'. This is done by observing the witnessing step of the source proof: a call to the main function requires *adding* the Φ subsidiary calls (*cf.* the **witnessing refinement** slot of fig. 5). Thus the identity of the first tuple component is provided by substituting the subsidiary calls in the target induction hypothesis for those in the source induction conclusion (depicted by $\mathcal{M}4$, fig.3), there by witnessing a value for u' of $u + v$. A similar analysis of the source proof could be performed to identify the second component of the target tuple corresponding to (**??**). However, a witness, v, for v' is provided by one of the target hypotheses and can hence be directly appealed to in order to witness a value for the remaining component. Once the witnessing steps have been completeted, the instantiated, or fertilized, conclusion is verified by appealing to the same tactics for unfolding *and* the same lemma, lemma 3, as used to verify the source induction step ($\mathcal{M}3$, fig.3). This completes the construction of the target proof, fig.3(b), which is then passed on to the OYSTER automatic program extraction process (§**??**).

So, by utilizing the eliminations and witnesses in the source proof induction, the PTS is able to automate the difficult tuple construction process which, within existing program transformation, systems has constituted a eureka step. We elaborate on this performance advantage in §**??**.

5.1.4 PTS Lemma Translation

Regarding the use of lemmas, the PTS is equipped with a simple translation procedure that turns a destructor type lemma of the form:

$$f_1(x) \quad = \quad f_2(f_1(x-a), f_1(x-b)), \ \ where \ b \geq a,$$

[18] In the general case, if $\Phi = n$ then a tuple of size n is constructed, and the value of n subsidiary calls would be required to construct a witness for the 1st component, $n-1$ for the second, and so forth.

into a constructor version of the following form:

$$f_1(x+b) \quad = \quad f_2(f_1(x+(b-a)), f_1(x)).$$

Hence there is no problem in using source proof lemmas that define a function $f(x)$ in terms of *predecessors* of x, since, if necessary, we can translate it into the equivalent lemma that defines $f(x)$ in terms of *successors* of x.

5.1.5 The Target (stepwise) Extract Program

The lambda calculus extract program, shown in fig.6, for the target stepwise proof is somewhat more esoteric than the more standard representation of the stepwise recursive Fibonacci that we gave in §**??**. The basic explanation of the p_ind proof construct was provided in §**??**. The unfamiliar construct is the $spread$ function. The $spread$ function takes a pair (first argument) and a list (second argument) specifying two variables and a term which may include them; on execution the function returns this term with the variables substituted by the elements of the pair.

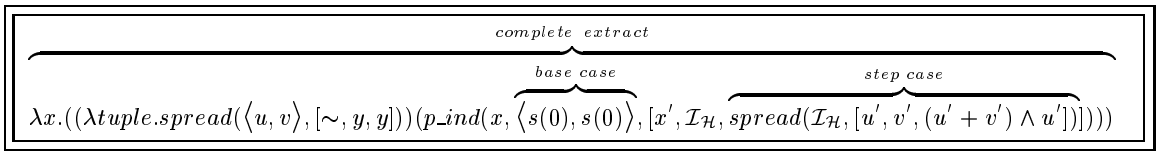


Figure 6: The stepwise extract for $\mathcal{FIB}$

So, regarding fig.6, the innermost $spread$ term (that constructed through the $\mathcal{P}_1$ branch of fig.3(b)) specifies that the two components, u and v, of the pair (tuple), $\mathcal{I}_\mathcal{H}$, whose existence is assumed through the induction hypothesis, are substituted, respectively, for u and v in the term $(u+v) \wedge u$. The outermost spread term (that constructed through the $\mathcal{P}_2$ branch of fig.3(b)) specifies that the output for Fibonacci is obtained by substituting the second element of the tuple, synthesized through $\mathcal{P}_1$, for y in the root node specification. Note that the stepwise extract, as in the stepwise proof, contains only a single evocation of the induction hypothesis, $\mathcal{I}_\mathcal{H}$. The recursive process generated by the stepwise extract is hence linear.

It is the use of tupling which allows us to construct such a linear process: the solution for Fibonacci corresponds to v in the above extract (i.e., the second argument of the first tuple component). Parameter u acts as an accumulator since its value in successive invocations accumulates the value(s) of the function. So, the process generated is *linear recursive* since, with u and v initialized to 1 and 0 respectively, the procedure applies the simultaneous "transformations" shown on the l.h.s. of the following informal equivalence (where $A \longmapsto B$ means P "transforms" to B),

$$\left\{ \begin{array}{lcl} u & \longmapsto & u+v \\ v & \longmapsto & u \end{array} \right\} \equiv \{\langle u, v\rangle \longmapsto \langle u+v, u\rangle\} \textit{ where } u = fib(i) \textit{ and } v = fib(i-1), \textit{(for some } i\textit{)}.$$

This represents a single recursive call where to obtain $\langle u+v, u\rangle$ we require a single evocation of the induction hypothesis construction, corresponding to $\langle u, v\rangle$.

So after applying this "transformation" n times then u and v will be equal to $fib(s(n))$ and $fib(n)$ respectively, i.e., (schematically),

$$\left\{ \begin{array}{lcl} u & \longmapsto & u+v \\ v & \longmapsto & u \end{array} \right\} \times n \equiv \langle fib(s(n)) + fib(n), fib(s(n))\rangle.$$

5.1.6 Scope of Induction Schema Transformations

In this section we provide an indication of the performance of the PTS as currently implemented. Although the PTS should currently be regarded as in an embryonic form, it is capable of linearizing, through the transformation of source proof induction schemas, a large class of program characterized by what Cohen describes as the *common generator redundancy*, CGR, class of programs [**?**]. This class is represented by the below schematic definition for a function f, with n self-recursive calls, and where d_1, d_2, ..., d_n, are *descent functions.* Descent functions are those functions which are applied to the main recursive arguments used in subsidiary calls.[19]

$$\begin{array}{rcl} f(x) & \Leftarrow & \textbf{if } b(x) \textbf{then } c(x) \\ & & \textbf{else } h(x, f(d_1(x)), ..., f(d_n(x))). \end{array}$$

[19] So, for example, there are two subsidiary recursive calls entered in the Fibonacci source course of values proof in order to satisfy the induction step, $p(x)$ and $p(p(x))$. The corresponding two descent functions for the two subsidiary calls are in both cases the predecessor function p.

The CGR class of programs are those programs where there exists a *common descent function*, δ, in terms of which each of d_1, d_2, ..., d_n can be defined. This means each descent function is related to each other through δ in that each is cashed out in terms of applying δ a certain number of times, i.e., $d_1 = \delta^i$ and $d_2 = \delta^j$, where δ^n is to be interpreted as the application of δ n times.

The general schematic function, shown above, for the CGR class of programs can hence be re-represented by $S1$ below:[20]

$$(S1)\ f(x) \Leftarrow \textbf{if } b(x) \textbf{ then } c(x) \textbf{ else } h(x, f(\delta^i(x)), ..., f(\delta^j(x)))$$

For the sake of brevity, we illustrated the transformation process using a fairly simple bi-linear instance of $S1$, namely Fibonnacci. However, the PTS will optimize any instance of $S1$.[21] For further examples the reader is referred to [**?**].

5.2 Example 2: Optimization By Transforming Nested Inductions

With the Fibonacci example of §**??** the optimization was achieved through transforming the source induction schema into a different schema with a more efficient computational rule. We now illustrate, by example, how the PTS is capable of transforming a source proof that involves a nested application of induction to a target proof with a single induction.

