Using Proof in Transformation Synthesis for Automatic Parallelisation

By
Andrew Cook
Submitted for the degree of Doctor of Philosophy
Heriot-Watt University
Department of Computing and Electrical Engineering
June 2001

I hereby declare that the work presented in this report was carried out by myself at Heriot-Watt University, Edinburgh, except where due acknowledgement is made.

(Candidate)

A. Ireland & G. Michaelson (Supervisor)

Date

Contents

List of Figures

This work is supported by UK EPSRC grant GR/L42889, "Parallelising compilation of Standard ML through prototype instrumentation and transformation".
I would like to thank my supervisors Andrew Ireland and Greg Michaelson, and Julian Richardson for his work on $\lambda Clam$.

Abstract

Parallel programming is very difficult because of the level of thought and planning it requires of programmers. The use of skeletons to execute higher-order functions in parallel is a promising approach to removing some of the burden of parallel programming. However higher-order functions seem restrictive and unnatural to programmers who prefer less abstract programming constructs.

This thesis addresses the problem of transforming a functional program written without the use of higher-order functions into one that contains them. We use proof, and in particular proof planning, to synthesise transformation rules which can be used to automatically introduce higher-order functions into a program.

Proof planning enables us to verify the equality of expressions in a functional language. These equalities can be used as transformation rules to change a functional program. Examining the verification proofs of transformation rules which equate program functions with expressions involving higher-order functions informs the development of a synthesis proof plan. We use middle-out reasoning to allow the proof to gradually instantiate a variable corresponding to unknown term structure. In this way we synthesise higher-order function expressions from existing program functions.

A prototype synthesis strategy has been implemented by extending the $\lambda Clam$ higher-order theorem prover. The evaluation of this prototype has yielded positive results and suggested extensions and improvements that might be made to it. The prototype synthesis system has been shown to find expressions involving the higher-order functions map, $foldr$, $foldl$ and $scan$.

Chapter 1

Introduction

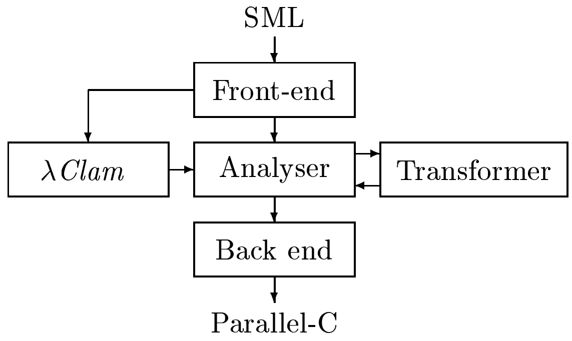
The research described in this thesis seeks to augment the capabilities of a parallel compiler with a theorem prover to allow the verification of transformation rules involving higher-order functions (HOFs). An interesting extension of this was to look into the possibility of using the theorem prover to automatically synthesise novel transformations instead of simply verifying user-defined ones. However, the verification of typical transformation rules used in the system proved relatively straight-forward and our main goal became the creation of transformation rules.

We have shown that creation of transformation rules can be achieved by extending the existing theorem proving technology. We use the rippling heuristic and a knowledge base of HOFs to synthesise expressions involving HOFs from program functions that lack them.

1.1 Context

This work is part of a project to build a Standard ML to parallel C compiler [MIP97, SBMK98, MSBK01]. The project is supported by the UK EPSRC grant GR/L42889. The overall structure of the compiler is shown in Figure 1. The front-end reads in and parses SML before passing it to the analyser. The analyser uses performance information about specific parallel architectures to and determines what transformations would produce efficiency gains. Efficiency gains for various examples are discussed in Hamdan's thesis [Ham00] SML is then passed to the transformer where changes in the code are made to increase the efficiency. The possible changes to the code are passed back to the analyser to pick the alternative which offers the most speed up. The final code is than passed to the back-end where it is compiled into parallel-C.

Figure 1: The structure of the SML to parallel C compiler



The work described here may be considered to sit between the front-end and analyser components. Through the synthesis of HOFs we suggest transformations which introduce potential parallelism.

The compiler uses program transformation as a means to produce parallel behaviour in a program. By introducing parallelism at the program level rather than at the specification level we separate semantic/algorithmic requirements from performance. Deriving an implementation from a specification is hard and if we add additional parallel performance requirements to it we increase the complexity and difficulty of producing an implementation. By allowing a programmer to take care of algorithmic issues by producing an executable SML program before attempting to introduce parallelism we gain extra information about the functions that we will identify as sites of potential parallelism. This also give more control to the user while shielding them from the complexities of parallel programming.

The compiler, in which our transformation synthesis system is an embedded component, exploits implicit parallelism and so the user is not aware of the choices being made with regard to parallel performance. Because it is unreasonable to expect user expertise in assessing parallel performance on an arbitrary architecture the system uses profiling and performance models in the analyser component to make these choices behind the scenes.

The compiler abstracts away from the details of hardware architecture and parallel processing allowing the programmer to focus on solving problems rather than worrying about the parallel performance. Instead the system focuses on optimising the parallel performance of the program that it is given. Our synthesis work assists this optimisation by providing a wide variety of transformations that the system can use to alter the parallel behaviour of the program, sometimes allowing the parallelisation of part of the program that would otherwise be not be amenable to transformation by the compiler.

There are complex problems that are effectively intractable by computer programs run on a single processor, for example geo-physical modelling, meteorology etc. However parallel programming is hard because it forces the programmer to think about many different, dynamic, interacting processes at the same time. Easing the burden of the programmer will increase their ability to solve hard or complex problems effectively. By allowing the implementation of faster, more robust parallel systems we also enable more efficient utilisation of resources.

The analyser component uses performance information about different parallel architectures to inform it about how HOF transformations are applied. The compiler is effectively customised by the performance information for each parallel architecture it is run on. In this way the compiler abstracts away from the specifics of the hardware architecture making it largely architecture independent. In the same manner, our synthesis of transformation rules is abstracted away from the hardware specifics. We concentrate on discovering a maximal number of possible transformation rules to allow the analyser component greater freedom to customise the program for the particular architecture it is running on. By producing an optimal parallel implementation tailored to the hardware architecture, in principal near linear speed up is possible [MSBK01].

1.2 Parallelism

In a traditional computer program every calculation is performed in sequence on the same processor. For parts of a program this is necessary because the one calculation depends on the results of the previous ones. But in some instances parts of the program may be completely independent of one another. In these cases the calculation may be done on separate processors at the same time which may yield the result more quickly than if they were done on the same processor in sequence.

Parallelism can be thought of as using multiple processors whereas concurrency is running different processes at the same time, this could use parallel parts of a single processor. Different models of parallelism are SIMD(single instruction, multiple data), MIMD(multiple instruction, multiple data) and SPMD(single program, multiple data). In the compiler project we use SPMD as it is a standard model for processor farms and divide and conquer[1].

Parallelism as a programming paradigm has not really become popular(except in special cases where the advantages are large such as numerical simulation and graphics) despite being around for a reasonable length of time and the advantages that it

[1] Farm and divide and conquer strategies will be discussed later in this section.

has. This is due to the lack of effective portability between the wide variety of parallel architectures, and the difficulty of producing and maintaining efficient parallel programs.

The problem with programming is caused by the necessity of explicitly using parallel constructs and the difficulty in visualising the execution of a parallel program. Parallelism involves different processes the order of execution of which is unpredictable. However, these processes may still have interdependencies through shared memory or message passing. This makes envisioning the execution of parallel programs very difficult. One way to increase the usability of the parallel paradigm is to abstract away from the details of explicit parallelism. Various ways of performing this abstraction are reviewed in a recent survey of parallel programming models [ST98].

1.2.1 Implicit Parallelism

Implicit parallelism is introduced behind the scenes by a compiler or optimiser so that the programmer is largely unaware of it. There has been surprisingly little work on implicit parallelism despite its usability advantages. The problem of introducing parallelism into an arbitrary program is very hard and previous approaches have either failed to produce speed up or restrict the expressiveness of the input language. Most recent work has been on languages with semi-explicit constructs which serve as signposts to help the compiler in producing a parallel implementation.

In imperative languages introducing parallelism is hard and requires data flow analysis to identify parallelism. Work has been done introducing parallelism into FORTRAN. The PTRAN (Parallel TRANslation) system [ABC+88] analyses a program to determine its data and control dependencies this information is then used to construct a parallel program satisfying these dependencies. Most recent work into parallel imperative languages centres around languages with explicit parallel constructs, like occam [INM84], or extensions to existing languages such as the MPI library [Mes93] for C.

In functional languages introducing parallelism is slightly easier because they do not allow side effects. Purely implicit approaches typically use evaluation transformation[2] to determine what parts may be done in parallel. These methods include exploiting parallelism in graph reduction, GRIP [Pey87], and rewriting for computation of expressions. Work has been done in parallelising Haskell [AAA+95] using this type of parallel evaluation strategy. This approach has the advantage of hiding all parallel behaviour from the programmer but it is very difficult to predict their parallel performance and generally don't produce much increase in performance.

Restricted approaches rely on the parallelisation of specific constructs in the programming language which encode common patterns of parallel behaviour. Although the fine details of parallelism are hidden the programmer must use the specific constructs (called skeletons) in their program if they are to exhibit parallel behaviour. This inhibits the expressiveness and ease of use of the programming language. These approaches while they are not as abstract and expressive as more dynamic methods have predictable performance and produce consistent performance enhancement. Examples of restricted implicit parallelism include NESL [Ble92], SAC [Gre99], and SISAL [MSA+83] all of which exploit parallelism associated with the manipulation of datatypes such as lists or arrays.

[2]Evaluation transformation is a way of introducing parallelism by creating a model of its evaluation and determining which parts of it are independent.

1.2.2 Skeletons

The algorithmic skeleton [Col89] approach allows us to abstract away from the details of parallelism while producing a fast parallel implementation. If the program is made up of pre-defined building blocks, or skeletons, where each skeleton corresponds to some known standard algorithm, then inter- and intra-block parallelism may be exploited relatively easily. Inter-block parallelism is between blocks, for example a pipeline between independent parallel components, and intra-block parallelism is within a block, for example splitting a component into separate parallel subcomponents.

Examples of skeleton functions are the HOFs which work over lists. For example the functions map and $fold$ (definitions of map, $fold$ and other HOFs can be found in Figure 2) can introduce parallelism into a program. A map function can be executed in parallel by a processor farm where each element in the list is sent to a free processor which calculates the new list value independently of the others. A divide and conquer approach can be used for $fold$ if the function being folded into the list is associative.

There are several very useful HOFs defined over lists, map, $fold$, $reduce$, $filter$, zip etc. map applies a function to every element in a list, $fold/reduce$ computes a value from the list by applying an operator(a binary function) to each element of the list and the result of the previous call, $filter$ selects values from a list if a test of that value returns true, and zip computes a new list from two others of the same length by applying an operator to corresponding values in the two lists[3]. These functions are all defined recursively with a base case and step case corresponding to those in the list definition(Figure 2).

Various properties of HOFs, such as the $map/fold$ equivalence (equation 1 below)

[3] zip is sometimes defined as a function which produces a list of pairs from a pair of lists [BW88]. This definition is a specialisation of the more general zip described here which is a HOF which applies a binary function to the elements of two lists (of the same length).

Figure 2: Definitions of HOFs on lists

$$
\begin{array}{rcll}
map\ f\ [\,] & = & [\,] & map \text{ base case} \\
map\ f\ h :: t & = & (f\ h) :: (map\ f\ t) & map \text{ step case} \\
foldr\ f\ b\ [\,] & = & b & foldr \text{ base} \\
foldr\ f\ b\ h :: t & = & f\ h\ (foldr\ f\ b\ t) & foldr \text{ step} \\
foldl\ f\ a\ [\,] & = & a & foldl \text{ base} \\
foldl\ f\ a\ h :: t & = & foldl\ f\ (f\ a\ h)\ t & foldl \text{ step} \\
reducer\ f\ b\ [d, e] & = & f\ d\ e & reducer \text{ base} \\
reducer\ f\ b\ h :: t & = & f\ h\ (reducer\ f\ b\ t) & reducer \text{ step} \\
reducel\ f\ a\ [d, e] & = & f\ d\ e & reducel \text{ base} \\
reducel\ f\ a\ h :: t & = & reducel\ f\ (f\ a\ h)\ t & reducel \text{ step} \\
scan\ f\ a\ [\,] & = & [\,] & scan \text{ base} \\
scan\ f\ a\ h :: t & = & (f\ h\ a) :: (scan\ (f\ h\ a)\ a\ t) & scan \text{ step} \\
filter\ f\ [\,] & = & [\,] & filter \text{ base} \\
filter\ f\ h :: t & = & if\ (f\ h) & \\
 & & then\ h :: (filter\ f\ t) & \\
 & & else\ (filter\ f\ t) & filter \text{ step} \\
zip\ f\ [\,]\ [\,] & = & [\,] & zip \text{ base} \\
zip\ f\ h1 :: t1\ h1 :: t1 & = & (f\ h1\ h2) :: (zip\ f\ t1\ t2) & zip \text{ step}
\end{array}
$$

and the distributivity of *map* over compose ($\circ$) (equation 2 below) may be used to change these skeleton functions in the program to produce an optimal implementation.

*map*s are implemented as processor farms but *fold*s can be implemented using divide and conquer strategies, so by applying the *map*/*fold* equivalence we change the way two functions are implemented on the parallel architecture.

If we have a *map*:

$$map\ f\ l$$

The $map/fold$ equivalence is:

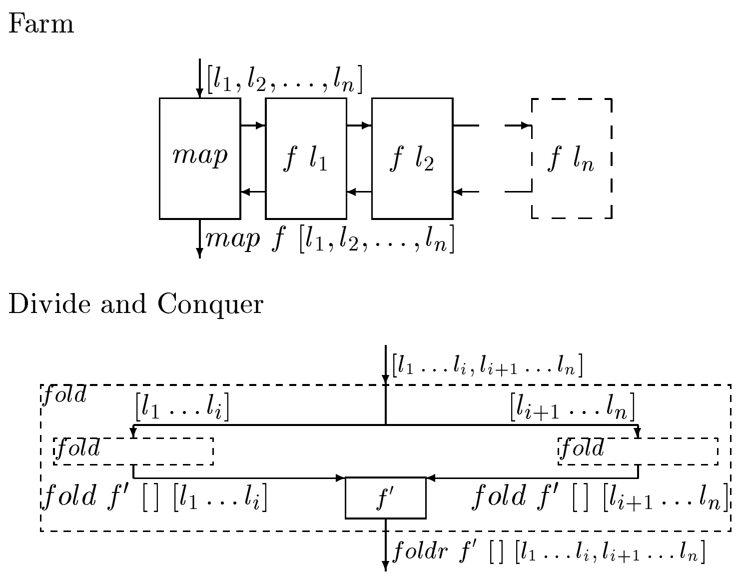
$$map\ f\ l = foldr\ \lambda x.\lambda y.((f\ x) :: y)\ []\ l \tag{1}$$

So we have two possible processor configurations. Figure 3 illustrates the possible parallel structures of $map\ f$, a farm and a divide and conquer.

A (map)farmer sends requests to worker(f) processors. Each worker computes the value of the function argument applied to individual members of the list. Then the farmer collects the results.

A $fold$ divide and conquer strategy divides the list into two portions. It recursively uses a divide and conquer strategy to split each half down further. Then the two halves of the list are joined together using the function argument, f.

Figure 3: Possible parallel structures of $map\ f\ l$



where $f' = \lambda x.\lambda y.((f\ x) :: y)$

Two composed maps will also have alternate implementations depending on how they are arranged. The distributivity property of map over $\circ$ allows the nested maps to be implemented in two different ways.