Our second example concerns the optimization of a program that computes the *factlist* function, $fctl$, with the following definition:

$$fctl(0) = [\,]; \tag{10}$$
$$fctl(n) = fact(n) :: fctl(n-1), \tag{11}$$

where the auxiliary function $fact$ is defined as follows:

$$fact(0) = 1; \tag{12}$$
$$fact(n) = n \times fact(n-1). \tag{13}$$

Here redundancy does not occur directly due to any self-recursive call but rather among the auxiliary recursive *fact* calls. This redundancy is exhibited by the symbolic dependency graph for $fctl$, the initial portion of which is shown in of fig.8. Recall from §**??** that a *symbolic* DG is based on the calling structure of subsidiary symbolic function calls (and is therefore potentially infinite in size). The multiple evocations of subsidiary calls, the redundancy pattern, is exhibited by more than one arrow directed at any particular node.

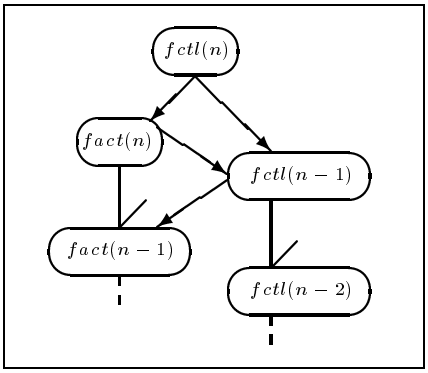


Figure 8: The symbolic DG for $fctl(n)$

A program for computing this inefficient procedure is synthesized from the following specification, $\mathcal{FL}$, for the factlist function:

$$\mathcal{FL}: \ \forall x, \exists l : list.\ fctl(x) = l.$$

As with the Fibonacci example, $fctl$ is defined through the use of lemmas, four in this case, which correspond to the terminating ((**??**) and (**??**)) and recursive ((**??**) and (**??**)) branches of the above definitions. However,

[20] The CGR class also covers the class of programs, referred to by Cohen as the *explicit redundancy* class, where $d_1 = d_2$.

[21] In fact $S1$ is a slight simplification since h may differ depending on which subsidiary call to which it is appled. thus the $\mathcal{MPTS}$ will also, for example, transform a source proofs of the following function $f'(n) = f'(n-1) \times (f'(n-3) + f'(n-4))$.

unlike the source synthesis proof for the Fibonacci function, $fctl$ is defined by a stepwise recursion schema – since $fctl$ function does not invoke itself more than once at each recursive call – and so is therefore most naturally synthesized using stepwise induction.

In fig.7 we provide a diagram that, as with fig.3, depicts the source and target proofs, and the (sub)structure mappings between them.[22] The redundancy manifests itself in the source proof, fig.7(a), in the form of the

[22]The same conventions apply to fig.7 as did to fig.3: symbols in the text in calligraphic font refer to the corresponding symbols in fig.7; terms such as "step($x = s(x')$)" denote that the induction variable, x, in the hypothesis is instantiated to $s(x')$ in the conclusion, and; terms such as "stepwise ind.(x') to $\mathcal{G}$" mean stepwise induction on x' is applied to $\mathcal{G}$. We also, due to space constraints, abbreviate some formulaes with $\ldots$, and omit some of the $\forall - intro$ applications.

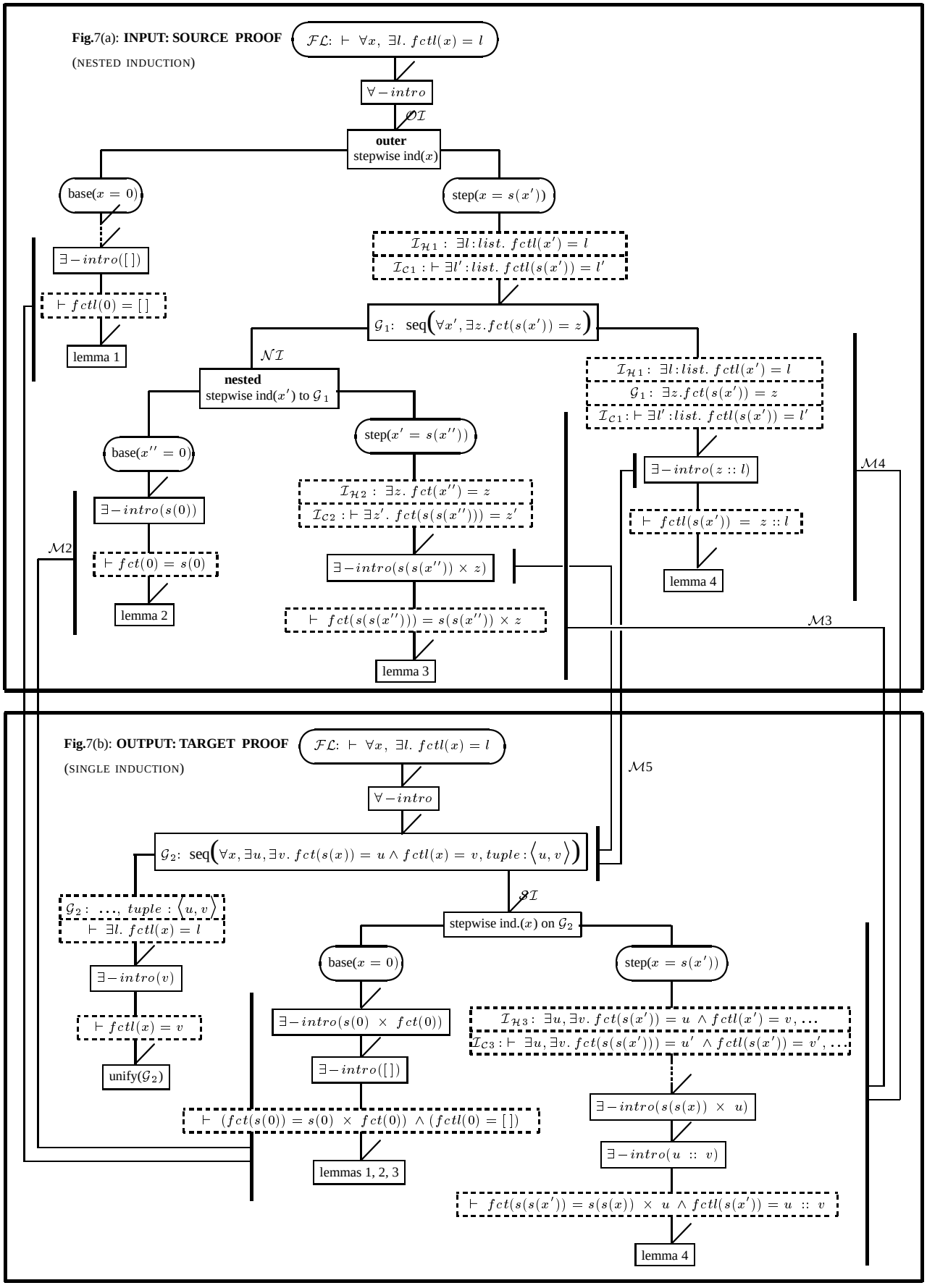


Figure 7: Schematic Representation of Source to Target Proof Mappings for *Factlist*

nested stepwise induction, $\mathcal{NI}$, required to synthesize an extract term for the auxiliary $fact$ call. The nested induction requires a prior sequencing step, at the induction step of the outer induction, to cut in the specification goal $\mathcal{G}_1$ for the $fact$ sub-routine:

$$\mathcal{G}_1\text{: } \mathsf{seq}\big(\forall x', \exists z.fct(s(x')) = z\big).$$

The nested schema means that for each recursive pass corresponding to the outermost induction, $\mathcal{OI}$, the source program must fully recurse on the innermost schema. This is also reflected by the dual nested recursion schema construct of the source proof extract program, a simplified representation of which is shown in fig.9: the p_ind function defines stepwise recursion and is evoked by the application of the corresponding induction. The nested p_ind structure mirrors the nested induction structure of the source proof. Thus if the induction variable, x' is 0 then the output is nil, otherwise the output is $z :: l$, where z is provided by the induction hypothesis $\mathcal{I}_{\mathcal{H}2}$, of the *nested* induction on x'', and l is provided by the induction hypothesis, $\mathcal{I}_{\mathcal{H}1}$, of the *outer* induction on x'. So the nested inductive proof provides an output, z, for $fact(s(x'))$, which is then used in the computation, $z :: l$, for $fctl(s(x))$ (i.e. $z :: l$ serves as a witness for the outer induction conclusion $\mathcal{I}_{\mathcal{C}2}$).

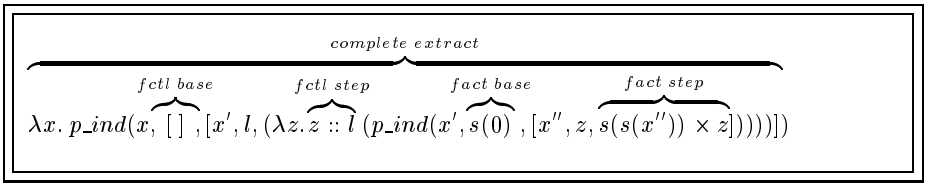


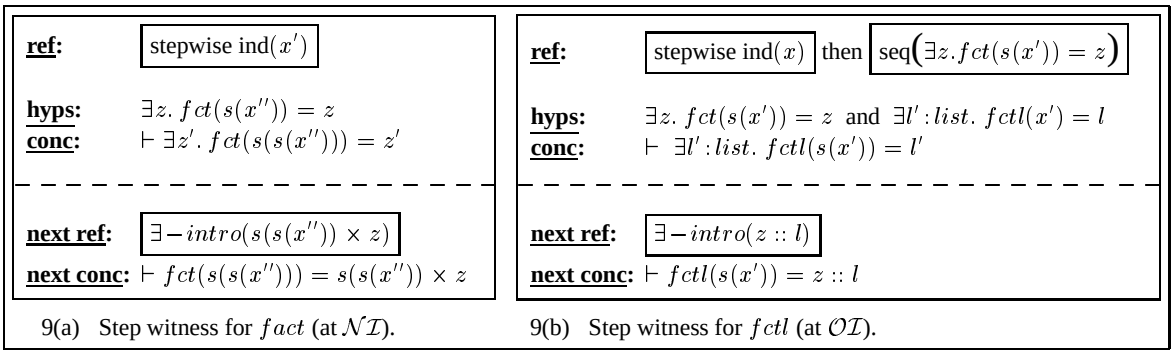
Figure 9: The Source Extract for $fctl$

So the task of the PTS transformation is to remove this nested induction, and thereby the redundancy caused by the nested recursion, by effectively specifying the auxiliary call at the level of the outermost induction.

5.2.1 Exploiting Dependency Information for the Target Construction

As with the source to target transformation of self-recursive functions, the optimization of the source auxiliary recursive $fctl$ function involves proof tupling and the exploitation of dependency information contained in the source proof. The step case existential witnesses of the inner and outer inductions of the $fctl$ source proof are expressed in terms of the source induction hypotheses (necessarily since the λ-function constructed is recursive). These witnesses are directly exploited in order to satisfy the single step case of the target proof.

In fig.10 we have represented the witnessing steps of both the source proof inductions (i.e. the outer and inner inductions of fig.7(a)). Fig.10(a) corresponds to the witnessing of the existential variable at the step case of the *nested* induction, and fig.10(b) to that of the *outermost* induction.



9(a) Step witness for $fact$ (at $\mathcal{NI}$). 9(b) Step witness for $fctl$ (at $\mathcal{OI}$).

Figure 10: The witnessing steps of the source $fctl$ proof

The PTS is able to determine from the above witnessing steps of the source proof, and from the subsequent unfoldings with the lemmas, that the recursive definition of the target tuple requires tabulating two function calls (i.e. $\Phi = 2$): there is one elimination performed on the respective induction hypothesis at each of the two inductions in order to provide a witness at the respective step cases (thus introducing recursion in the main and auxiliary functions being constructed). The actual witnesses tell us that the first is an occurrence of the auxiliary $fact$ function which takes the same argument, n, as in the head of the definition. The other tabulation is a subsidiary $fctl$ call which takes the predecessor, $n - 1$, of the argument n in the head of the definition.

The two arcs corresponding to $\mathcal{M}5$ of fig.7. depict the mapping of information from the source proof in order to identify the requisite tuple. The target definition is given the hypothesis label $tuple$ and, as in the Fibonacci example, is expressed as a conjunction and sequenced into the target proof as a new fact $\mathcal{G}_2$:

$$\mathcal{G}_2\colon\ \mathbf{seq}\big((\forall x, \exists u, \exists v.\ fct(s(x)) = u \wedge fctl(x) = v),\ tuple : \langle u, v\rangle\big).$$

Stepwise induction is then performed on the sequenced in goal (where the induction variable, x, is the same as that for the outermost application of induction in the source proof).

At the induction step of the target proof, $s(x)$ in the hypothesis, $\mathcal{I}_{\mathcal{H}3}$, is instantiated to $s(s(x))$ in the conclusion, $\mathcal{I}_{\mathcal{C}3}$, yielding:

$$(\exists u', \exists v'.\ fct(s(s(x))) = u' \quad \wedge \quad fctl(s(x)) = v'),\ tuple : \langle u', v'\rangle. \tag{14}$$

Both the tuple components (conjuncts) u' and v', of **(??)**, unfold to terms that are provided by mappings from the source proof:

- $fact(s(s(x)))$ is equivalent to $s(s(x)) \times fact(s(x))$ where $fact(s(x))$ matches the hypothesis $u = fact(s(x))$. Hence we require a witness value for the first tuple component of $s(s(x)) \times u$. This is obtained by mapping across the witness for the source nested induction and substituting u' for z. In fig.7 this corresponds to $\mathcal{M}3$.
- $fctl(s(x))$ is equivalent to $fact(s(x)) :: fctl(x)$ where $fact(s(x))$ matches the hypothesis $u = fact(s(x))$, and where $fctl(x)$ matches the hypothesis $v = fctl(x)$. Hence we witness a value for the second tuple component of $u :: v$. The PTS obtains this witness simply by substituting the target hypothesis labels, u' and v', for the labels, z and l', in the step case witness of the *outermost* source induction (depicted by $\mathcal{M}4$ of fig.7)

As with the previous examples, the base case witnesses are mapped across, one on one, from the source, as are the lemma applications required for verifying both the base and step case witnesses ($\mathcal{M}1$ and $\mathcal{M}2$ of fig.7).

The completed target proof constructed by the PTS, corresponding to fig.7(b), is then passed on to the OYSTER extraction process.

5.2.2 The Target (stepwise) Extract Program

The target program construction is shown below in fig.11.