If we have two composed maps:

$$map\ g\ (map\ f\ l)$$

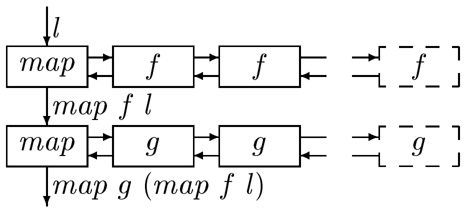
Distributivity of *map* over $\circ$ applied to the example yields:

$$map\ g\ (map\ f\ l) = map\ \lambda x.(g\ (f\ x))\ l \tag{2}$$

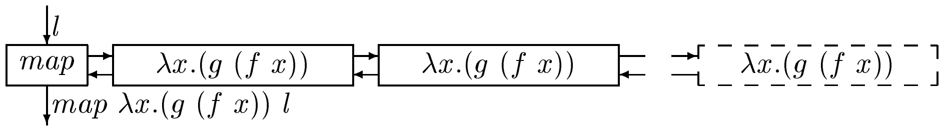
So we have two possible processor farm configurations. Figure 4 illustrates the possible parallel structures of $map\ g\ (map\ f\ l)$.

Figure 4: Two possible parallel structures of the nested map $map\ g\ (map\ f\ l)$

$map\ g\ (map\ f\ l)$ has a processor farm of the form:



And $map\ \lambda x.(g\ (f\ x))\ l$ has a processor farm of the form:



Problems with this approach come when the expressiveness of the programming language is diminished because the programmer is forced to use the pre-defined skeletons. Furthermore in [ST98] Skillicorn claims that there is no natural way to develop algorithmic skeleton programs from higher-level programming constructs or at the skeleton level.

1.3 Functional Programming

In the functional programming paradigm, variable assignments, which are common in imperative programming, are not allowed. The consequence is that once an identifier has been declared it carries the same value for the entire scope of its existence. This simple rule gives functional languages many properties that set them apart from imperative languages. They will obey the Church-Rosser property [Han94][4] so reduction can be performed on arbitrary subexpressions rather than straight evaluation. There is no concept of change of state in the program because values don't change over time. This makes it much easier to reason about functional languages than about imperative languages because the structure of the language more closely matches that of the reasoning logic of most theorem provers. Other useful properties, of some functional languages, include pattern matching, lazy evaluation, and polymorphic typing. Pattern matching in function definitions gives a close relationship between the program and data structure which allows a more elegant expression of the program. Lazy evaluation (values are never calculated until they are needed) allows the manipulation of potentially infinite data structures. Polymorphic typing allows types to be parametrised with other types and functions to be defined over sets of similar types instead of on a single concrete type. However functional languages are often not as efficient as imperative languages because the structure of the language is more distant from native machine code.

The first language to use functional components was LISP (LISt Processing) [McC60] which was imperative but had a purely functional subset. ISWIM (If you See What I Mean) [Lan66] followed LISP and was developed to overcome LISP's over reliance on lists, poor storage allocation and excessive bracketing. SASL (St. Andrews Standard Language) [Tur75] was weakly typed and used lists as its only datatype. A second version of SASL was later developed including pattern matching and lazy constructors. Hope [BMS80] is a pure functional language which provides strong typing and a polymorphic higher-order type system. ML (Meta Language) was developed as a meta language for the LCF [GMW79b] proof system but later matured into SML (Standard

[4] page 34

ML) [MTH90], a language in its own right. SML is actually an imperative language supporting assignment and iteration. However SML contains a widely used pure functional subset including modules for creating large systems. Miranda [Tur85] is based on ideas from SASL and like Hope and SML has polymorphic types. Unlike earlier languages Miranda is fully lazy. Haskell [eee$^+$92] is a relatively new language providing polymorphic types and fully lazy evaluation. Like SML, Haskell incorporates a module system.

1.3.1 Theory of lists

The theory of lists [Bir87] links the practical and formal aspects of functional programming. Lists are an extremely important part of any functional language allowing the storage and manipulation of collections of data.

Lists are recursively defined, usually as an empty list or an element added to the front of another list.

$$
\begin{array}{lcll}
\text{list} & = & [\,] & \text{Empty list} \\
\text{list} & = & \text{h::t} & \text{Element}(h) \text{ added to list}(t)
\end{array}
$$

Many important functions on lists can be defined using these functions. For example functions to find the length of a list, to sort a list, to find the maximum or minimum element of a list may all be defined in terms of *folds*.

foldr, *foldl*, *reducer* and *reducel* are examples of a family of functions which produce a value from a list by "folding" together the elements of the list by applying a binary function to them. The *reduce* functions are similar to the *fold* versions except that they do not use a user specified base case. In the 'r' functions the parentheses around the computation are right associative while in the 'l' functions it is left associative:

$$foldr \oplus \text{b } [e_1,e_2,\cdots,e_n] = (e_1 \oplus(e_2 \oplus(\cdots\oplus(e_n \oplus \text{b})\cdots)))$$

$$foldl \oplus \text{b} \; [e_1,e_2,\cdots,e_n] = ((\cdots((\text{b} \oplus e_1) \oplus e_2)\oplus\cdots)\oplus e_n)$$

BMF(Bird-Meertens Formalism) is a notation for representing functional programs. It is designed to reduce the amount of clutter and to represent definitions in a compact, precise and hopefully more readily understandable form. The symbols $\oplus$, $\otimes$ and $\odot$ are used infix to denote arbitrary binary functions. $\circ$ is used as the function composition operator and $+\!\!+$ is the list concatenation symbol. HOFs such as map ($*$) and reduce ($/$) are also represented in the BMF. The identity element for an operator $\oplus$ is written as $1_\oplus$[5]. Figure 5 illustrates the *reals* (*sizes l*) expressed in BMF.

Figure 5: The $reals(sizes)$ example expressed in BMF

$reals\ (sizes\ l)$	$\rightarrow$	$(sizes\ \circ\ reals)\ l$
$map\ real\ (map\ size\ l)$	$\rightarrow$	$((*\ size)\ \circ\ (*\ real))\ l$
$map\ \lambda x.(real\ (size\ x))\ l$	$\rightarrow$	$*\ (size\ \circ\ real)\ l$

$\lambda x.(real\ (size\ x))$ indicates the composition of the two functions *real* and *size* so that for any argument *size* is applied to it then *real* applied to the result.

For clarity we will use the names of the HOFs instead of their BMF representations.

Because of their recursive definitions, properties of lists and the HOFs on lists are proved, directly or indirectly, using induction. Various interesting properties of the list manipulation functions are true only if some property of their argument function holds. It can be shown that $foldr$ and $foldl$(Figure 2) give the same result for the same arguments if the operator they apply to the list is associative and the base case element given is the identity element for the operator(Figure 6).

This property of folds is important because if the function used ($\oplus$) is associative then both $foldr$ and $foldl$ can be represented by a similar parallel implementation. In

[5] The identity element for an operator is the value that when given as one of the arguments of the operator the result will always be the other argument. $1_\oplus \oplus e = e$ and $e \oplus 1_\oplus = e$

Figure 6: Associativity in $foldr$ and $foldl$

The expression:

$$\begin{aligned} foldr \ \oplus \ b\,[e_1, e_2, \cdots, e_n] &= foldl \ \oplus \ b\,[e_1, e_2, \cdots, e_n] \\ (e_1 \oplus (e_2 \oplus (\cdots \oplus (e_n \oplus b) \cdots))) &= ((\cdots((b \oplus e_1) \oplus e_2) \oplus \cdots) \oplus e_n) \end{aligned}$$

is true if $\oplus$ is associative.

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

and b is the identity element of $\oplus$

$$b = 1_{\oplus}$$
$$b \oplus x = x \text{ and } x \oplus b = x$$

fact the parallel divide and conquer strategy only works when this is the case. This property is linked closely to the idea of homomorphisms [Bir87]. h is a homomorphism iff:

$$h(x +\!\!+ y) = h(x) \ \oplus \ h(y)$$

Since the concatenation can take place anywhere in a list, because $+\!\!+$ is associative, the operator $\oplus$ needs to be associative. $h([\,])$ must be equal to the identity element of $\oplus$ so that $h(x +\!\!+ [\,])$ and $h([\,] +\!\!+ y)$ will have the correct values. These properties mean that any instance of $fold$ which has the $foldr = foldl$ property is a homomorphism. Any map will also be a homomorphism where $\oplus = +\!\!+$, since map works over each element independently and the base case for map is $[\,]$ which is the identity element for $+\!\!+$. Each homomorphism h can be defined by two functions f and $\oplus$ where:

$$h\ l = fold \ \oplus \ 1_{\oplus} \ (map\ f\ l)$$

Skillicorn [Ski95] uses homomorphisms and the BMF as the basis for a parallel software development technique. By insisting that functions are expressible as homomorphisms he ensures that programs exhibit good parallel behaviour.

The expression of programming constructs as homomorphisms is important for parallelising programs using skeletons. The HOF map is expressible as a homomorphism. Also the $fold$ and $reduce$ are expressible as homomorphisms when their function arguments are associative. This is the condition that needs to be satisfied to enable a parallel divide and conquer strategy for $fold$ and for the foldr/foldl equivalence (and the corresponding reducer/reducel equivalence) to be true.

The Boom hierarchy [Bir87] describes a hierarchy of types, trees, lists, bags and sets, which are closely linked. In the Boom hierarchy lists are defined as the concatenation of two trees[6] rather than the addition of an element on to a list. This means that there is some ambiguity in the de-construction of lists. This ambiguity arises from the ability to split the list at any point; a choice of where to split the list must be made instead of it being fixed as the isolation of the first element. Without the possibility of splitting the list at any point the relationships between the types would not be so strong. The Boom hierarchy says that the types differ only in the properties of the concatenation operator. A list is a tree where $++$ is associative so that the exact bracketing is disregarded. A bag is a list where $++$ is commutative so the exact order of the elements is disregarded. A set is a bag where $++$ is idempotent[7] so that the repeated elements are disregarded. In [Bac88] various properties about these relationships are proved. Especially interesting are the proofs that properties of one datatype can hold in the others as long as certain side conditions hold. The Boom hierarchy is interesting to us because it shows that if we can introduce parallelism for one type then that parallelism might also be exploited for other types. Though we do not take advantage this property in this thesis it remains an interesting topic for further work because the techniques we develop may be adapted

[6] The concatenation of two trees forms a new tree with each of the two original trees as complete subtrees of the root of the resulting tree.

[7] idempotent means that the result of an application to two identical elements is the same as a single such element. In the case of $\oplus$, $a \oplus a = a$

for use on other datatypes(section 8.4.7).

The side condition distinguishing lists from trees is an important one. The associativity of a function is important in forming a homomorphism. A tree structure may still have a parallel implementation but its structure explicitly determines which parts of the tree may be done in parallel. Given a well balanced tree, one in which at each node in the tree the subtrees are of roughly equal size, it may be possible to produce a parallel implementation as efficient as for a list but there may be no way of ensuring a well balanced tree structure.

Properties of lists are central to our manipulation of potential parallelism in the transformer. The use of HOFs on lists provides considerable potential for manipulating parallelism.

Standard properties such as the distributivity of *map* over compose are used in BMF to transform programs. The functions described in Figure 2, in particular map, foldr, foldl, scan and reduce, are useful in exploiting parallelism and their properties can be used as the basis of the transformation rules. Gorlatch describes how transformations in the BMF, specifically on the functions *scan* and *reduce*, can be used to increase possible parallelism [GL98].

The Boom hierarchy [Bac88] is interesting in that it seems to promise a framework for extending the transformation rules for exploiting parallelism to work over a larger set of types. The proof of some property, and the corresponding transformation rule, about one type is the same as the proof of a comparable property of another with a few side conditions added.

The BMF relies on the $++$ constructor to produce desirable parallelisable behaviour. However most functional programming languages use the :: operator as list constructor because it allows pattern matching to be unambiguous, it separates the first single element from the rest of the list rather than indicating the concatenation of two lists of arbitrary length. Unfortunately it is the ambiguous behaviour of ++ which produces the desirable parallel behaviour. To enable the exploitation of parallel behaviour lists constructed using :: may have to be converted to constructs which use a constructor more like ++. In the compiler project (section 1.1) the aim is to compile programs written in SML into an parallel implementation and this conversion can be done as part of this process. In this thesis we will take lists constructed with :: and ++ to be the same.

1.3.2 SML

ML was originally a meta-language for the LCF theorem prover [GMW79a]. It has become a more general purpose language and is now in wide usage. Our project is based on parallelising Standard ML [HMM86].

SML includes imperative features such as assignment but has a pure functional subset free from assignment and side-effects. SML is a strict language; eager evaluation is used rather than lazy evaluation.

SML is a polymorphic language where types of functions and data-types can be represented by type variables. This allows the same function to be applied to structures of different types. For example a function defined on general lists will work on lists of integers as well as on lists of reals. SML is also strongly typed so that all object types can be determined at compile time [Ull94].

In SML, types of functions can be either tupled or curried, for example: $(a, b) \to c$ or $a \to (b \to c)$. Curried types allow partial application; the application of only one argument of a function with more than one argument yielding a result which is a specialised function needing one less argument. For example the application of the function of type $a \rightarrow (b \rightarrow c)$ to a particular a yields a function of type $b \rightarrow c$.

SML has many properties which make it a good language for transformation and synthesis for parallelisation, in particular support for HOFs, curried function types, and recursion. HOFs are essential for the compiler project (section 1.1) since the parallelism is dependent on their manipulation. Currying is important for the expression of argument functions to HOFs because an argument function may take the form of a partially evaluated function.

The choice of SML for use in the project was made primarily for pragmatic reasons. However SML, being a strict functional language, is well suited to the processes of verification and synthesis.

1.4 Contributions

This thesis presents several contributions. They are as follows:

1. We have shown that properties of HOFs can be verified using the *Clam* or λ*Clam* theorem provers and in doing so extended their corpus of examples.

2. We have extended the capabilities of *Clam* with a critic to unblock the fertilisation step.

3. We have developed a plan for synthesising expressions involving HOFs from expressions that lack them.

4. We have implemented our synthesis plan in the λ*Clam* proof planner.

5. We have evaluated the implemented plan on examples with multiple sites of potential HOFs.

6. To facilitate the integration of synthesis into the parallel compiler, we have implemented a system for translating ML functions into λ*Clam* definitions and a

λ*Clam* portrayal system for outputting expressions in ML syntax.

1.5 Structure of This Document

The above contributions are developed in the following chapters.

Chapter 2 provides an overview of relevant technologies, in particular program transformation and synthesis, theorem proving technology, proof planning and the rippling heuristic which we use to guide synthesis and to verify transformations.

Chapter 3 documents the verification of various transformation rules and the implementation of the fertilisation critic which can unblock verification proofs.

Chapter 4 provides a record of the evolution of the synthesis plan from its initial conception, through each major change, to the implemented plan and a proposed plan.

Chapter 5 describes the implementation of the synthesis plan and the alterations made to λ*Clam* to allow the synthesis to successfully produce HOF transformations with multiple sites of potential HOFs.

Chapter 6 discusses the results from three examples. In particular the times taken to produce multiple results for each example using the backtracking of λ*Clam* are analysed.

Chapter 7 describes the two translators needed to convert the program syntax(SML) into the syntax of the λ*Clam* theorem prover and back again. This makes the integration of our synthesis plan into the parallel compiler project possible.

Chapter 8 surveys our achievements and describes ways in which the work could be improved and extended.

Chapter 2

Background

2.1 Program Transformation

Program Transformation ranges from program optimization(section 2.1.1) to transforming an unexecutable high-level specification into an executable program [MW79] (section 2.2). Partsch's survey [PS83] provides a good overview and comparison of the various transformation systems.

2.1.1 Fold/Unfold Methodology

The fold/unfold methodology (FUM) [Dar77] is a method of transformation used to transform functional programs. The FUM consists of six rules given in Figure 7.

The FUM is problematic to mechanise. While most of the steps are straight forward the definition rule involves introducing new structure. Applications of the definition rule, or other rules which introduce new structure, are called *eureka steps* because their application requires inspiration on the part of the user. The FUM is not complete and contains the possibility of non-termination. Burstall and Darlington [Dar77] explored the use of a few simple heuristic strategies to guide transformation, based on observations of transformations done by hand on examples. They produced a semi-automatic system which relied on user inspiration to provide the eureka steps. Darlington's paper [Dar81] describes the behaviour and implementation of the system.