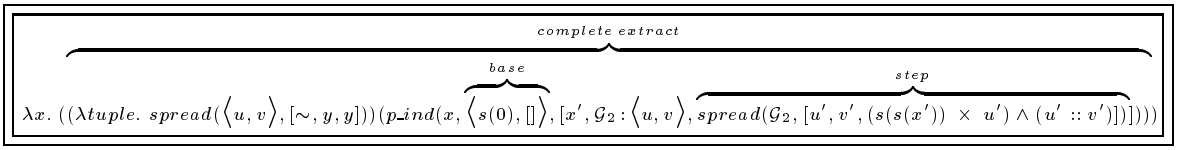


Figure 11: The target extract for $fctl$

Note that just as the source proof – fig.7(a) – contained two stepwise inductions, with the nested induction being applied at the step case of the outermost induction, and the target proof – fig.7(b) – contains only a single induction (on a tuple structure), so the source extract program – fig.9 – contains a dual nested recursion schema, with the nested recursion being applied at the step case of the outermost recursion, and the target extract program – fig.11 – contains only a single dual recursion (on a tuple structure).

5.2.3 Scope of Loop Removal Transformations

The situation for proof tupling *auxiliary recursive* functions is different from that of functions which contain only self-recursive calls in the body of the definition. CGR functions which are *auxiliary* recursive fit the following schematic definition $S2$:

$$S2\ \ f(x) \Leftarrow \mathbf{if}\ b(x)\ \mathbf{then}\ k(x)\ \mathbf{else}\ h(x, f_1(\delta^i(x)), ..., f_n(\delta^j(x)))$$

where there is at least one auxiliary function call in the body of $S2$. So for a bi-linear instances of $S2$, such as the $factlist$ function, the following holds: $(f = f_1 \vee f = f_2) \wedge f_1 \neq f_2$. The PTS is, however, capable of performing tupling transformations on any instances of $S2$.

As we illustrated in §**??**, the PTS will also transform functions where, regarding $S2$, one or more of the functions, $f_1, ..., f_n$, in the body of $S2$ is an instance of $S1$. This increases the performance of the PTS since the scope of transformable functions is not soley those that pertain to $S1$ or $S2$, but in addition those that pertain to some combination of $S1$ and $S2$. A thorough account of example transformations can be found in [**?**].

5.3 Example 3: Loop Removal By Transformation of $(+1)s$ to $(+2)s$ Induction

Consider the following variant of $factlist$:

$$factlist(s(n)) = fact_2(s(n)) :: factlist(n), \tag{15}$$

where the auxiliary function $fact_2$ is a $(+2)s$ recursive function thus:

$$\begin{aligned} fact_2(0) &= s(0); \\ fact_2(s(0)) &= s(0); \\ fact_2(s(s(n))) &= s(s(n)) \times fact_2(n), \end{aligned}$$

and where the PTS constructs a target tuple of length 3, where one component is the subsidiary $factlist$ call and the remaining two components are the 2 subsidiary calls for the $fact$ computation.

$(+2)s$ induction is best suited to construct the auxiliary $fact_2$ function since $fact_2$ is naturally a $(+2)s$ definition. The schema for $(+2)s$ induction is as follows:

$$\frac{\vdash P(0) \qquad \vdash P(s(0)) \qquad \forall v : pnat.\ P(v) \vdash P(s(s(v)))}{\vdash \forall x : pnat.\ P(x)}.$$

So in order to synthesize a program which computes the $factlist$ varient (**??**), we must construct a proof where in a $(+2)s$ induction is nested within (at the step case of) an outer $(+1)s$ induction. The nested $(+2)s$ inductive proof is almost identical to the nested $(+1)s$ proof of example 2 (*cf.* fig. 9(a)). The only difference is that the recursive argument in the goal conclusion is two, rather than one, applications of the successor function out of step with the recursion argument in the induction hypothesis. In fig.12 below we show the corresponding $(+2)s$ induction node:

refinement:	$\boxed{(+2)s\ induction(x')}$
induction hypothesis:	$\exists z.\ fact_2(x'') = z$
induction conclusion:	$\vdash \exists z'.\ fact_2(s(s(x''))) = z'$
witnessing refinement:	$\boxed{\exists - intro(s(s(x'')) \times z)}$
fertilized conclusion:	$\vdash fact_2(s(s(x''))) = s(s(x'')) \times z$

Figure 12: Source nested $(+2)s$ induction (for $fact_2$ construction).

To perform the proof tupling transformations on such a nested induction, the PTS needs to tabulate 2 $fact_2$ function calls, along with the $factlist$ call. That the target tuple includes 2 $fact_2$ function calls is determined by precisely the same reasoning that is used to form a target tuple for the Fibonacci example: the body of the step case definition for $fact_2$ contains a self recursive call to $fact_2$ that is 2 applications of the common generator function, in this case s, out of step with the head of the definition. This is clearly illustrated by replacing z in the **next conclusion** slot, of fig.12, by the hypothesis that it labels thus:

$$\vdash fact_2(s(s(x''))) = s(s(x'')) \times fact_2(x).$$

Hence, the optimization of the $fact_2$ function requires a tuple of two components (i.e., $\Phi = 2$), where the tabulations would correspond to $fact_2(s(n))$ and $fact_2(n)$. Since $fact_2$ appears as the auxiliary function call of $factlist$, then the required target tuple contains three components (i.e., $\Phi = 3$), and the PTS sequences the following goal into the target proof:

$$\big((\exists u,\ \exists v,\ \exists w.\ fact_2(s(x)) = u \ \wedge\ fact_2(x) = v \ \wedge\ fctl(x) = w),\ tuple\colon \langle u, v, w\rangle\big).$$

Note that, in effect, in performing the above source to target transformation we have *both*:

- transformed a source proof with a nested induction to a target proof with a single induction (employed on a tuple); and
- in doing so, transformed the (nested) $(+2)s$ induction into a standard $(+1)s$ stepwise induction.

Hence proof tupling on source proofs that contain a nested induction structure, where either of the inductions is in itself susceptible to optimization through tupling, is tantamount to combining the transformation of induction schemas with the merging of nested inductions.

6 Merits and Applications of Proof Tupling and Comparisons with Program Tupling Transformations

In §**??** we mentioned that one of the most influential strategies for program transformation is the *unfold/fold* technique [**?**]. This technique is employed within Darlington's interactive NLP program transformation system, and used by Chin to perform *automatic* tupling transformations [**?**].

In §**??** we identified three key steps for transformation using the *unfold/fold* strategy. These steps correspond to the most difficult aspects as far as *automation* is concerned, and in NLP, and similar systems, require some form of user guidance:

- Lemma generation: the introduction of an appropriate function definition in terms of the source definition. The provision of such explicit definitions, where the target is defined in terms of the source, generally constitute the well known *eureka* step in unfold/fold transformations, and are notoriously difficult to automate [**?**]. The unfold/fold strategy is motivated by the observation that significant optimization of a (declarative) program generally implies the use of a new recursion schema. This process usually depends on the *user* providing the requisite explicit target definition. The strategy then proceeds to evaluate the recursive branches of the target definition, primarily through unfolding with the source definitions, until a fold (match) can be found with the explicit definition.
- Folding: when to fold the eureka definition with the source definition. This requires using matching as a means of testing for the successful folding of the target function definition with the source definition.
- Application of laws: for example, when to apply associativity.

In subsequent sections we discuss the differences, and advantages, that the PTS approach to optimization has over unfold/fold style program development.

6.1 The Reduced Workload Regarding Dependency Analyses

To understand how the proof tupling approach circumvents the need to produce and analyze dependency graphs we shall briefly describe an existing program transformation system that employs the tupling technique.

Recently, Chin, a student of Darlington's, has described several methods for automatic program transformation within the HOPE$^+$ system [**?**]. By an analysis of *symbolic dependency graphs*, based on [**?**], Chin is able to describe an automatic procedure for finding a pair of *matching tuples* by the unfolding of selected calls to the source program, and then using matching as a means of testing for successful folding. This is a significant achievement and represents the first successfull attempt to automate the notoriously difficult unfold/fold eureka steps. Chin's automatic tupling method is best described by example (we shall remain with the Fibonacci function).