Figure 7: Rules of the fold/unfold methodology

1.	Definition:	Enables the introduction of a new recursion equation whose left hand side is not an instance of the left hand side of any previous equation.
2.	Instantiation:	Enables the introduction of a substitution instance of an existing equation.
3.	Unfolding:	If $E = E'$ and $F = F'$ are equations and there is some occurrence of E in F then it can be replaced by E' to obtain F'' and the equation $F = F''$ added to the set of equations.
4.	Fold:	If $E = E'$ and $F = F'$ are equations and there is some occurrence of E' in F' then it can be replaced by E to obtain F'' and the equation $F'' = F$ added to the set of equations.
5.	Abstraction:	A 'where' clause may be introduced, abstracting out of an equation $E = E'$ the variables that are required. $E = E'[u_1/F_1, \cdots, u_n/F_n]$ where $(u_1, \cdots, u_n) = (F_1, \cdots, F_n)$
6.	Laws:	This rule allows the application, on the right hand expression of any equation, of any laws or axioms known about a primitive.

The FUM as described is not complete, because certain transformations may not be made but are feasible if the transformation is made in reverse (transforming the result to get the original). The addition of a seventh rule, redefinition, was suggested [Dar77] which allows a sequence of transformations to be reversed. With this rule the system may be considered to be complete.

Consider the following two equations:

$$f(0) = 0 \qquad (1)$$

$$f(n+1) = f(n) \qquad (2)$$

The equation $f(n) = 0$ is equivalent but the original rules (1 and 2) cannot produce it from the two equations. However as transformation in reverse is possible, the two original equations can be obtained from the desired equation.

$$\begin{array}{rcll} f'(n) & = & 0 & \text{definition} \quad (3) \\ f'(0) & = & 0 & \text{instantiate} \quad (4) \\ f'(n+1) & = & 0 & \text{instantiate} \quad (5) \\ f'(n+1) & = & f'(n) & \text{fold 3 with 5} \quad (6) \end{array}$$

From equation 3, the desired equation, equations 4 and 6, the original equations, have been produced. By the redefinition rule equations 4 and 6 can be transformed to 3 because 3 can be transformed to 4 and 6.

Later on Feather [Fea82] looked at the application of the FUM to larger problems by using 'meta-programs' to guide the transformation process. The system he developed introduced additional levels of abstraction above the FUM. The transformation strategy level is for describing overviews guiding the transformation of programs. The transformation tactic level was a lower level concept dealing with individual functions and how performance increases could be brought about in them. Feather proposed three tactics, combining(*fusion*), *tupling* and *generalizing* which are described in section 2.1.3. The pattern-directed transformation level was a lower-level layer dealing with the application of the FUM guided by the transformation tactic which describes the objective of the transformation. Pettorossi and Proietti recently wrote a good overview of functional and logic program transformation techniques and strategies [PP96].

Interactive program development systems like CIP [F.L85, FP87, BMPP89] (Computer aided, Intuition guided Programming), PROSPECTRA [KB90] (PROgram development by SPECification and TRAnsformation) and KIDS [Smi90] (Kestrel Interactive Development System) use the FUM and other transformation methods to refine a general specification into an efficient executable program. These systems typically use a higher-order algebraic specification language (CIP-L in CIP, Panda-S in PROSPECTRA) as their starting point. The specification is transformed into an executable program then performance enhancing transformation tactics (described in section 2.1.3) can be used to optimise it. Such systems typically either suffered from a lack of integrated theorem proving technology or the need for expert user interaction and so are unable to perform correct automatic program transformation.

Theorem provers including higher-order capabilities like *Isabelle* [Pau86] and HOL [Gor88] were used to produce systems which apply only provably correct transformations [Bas98]. Such systems will produce programs which will always be correct with respect to their specifications. However the emphasis of such systems is often on correctness rather than usability and so they require some knowledge of proof on the part of the user. More recently there have been attempts to combine CIP type program development systems with theorem provers like *Isabelle* using a high-level user interface system [LKKB99], to make the user interaction less dependent on knowledge of proof.

These types of transformational program development systems have been developed for different types of environments. CIP and PROSPECTRA manipulate mainly functional programs though they can be used to generate imperative iterative loops from tail recursive programs. Imperative program development is harder than for functional programs because of the possibility of side effects but the use of temporal logic and control flow graphs has been suggested to overcome this [LdM01]. Systems for developing concurrent and distributed programs have also been attempted using temporal logic [OR93].

2.1.2 Proof Driven Transformation

It is possible to synthesise programs from a high level specification by constructing a proof of the internal consistency of the specification in a certain form and extracting from the proof a functional implementation which meets the specification. This synthesis method is described in section 2.2.1.

With the "proofs as programs" principal [Con83], the program is constructed out of the proof. If the proof is transformed then the constructed program will be also be transformed. By constraining the manipulation of the proof in different ways it is possible to transform a program into a more efficient one . The programs are guaranteed to be functionally equivalent if the proof is rigorous since the goal of the proof (the specification of the program) is the same for both the original and transformed proofs. Madden [Mad92] uses the extra information available in the proof to automate this process of program optimisation for tupling[1] transformations.

[MBS99] provides an overview of program optimisation through synthesis proof transformation and describes an *Oyster* (section 2.4.6) based transformation system for transforming such proofs. This system works by identifying the key points at which a synthesis proof can be altered to produce beneficial results. These key points are the application of an induction scheme, the type and structure of objects that witness the induction cases and the definitions used to complete verification.

Richardson wrote his PhD thesis [Ric95] on the application of the proof planning framework(section 2.3.4) to program transformation by transformation of datatypes. He describes how *Clam* (section 2.4.6) can be used to perform transformations on constructive synthesis proofs.

[1] See section 2.1.3

2.1.3 Transformation Tactics

There are several different tactics used to optimise programs. Among these are *memoisation*, *tupling*, *generalisation*, *introducing accumulators*, *finite differencing*, *fusion*, *abstract datatype transformation*, and *partial evaluation*. Busvine surveyed these various techniques in a technical report [Bus90]. Wegbriet [Weg76] describes the concept of goal directed transformation to produce a program which is more efficient than the original. He uses partial evaluation and other transformation techniques to achieve this.

Memoisation [Mic68] is a way to improve program efficiency by eliminating multiple instances of the same function call. In a functional language, if the arguments and function of two function calls are the same the result will always be the same. This means that if the result from a function call can be stored then the call need not be calculated again but can simply be retrieved from the stored value. Storing the result of all function calls may increase time efficiency but is very space inefficient. To limit the size of the look up table scheduling strategies can be used [Hil76]. Various other strategies can be employed to determine or estimate exactly which function calls will be needed in the future. These include local memoisation, exact tabulation and over tabulation [Bir80]. Cohen [Coh83] uses descent conditions and directed acyclic dependency graphs to determine which results of recursive calls will be used more than once. This is used to introduce transformations which will introduce a look up table of values which use the minimal amount of space but will eliminate all redundant recursive calls.

Tupling allows two different but linked values to be calculated without the duplication of computation. One function is used to calculate two values which are arrived at via the same intermediate results and return both values in a tuple. By calculating both values together we eliminate duplicated calculation of the intermediate result. Tupling is mentioned in [Fea82] and [PP96].

Generalisation of a function can help to increase its efficiency indirectly by transforming it into a function which is amenable to further transformation [Dar77]. An example of generalisation is the conversion of a recursive function into tail recursive form, by introducing an accumulator. A tail recursive function may not necessarily be any more efficient but it may be converted easily into an efficient iterative loop. A continuation is a data structure which represents the future course of computation [Wan80] and the introduction of a continuation may be considered to be like the introduction of a higher order accumulator. With a continuation a function which calculates the return value is built up during the recursion and is evaluated on reaching the base case. This may not seem a particularly efficient way of doing things since a growing function has to be passed around but often the structure of the function can be simplified to produce a more efficient version. The use of continuations allows the use of global context in subsequent transformations.

Finite differencing can be used when a complex function is called a number of times with successive numbers and the relationship between the results of successive calls is the result of a simpler, more efficient function. By using the simple function to compute the output of subsequent calls to the complex function the efficiency of the program can be increased [PK82].

Fusion works over function composition, where the result of one function call is used as the argument of another [Dar77, Fea82]. Fusion combines the functions together into a new function. This eliminates the calling overhead for one function, more if there are multiple nested function calls. The advantages of using fusion can be more than just eliminating function calling overheads. The combined function can also remove any intermediate structure(called deforestation [Wad88]). This structure would be built up in one function and deconstructed in another to enable the transfer of information from one function to the other in the original program. In the fused function such structure is unnecessary and can be eliminated [Wad88].

Abstract data types, due to their nature, are often implemented inefficiently. A programmer might define a mapping from a more efficient concrete type to the abstract type, an abstraction function. The abstract type can be used freely and the compiler should be able to transform the abstract manipulation functions to functions which manipulate the corresponding concrete type. Examples of abstract data type transformation can be found in [Dar77, FH88].

Partial evaluation may be done when an argument of a function is known at compile time. The function may be partially evaluated with this argument in order to obtain a more efficient function. Even if the exact value of an argument is not known the structure of the argument may be known and this can allow some specialisation and efficiency increase. The fusion of two functions can be considered a special case of partial evaluation [Wad88]. The result of the inner function produces some known structure which can be used to partially evaluate the outer one.

Transformation Tactic Summary

In relation to the introduction of exploitable parallelism into functional programs some of the techniques described above may be useful while others are not.

Memoisation is probably not particularly useful in the context of parallel computation since maintaining a large table of values and keeping it synchronised across multiple processors is much too complicated. However the technique described by Cohen [Coh83], which produces multi-accumulator functions, might be useful in producing efficient functions without using memo tables.

The advantages of *tupling* are probably diminished in a parallel setting and may even interfere with the introduction of parallelism. *Tupling* eliminates duplicated computation by calculating two values, which are arrived at via the same intermediate structure, together and returning a tuple. In a parallel context the duplication of calculation may have less impact because it might be performed on separate processors simultaneously. Also by forcing the calculation of two values by one function there is the potential to interfere with benefits that might occur by performing each function separately on different processors. *Tupling* may be used in conjunction with parallelisation techniques but care should be taken that it does not interfere.

Generalisation has some importance to the introduction of parallelism and the synthesis of HOFs. There are situations where a function appears not to be amenable to transformation but with the introduction of another argument in place of a hard wired constant the function may be made to fit the criteria for parallel transformation. This type of generalisation is necessary in the proofs of functions which use accumulators to allow synthesis of functions within the accumulator (section 4.6).

Finite differencing seems to be at odds with the concept of parallel computation as the idea behind finite differencing depends on the calls to a function being carried out in sequence with the result of one call available to the next. This technique appears to be of little use in a parallel framework.

Fusion, promises the elimination of expensive calling overheads but the eager application of fusion can restrict the malleability of the program with respect to parallelisation. Fusing two functions together may restrict the ability of synthesis to discover possible HOFs especially if intermediate recursive data structures are eliminated. Fusion like transformations can be applied after HOF introduction by properties such as map/compose distributivity (section 3.3.2). Fusion, like tupling may be used in a parallel environment but care should be taken that the parallel behaviour is not restricted due to overeager application of the fusion technique.

The optimisation of *abstract datatypes* is an interesting concept in relation to parallelism. It may be possible to envision ways of parallelising user defined recursive data structures and various similarity properties have been shown about some data types using BMF [Bac88](see section1.3). However, given an arbitrary data structure, automatically extracting parallelism from it in the form of algorithmic skeletons is very hard.

Partial evaluation seems to be more or less the converse of function generalisation. Instead of introducing extra arguments into the functions, arguments are removed as they are absorbed into the function specialising it. The process of specialisation may decrease function calling overheads but it may also interfere with the introduction of parallelism. This specialisation reduces function calling and could prevent the automatic discovery of situations where such functions might be run in parallel.

These transformation tactics don't seem to have direct relevance to the problem of deriving parallelism from the discovery of HOFs. Although most of these techniques may be used successfully in conjunction with our technique there is great potential for them to interfere with it if used too eagerly. These techniques tend to work by specialising the program into a more efficient form, our technique of finding HOFs performs a form of generalisation on the program and at this level seems to be in direct opposition to most of these techniques. The introduction of HOFs will probably cause a sequential program to be slightly less efficient; it is only in the context of parallelism that the advantages of using HOFs are seen. The techniques of *generalisation* and *abstract datatypes* may be interesting transformations to perform in conjunction with the discovery of HOFs but we do not examine this here.

2.1.4 Transformation Verification

When performing program transformation it is important to have some confidence that the result will perform the same function as the original program. There are various approaches to this problem: transformation rules used can be proven to be correctness preserving; the resulting program can be proven to meet some specification; the transformation can be done using a proof engine to ensure that transformations are applied consistently.

Burstall and Darlington's transformation engine using their FUM requires some user interaction to transform programs. Their methodology has been proven to be correct dependent on the correctness of input given by the user. Huet and Lang [HL78] present an extension of Burstall and Darlington's work which uses transformation rules composed of second order patterns. Their technique includes the ability to formally prove the correctness of individual transformation rules.

Shankar [Sha96] uses the verification system PVS to mechanise application of the fusion and continuation transformation strategies. The use of PVS provides verification of the validity of these transformation strategies. There is also the possibility that other transformation strategies could be mechanised and verified in the same way.

Transformation verification has been done in the higher-order theorem provers Isabelle and HOL as part of transformational program development systems. These systems typically use theorem proving technology in an attempt to produce systems similar to CIP and PROSPECTRA but with more emphasis on provable correctness.

Transformation systems based on the proofs as programs principal rely on the transformation of synthesis proofs to change the synthesised program. These systems gain a correctness guarantee as a side effect of their use of proof synthesis to perform transformation.

Early synthesis work [BM73, WL69] relied heavily on theorem proving techniques to ensure correctness. Later special purpose synthesis systems were built which did not utilise theorem proving [BD77, MW79]. Manna and Waldinger [MW80] reunited synthesis and theorem proving combining unification, mathematical induction, and transformation rules.

Barstow's knowledge-based synthesis technique [Bar77] uses a large data base of information to perform program synthesis. Later Barstow [Bar79] proposed the introduction of a theorem prover so that the rules of the system could be reasoned about and various properties proved about the synthesised programs.

2.2 Program Synthesis

An overview of various synthesis methods is given in [DL94]. The following survey is primarily concerned with the synthesis of logic programs but the general techniques of deductive and inductive synthesis have been applied to functional and imperative languages as well. The technique of constructive program synthesis seems to be particularly suited to the synthesis of functional and logic programs and consequently is the focus here.

2.2.1 Constructive Synthesis

It is possible to synthesise programs from a high level specification by constructing a proof of its internal consistency and extracting from it a functional implementation which meets the specification. The "proofs as programs" [Con83] principal relies on the Curry-Howard Isomorphism which relates inference rules and functional terms of λ-calculus. This ensures that the program is correct with respect to the specification while the concept of "propositions as types" ensures that the program constructed is type correct [Abr94]. $t : T$ may be viewed in different ways:

t is a proof of T

t is an element of type T

t is a program meeting the specification T

These relationships guarantee that the program extracted from the proof of T is both type correct and that it is correct with respect to the specification.

Sequent calculus is used to construct these proofs, pure existence proofs[2] must be eliminated by using a constructive logic or restricting the proof. In a constructive logic, evidence must be provided for a proof to be valid. It is not enough to show $A \vee B$; a complete proof of either A or B must be provided and identified as such. For instance, a proof of $A \vee \neg A$(which is classically true) must consist of a proof of either A or $\neg A$ and an indication of which has been proven. This ensures that the proof contains evidence that one of A and $\neg A$ actually holds so that a program or program fragment can be constructed from it.

The proof is constructed by a process of refinement starting with a specification or goal and working backwards to produce a proof tree the leaves of which will be axioms. Each proof rule has a construction rule associated with it which builds an extract term corresponding to the program being constructed. So each proof step used in constructing the proof tree not only expands the boundary of the proof tree but also builds up the extract term. The leaves of the proof tree are axioms or previously conjectures.