The initial portion of the symbolic DG for Fibonacci is shown below in fig.13. As with the symbolic DG for the factlist function, fig.9, redundancy is exhibited by more than one arrow directed at any particular node.

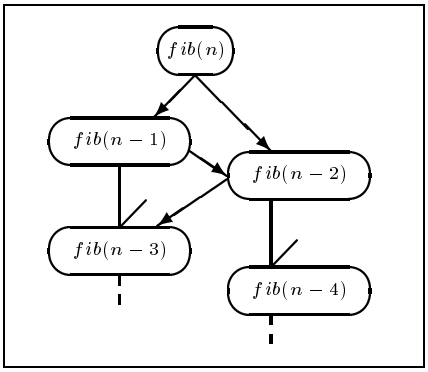


Figure 13: The symbolic DG for $fib(n)$

The main idea taken from [**?**] is that:

> An appropriate eureka tuple can be found if and only if there exists a *progressive sequence* of *cuts* that *match* one another, in the function's dependency graph.

A *cut* is defined as a subset of nodes across a dependency graph that when removed will divide the graph into two disconnected halfs. A *progressive sequence* of cuts is a sequence of cuts ordered according to size (i.e., according to the number of nodes in the subset). A pair of cuts *match* if a consistent substitution can be obtained when each function call of the first cut is matched with the corresponding function call of the second cut.[23]

The finding of an appropriate *eureka tuple* depends on the notion of a continuous sequence of cuts. This is defined in [**?**] as follows:

> "A *continuous* sequence of cuts, $cut_1, cut_2, ..., cut_N$, is a successive series of cuts which starts with the root node as its first cut. This sequence successively obtains the next cut by giving up a subset of nodes... from the *topmost set* of the current cut in order to acquire the children for the next cut."

The topmost set of a cut is defined as a set of nodes whose ancestors are not present in the cut itself.

Returning to the example and starting with the main function call, Chin's analysis replaces $fib(n)$, the first cut, with its two subsidiary calls, $\langle fib(n-1), fib(n-2) \rangle$. This gives us the second cut. The analysis then proceeds by unfolding only that call in a cut which is *not* a subsidiary call of the other call, i.e., the topmost item. So, since the function call $fib(n-2)$ is a subsidiary call of $fib(n-1)$, only $fib(n-1)$ is unfolded. This gives the third cut, $\langle fib(n-2), fib(n-3) \rangle$. The third cut matches the second cut, thus providing the analysis with a matching tuple.

Chin's process is essentially the same as that described for Darlington's *unfold/fold* tupling technique: the unfold/fold steps required for the tupling transformation are achieved by locating a pair of matching tuples by the unfolding of appropriately selected calls and then using matching as a means of testing for successful folding.

The main difference between Chin's and Darlington's systems is that the use of such selection ordering allows for a considerable degree of automation, since once this analysis succeeds the main task of the tupling transformation – finding a successful fold – will have been achieved.

6.1.1 Comparison with Proof Tupling

Chin's DG analysis tells us two things:

1. firstly, *the number*, Φ, of subsidiary calls of the main function calls required to form the tuple (i.e., the determination of the tuple size); and

2. secondly, *which* subsidiary calls are to be tabulated.

An advantage of *proof tupling* is that *both* of these things, required for the tuple formation, are contained in the source proof. This means that they can readily be abstracted from the proof and exploited for the construction of the target tuple *without* any additional dependency graph construction and analysis procedures. This will

[23] These terms are formally defined in [**?**].

always be the case for tupling transformations since the eliminations performed on the induction hypothesis in the source will always provide an accurate measure of what recursive calls are (a) required to compute the source course of values procedure, and (b) require tabulation in order to compute the target stepwise procedure. Returning to the Fibonacci example, the required information is read directly from the witness,

$$\exists - intro(y_1 + y_2),$$

of the source induction step. In OYSTER notation this witness is specified in terms of the eliminations on the induction hypothesis $\mathcal{I}_{\mathcal{H}}$:

$$(intro(\mathcal{I}_{\mathcal{H}}\ of\ p(x) + \mathcal{I}_{\mathcal{H}}\ of\ p(p(x)))),$$

This tells us precisely the number of, 2, and the identity of, $p(x)$ and $p(p(x))$ the eliminations on $\mathcal{I}_{\mathcal{H}}$ performed in the source induction in order to introduce recursion in the source function. In the general case, the dominant function of the first tuple component will always be that employed at the induction step of the source (where the number of tuple elements corresponds to the number of source proof eliminations on the induction hypothesis).

Note also that no extensive search is involved in the analysis of the source proof in order to determine Φ and to witness a value for the tuple components. The portions of the source proof that are accessed for the analysis correspond to specific semantic units: the specification, the application of induction, the induction base and step cases, the unfolding step, and the witnessing rule. These are clearly represented as distinct sub-lists within the rule-tree abstractions (§**??**) and the PTS knows precisely where to look in order to access any of the aforementioned units. For example, the induction step will always correspond to that rule applied at the deepest node of the decision tree employed to separate the various cases. So, within the rule-tree, the induction step occurs as the last case of a nested case analysis.

So, unlike program tupling, the PTS proof tupling optimizations do not require the construction of a (potentially infinite) dependency graph, nor does it require any procedures for searching the dependency graph in order to find a *matching tuple*.

6.1.2 Tuples As Conjunctions

Within the object-level OYSTER proofs the tuples are represented simply as conjunctions (hence a tuple $\langle A, B, C \rangle$ is represented as $A \wedge B \wedge C$). Hence, we bypass the need to invent new data-types for tuples solely for the purposes of transformation. This means we avoid the charge that (program) tupling techniques rely heavily on the somewhat ad hoc requirement to introduce tuples, memo tables or similar objects, and that we do not require arbitrarily complex tabulating constructs. For example, program transformations within Darlington's FPE environment automate, to some extent, the construction of the eureka tuple by incorporating a large table and managing system [**?**]. However, this causes considerable inefficiency since it has the effect of carrying round potentially huge open-ended tuple structures whose length is tailored to the functions needs.

6.2 Further Advantages Regarding Search, Control and Correctness

The fact that the PTS transformation tactics are (partially) specified at the meta-level, in terms of syntactic pre- and post-conditions, reduces the amount of search that would be involved if the target proof were constructed at the object-level. In other words, since we can regard the rule-trees, together with pre- and post-conditions, as proof plans then a general advantage of performing *tactic transformations* – i.e., meta-level transformations on the object-level tactics – is that the transformation space is equivalent to a planning search space which is far smaller than the object-level search space.

As well as the way that dependencies are sought during tupling transformations, further factors which play a beneficial role regarding search and control include the means by which the target recursive step is completed, and the form of equation development used all have a significant effect on the amount of search involved during the transformation.

We shall consider in turn how the PTS reduces the search involved with each of these factors in comparison with previous program tupling systems (notably [**?**, **?**]

6.2.1 Derivational Form: Folding Vs. Fertilization

Darlington's NLP, and Chin's HOPE$^+$, tuple analysis is motivated by the desire to find a matching tuple which can be used for *folding*. This can involve extensive search. To illustrate this property, we display, in fig.14, the unfold/fold derivation of the efficient Fibonacci procedure:

• Equational def. of fib:

(1)	$fib(0)$	$=$	1	Given
(2)	$fib(1)$	$=$	1	Given
(3)	$fib(x+2)$	$=$	$fib(x+1)+fib(x)$	Given

• Derivation of auxilliary tuple function g:

(4)	$g(x)$	$=$	$\langle fib(x+1), fib(x)\rangle$	Eureka — Definition
(5)	$g(0)$	$=$	$\langle fib(1), fib(0)\rangle$	Instantiation
(6)		$=$	$\langle 1, 1\rangle$	Unfolding with 1 and 2
(7)	$g(x+1)$	$=$	$\langle fib(x+2), fib(x+1)\rangle$	Instantiate 4
(8)		$=$	$\langle fib(x+1)+fib(x), fib(x+1)\rangle$	Unfold with 3
(9)		$=$	$\langle u+v, u\rangle$ **where** $\langle u, v\rangle == \langle fib(x+1), fib(x)\rangle$	Abstract
(10)		$=$	$\langle u+v, u\rangle$ **where** $\langle u, v\rangle == g(x)$	**Fold** with 4

• Derivation of fib in terms of g:

(11)	$fib(x+2)$	$=$	$u+v$ **where** $\langle u, v\rangle == \langle fib(x+1), fib(x)\rangle$	Abstract 3
(12)		$=$	$u+v$ **where** $\langle u, v\rangle == g(x)$	**Fold** with 4

Figure 14: Unfold/Fold development of efficient Fibonacci

The development of the target terminating branch is straightforward. Regarding the recursive branch, unfolding must be performed in order to obtain the explicit definition, (8), from the eureka definition (4). A fold step is now required so as to introduce a recursion into (8). The search for a fold involves observing that all the components necessary to match the above equation are present within the initial definition, (4), for the auxiliary function g. Hence (8) is re-written using unfolding and *where abstraction*, to (10) which easily folds with the eureka definition (4) yielding the desired optimized recursive definition (10).[24]

The derivation of fig.14 illustrates how, within unfold/fold style systems, the head of the developing equations remains constant, and it is only the body that is modified, i.e., re-write rules are only applied to the left hand side of equations. This form of equation development, together with the formal definition of folding [**?**]:

> If $E = E'$ and $F = F'$ are equations and there is some occurrence in F' of an instance of E', replace it by the corresponding instance of E obtaining F''; then add the equation $F = F''$,

means that, throughout the equation development, the same equation head is retained. Hence folding with the source equations is a necessary requirement at some point in order to introduce a recursion into the tail of the developing equations. There is not, however, any procedure for knowing when to halt unfolding and introduce a fold (nor when to perform a *forced fold*). Thus the folding requirement presents control problems, and is one primary reason why user guidance is usually required in such systems in order to avoid flawed attempts at folding. The other reason being the provision of the eureka step corresponding to the generation of the auxiliary tuple. Note that, regarding fig.14, the control problem is, in fact, doubled since following the first fold, (10), further (forced) folding is required, at steps (11) and (12), to express fib in terms of g.

An advantage of the PTS transformations is that they inherit the properties of theorem proving: inductive proofs are driven by the heuristic requirement to find a fertilization: the proof construction is developed in a bi-directional manner since both sides of the induction conclusion can be re-written in the search for matching (unifiable) terms. The simplest way to illustrate this is to employ meta-variables (in upper-case) for those "unknown" portions of the proof (corresponding to the initial eureka step and the witnessing steps). We also adopt the standard conventional notation for tuples, rather than use OYSTER's conjunctive representation, and use a $where$ construct to refer to the induction hypothesis. These changes do not alter the bi-directional form of the proof development, but rather makes it easier to see and compare with the unfold/fold style derivation of fig.14. A characterizing feature of tupling proofs is that the recursive definition will consist of some, as of yet unknown, function(s) applied to the tuple components of the induction hypothesis [**?**]. Hence we shall use the meta-variables to represent such functions in our comparative illustration, fig.15 below, of the target Fibonacci proof (we show only the induction step case of the auxiliary proof, corresponding to steps (4) to (14) of fig.14).

[24] Abstraction consists of replacing parts of an expression, in the body of an equation, by variables, and then defining these variables in a *where* clause. The combination of unfolding and abstraction is sometimes referred to as *forced* folding.

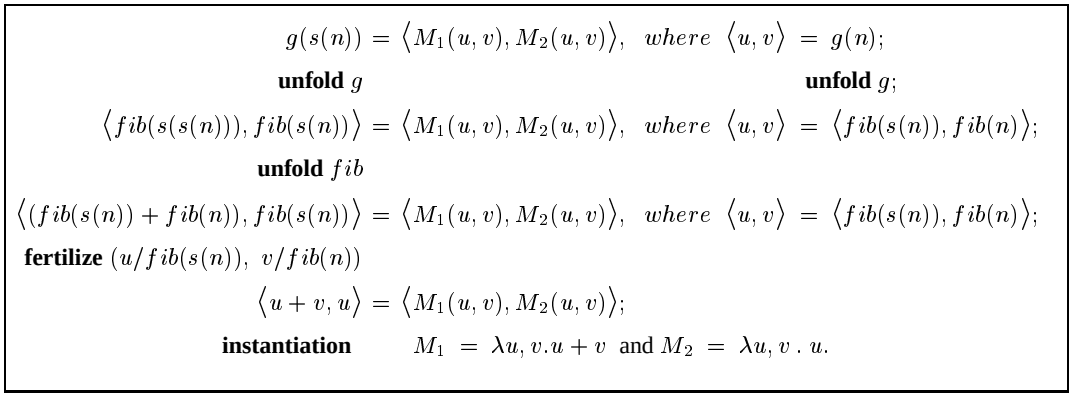


Figure 15: Parallel development of induction conclusion

The important feature to note is the "parallel" development of both head and body towards a unifiable pattern, such that induction terms may be eliminated from the conclusion. This means that since we can modify both sides of the equation we can avoid the decision(s) as to when, and with what, to fold. That is, we can limit the process to the iterative application of unfolding with equational definitions. This significantly reduces the search space, and on the available evidence is much easier to control (it is precisely what is formally captured by the rippling technique, §**??** and §**??**).

6.2.2 Law Application

A notorious problem with unfold/fold is that there is no principled means of applying semantic laws. That many unfold/fold transformations may require the numerous and somewhat arbitrary application of laws, for which any overall strategy is difficult to characterize, means that user-interaction is usually required. Thus an advantage of operating within a proof theoretic framework is the capability to automatically form and apply rewrites from semantic laws. By semantic laws we usually mean lemmas such as the *associativity of append*, rather than the lemmas used for the purposes of verification in our examples. Several examples of such principled law application can be found in [**?**].

6.2.3 Correctness

More recent incarnations of the unfold/fold strategy have been shown to be correctness guaranteed for specified classes of functions (*cf.* [**?**] and [**?**]). However, each extension to the class of functions requires a corresponding extension to the correctness procedures, and this leads to a considerable work overhead (proportional to the range of transformations – or *generality* – of the system).

This is not a problem regarding the PTS, and any future extensions thereof: synthesis proofs must contain a verification proof that the extract term computes the task described by the specification. Thus, extract programs are correct with respect to the complete specifications of the synthesis proofs from which they are extracted. Hence the correctness of *all* transformations is ensured without having to additionally provide, or extend, any correctness criteria, or proof, each time we extend the range of programs to which the transformations are applicable.

Stricktly speaking, we have only addressed *partial* correctness. Total correctness involves providing termination conditions in addition to ensuring that the output program computes the desired function. As stated in the previous section, a problem with controlling unfold/fold transformations is knowing when to stop unfolding and introduce the crucial fold step into the derivation. This can lead to an infinite regression of unfolding and lemma applications. In the case of proof transformation, termination simply corresponds to the completion of the target proof: when the rewriting of the induction conclusion has been successfully driven toward fertilization with the hypothesis. Unlike folding, fertilization is well-founded.