[2] Proofs which show the existence of a solution to the specification without producing an example of it

The logical connectives used in the sequent calculus and the rules concerning them correspond to constructs in a functional programming language. The construction rules use the correspondence to build the program in the λ-calculus. Implication($\supset$) corresponds to function($\rightarrow$), and the rule for implication has associated with it a construction rule producing a function call construct. One rule for the universal quantifier($\forall$) constructs a function declaration construct. Other constructs are built in a similar way.

The cut rule, although unnecessary in some logics, is required for lemma application and generalisation in program synthesis where induction is involved. The cut rule can be used to add new hypotheses to the proof corresponding to lemmas which may be needed.

Introducing induction into the proof allows the construction of recursion in the program. Different induction schemas using different inference rules will produce different types of recursion in the constructed program.

The programs as proofs principal allows synthesised programs to be transformed via manipulation of their constructive synthesis proofs. This can be exploited as a program transformation technique(section 2.1.2).

2.2.2 Deductive Synthesis

In deductive synthesis [DL94] a specification is taken and a program is deduced from it by the application of predefined deduction rules.

Most deductive synthesis methods use first order logic sentences for the specification because program clauses can be deduced directly from the specification and if the deduction rules can be shown to be correct then the result will be correct. Otherwise more sophisticated methods, such as theorem proving, may be required to ensure that the synthesised program is correct with respect to the specification.

The methods described by [Hog81] and [LP90] start with a set of if and only if formulae and use symbolic evaluation, logical deduction or a resolution like mechanism to deduce a program.

Work by Waldinger and Lee in functional program synthesis [WL69] relied on resolution theorem proving. The difficulties with representing mathematical induction in these theorem provers made the synthesis of programs involving recursion difficult. Manna and Waldinger [MW79] abandoned the use of a theorem prover and implemented a system which directly used transformation rules to refine a specification into a runnable program. Theorem proving systems became much more able to handle mathematical induction and subsequently Manna and Waldinger [MW80] again integrated their techniques with a theorem prover.

The FUM presented by Burstall and Darlington [BD77] was designed as functional program transformation system. In [Dar78] Darlington shows how the same rules can be used for program synthesis. The practical aspects of Burstall and Darlington's system are described in [Dar81].

Barstow [Bar77] presented a system for synthesising LISP programs from specifications using a knowledge-based approach. The system refines the specification using a set of rules until a runnable program is produced. Barstow uses a very large set of rules which encode specific knowledge about the programming language constructs and specific constructs in the target language. This contrasts with Burstall and Darlington, and Manna and Waldinger who both use a small set of general rules. Barstow explored the use of theorem proving in the knowledge-based system [Bar79] and the inclusion of efficiency information for guiding the synthesis [KB81].

Dershowitz describes a set of synthesis rules for the deductive synthesis of imperative programs [Der85]. He concentrates on synthesising imperative loops through the techniques of finding invariant assertions and subgoal assertions.

2.2.3 Middle-Out Synthesis

Included in the deductive synthesis style is the work by Kraan [BBKM93, KBB93a, KBB93b, Kra94, KBB96] using middle out reasoning and proof planning(see sections 2.3.4 and 2.3.4) to guide a synthesis proof. The proof of the equality between a program and its specification is set up but the program is represented by a metavariable, a variable introduced at the planning level to represent some arbitrary terms structure, which will be instantiated during the course of the proof.

This method uses a constructive style of proof, but this type of synthesis using middle out reasoning is deductive because it starts with a specification and during the course of the proof the structure of the program is gradually deduced as metavariables are instantiated.

Kraan uses middle-out reasoning and constructive proof to synthesise a pure logic program from an unexecutable specification. Kraan formally defines the concept of pure logic programs in [KBB96]. She illustrates the synthesis process using a *subset* function, whose specification is:

$$\forall i, j.subset(i, j) \quad \leftrightarrow \quad (\forall x.member(x, i) \rightarrow member(x, j))$$

The definition of *member* is:

$$\forall x, l.(member(x, l) \quad \leftrightarrow \quad \exists h, t.l = (h :: t) \wedge (x = h \vee member(x, t))$$

The original uninstantiated program is:

$$\forall i, j.subset(i, j) \quad \leftrightarrow \quad \mathcal{P}(i, j)$$

where $\mathcal{P}$ is the metavariable representing the program body. Recursion analysis[3] suggests a one step induction; induction on i constrains the form of recursion in the program yielding:

$$\begin{aligned} \forall i, j.subset(i,j) \quad \leftrightarrow \quad & i = [\,] \wedge \mathcal{B}(j) \vee \\ & \exists h, t.i = (h :: t) \wedge \mathcal{S}(h,t,j) \end{aligned}$$

The base case of the induction is:

$$subset([\,],j) \quad \leftrightarrow \quad (\forall x.member(x,[\,]) \rightarrow member(x,j))$$

Using symbolic evaluation this becomes:

$$subset(nil,j) \quad \leftrightarrow \quad true \tag{3}$$

By appealing to the program specification we complete the base case proof and instantiate the base case of the program:

$$\begin{aligned} subset([\,],j) \quad \leftrightarrow \quad & [\,] = [\,] \wedge \mathcal{B}(j) \vee \\ & \exists h', t'.[\,] = h' :: t' \wedge \mathcal{S}(h',t',j) \\ subset([\,],j) \quad \leftrightarrow \quad & \mathcal{B}(j) \end{aligned} \tag{4}$$

By unifying equations 3 and 4 the instantiation $\lambda u.true$ is found for $\mathcal{B}$ so the program is now:

$$\begin{aligned} \forall i, j.subset(i,j) \quad \leftrightarrow \quad & i = [\,] \wedge true \vee \\ & \exists h, t.i = (h :: t) \wedge \mathcal{S}(h,t,j) \end{aligned}$$

The step case of the proof is rippled (see section 2.4) and simplified to:

$$subset\ h :: t\ j \quad \leftrightarrow \quad member\ h\ j \wedge subset\ t\ j \tag{5}$$

[3]Recursion analysis is a technique for examining the recursive structure of a conjecture to decide what form of induction rule is most likely to be successful in proving it.

This residue (equation 5) corresponds to the step case of the program being synthesised. However the residue must be shown to follow from the program definition; this will further instantiate the body of the program. The program is appropriately instantiated and simplified:

$$\begin{aligned} subset(h :: t, j) &\leftrightarrow \quad h :: t = [\,] \wedge \mathcal{B}(j) \vee \\ &\qquad \exists h', t'.h :: t = h' :: t' \wedge \mathcal{S}(h', t', j) \\ subset(h :: t, j) &\leftrightarrow \quad \mathcal{S}(h, t, j) \end{aligned} \tag{6}$$

Unifying the equations 5 and 6 instantiates $\mathcal{S}$ with $\lambda u, v, w.member(u, w) \wedge subset(v, w)$. This completes the step case and produces a fully instantiated program:

$$\begin{aligned} \forall i, j.subset(i, j) \quad \leftrightarrow \quad & i = [\,] \wedge true \vee \\ & \exists h, t.i = (h :: t) \wedge member(h, j) \wedge subset(t, j) \end{aligned}$$

Kraan uses the inductive theorem prover *Clam* to automate this synthesis process. Rippling and *Clam* are described in section 2.4.

An extension to this work has been recently presented [LRS00] using higher-order logic in the λ*Clam* proof planning engine instead of the first order logic of the *Clam* proof planner.

Smaill, Green [SG95] and Armando [ASG97] used *Clam* as the basis for the synthesis of functional programs. Induction using the rippling heuristic is used to guide the synthesis process. Middle-out reasoning is also used to address the difficulties of proving existential theorems. [ASG97] uses a unification algorithm to demonstrate synthesis of recursive programs within *Clam*.

Recently Stark [Sta00, SI99] describes a system called *Bertha* which uses middle-out reasoning and the proof planning framework of *Clam* to constrain the search in imperative program synthesis.

2.2.4 Inductive Synthesis

Inductive synthesis [DL94, Plo70, Plo71] is a method based on techniques from Artificial Intelligence, called inductive inference, where conclusions are induced from incomplete data. A set of examples may be used to indicate how a program should behave, and a program may be synthesised from these. Because the program is synthesised from is incomplete data it may deviate from its expected behaviour if given an unusual set of inputs.

Inductive synthesis is related to the concept of generalisation just as deductive synthesis is related to the concept of specialisation. In inductive synthesis a set of specific examples or other data is given and a general program is synthesised from it; a program which will produce the correct results from the specified inputs. In deductive synthesis however we have a general unrunnable specification and synthesis will produce a specific runnable program from it.

Synthesis can be done in a bottom-up or top-down fashion [DL94, Plo71]. Bottom-up synthesis will produce least general solutions first and refine the results. An example of a least general program is a program which has cases for each of the specified examples and will produce garbage results for any other inputs. Top-down on the other hand will produce the most general solutions first.

2.3 Automated Theorem Proving

Mathematicians have been conjecturing theories for centuries about the properties of mathematics. To prove or disprove a property they try to show that it follows logically from some generally accepted facts or axioms. Proofs consist of reasoning steps to show how one formula follows from another until the theorem that is to be proved is reached. Often these proofs are long and very involved. Because of this it is easy for human mathematicians to make mistakes in proofs.

Computers are very good at doing simple, repetitive work without making mistakes. With the use of a computer the tedious, repetitive parts of proof can be automated. One might think that getting a computer to produce a logical proof would be easy since computers seem to be far more logical(at times) than humans. However the process of producing a proof requires a great deal of search and if we are to arrive at a solution in an acceptable time the proof often needs to be approached in a more intelligent way. The search spaces required in many proofs can be extremely large or even infinite so proofs need to be guided. Until we can teach computers to intelligently apply logic to a problem the guidance for proof construction must come from human users who either program the computer to find certain classes of proofs or interact with them during the course of a proof.

2.3.1 Resolution

Resolution [Rob65] is a method for proving properties of first-order theorems. In a first order predicate calculus the resolution method consists of one resolution rule and proceeds by repeatedly applying it to the problem.

The resolution rule is:

$$\frac{(\mathcal{A} \vee \mathcal{B})\ (\mathcal{C} \vee \neg\mathcal{D})}{\mathcal{A}\phi \vee \mathcal{C}\phi}$$

where $\mathcal{A}$ and $\mathcal{C}$ are disjunctions of formulae, $\mathcal{B}$ and $\mathcal{D}$ are atomic and ϕ is the most general unifier of $\mathcal{B}$ and $\mathcal{D}$.

Resolution requires the goal to be transformed into Skolem form [4] and being changed to conjunctive normal form[5]. Resolution is complete, so if a proof exists one can be

[4]Eliminating the existential quantifiers.

[5]Conjunctive normal form consists of atomic formulae(A, $\neg A$) grouped together using logical and($\wedge$) into clauses which are joined by logical or($\vee$).

found by using only the resolution rule. However if the conjecture is invalid then resolution may not terminate.

The resolution rule is similar to modes ponens but the use of unification makes is possible to consider all instantiations of the rule at once. A unifier of two formulae is a substitution which when applied to the two terms renders them identical. The most general unifier is the unifier which will substitute only those terms in each term which it is necessary to make the two terms identical. In first order logic if two terms have a unifier then they will have a most general unifier [Rob65]. Unification is decidable and the most general unifier found will be unique. Even with unification automatically applying the resolution rule gives rise to a huge search space [Rob65].

Many methods have been suggested for reducing or restricting the search space for resolution theorem proving. One particular problem is that of generating duplicate or redundant clauses. Duplication of clauses is described in [LP92] which describes a hyper-linking strategy for eliminating such duplication. Techniques for detecting and restricting the generation of redundant clauses, ones that simply are not used in the course of the proof, are described in [Ben92] and [SA92]. Plotkin [Plo72] suggested performing resolution within an equational theory by incorporating equality into the unification process.

When equality is introduced into the calculus it may be handled like any other predicate, however this is inefficient as the appendix of [RW69] demonstrates. Instead an extra proof rule, paramodulation [RW69], may be introduced which will replace an expression by an equivalent one. Closely linked is the concept of demodulation [WRCS67] which reduces the number of retained clauses and increases the efficiency of the proof search.

2.3.2 Equational Methods

A set of rules may be used to rewrite term structure in a formula to be proven. Rewriting replaces an instance of the left hand side of the rewrite with the appropriate instance of the right hand side.

A rewrite step(denoted by $\rightarrow$) can be represented by the following rule:

$$\frac{(\mathcal{P}(\mathcal{A}'))\ (\mathcal{A} \rightarrow \mathcal{B})}{\mathcal{P}(\mathcal{B}\phi)}$$

ϕ is the most general unifier such that $\mathcal{A}\phi$ equals $\mathcal{A}'\phi$. $\mathcal{P}(\mathcal{A}')$ represents some expression of which $\mathcal{A}'$ is a subexpression.

A formula is rewritten and reduced to some unique form; if this form is an axiom then a proof has been found. However, given that there is a proof, ensuring that the form reached is an instance of an axiom is difficult.

An important subset of this type of reasoning is reasoning about equations. Conjectures are equations and so contain '='. A proof can be obtained by reducing the equation to an instance of '$A = A$', the reflexivity axiom. Equational methods work by taking the equation to be proven and transforming its subterms until it is proven or cannot be transformed any more. If both sides of an equation reduce to the same form then they have been proven to be equal [Bun83].

To ensure success, certain properties of the rewrite rules must be observed. The set of rewrite rules must be terminating and confluent. To be terminating there must be no sequence of rules which could potentially continue to be applied forever. Confluence says that if two different rules are applied to the same formula then there are sequences of rewrites which take the two results and produce the same result. In effect, the set of rewrite rules should produce a *unique* result for equal formulae. If a set of rules is both terminating and confluent it is said to be complete. The presence of commutativity and other similar laws preclude termination since repeated application of the law may result in a result identical to the original expression. A continual loop of rewrites is therefore possible, for instance by swapping two arguments over and back repeatedly, and the set of rules is not terminating.

To overcome the problems of termination and confluence, an attempt can be made to prove these properties about a set of rules or add to them to obtain such a set. Knuth and Bendix [KB70] described a completion algorithm which takes a set of equations and returns an equivalent complete set of equations. The problem of nonterminating equations like associativity and commutativity may be overcome if we merge these properties into the unification process [Sti81].

2.3.3 Proof Tactics

Both equational reasoning and resolution are *uniform proof procedures.* Uniform proof procedures work by using a single proof tactic and repeatedly applying it until a proof has been found; in resolution, the resolution rule and in equational reasoning, the rewriting rule. This use of only a single tactic without any way of intelligently traversing huge search spaces makes it difficult to prove complicated theorems in an acceptable time.

In contrast, tactic based theorem provers [GMW79a, BvHHS90a] use many different tactics to manipulate the goal. It might seem that a system using many tactics would produce a larger search space than a system with just one. However, the tactics are more constrained than the general purpose methods involved in uniform proof procedures. In proof search there are many repeated patterns of the application of proof tactics. Because of this, the rules in tactic based theorem provers, such as LCF [GMW79a], may be joined in various ways to form a new tactic to help guide the search for a proof

[Pau83]. The resulting tactic may need to be altered for different proofs and types of proof to enable the theorem prover to find a proof quickly.

Reduction is one strategy for ensuring termination of the proof search. By defining a measure on the goal, which is discrete[6] and has a lower bound[7] (usually zero), and restricting the rules which may be applied to the ones in which the measure is strictly decreased termination is guaranteed. Reduction does not ensure that a proof will be found if one exists. The restriction of measure decreasing rules may lead to dead ends where the current proof attempt has failed. In this event the prover must backtrack over the proof tree and select differently at points in the proof where more than one rule was applicable.

2.3.4 Proof Planning

Proof planning [Bun88a] is an extension to tactic based theorem proving which constrains the search space of a proof attempt by using meta-level information about the proof. This meta-level information provides heuristic search control information which may be used to guide a proof attempt. The proof is planned at the meta-level and then that plan is translated into a low-level tactic which is executed by the tactic theorem prover to obtain the object-level proof. The plan does not guarantee a proof because the resulting tactic when applied may not be able to construct a proof of the conjecture.