6.3 Generality: Exploiting Proof Plans

In §**??** we remarked that since the majority inductive proofs pertain to the same (formal) pattern that the PTS design need not be altered for disparate inductive proof transformations (thus the majority of proofs employing course of values induction can be transformed into an equivalent, but more efficient, stepwise inductive proof). That there is a high degree of similarity in the overall shape of the inductive *proof trees* (and in the strategy

employed in inductive proofs) requires some explanation if our claims concerning the generality of the PTS design are to be justified. This will also be relevant to the subsequent section on future research.

Inductive proofs, including the source and target Fibonacci proofs, invariably involve a process whereby formulae are unfolded by replacing terms by suitably instantiated definitions. The proliferation of this process such that recursive terms are gradually removed from the recursive branches – by the repeated unpacking of induction terms – is part of the (heuristic) process known as *rippling*. A simple examples of this would be the application of the recursive branch of the $append$ definition:

$$append(e :: l_1, l_2) \quad \Rightarrow \quad e :: append(l_1, l_2)$$

The terms $append(l_1, l_2)$ would unify (fertilize) with the respective induction hypothesis. Thus the goal of rippling is precisely that of the induction step: to reduce the induction step case to terms which can be fertilized with those in the induction hypothesis, or those in subsequent derivations of the induction hypothesis.

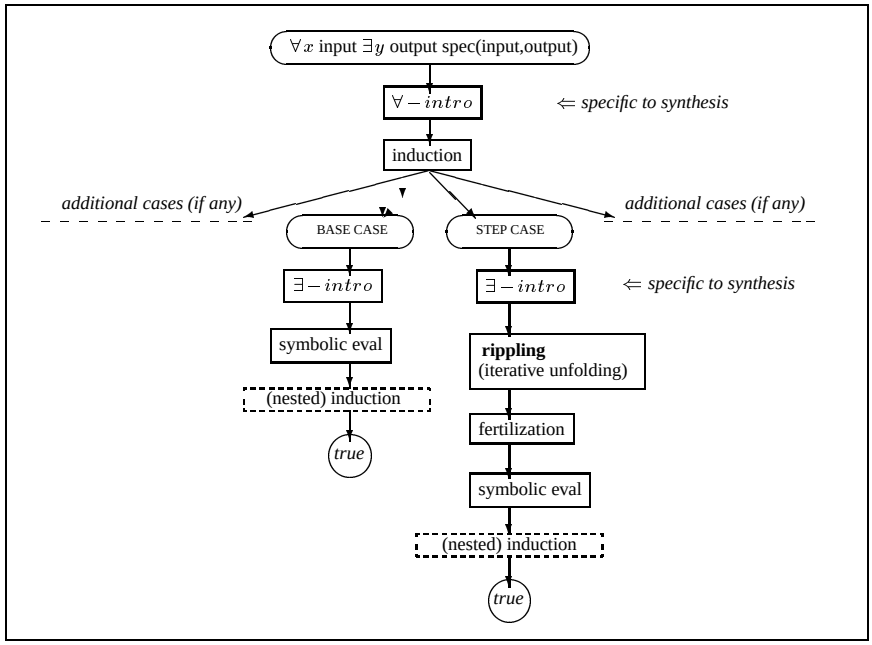


Figure 16: Proof plan for induction strategy

This common pattern to inductive theorem proving allows for the construction of a general induction *proof plan*, specified at the meta-level, which can then be used for guiding a whole gamut of object-level proofs. In fig.16 we have represented the key decisions and choice commitments made during a typical inductive proof. These will involve applying one of the numerous OYSTER induction rules and then witnessing the existential quantifier, using $\exists - intro$, at each of the induction cases (where, as indicated in fig.16, the application of the $intro$ rules are specific to inductive *synthesis* proofs). We have indicated, within dashed boxes, that, following the witnessing steps of the (outermost) induction, there may occur a further *nested* induction. These will take the same format as the outermost induction. Finally, we must verify that the instantiated schema will yield a recursive schema that will compute the input-output relation specified in the main conjecture.

The fact that inductive proofs invariably pertain to this common form increases our expectations that there will be no need to build into the PTS ad hoc and diverse mechanisms for dealing with substantially different patterns of proof.

In §**??** we briefly discuss directly exploiting proof plans for the purposes of proof transformation.

6.4 Applications and Future Research

In this section we consider the applications (potential and real), and future avenues of research, regarding proof transformations.

6.4.1 Optimizing Recursion

The vast majority of commercial software involves the computation of recursive functions, and to prove theorems about such functions it is necessary to use mathematical induction. To manipulate such proofs, whether or

not the aim is to optimize associated program constructs, requires the machinery for correct and well-founded induction transformations. This research, albeit embryonic, makes a first inroad into this requirement: the more that theorem proving, and in particular inductive theorem proving, forms the basis of automatic programming then the more that proof transformation becomes a viable means for providing automatic, correctness guaranteed optimization.

Anticipated future applications of this research include the optimization of electronic circuit design and the optimization of computer configurations. This is because both these design problems can be formally cast as processes of inference [**?**, **?**]. Thus, we can apply the same automated theorem proving techniques that we use for high quality software production.

6.4.2 Software Quality: Efficiency and Reliability

As stated at the outset, §**??**, the research described herein addresses both the reliability and efficiency, as well as the automatability, criteria of developing high quality software using formal methods. Formal methods allows us to employ the better understood techniques of theorem proving to guarantee these criteria.

In this paper we used simple examples of linearization and the removal of nested recursion to illustrate the methodology. However, more complex optimizations are possible by using different (non-primitive) inductions to construct the target proof: in [**?**] we explain how linear procedures can be optimized to logarithmic procedures through proof transformation by using the method of *matrix multiplication* and replacing the *stepwise* induction employed in the source proof by a target *divide and conquer* induction.

Future anticipated extensions include the systemization of more esoteric induction transformations involving schemas such as induction based on the construction of numbers as products of primes [**?**].

6.4.3 An Aid for Synthesis

On empirical evidence alone, there appears to be an inverse relation between, on the one hand, the efficiency of the recursive process generated by an extract, and on the other, the complexity of the proof from which it was extracted.[25] This evidence has been gleaned from a study of synthesizing several sorting algorithms in the NUPRL system where the extracts corresponding to various synthesized sorting algorithms are compared with the syntactic density of the associated proofs [**?**]. Further evidence is provided from research regarding pruning inductive proof trees in order to adapt the associated extract program [**?**, **?**]. So, for example, transformations which increase the syntactic complexity of a source course of values proof, by performing proof transformations that cut in (or sequence) an additional sub-proof, will decrease the complexity of the recursive behaviour of the extract programs (from exponential to linear). One practical contribution of a proof transformation system is, therefore, that it enables the synthesizer (human or mechanical) to construct short, elegant proofs, without clouding the design process with efficiency issues, and then to transform them into opaque proofs that yield efficient programs.

The inverse complexity relation is something which merits further attention but for which, as of yet, there is only empirical justification and a quasi-theoretical foundation [**?**]. Intuitively speaking, however, the extra complexity associated with a target proof can be thought of as additional information required to compute the specified input/output relation *efficiently* as opposed to simply ensuring that the specified input/output relation is computed.