Methods

Proof planning proceeds through the application of *methods* which when combined make up the proof plan. The methods are arranged hierarchically with a high-level method calling on lower-level methods, or submethods, to perform steps as part of its overall function. Knowledge of the structure of proofs can be used to inform the order

[6] So that rules cannot decrease it by an ever decreasing amount.
[7] so that it cannot decrease forever.

in which the methods are tried. The order of application of methods can be controlled by the use of *methodicals*. Methodicals can direct the order of application of individual methods. For instance methodicals can encode which methods are chosen in preference to others or ensuring that one method is always tried immediately after the successful application of another.

Each method contains heuristic information in the form of *preconditions* about when it should be applied. Preconditions indicate when a method is applicable and encode heuristics for guiding the proof search. These preconditions help to guide the search for a proof and help prevent the construction of tactics which do not produce a proof. In λ*Clam* each method consists of a name, its input, preconditions, postconditions, its output and the corresponding tactic. The method's name identifies it and allows it to be used within the planner in conjunction with other methods and methodicals. The input to the method specifies the type of goal that the method applies to. It also can make the individual parts of the goal available for manipulation by the method. Preconditions specify the heuristic conditions that need to be true about the goal in order for the method to apply. Postconditions indicate conditions that need to be true after the goal has been manipulated. The output specifies the form of the goal after the method has applied and the tactic is the tactic used by the underlying proof system that corresponds to the method.

Figure 8: λ*Clam*'s ripple method

```
method1  ripple_inout_once
         ( ripGoal (Hyps >>> Goal) Skel E1 )
         ( ripple_step Rule inout Skel (Goal @@ E1) (After @@ E2) )
         true
         ( ripGoal (Hyps >>> After) Skel E2 )
         (rewr_plan Rule).
```

The ripple method shown in figure 8 demonstrates the different method slots. The method's name is `ripple_inout_once`, its input must be(unify with) a `ripGoal` which indicates that the method can only be used during rippling. This method has only one precondition, `ripple_step`, which determines if there is an applicable wave rule and applies it. There are no postconditions so the postcondition slot is occupied by `true`. The method outputs the rewritten goal. The final slot indicates the tactic and rewrite rule that correspond to the ripple method.

This method based approach has been applied to a number of areas, in particular induction. The *rippling* heuristic guides the step case of an induction proof towards a situation where the induction hypothesis can be used to rewrite the induction conclusion. The usefulness of proof planning and rippling in constraining proof search has been shown in theorem provers such as *Clam* [BvHHS90b]. The rippling heuristic is explored further in section 2.4.

Critics

By analysing a failure in a proof valuable information may be found pertaining to why the proof failed. Critics find and use this information to attempt to patch the proof and allow it to continue. Critics are very useful in situations where human interaction would otherwise be necessary.

Critics are tied to particular methods and it is the pattern of failure of the method which indicates when a critic may be applicable. The failure of certain method preconditions may suggest that extra term structure should be added to the goal, the failure of others may suggest that a different induction scheme should be used in the proof.

The *Clam* theorem prover has been extended to include a critics mechanism. Critics are explored in more detail in section 2.4.5.

Middle-Out Reasoning

In some proofs, steps must be made which introduce new term structure. These steps are called eureka steps and are necessarily very hard to automate since there may be a very large number, potentially an infinite number, of steps which could be made at a branching point. Steps such as introducing objects for existentially quantified variables or eliminating universally quantified variables are parametrised by a particular term and it is not always obvious what this term should be. Sometimes we can make an educated guess but it is still only a guess. The term structure we introduce in the eureka step will be needed later on in the proof and if at that point our guess turns out not to be useful then the proof will have to backtrack, speculate a different eureka step or fail.

In the proof planning framework, the meta-level information used to control the search for a proof allows a very powerful technique called *middle out reasoning* to be used. Middle out reasoning [Hes91] allows the selection of term structure to be introduced in a eureka step to be postponed until a later stage in the proof. By introducing a metavariable, to represent some arbitrary term structure in the proof, the decision of what exactly it is can be deferred. At the time that this bit of term structure is needed the instantiation of this metavariable may be constrained by the proof.

One example of middle out reasoning can be seen in the lemma speculation critic (section 2.4.5) for the *Clam* theorem prover [Ire96]. This critic is invoked when there is a missing lemma to be found but the form of this lemma is at least partially unknown. Meta-variables are substituted for the parts of the lemma which are not known, the lemma is applied and the plan proceeds. At some later point the metavariables are instantiated and a plan of the lemma is then attempted. If this side plan succeeds, the main plan continues as before.

There are problems with speculating using middle-out-reasoning. Search control structures must be in place to avoid nontermination. Because most logics do not allow metavariables, the proof has to be strictly controlled or verified after it is complete. Also unification is a problem; since the metavariable is higher-order a unification algorithm which can handle higher-order variables is required.

2.3.5 Higher Order Unification

Higher order unification [Hue75] is central to the process of middle out reasoning but is undecidable. A most general unifier is that a unifier which unifies two terms by instantiating only those subterms which must be made concrete for the terms to be the same. A most general unifier is desirable because it facilitates future unification by not constraining variables more than necessary. Unlike in first order unification the concept of most general unifier is difficult to define when applied to higher order unification. There may be an infinite number of unifiers and an infinite number of most general unifiers.

Most practical higher-order unification algorithms will terminate with either a most general unifier or a pre-unifier [Hue75] if the terms are unifiable. The pre-unifier algorithm is a unifiability algorithm which is a semidecidable problem, as opposed to an undecidable one. The algorithm should terminate if the terms are unifiable and should return a unifier for them.

The nonterminating properties of higher-order unification are caused by the presence of flexible-flexible(flex-flex) pairs [Hue75] which give rise to infinite sets of unifiers. The termination properties of the algorithm are improved if we do not attempt to unify these pairs of variables and simply regard them as constraints. The result of ignoring these flex-flex pairs is a pre-unifier which unifies the two terms but retains the flex-flex pairs.

The rules of this algorithm are *delete*, *decompose*, *eliminate*, *imitate* and *project* [Hue75]. *Delete* removes an identity($x = x$) from the set. *Decompose* allows the unification of two rigid terms which have the same function by unifying the arguments of the two terms. *Eliminate* allows the unification of symbols to be propagated through the set. *Imitate* and *project* are described below.

There are two main ways to unify two terms using the *imitation* and *projection* rules. *Imitation* is where a function symbol is made to take the form of a rigid term whereas *projection* is where an argument becomes the rigid term.

If we take the unification of the term $F(Y)$ with $c(X) + c'(X)$ by using *imitation* we obtain instantiations for F and Y:

$$\begin{array}{rcl} F(Y) & = & c(X) + c'(X) \\ & \Downarrow & \\ F & = & \lambda Z.c(Z) + c'(Z) \\ Y & = & X \end{array}$$

The function F *imitate*s the rigid term$(c(X) + c'(X))$ and so becomes the function $\lambda Z.c(Z) + c'(Z)$ and F's argument Y becomes X the argument to $\lambda Z.c(Z) + c'(Z)$.

Using the same example we obtain two different instantiations for F and Y:

$$\begin{array}{rcl} F(Y) & = & c(X) + c'(X) \\ & \Downarrow & \\ F & = & \lambda Z.Z \\ Y & = & c(X) + c'(X) \end{array}$$

The argument Y takes the form of the rigid term $(c(X) + c'(X))$. Y becomes the the rigid term and F becomes the identity function $\lambda Z.Z$.

2.3.6 Inductive Theorem Proving

Often in mathematics, and especially when considering computer programs, we want to reason about repetition. Repetition can arise from recursively defined functions and datatypes, iteration in programs or other sources. This repetition can be unbounded or dependent on unknown factors such as the data given to a program or other user input.

To reason about possibly infinite repeating structures it is usually essential to use mathematical induction. When induction is added to a logical theory automation becomes difficult [Bun99]. Logical theories which include induction are usually incomplete and proofs may require generalisation of conjectures or the introduction of lemmas. These problems can introduce potentially infinite branching points to the search space. At any stage one of an infinite number of induction rules could be constructed and applied, any one of an infinite number of lemmas could be introduced or a conjecture could be generalised in an infinite number of ways. When dealing with induction these steps can be essential so in inductive theorem provers special search control techniques need to be used to constrain their use.

There are two main approaches to automating mathematical induction, *explicit* and *implicit*. Explicit induction involves the use of explicit induction rules added to the logic [Wal94]. Because there are a potentially infinite number of induction rules that can be applied there is usually a method for dynamically constructing and applying induction rules [Wal93]. Walther's estimation calculus for proving the termination of functional programs is also a powerful tool for ensuring the well foundedness of induction rules. A *well founded* induction rule is one for which there is a well founded order in which the induction hypothesis is less than the induction conclusion. This helps in showing the termination of the induction step case when the proof is only allowed to proceed by means of strictly measure reducing steps. Gow [GBG99] extends the estimation calculus to features commonly found in induction rules. Implicit induction involves the introduction of inductive results to the theory to substitute for the use of explicit induction rules and then using a Knuth-Bendix completion algorithm (section 2.3.2) to verify that the extended rule set is terminating and confluent [KNZ91].

Theorem provers which use induction include Nqthm [BM88], PVS [ORS92], EVES [CKM+92], INKA [BHHW86], Oyster/*Clam* [BvHHS90a] and λ*Clam* [RSG98], which use explicit induction, SPIKE [BKR92], which uses implicit induction, and RRL [KZ95], which uses both explicit and implicit induction.

We discuss λ*Clam* and the heuristics it uses to guide induction in greater detail in section 2.4.

2.3.7 Theorem Proving Summary

Theorem proving has been around for a long time. Various approaches have been taken of which all require extensive, if not infinite, search to prove complex conjectures. Theorem proving is hard but search control helps automated theorem provers to find proofs more quickly and easily.

Proof planning is a heuristic search control framework which uses knowledge of common patterns in proof to guide the search for a proof. Through the use of higher-order logic and middle-out reasoning, theorem provers are able to prove more difficult examples automatically. Proof planning has been successfully applied to the problems involved with inductive theorem proving in theorem provers such as *Clam* and λ*Clam*.

In the next section we will discuss some of the heuristics and techniques used within the proof planning framework to perform inductive proofs.

2.4 Rippling

The proof planning approach has been applied to a number of areas, in particular induction. The usefulness of the rippling heuristic in constraining proof search in inductive proofs has been shown in theorem provers such as *Clam* [BvHHS90b].

The rippling heuristic works by annotating the induction conclusion with the differences between the induction conclusion and induction hypothesis. The differences between the hypothesis and conclusion are called wave-fronts and the similarities are called the skeleton.

For example the proof of the equivalence of *squares* and *map square*.

$$squares\ l = map\ \lambda x.(x * x)\ l$$

The differences between the induction conclusion and the induction hypothesis are the wave-fronts and are annotated in grey($\boxed{h ::}$). The similarities are unannotated and remain white. A similarity inside a wave-front is called the wave-hole($\boxed{t}$). The following goal is from the induction step case.

$$squares\ t = map\ square\ t \vdash$$
$$squares\ \boxed{h :: \boxed{t}}^{\uparrow} \quad = \quad map\ square\ \boxed{h :: \boxed{t}}^{\uparrow}$$

Wave rules are symbolic evaluation rules which have been annotated so that the wave-front is moved, or *rippled*, according to guidelines which preserve the entire induction hypothesis within the induction conclusion and ensure that each step is measure reducing[8]. These guidelines ensure that the rippling process terminates [BW96]. Example wave rules which apply to the above example are:

$$squares\ \boxed{h :: \boxed{t}}^{\uparrow} \quad \Rightarrow \quad \boxed{(square\ h) :: \boxed{(squares\ t)}}^{\uparrow}$$
$$map\ f\ \boxed{h :: \boxed{t}}^{\uparrow} \quad \Rightarrow \quad \boxed{(f\ h) :: \boxed{(map\ f\ t)}}^{\uparrow}$$

[8]Information on the the measure used in rippling can be found in [BW96]

Wave rules are then applied to the annotated goal in order to isolate the skeleton so that the conclusion can be rewritten using the hypothesis. [BSvH+93] describes several types of rippling.

2.4.1 Rippling Out

Rippling out or longitudinal rippling moves the wave fronts outwards so the induction conclusion will eventually be wholly contained in a wave-hole. At this point the induction hypothesis can be used to rewrite the induction conclusion by replacing the wave-hole with *true*(*strong fertilisation*).

In the *map squares* example we see both sides of the equation being first separately rippled out. The entire equation is then rippled joining the two sides of the equation. Finally the induction hypothesis is contained entirely within the wave hole and can be replaced with *true*.

We use the following wave rules

$$squares\ \boxed{H :: \underline{T}}^{\uparrow} \Rightarrow \boxed{(square\ H) :: \underline{(squares\ T)}}^{\uparrow} \qquad (7)$$

$$map\ F\ \boxed{H :: \underline{T}}^{\uparrow} \Rightarrow \boxed{(F\ H) :: \underline{(map\ F\ T)}}^{\uparrow} \qquad (8)$$

$$\boxed{(F\ A\ \underline{B})}^{\uparrow} = \boxed{(F\ C\ \underline{D})}^{\uparrow} \Rightarrow \boxed{(A = C) \land \underline{(B = D)}}^{\uparrow} \qquad (9)$$

to rewrite the conjecture:

$$
\begin{array}{rcll}
\multicolumn{3}{l}{squares\ t = map\ square\ t \vdash} & \text{(ih)} \\
squares\ \boxed{h :: \underline{t}}^{\uparrow} & = & map\ square\ \boxed{h :: \underline{t}}^{\uparrow} & 7 \\
\boxed{(square\ h) :: \underline{(squares\ t)}}^{\uparrow} & = & map\ square\ \boxed{h :: \underline{t}}^{\uparrow} & 8 \\
\boxed{(square\ h) :: \underline{(squares\ t)}}^{\uparrow} & = & \boxed{(square\ h) :: \underline{(map\ square\ t)}}^{\uparrow} & 9 \\
\multicolumn{3}{c}{\boxed{(square\ h) = (square\ h) \land \underline{(squares\ t) = (map\ square\ t)}}^{\uparrow}} & \text{ih} \\
\multicolumn{3}{c}{(square\ h) = (square\ h) \land true} &
\end{array}
$$

2.4.2 Rippling Sideways

Rippling sideways, or transverse rippling, is useful where there is a universally quantified variable in the goal. These variables are called *sinks*. Because sinks are universally quantified they can be used to absorb unwanted structure in the induction conclusion. Rippling sideways moves the wave-front around a sink. The wave-front is then rippled in(see section 2.4.4) towards the sink so that it can be absorbed into it.

Both rev and $foldl$ use an accumulating parameter to build up a result value. Such accumulators often produce situations where transverse rippling must be used. The wave front on both sides of the equation can be rippled into the accumulating parameter, which is a sink, allowing for the proof to proceed. For example consider:

$$\forall w.(rev\ w\ t = foldl\ \lambda x.\lambda y.(x :: y)\ w\ t) \vdash$$
$$\forall w.(rev\ \lfloor w \rfloor\ \boxed{h :: \boxed{t}}^{\uparrow} \quad = \quad foldl\ \lambda x.\lambda y.(x :: y)\ \lfloor w \rfloor\ \boxed{h :: \boxed{t}}^{\uparrow})$$

The accumulator w is universally quantified in the induction hypothesis so it is annotated as a sink. Rippling sideways is performed to move the wave-front from around the t to around the sink w. We ripple with the following wave rules:

$$rev\ \lfloor W \rfloor\ \boxed{H :: \boxed{T}}^{\uparrow} \Rightarrow rev\ \boxed{H :: \boxed{\lfloor W \rfloor}}^{\downarrow}\ T \tag{10}$$
$$foldl\ F\ \lfloor W \rfloor\ \boxed{H :: \boxed{T}}^{\uparrow} \Rightarrow foldl\ F\ \boxed{F\ H\ \boxed{\lfloor W \rfloor}}^{\downarrow}\ T \tag{11}$$

Rippling produces:

$$\forall w.(rev\ w\ t = foldl\ \lambda x.\lambda y.(x :: y)\ w\ t) \vdash$$
$$\forall w.(rev\ \boxed{h :: \boxed{\lfloor w \rfloor}}^{\downarrow}\ t \quad = \quad foldl\ \lambda x.\lambda y.(y :: x)\ \boxed{h :: \boxed{\lfloor w \rfloor}}^{\downarrow}\ t)$$

When the wave front is immediately around the sink it has been fully rippled and the wave front is said to have been absorbed into the sink. The previous line could also be written as:

$$\forall w.(rev \left\lfloor \boxed{h :: \boxed{w}} \right\rfloor t{=}foldl\ \lambda x.\lambda y.(y :: x) \left\lfloor \boxed{h :: \boxed{w}} \right\rfloor t)$$

to reflect this change in the status of the wavefront.

With the differences all rippled into sinks the induction hypothesis can apply because it can unify the quantified variable with the structure within the sink. Sinks of the same variable must contain syntactically identical structure because the variables must both be instantiated to the same thing.

2.4.3 Hybrid Rippling

In some cases a ripple may be hybrid, rippling structure both outwards and sideways.

In the *isums* example:

$$isums\ w\ l = scan\ \lambda x.\lambda y.(x + y)\ l$$

after rippling there are differences both around the sink variable and outside the skeleton. The wave front is split in two and one part rippled sideways and one rippled out. This is called hybrid rippling because both transverse and longitudinal rippling occur in the same step.

We use hybrid rippling with the following wave rules:

$$isums\ \lfloor W \rfloor\ \boxed{H :: \boxed{T}}^{\uparrow} \Rightarrow \boxed{H + \boxed{(isums\ \boxed{(H + \boxed{\lfloor W \rfloor})}^{\downarrow}\ T)}}^{\uparrow} \tag{12}$$

$$scan\ F\ W\ \boxed{H :: \boxed{T}}^{\uparrow} \Rightarrow \boxed{F\ W\ \boxed{(scan\ F\ \boxed{(H + \boxed{\lfloor W \rfloor})}^{\downarrow}\ T)}}^{\uparrow} \tag{13}$$

$$
\begin{array}{lr}
\forall w.(isums\ w\ t = scan\ \lambda x.\lambda y.(x+y)\ w\ t) \vdash & \text{(ih)} \\
\quad \forall w.(isums\ \lfloor w \rfloor\ \boxed{h :: t}^{\uparrow} = scan\ \lambda x.\lambda y.(x+y)\ \lfloor w \rfloor\ \boxed{h :: t}^{\uparrow}) & 12 \\
\forall w.(\ \boxed{(h+w) :: (isums\ \boxed{(h+\lfloor w \rfloor)}^{\downarrow}\ t)}^{\uparrow} = scan\ \lambda x.\lambda y.(x+y)\ \lfloor w \rfloor\ \boxed{h :: t}^{\uparrow}) & 13 \\
\forall w.(\ \boxed{(h+w) :: (isums\ \boxed{(h+\lfloor w \rfloor)}^{\downarrow}\ t)}^{\uparrow} = & \\
\qquad \boxed{(h+w) :: (scan\ \lambda x.\lambda y.(x+y)\ \boxed{(h+\lfloor w \rfloor)}^{\downarrow}\ t)}^{\uparrow}\ ) & 9 \\
\forall w.(\ \boxed{(h+w) = (h+w) \wedge (isums\ \lfloor (h+w) \rfloor\ t) = (scan\ \lambda x.\lambda y.(x+y)\ \lfloor (h+w) \rfloor\ t)}\ ) & \text{ih} \\
\quad \forall w.((h+w) = (h+w) \wedge true) &
\end{array}
$$

2.4.4 Rippling In

Rippling in is described in [BSvH+93]. This occurs when one side of an equation is fully rippled but the other side is blocked. The induction hypothesis can be used to rewrite the fully rippled side and then the result may be rippled inwards to match the blocked structure on the other side. This technique was later replaced by a critic which would calculate the lemma needed to unblock rippling(section2.4.5).

2.4.5 Critics

An extension to the proof planning framework is the critics mechanism proposed in [Ire92]. The first experiments with critics as implemented in *Clam* were reported in [Ire96].

Critics use information gained from a failed proof attempt to patch the proof so that it avoids or proceeds past the point of failure. Critics are very useful in the discovery of eureka steps. The *Clam* theorem prover has been extended to include critics for lemma calculation, lemma speculation, generalisation and induction revision and sink equalization.

Generalisation

Where rippling is blocked but a transverse wave rule would apply if a constant in the goal were a variable, then generalisation might be tried. In a transverse wave rule a wave-front is moved sideways from around the induction variable to around a sink so that it may be rippled into it. However, if there is no sink in this position but only a constant then the ripple is blocked.

To counter this the conjecture is generalised. We abstract away from the constant and replace it with a universally quantified variable. The proof can then proceed by rippling the wave-front sideways and into the newly introduced sink.

Induction Revision

Induction revision occurs when rippling is blocked but there is a partially applicable wave-rule. If this wave rule requires extra wave-fronts to be present to apply then this suggests an induction revision. The extra wave fronts in the wave rule might be added by changing the induction scheme to a two step induction, for example. This will add extra structure to the step case goal and possibly allow the partially applicable wave rule to apply at the point at which the proof became blocked.

Lemma Calculation

Lemma calculation is applicable where one side of an equation is fully rippled and the other is blocked. The critic will calculate a lemma corresponding to a possible wave rule applicable to the blocked side of the equation. This lemma is proven and applied to the blocked ripple goal to unblock the rippling process.

Lemma Speculation

Lemma speculation is used when neither side of an equation is fully rippled. In this case there is no way to determine what the lemma should be at the point of failure so we delay the choice of lemma until we have more information. To do this we use middle out reasoning and speculate a schematic lemma, that is one which contains metavariables in place of the structure we do not know. We can use the rippling process to constrain the schematic wave-rule we produce because we know the skeleton structure and we know about the possible forms of rippling that might be applied.

One of the wave rules is chosen to rewrite the goal. The wave rules introduce metavariables into the goal and further rippling instantiates them. Once the metavariables are instantiated and the schematic lemma has become concrete it must then be proven.

2.4.6 *Clam* and λ*Clam*

Proof planning is supported by the *Clam* and λ*Clam* theorem provers which were chosen at the start of the project as possible platforms to enable transformation in the compiler.

Clam is built on top of *Oyster* which is a reimplementation of the NuPRL program development system [CAB+86] in Prolog. λ*Clam* is a higher-order version of *Clam* written in λProlog.

The original idea for rippling was presented in [Bun88b] as automatic search control for the NuPRL program synthesis system. NuPRL is based on Martin-Löf Intuitionistic Type theory [ML79]. This provides the constructive logic framework which is necessary for the program synthesis(see section 2.2.1) that NuPRL was designed for. The proof plans are represented by LCF style tactics.

Oyster is reimplementation of NuPRL implemented in the logic programming language Prolog. *Oyster* is an interactive proof editor so the search for a proof must be guided externally by interaction with a human or another program. *Clam* is built on top of *Oyster* and provides tactics which guide the proof search. These tactics are described by methods which specify their preconditions and postconditions. The *Clam* methods encode the heuristics necessary for the automated search for a proof plan. This meta-level proof plan, once created, may be translated into an object-level proof by using the corresponding tactics to guide the *Oyster* proof system.

Methods were implemented in *Clam* for using rippling out to guide inductive proof [BvHS88]. These were later updated to include methods for multi-wave rippling, rippling in, rippling sideways, destructor style rippling, and existential rippling [BSvH+91]. Synthesis and middle-out reasoning capabilities were then added to *Clam*. A critics mechanism was added to *Clam* to make productive use of the occasional failure of rippling [IB95]. The lemma calculation critic effectively replaces the rippling in mechanism described in the earlier paper.

Many of the tasks *Clam* was used for have been naturally higher-order in nature. Synthesis and verification techniques such as critics rely on the use of middle-out reasoning and higher-order unification. In *Clam* these techniques were added on and are not naturally integrated into the theorem proving framework. λ*Clam* is a higher-order version of *Clam* written in λProlog [MN88] the higher-order version of Prolog.

λProlog is an extension of Prolog. In λProlog typed λ-terms are used instead of first order terms and implication and both universal and existential quantification are allowed in goals. The λ calculus used in λProlog is restricted to Church's simply typed λ-terms; this allows higher-order unification to be usable. λProlog is based on hereditary Harrop formulas instead of the Horn clauses of Prolog.

The *Clam* methods for rippling were transferred to λ*Clam*. The higher-order unification built in to the λ*Clam* system allows middle-out reasoning to be used much more naturally in a similar proof planning framework to that of *Clam* [RSG98].

Chapter 3

Verification

3.1 Introduction

As a central part of the parallel compiler project(see section 1.1) a theorem prover was to be used to prove new transformation rules correct. This adds some notion of correctness to the compiler and also provides a rich source of new inductive conjectures for testing the proof plan for induction existing in the *Clam* and λ*Clam* theorem provers. New rules would be provided by the user of the system and verified before being added to a library of transformation rules. These rules were then to be used to transform an input program to obtain the optimum parallel performance.

These new rules would typically perform the transformation of a user defined function into an expression involving HOFs. For example:

$$squares\ l \Rightarrow map\ \lambda x.(x * x)\ l$$

where *squares* is defined as:

$$\begin{aligned} squares\ [\,] &= [\,] \\ squares\ (h :: t) &= (h * h) :: (squares\ t) \end{aligned}$$

The HOF in the resulting expression could then be used to produce parallelism. We use HOFs because introducing parallelism directly is very difficult without some strategy to identify where it might be effective. Finding HOFs effectively identifies the processing of data structures where processing each element does not depend on processing other elements. HOFs provide a good template for parallelism while still remaining close to the functional source. By using HOF transformations to introduce parallelism we are able to find sites of potential parallelism relatively easily and we also gain the power and flexibility of being able to selectively transform the program.

Rules which introduce HOFs would be the main class of transformations in the library but there would also be others which manipulate existing HOFs to produce an optimal arrangement. This class of rules can also be verified quite easily by *Clam*. Examples are the $map/foldr$ equivalence and the map distributivity property. The $map/foldr$ equivalence can be used to transform an expression involving the HOF map into one involving $foldr$:

$$map\ \lambda x.(f\ x)\ l = foldr\ \lambda y.\lambda z.((f\ y) :: z)\ []\ l$$

The distributivity property of map can be used to transform two nested maps into a single map:

$$map\ g\ (map\ f\ l) = map\ \lambda x.(g\ (f\ x))\ l$$

The fertilisation step of a ripple proof can become blocked if wave fronts are rippled into sinks and the structure in the sinks is not identical. Bundy[1] suggested the use of a critic to solve this problem. We implemented this critic extending the capabilities of *Clam*.

The problem of verification of transformation rules seemed reasonably simple because existing proof tools such as *Clam* could already prove conjectures of the type that were likely to arise. However, the user would have to add new transformations for new

[1]Personal communication.

functions which detracts from the principal that they should not have to think about component characteristics while programming. Automatic synthesis of transformation rules would allow the user to completely stand back from the transformation done by the compiler. From a research perspective, the idea of automatically synthesising new correct transformation rules is very interesting. With the switch in emphasis to transformation rule synthesis, using the higher-order theorem prover λ*Clam* (section 2.4.6) became attractive. λ*Clam*'s use of metavariables, higher-order unification and polymorphic typing made it a good choice for performing synthesis.

We performed verification in *Clam* because it was an established system which has been well tested and has an implemented critics mechanism. For verification problems especially ones involving induction *Clam* is a good theorem prover. When we switched our emphasis to synthesis the λ*Clam* theorem prover seemed like the natural choice. Although *Clam* has been used for synthesis the emphasis of our work on HOFs suggested that a theorem prover more suited to handling higher-order constructs might be needed. λ*Clam* is written in λProlog which uses higher-order unification while *Clam* is written in Prolog which uses first-order unification this gives λ*Clam* direct access to unification algorithm needed for synthesis. Also λProlog is typed where Prolog is not this allows the type system to constrain the proof search. This also allows the use of types in restricting the choice of HOF. Although we didn't use this typing to directly influence the HOF choice, the presence of the type system may help to narrow the number of candidate HOFs we have to choose from.

Though the main emphasis was changed to synthesis, the verification work informed the development of the synthesis plan described in chapter 4. The $map/foldr$ equivalence provided insight into the process of producing multiple transformation rules from a single problem. The *map* distributivity property appeared in the *mergesort* example(section 6.3.5) and helped to produce an important refinement of the synthesis plan. Also the concept of equating sinks used in the fertilisation critic was used in the synthesis plan to allow the discovery of extra nested HOFs.

3.2 Verification of New Transformation Rules

The majority of rules that may be added involve the transformation of user defined functions into expressions involving HOFs. For example if there is a user defined function which squares every element of a list the user may wish to express it in terms of *map* or *foldr* in order to have the function evaluated using multiple processors. This function may be implemented in SML as follows:

```
fun squares [] = []|
    squares (h::t) = (h*h)::(squares t)
```

In the bare SML syntax this is represented by:

```
val rec squares = fn arg1 =>
  (fn nil => nil |
      ::{#1 = h, #2 = t} =>
      ::{#1 = *{#1 = h,#2 = h}, #2 = squares t}
  ) arg1
```

Translated into rewrite rules for *Clam* or λ*Clam* the `squares` definitions will look like:

$$squares\ [] = [] \tag{14}$$

$$squares\ (h :: t) = (times\ h\ h) :: (squares\ t) \tag{15}$$

Information on the automatic transformation of bare SML syntax into λ*Clam* rewrite rules can be found in section 7.2.3.

The *squares* function can be transformed into an expression involving *map*. The definition of the HOF *map* in λ*Clam* is as follows:

$$map\ f\ [] = [] \tag{16}$$

$$map\ f\ (h :: t) \quad = \quad (f\ h) :: (map\ f\ t) \tag{17}$$

The transformation rule for replacing *squares* with the equivalent *map* expression is:

$$\forall l.(squares\ l \quad \Rightarrow \quad map\ \lambda x.(x * x)\ l)$$

The proof of this conjecture is relatively simple in either *Clam* or λ*Clam*. Induction:

Base Case

$$\begin{array}{rcll}
squares\ [] & = & map\ \lambda x.(x * x)\ [] & 14 \\
{[]} & = & map\ \lambda x.(x * x)\ [] & 16 \\
{[]} & = & [] & \text{eq} \\
 & true & &
\end{array}$$

Step Case

$$\begin{array}{rcll}
\multicolumn{3}{l}{squares\ t = map\ \lambda x.(x * x)\ t \vdash} & \text{(ih)} \\
squares\ \boxed{(h :: \boxed{t})}^{\uparrow} & = & map\ \lambda x.(x * x)\ \boxed{(h :: \boxed{t})}^{\uparrow} & 15 \\
\boxed{(h * h) :: \boxed{(squares\ t)}}^{\uparrow} & = & map\ \lambda x.(x * x)\ \boxed{(h :: \boxed{t})}^{\uparrow} & 17 \\
\boxed{(h * h) :: \boxed{(map\ \lambda x.(x * x)\ t)}}^{\uparrow} & = & \boxed{(h * h) :: \boxed{(map\ \lambda x.(x * x)\ t)}}^{\uparrow} & \text{ih} \\
(h * h) :: (map\ \lambda x.(x * x)\ t) & = & (h * h) :: (map\ \lambda x.(x * x)\ t) & \text{eq} \\
 & true & &
\end{array}$$

It is possible that more complex transformation rules may be input by the user and *Clam* may or may not be able to prove them. However, the main advantage of using these transformation rules is for introducing HOFs and parallelisation into a program. The fact that the *squares* rule, and others like it, are so simple to verify gives us some confidence that many of the important transformation rules can be verified without too much effort.

Although the user does not have to worry about the specific details of the parallel implementation of their program, they are forced to add new transformation rules for functions that they write. Because of this, programming with awareness of parallelisation has been replaced by programming with awareness of HOFs. To eliminate the need for the user to add new transformation rules and allow them to stand back from parallelisation/HOF issues, the automatic synthesis of transformation rules is important.