6.4.4 Exploiting Proof Planning

The automatic CLAM proof-planning system *formally* encapsulates, in a meta-logic, the common shape of inductive proofs discussed in §**??**. The system automatically constructs meta-level *proof plan* representations from proof specifications [**?**]. These proof plans can then be used to guide the object level synthesis/verification, with the advantage that the planning search space is considerably smaller than the object-level OYSTER search space. The proof plans can then be used, as a general strategy, to guide the refinement of specific specifications [**?**, **?**]. Of particular significance is the systemization of the rippling re-writing process: definitional equations are converted into appropriate re-write rules through a special annotation process. The annotations mark the differences between the two sides of the equation. The annotated rewrite rule so formed can then be matched against proof (sub)goals and the (sub)goal rewritten accordingly.[26]

[25] This is despite the fact that human theorem provers are usually trained to find short, elegant proofs rather than long opaque ones.

[26] This very brief outline is only barely representative of the current state of rippling, and of its use in automatic proof plan formation. For full details the reader should consult [**?**].

Anticipated future research includes extending the PTS to be fully compatible with the CLAM system. This means that any source to target proof transformations can exploit the proof planning facilities thus leading to greater generality and automatability of the class of optimizations amenable to the system. At present the PTS must constantly access the source proof in order to complete the target proof. The adapted version will need only to access the source to obtain information such as tuple identification and induction witnesses. The target proof can then be completed using the automatic reasoning systemized in the CLAM proof-planning system.

7 Summary

We described the fundamentals of a working synthesis proof transformation system. The novel aspect of this research is that program optimization is achieved through the transformation of *synthesis* proofs. In particular, recursive programs are optimized by transforming inductive synthesis proofs. Techniques from the field of program transformation may be used to transform the computational content of a proof. An important technique for transforming exponential behaviour into linear behaviour is *tupling*. The PTS, unlike other existing transformation systems, performs this technique on (synthesis) proofs. The system satisfies the desirable properties for a transformation system of correctness, generality, automatability and the means to guide search through the transformation space.

The benefits of the proof transformation approach include the fact that extra information contained in the proofs, but not programs, can be exploited to automatically guide the transformations. In particular: proofs contain a verification component, and; dependency information abstracted from the source proof guides the transformations without the need for any extensive dependency graph analysis.

The source and target programs of traditional program transformation systems do not have a formal specification present, nor, as mentioned above, a verification component. This means there is no immediate means of checking that the target program meets the desired operational criteria. Regarding proof transformation, all transformed programs are correct with respect to their specifications, and we ensure that the target computes the same specified input/output relation as the source (only more efficiently).

With the more traditional program development systems which employ the unfold/fold technique, it is the *automation* of the lemma generating procedures and, *in particular*, the subsequent folding with the lemmas, that have proved, to date, difficult to automate. We described how, within the context of proof transformation, target tuple definitions can be automatically generated by analysing source definitions. The problem of *folding* has been circumvented within the proof transformations since, due to the sequent calculus notation and the manner in which proofs are refined, we need use only unfolding: recursive terms, corresponding to source proof induction terms, are eliminated from the target recursive branches, corresponding to the target proof induction branches, by unfolding until fertilization applies.

The source and target programs of traditional program transformation systems do not have a formal specification present. This means there is no immediate means of checking that the target program meets the desired operational criteria. Regarding proof transformation, all transformed programs are correct with respect to their specifications, and we ensure that the target computes the same specified input/output relation as the source (only more efficiently).

An important commitment regarding the recursive behaviour of an extract program is the choice of induction schemata (and how the cases are satisfied). By exploiting the common structure of OYSTER inductive synthesis proofs we can transform the induction schema employed in a proof yielding an inefficient program into a schema such that the new target proof yields a more efficient program. Transformation is achieved through the application of *proof transformation tactics* to internal representations of the OYSTER proofs. Since we can provide a general proof plan for inductive (synthesis) proofs, then we can build general transformation tactics for optimizing the recursive programs that they synthesize.

References

[1] David A. Basin. Extracting circuits from constructive proofs. Research Paper 533, Dept. of Artificial Intelligence, Edinburgh, 1991. Also appeared in Proceedings of the IFIP-IEEE International Workshop on Formal Methods in VLSI Design, Miami USA, 1991.

[2] B. Bjerner. *Time Complexity of Programs in Type Theory*. PhD thesis, University of Göteborg, 1989.

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

[4] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, 1988. Perspectives in Computing, Vol 23.

[5] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[6] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[7] A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill, and A. Stevens. A rational reconstruction and extension of recursion analysis. In N.S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 359–365. Morgan Kaufmann, 1989. Also available from Edinburgh as DAI Research Paper 419.

[8] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[9] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, 1977.

[10] W.N. Chin. *Automatic Methods for Program Transformation*. PhD thesis, Imperial College, 1990.

[11] N. H. Cohen. Eliminating redundant recursive calls. *ACM Transactions on Database Systems*, 5 No. 3:265–299, 1983.

[12] H.B. Curry and R. Feys. *Combinatory Logic*. North-Holland, 1958.

[13] J. Darlington. *A Semantic Approach to Automatic Program Improvement*. PhD thesis, Dept. of Artificial Intelligence, Edinburgh, 1972.

[14] J. Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16(3):1–46, August 1981.

[15] J. Darlington. A functional programming environment supporting execution, partial evaluation and transformation. In *PARLE 1989*, pages 286–305, Eindhoven, Netherlands, 1989.

[16] W.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[17] Helen Lowe. The Use of Theorem Proving Techniques in Expert Systems for Configuration. In J.-C. Rault, editor, *Proceedings of the Eleventh International Workshop on Expert Systems and their Applications, Avignon*. EC2, May 1991. Also available from Edinburgh as DAI Research Paper 536.

[18] P. Madden. A NuPRL synthesis of several sorting algorithms: Towards an automatic program transformation system. Research Paper 356, Dept. of Artificial Intelligence, Edinburgh, 1987.

[19] P. Madden. The specialization and transformation of constructive existence proofs. In N.S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1989. Also available as DAI Research Paper No. 416, Dept. of Artificial Intelligence, Edinburgh.

[20] P. Madden. *Automated Program Transformation Through Proof Transformation*. PhD thesis, University of Edinburgh, 1991.

[21] P. Madden. Formal methods for automated program improvement. In B. Nebel and L. Dreschler-Fischer, editors, *KI-94: Advances in Artificial Intelligence. Proceedings of 18th German Annual Conference on Artificail Intelligence*, Saarbrueken, Germany, September 1994. Springer-Verlag. A longer version is available from the Max-Planck-Institut as MPI-I-94-38.

[22] P. Madden. Linear to logarithmic optimization via proof transformation. Research paper MPI-I-94-240, Max-Planck-Institute für Informatik, 1994.

[23] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[24] Per Martin-Löf. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hanover, August 1979. Published by North Holland, Amsterdam. 1982.

[25] D. Michie. Memo functions and machine learning. *Nature*, 218:19–22, 1968.

[26] A. Pettorossi. A powerfull strategy for deriving programs by transformation. In *ACM Lisp and Functional Programming Conference*, pages 405–426, 1984.

[27] A. Stevens. A rational reconstruction of Boyer and Moore's technique for constructing induction formulas. In Y. Kodratoff, editor, *The Proceedings of ECAI-88*, pages 565–570. European Conference on Artificial Intelligence, 1988. Also available from Edinburgh as DAI Research Paper No. 360.

[28] H. Tamaki and T. Sato. A transformation system for logic programs that preserves equivalence. Technical Report TR-018, ICOT, 1984.

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

[30] S. S. Wainer. Logical and recursive complexity. Technical Report 31/90 (Preprint Series), Center for Theoretical Computer Science, University of Leeds, 1990.