3.3 Alternate HOFs and Distributivity Transformations

Some useful transformations pertain not to the expression of individual functions using HOFs but to the nature and arrangement of the HOFs themselves. These transformations may be used to achieve the optimal arrangement of HOFs in the program and so should be available for use. These transformations also need to be verified before being added to the library of transformation rules.

3.3.1 $map/foldr$ Equivalence

An expression involving the HOF map can always be represented by an equivalent $foldr$ expression by using the :: function in the function argument of the $foldr$ [BW88, p.72]:

$$\forall f.(map\ \lambda x.(f\ x)\ l \Rightarrow foldr\ \lambda y.\lambda z.((f\ y) :: z)\ []\ l) \tag{18}$$

The definition of $foldr$ is as follows:

$$foldr\ f\ b\ [] \quad = \quad b \tag{19}$$

$$foldr\ f\ b\ (h :: t) \quad = \quad f\ h\ (foldr\ f\ b\ t) \tag{20}$$

Since map and $foldr$ are implemented differently in the parallel system, using the $map/foldr$ equivalence as a transformation rule allows the behaviour of the program to be altered depending on the whether we use a map or a $foldr$. map is implemented by a farm whereas $foldr$ is implemented by a divide and conquer strategy. The transformer component of the compiler (section 1.1) will use performance information to determine whether or not to apply this transformation. However, the use of a divide and conquer strategy depends on the associativity of the function argument and the :: used in the foldr function argument is not associative. To overcome this :: may be translated into ++ which is associative by the following rule:

$$h :: t \Rightarrow [h]{+\!\!+}t$$

This translation will allow the map expression to be implemented using a divide and conquer strategy and also allows it to be expressed using $foldl$ instead of $foldr$. Performance information can be used to decide which of the implementations is best in any given situation.

The proof of the $map/foldr$ equivalence rule proceeds as follows:

Induction:

Base Case

$$\begin{array}{rcll}
\forall f.(map\ \lambda x.(f\ x)\ [] & = & foldr\ \lambda y.\lambda z.((f\ y) :: z)\ []\ []) & 16 \\
\forall f.([] & = & foldr\ \lambda y.\lambda z.((f\ y) :: z)\ []\ []) & 19 \\
\forall f.([] & = & []) & \text{eq}
\end{array}$$

$$true$$

Step Case

$$
\begin{array}{rcll}
\multicolumn{3}{l}{\forall f.(map\ \lambda x.(f\ x)\ t = foldr\ \lambda y.\lambda z.((f\ y) :: z)\ []\ t) \vdash} & \text{(ih)} \\
\forall f.(map\ \lambda x.(f\ x)\ \boxed{(h :: \boxed{t})}^{\uparrow} & = & foldr\ \lambda y.\lambda z.((f\ y) :: z)\ []\ \boxed{(h :: \boxed{t})}^{\uparrow}\ ) & 17 \\
\forall f.(\ \boxed{(f\ h) :: \boxed{(map\ \lambda x.(f\ x)\ t)}}^{\uparrow} & = & foldr\ \lambda y.\lambda z.((f\ y) :: z)\ []\ \boxed{(h :: \boxed{t})}^{\uparrow}\ ) & 20 \\
\forall f.(\ \boxed{(f\ h) :: \boxed{(map\ \lambda x.(f\ x)\ t)}}^{\uparrow} & = & \boxed{(f\ h) :: \boxed{(foldr\ \lambda y.\lambda z.((f\ y) :: z)\ []\ t)}}^{\uparrow}\ ) & \text{ih} \\
\forall f.((f\ h) :: (foldr\ \lambda y.\lambda z.((f\ y) :: z)\ []\ t) & = & (f\ h) :: (foldr\ \lambda y.\lambda z.((f\ y) :: z)\ []\ t)) & \text{eq} \\
 & true & &
\end{array}
$$

The knowledge of the $map/foldr$ equivalence helped inform the development of the synthesis plan(Chapter 4). In performing synthesis on functions that can be represented as a map expression there will always be the possibility of finding an extra expression that involves $foldr$ instead of map. To find both of these possibilities a choice point is present in the synthesis plan to allow the planner to choose either solution.

3.3.2 *map* Distributivity

Two composed maps can be combined into one using a distributivity property [BW88, p.62]. The map distributivity property is as follows:

$$\forall f.\forall g.(((map\ f)\ \circ\ (map\ g))\ l \quad = \quad map\ (f\ \circ\ g)\ l)$$

The $\circ$ represents compose, the function that takes two other functions and applies one to the result of the other:

$$(f\ \circ\ g)\ l \quad = \quad g\ (f\ l)$$

This distributivity property can be rewritten as:

$$\forall f.\forall g.(map\ g\ (map\ f\ l) = map\ \lambda x.(g\ (f\ x))\ l) \tag{21}$$

This distributivity property allows instances of composed maps to be implemented in two different ways. One using two maps and one using only one. Performance information can then be used to decide which of the two *map* arrangements to use in the implementation.

The proof of the map distributivity property is as follows:

Induction:

Base Case

$$\begin{aligned}
\forall f.\forall g.(map\ g\ (map\ f\ []) &= map\ \lambda x.(g\ (f\ x))\ []) && 16 \\
\forall f.\forall g.(map\ g\ [] &= map\ \lambda x.(g\ (f\ x))\ []) && 16 \\
\forall f.\forall g.([] &= map\ \lambda x.(g\ (f\ x))\ []) && 16 \\
\forall f.\forall g.([] &= []) && \text{eq}
\end{aligned}$$

$$true$$

Step Case

$$\begin{aligned}
&\forall f.\forall g.(map\ g\ (map\ f\ t) = map\ \lambda x.(g\ (f\ x))\ t) \vdash && \text{(ih)} \\
\forall f.\forall g.(map\ g\ (map\ f\ h :: t) &= map\ \lambda x.(g\ (f\ x))\ h :: t) && 17 \\
\forall f.\forall g.(map\ g\ (f\ h) :: (map\ f\ t) &= map\ \lambda x.(g\ (f\ x))\ h :: t) && 17 \\
\forall f.\forall g.((g\ (f\ h)) :: (map\ g\ (map\ f\ t)) &= map\ \lambda x.(g\ (f\ x))\ h :: t) && 17 \\
\forall f.\forall g.((g\ (f\ h)) :: (map\ g\ (map\ f\ t)) &= \\
&\quad (g\ (f\ h)) :: (map\ \lambda x.(g\ (f\ x))\ t)) && \text{ih} \\
\forall f.\forall g.((g\ (f\ h)) :: (map\ \lambda x.(g\ (f\ x))\ t) &= (g\ (f\ h)) :: (map\ \lambda x.(g\ (f\ x))\ t)) && \text{eq}
\end{aligned}$$

$$true$$

Mergesort

After the implementation of the first prototype (chapter 5), the *mergesort* example:

$$mergesort\ x = mergeall\ (sortmall\ x) \tag{22}$$

$$mergeall\ [] = [] \tag{23}$$

$$mergeall\ (h :: t) = merge\ h\ (mergeall\ t) \tag{24}$$

$$sortmall\ [] = [] \tag{25}$$

$$sortmall\ (h :: t) = (sortm\ h) :: (sortmall\ t) \tag{26}$$

$$sortm\ [] = [] \tag{27}$$

$$sortm\ (h :: t) = insertm\ h\ (sortm\ t) \tag{28}$$

produced unexpected results(section 6.3.5) by combining a map nested within a $foldr$ instead of treating the two functions separately. Knowledge of the map distributivity property allowed these to be identified as correct alternate results.

These unexpected results were produced because the planner took advantage of a property of map and $foldr$ similar to the map distributivity property:

$$\forall f.\forall g.(foldr\ f\ b\ (map\ g\ l) = foldr\ \lambda x.\lambda y.(f\ (g\ x)\ y)\ b\ l) \tag{29}$$

The $mergesort$ example highlighted the existence of this property allowing it to be verified as a new transformation rule.

Induction:

Base Case

$$\begin{array}{rcll}
\forall f.\forall g.(foldr\ g\ b\ (map\ f\ [\,]) & = & foldr\ \lambda x.\lambda y.(g\ (f\ x)\ y)\ b\ [\,]) & 16 \\
\forall f.\forall g.(foldr\ g\ b\ [\,] & = & foldr\ \lambda x.\lambda y.(g\ (f\ x)\ y)\ b\ [\,]) & 19 \\
\forall f.\forall g.(b & = & foldr\ \lambda x.\lambda y.(g\ (f\ x)\ y)\ b\ [\,]) & 19 \\
\forall f.\forall g.(b & = & b) & \text{eq} \\
& true & &
\end{array}$$

Step Case

$$
\begin{array}{rcll}
\multicolumn{3}{l}{\forall f.\forall g.(foldr\ g\ b\ (map\ f\ t) = foldr\ \lambda x.\lambda y.(g\ (f\ x)\ y)\ b\ t) \vdash} & \text{(ih)} \\
\forall f.\forall g.(foldr\ g\ b\ (map\ f\ \boxed{h :: \underline{t}}^{\uparrow}) & = & foldr\ \lambda x.\lambda y.(g\ (f\ x)\ y)\ b\ \boxed{h :: \underline{t}}^{\uparrow}) & 17 \\
\forall f.\forall g.(foldr\ g\ b\ \boxed{((f\ h) :: \underline{(map\ f\ t)})}^{\uparrow} & = & foldr\ \lambda x.\lambda y.(g\ (f\ x)\ y)\ b\ \boxed{h :: \underline{t}}^{\uparrow}) & 20 \\
\forall f.\forall g.(\boxed{g\ (f\ h)\ \underline{(foldr\ g\ b\ (map\ f\ t))}} & = & foldr\ \lambda x.\lambda y.(g\ (f\ x)\ y)\ b\ \boxed{h :: \underline{t}}^{\uparrow}) & 20 \\
\forall f.\forall g.(\boxed{g\ (f\ h)\ \underline{(foldr\ g\ b\ (map\ f\ t))}} & = & \boxed{g\ (f\ h)\ \underline{(foldr\ \lambda x.\lambda y.(g\ (f\ x)\ y)\ b\ t)}}^{\uparrow}) & \text{ih} \\
\forall f.\forall g.(\boxed{g\ (f\ h)\ \underline{(foldr\ \lambda x.\lambda y.(g\ (f\ x)\ y)\ b\ t)}} & = & & \\
 & & \boxed{g\ (f\ h)\ \underline{(foldr\ \lambda x.\lambda y.(g\ (f\ x)\ y)\ b\ t)}}^{\uparrow}) & \text{eq} \\
 & true & &
\end{array}
$$

This $map/fold$ distributivity property along with the map distributivity property informed the final refinement of the synthesis plan (section 4.7) and enabled it to find results both from two composed HOFs and from the equivalent single HOF with the composition in its function argument.

3.4 Fertilisation Critic

In the *Clam* theorem prover the rippling heuristic is used to guide the progress of inductive proof. Rippling directs the application of rewrite rules towards a situation where the induction hypothesis can be used to rewrite the induction conclusion. This rewriting of conclusion with hypothesis is called fertilisation.

There are situations where rippling makes progress towards fertilisation but then the proof attempt becomes blocked on the fertilisation step. Bundy[2] suggested using a critic to unblock the fertilisation step. Boulton[3] indicated the possible need for the use of larger context in the critic proposed by Bundy.

The fertilisation critic was implemented in *Clam*.v3.2 as Bundy originally suggested

[2] Personal communication.
[3] Personal communication.

without the use of context suggested by Boulton. The fertilise critic successfully unblocked the fertilisation step in the example proof attempt and should be applicable to other similar examples.

Armando [ASG97] developed a sink-equalisation method as part of his work on the synthesis of recursive functional programs. However, Armando's method only uses simple rewriting to unblock the fertilisation step. Armando's method is discussed in more detail in section 8.2.2.

3.4.1 The Problem

This critic specifically applies to rippling sideways(section 2.4.2). When a conjecture is rippled sideways, the wave-front is moved to around a universally quantified variable. The aim of rippling sideways is to instantiate this variable with the differences between the induction hypothesis and the induction conclusion and so allow fertilisation to take place. The fertilisation process becomes blocked if the wave structures rippled towards two instances of the same variable are not identical. In this case the variable cannot be instantiated with the wave structure because a single variable cannot be instantiated to two different things.

In some cases the sinks are structurally different but are can be shown to be equal. An identity check is not sufficient in these cases.

This problem is illustrated by the proof of the following conjecture($sumthm$):

$$\forall b.\forall l.(sum\ b\ l \quad = \quad foldl\ \forall x.\forall y.(x+y)\ b\ l) \tag{30}$$

The definitions of, sum, and $foldl$ are as follows:

$$sum\ b\ [] \quad = \quad b \tag{31}$$

$$sum\ b\ h :: t \quad = \quad sum\ (b+h)\ t \tag{32}$$

$$foldl\ f\ b\ [\,] \quad = \quad b \tag{33}$$

$$foldl\ f\ b\ h :: t \quad = \quad foldl\ (f\ h\ b)\ t \tag{34}$$

The proof proceeds by induction on x. The base case is trivial. The step case is as follows:

$$\begin{array}{rcll}
\multicolumn{3}{l}{sum\ b\ t = foldl\ \forall x.\forall y.(x+y)\ b\ t \vdash} & \text{(ih)} \\
sum\ \lfloor b \rfloor\ \boxed{h :: \underline{t}}^{\uparrow} & = & foldl\ \forall x.\forall y.(x+y)\ \lfloor b \rfloor\ \boxed{h :: \underline{t}}^{\uparrow} & 32 \\
sum\ \left\lfloor \boxed{(\underline{b}+h)}^{\downarrow} \right\rfloor\ t & = & foldl\ \forall x.\forall y.(x+y)\ \lfloor b \rfloor\ \boxed{h :: \underline{t}}^{\uparrow} & 34 \\
sum\ \left\lfloor \boxed{(\underline{b}+h)}^{\downarrow} \right\rfloor\ t & = & foldl\ \forall x.\forall y.(x+y)\ \left\lfloor \boxed{(h+\underline{b})}^{\downarrow} \right\rfloor\ t & \text{blocked}
\end{array}$$

The plan ripples until all the wave terms are contained within sinks. But fertilisation is blocked because the terms $(b+h)$ and $(h+b)$ which have been rippled into the z sinks are not identical. The sinks are equal However, and by rewriting one of the sinks with the commutativity lemma:

$$(x+y) \quad = \quad (y+x) \tag{35}$$

we can unblock the fertilisation.

However, in general, a lemma may not be available to the planner and may need to be proven before it can be used to unblock fertilisation.

3.4.2 Solution

The critics mechanism in *Clam*.v3.2 provides a possible solution to a blocked fertilisation step. By invoking a critic when the fertilise method fails, it is possible to find and prove the lemma needed to patch the plan and unblock the fertilisation step.

Fertilise Method

Methods in *Clam* are defined using the `method` predicate. The arguments to `method` are the method name, the input goal, pre conditions, post-conditions and output goal.

Methods are defined mostly by changing the pre-conditions which determine when a method is applicable.

The critics mechanism in *Clam*.v3.2 is invoked when the preconditions to a method fail in a specific pattern. So a critic is only invoked when certain preconditions to the method fail and others succeed. By analysing the failure of preconditions like this it can determine if a critic should be tried in the specific circumstances.

In order for the critic to detect the partial failure of the fertilise method, the fertilise method's preconditions needed to be changed. In the original fertilise method the only precondition was an induction hypothesis match(`induction_hyp_match`). The failure of this condition is not useful in detecting the particular case when the fertilisation critic should be invoked, so the preconditions needed to be changed.

To allow the critic to detect when it is applicable, an extra precondition was added to the fertilise method. The new precondition `all_waves_sunk` checks if all the wave-fronts in the goal reside within sinks. `all_waves_sunk` succeeds when the induction conclusion has been fully rippled and all the annotated wave terms have been rippled into sinks; this is when fertilisation should occur. Implicit in the existing precondition `induction_hyp_match` is the condition that the goal is fully rippled; in the rippling sideways case this means `all_waves_sunk`. Although `induction_hyp_match` subsumes `all_waves_sunk` the latter needs to be present to allow *Clam* to detect partial failure.

To have `all_waves_sunk` and `induction_hyp_match` in the preconditions seems superfluous because `induction_hyp_match` is sufficient. The alternative is to split `induction_hyp_match` into `all_waves_sunk` (which is necessary to detect partial failure) and another new predicate to check whether all the sinks are identical. However, this is likely to be much less efficient than `induction_hyp_match`. If at this point there is an induction hypothesis match then fertilisation can occur; if not then the fertilisation is blocked and the critic should be invoked.

Fertilise Critic

The fertilise critic itself determines the content of which sinks should be equal. In some cases there may be more than one pair. The critic then constructs a lemma corresponding to the equality of the sinks and plans the proof of this lemma. It rewrites one of the sinks with the lemma, using a special purpose rewrite method (`general_rewrite`), and in so doing allows fertilisation to take place.

First the critic extracts the sinks from the goal and groups them according to the sink variable. Next it eliminates the trivial sinks, in which the sink variable is equivalent to the entire contents of the sink itself[4]. It uses this information about the sink variables to determine the equalities which need to be demonstrated.

The critic constructs a list of lemmas corresponding to the calculated equalities and picks one of the lemmas to plan. It then verifies the lemma by either determining it to be a previously verified lemma or by constructing and verifying it from scratch. The critic then uses the verified lemma to unblock the goal.

Rewrite Method

The special purpose rewrite method (`general_rewrite`) uses a lemma to rewrite the content of a sink. This method is used in order to supply *Oyster* with some record of the rewrite step which is performed at the end of the critic to unblock fertilisation.

The method performs a straight rewrite in a ripple proof so has the power to disturb the rippling process. In the context of the new critic this is not the case because it applies the method in a very controlled manner. The method will be used to rewrite the contents of a sink and so will not interfere with rippling.

[4]This may not be wise since an equality may need to be constructed using one of these trivial sinks(see section 3.4.4).

3.4.3 Results

The example, *sumthm*, may seem to be an artificial since the reversal of x and y in the function argument of the $foldl$ would easily eliminate the problem. However, this type of problem arises often in theorem proving and the example serves to illustrate that the critic works.

Without the critic the proof will get to this stage and become blocked:

$$sum \ \lfloor (b+h) \rfloor \ h :: t \quad = \quad foldl \ \forall x . \forall y . (x+y) \ \lfloor (h+b) \rfloor \ t$$

Fertilisation is blocked because the contents of the sinks are not equivalent.

With the fertilisation critic in place, *Clam* will detect the blocked fertilisation step and calculate and plan the commutativity lemma:

$$(y+x) = (x+y)$$

It would then use it to rewrite one of the sinks in the goal. This unblocks the fertilisation step allowing the plan to complete.

3.4.4 Criticising The Critic

The addition of the extra precondition `all_waves_sunk` to the fertilise method excludes the case where the waves are rippled outwards. This means that this fertilise method can only be used for rippling sideways proofs. Another fertilise method was added with the extra precondition `all_waves_beached` to allow outward rippling. However, this still excludes cases involving hybrid rippling where some waves have been rippled out and some rippled sideways. The ideal is a generalised fertilise method which would deal with all cases.

At the moment the fertilise critic picks one lemma, plans it, uses it and continues. There may be cases where more than one lemma needs to be planned in order to unblock the goal but the fertilisation critic as it stands cannot handle these cases.

The use of context in showing the sinks equal has not been addressed in this implementation of the critic but is an interesting extension.

3.4.5 Summary

The fertilise critic is an implementation of Bundy's suggestion. It detects the failure of the fertilisation step due to two sinks not being identical and attempts to show them to be equal. It uses a pre-loaded lemma or plans the needed lemma and uses it as the basis of a rewrite step which allows the fertilisation step to take place.

The idea of equating sinks is used later on in the synthesis plan to enable the synthesis of expressions that have been rippled into sinks. In the fertilisation critic we equate the contents of two sinks, prove their equality and use it to rewrite one of the sinks. Here we use this idea slightly differently. In the synthesis plan the contents of both sinks in the goal are set to the value of some new variable. The equality of the sinks is set up and is used as the basis of a recursive call to the synthesis plan.

This treatment of sinks within the synthesis problem can inform the implementation of a replacement for the fertilisation critic. A method could be used to set up the sink equality subproblem within the ripple goal. Along with the use of substitution axioms this could be used to overcome all of the problems with the critic implementation mentioned above.

3.5 Conclusions

The verification of typical HOF introducing transformation rules was relatively straightforward. However, since each of these rules would need to be entered by the user this way of introducing sites of potential parallelism is not much better than requiring the user to use HOFs or parallel constructs in their programs. This realisation indicated the need for automatic HOF synthesis. However, the structure of these verification proofs served as a template for the automatic synthesis of HOF introducing transformation rules from programs.

Verification of properties of HOFs like the distributivity of *map* over $\circ$ and the $map/fold$ equivalence allows these properties to be added to the compiler as transformation rules. The use of these rules can allow many different arrangements of HOFs that can be exploited to find the optimal parallel implementation. These verification proofs also informed the development of the synthesis plan enabling it to find more results from composed functions.

The fertilise critic, although not directly relevant to synthesis of HOFs, highlights a problem with fertilisation that arose later in the synthesis plan. The rewriting of sinks approach used in the critic suggested how this problem could be overcome in the synthesis. The altered solution used in the synthesis suggests how the inadequacies of the fertilise critic itself might be overcome.

3.6 Summary

In order to examine the problem of verifying possible parallelism increasing transformation rules several properties of HOFs were shown to be true and a fertilisation critic was implemented to aid the verification process. The properties we verified either facilitated the production of an optimal parallel implementation by allowing the rearrangement of existing HOFs or increased possible parallelism by introducing new HOFs. These verification problems turned out to be very simple and we turned our attention to the problem of synthesising the class of HOF introducing transformations. The next chapter details the evolution of our synthesis plan from a naïve first attempt to its final form.

Chapter 4

Synthesis plan

To produce a parallel implementation of a program using skeletons the program must contain the HOFs that those skeletons correspond to. HOFs are relatively difficult to understand and work with so they are not often used. It may be desirable to introduce HOFs into programs to increase their parallel performance.

We use synthesis techniques to extract expressions containing HOFs from existing program functions. Where Kraan [KBB96] synthesises from a general unexecutable specification we use executable SML as our specification.

In order to gain some insight into the possible mechanism for the synthesis we looked at the verification proofs (section 3.2) of the equality of the original function and the equivalent HOF expression. These provide a guide to how HOF expressions may be synthesised from program functions.

This chapter outlines the evolution of the synthesis plan from its initial form through each of the major changes to its final form.

4.1 Examples

To illustrate the synthesis plan, we will use the $squares$ example to show its progression. Where necessary to show the rippling sideways and hybrid case we will use $reverse$ and $isums$.

4.1.1 *squares*

squares is a function which will square all the elements of a list. It does this by squaring the first element of the list and recursively applying itself to the rest of the list. The *squares* function can be represented by an expression involving the HOF *map*. *map* takes a function applying it to the first element of a list and recursively applying itself to the rest of the list. So by giving the function $\lambda x.(x * x)$ as the function argument of *map* we achieve the same effect as the original *squares* function.

$$\lambda x.(map\ \lambda y.(y * y)\ x)$$

The synthesis plan will synthesise the expression involving *map* from the definition of the *squares* function.

Anything that can be represented as a *map* expression may also be represented as a *foldr* expression (see section 3.3.1). So the *squares* function can be represented by the *foldr* expression:

$$\lambda x.(foldr\ \lambda y.\lambda z.((y * y) :: z)\ []\ x)$$

The definition of *squares* can be found in section 3.1, and definitions for *map* and *foldr* in figure 2.

4.1.2 *reverse*

The *reverse* function reverses the order of the elements in a list. It does this by taking the first element of the list and putting it on the front of another list. This way the first element to be put onto the second list will be at the end of it and the last element to be copied will be on the front. The auxiliary function *rev* takes two arguments; the result list and the original list. It recursively applies itself to the result list with the first element of the original list attached and the original list with the first element removed.

$$\begin{aligned} reverse\ l &= rev\ []\ l \\ rev\ b\ [] &= b \\ rev\ b\ (h :: t) &= rev\ (h :: b)\ t \end{aligned}$$

rev can be represented by an expression involving the HOF $foldl$. The definition of $foldl$ can be found in figure 2. So by using $\lambda x.\lambda y.(y :: x)$ as the function argument we achieve the same effect as the rev function.

$$\lambda x.\lambda y.(foldl\ \lambda z.\lambda u.(u :: z)\ x\ y)$$

To synthesise a HOF expression involving $reverse$ we must first rewrite and generalise the conjecture:

$$\begin{aligned} \exists M.(reverse\ l &= M\ l) && 36 \\ \exists M.(rev\ []\ l &= M\ l) && \text{gen} \\ \exists M.\forall y.(rev\ y\ l &= M\ y\ l) \end{aligned}$$

This preprocessing of $reverse$ allows the synthesis to perform sideways rippling on rev and synthesise functions within the new variable y[1].

4.1.3 *isums*

The $isums$ function takes a list and constructs a list of the sums of every prefix of the list. So the resulting list will contain the sum of the first element, the sum of the first two elements, the sum of the first three elements etcetera. It does this by using a base case which contains the sum of the previous prefix and adding the next element on to

[1] Unfortunately the $\lambda Clam$ implementation cannot handle this generalisation step so these steps were performed by hand on this example.

it. Then recursively applying itself to the rest of the list using the new sum as the base case and adding the sum to the start of the resulting list.

$$
\begin{aligned}
isums\ b\ [] &= []\\
isums\ b\ (h :: t) &= (h+b) :: (isums\ (h+b)\ t)
\end{aligned}
$$

isums can be represented by an expression involving the HOF *scan*. *scan* takes a function argument and applies it to the base case and the first element of the list. It then recursively applies itself to the rest of the list using the result as the base case. The result is added to the list resulting from the recursive application. The definition of *scan* can be found in figure 2. So by using $\lambda z.\lambda u.(z+u)$ as the function argument to *scan* we get an expression equivalent to *isums*.

$$\lambda x.\lambda y.(scan\ \lambda z.\lambda u.(z+u)\ x\ y)$$

4.2 The Aims of The Plan

The purpose of the synthesis is to find expressions involving HOFs that are equivalent to the original function. The plan progressed from an original idea to an implementable solution to the problem. Each plan may be evaluated with respect to a number of criteria.

1. Solutions. The more solutions the plan can find for a particular problem the better. The search for the most optimal solution is not of particular concern to us; we do not consider the performance information when performing the synthesis but defer that analysis to later. However, the more solutions we find the more likely it is that an optimal solution is among them; performance information can then be used when the transformations are being applied to the program to select

the optimal one. The ways of finding more solutions are using new HOFs which can be used in place of existing ones and introducing new choice points into the plan so that the search is as broad as possible without being unmanageable.

2. Generality. The more types of examples the plan can find solutions for the better. This typically involves using new HOFs where there are no solutions involving the existing HOFs or changing the plan to enable it to work with examples whose structure is different from the existing ones.

3. Feasibility. The easier the plan is to implement in our chosen environment ($\lambda Clam$) the better. This may mean that the existing structures in $\lambda Clam$ may be used in the synthesis or that a particular new construct would fit well into the $\lambda Clam$ environment.

We first revisited the existing plan for verification and based the synthesis plan around that.

4.3 The Verification Plan

Consider:

$$\forall x.(squares\ x = map\ \lambda y.(y * y)\ x)$$

We perform induction on the variable x and the step case proceeds via rippling (section 2.4).
Step Case

$$squares\ t = map\ \lambda y.(y * y)\ t \vdash$$

$$\begin{aligned}
squares\ \boxed{(h :: \boxed{t})}^{\uparrow} &= map\ \lambda y.(y * y)\ \boxed{(h :: \boxed{t})}^{\uparrow} \\
\boxed{(h * h) :: \boxed{(squares\ t)}}^{\uparrow} &= map\ \lambda y.(y * y)\ \boxed{(h :: \boxed{t})}^{\uparrow} \\
\boxed{(h * h) :: \boxed{(squares\ t)}}^{\uparrow} &= \boxed{(h * h) :: \boxed{(map\ \lambda y.(y * y)\ t)}}^{\uparrow}
\end{aligned}$$

If there is an identical outer function on each side we may split the goal apart at this point using substitution axioms.

$$(f \; a_1 \cdots a_n) = (f \; b_1 \cdots b_n) \Rightarrow a_1 = b_1 \;\; \cdots \;\; a_n = b_n$$

where $\Rightarrow$ denotes rewriting.

The outer function(f) in the *squares* example is ::.

$$\boxed{(h * h) = (h * h) \wedge \boxed{\forall y.((squares\ t) = (map\ \lambda y.(y * y)\ t))}}^{\uparrow}$$

Now that the goal is fully rippled we may fertilise to remove the complete wave-hole from the goal.

$$\forall h.((h * h) = (h * h))$$

What is left is a residue which may require a further induction to prove. In this case However, it is trivial.

The base case is then tackled using the same techniques as the main proof. In this case further induction is not necessary and the base case is completed by the application of rewrite rules.

Base Case

$$\begin{aligned} squares\ [] &= map\ \lambda y.(y * y)\ [] \\ [] &= map\ \lambda y.(y * y)\ [] \\ [] &= [] \end{aligned}$$

In the rippling sideways and hybrid rippling cases the step case proofs go as follows:

$$\forall y.(rev\ y\ t = foldl\ \lambda w.\lambda x.(x :: w)\ y\ t) \vdash$$

$$\begin{aligned}
\forall y.(rev\ \lfloor y \rfloor\ \boxed{(h :: \underline{t})}^{\uparrow} &= foldl\ \lambda w.\lambda x.(x :: w)\ \lfloor y \rfloor\ \boxed{(h :: \underline{t})}^{\uparrow})\\
\forall y.(rev\ \left\lfloor \boxed{(h :: \underline{y})}^{\downarrow} \right\rfloor\ t &= foldl\ \lambda w.\lambda x.(x :: w)\ \lfloor y \rfloor\ \boxed{(h :: \underline{t})}^{\uparrow})\\
\forall y.(rev\ \left\lfloor \boxed{(h :: \underline{y})}^{\downarrow} \right\rfloor\ t &= foldl\ \lambda w.\lambda x.(x :: w)\ \left\lfloor \boxed{(h :: \underline{y})}^{\downarrow} \right\rfloor\ t)
\end{aligned}$$

and

$$\forall y.(isums\ y\ t = scan\ \lambda w.\lambda x.(w + x)\ y\ t) \vdash$$

$$\begin{aligned}
\forall y.(isums\ \lfloor y \rfloor\ \boxed{(h :: \underline{t})}^{\uparrow} &= scan\ \lambda w.\lambda x.(w + x)\ \lfloor y \rfloor\ \boxed{(h :: \underline{t})}^{\uparrow})\\
\forall y.(\boxed{(y + h) :: \underline{(isums\ \left\lfloor \boxed{(\underline{y} + h)}^{\downarrow} \right\rfloor)}}^{\uparrow} &= scan\ \lambda w.\lambda x.(w + x)\ \lfloor y \rfloor\ \boxed{(h :: \underline{t})}^{\uparrow})\\
\forall y.(\boxed{(y + h) :: \underline{(isums\ \left\lfloor \boxed{(\underline{y} + h)}^{\downarrow} \right\rfloor)}}^{\uparrow} &= \boxed{(y + h) :: \underline{(foldl\ \lambda w.\lambda x.(w + x)\ \left\lfloor \boxed{(\underline{y} + h)}^{\downarrow} \right\rfloor\ t)}}^{\uparrow})\\
\forall y.(\boxed{(y + h = y + h) \wedge \underline{(isums\ \left\lfloor \boxed{(\underline{y} + h)}^{\downarrow} \right\rfloor) = (foldl\ \lambda w.\lambda x.(w + x)\ \left\lfloor \boxed{(\underline{y} + h)}^{\downarrow} \right\rfloor\ t)}}^{\uparrow})\\
\forall y.(h + y &= h + y)
\end{aligned}$$

Fertilisation can only take place when the sink terms ($\lfloor \ldots \rfloor$) are identical.

The base cases for *reverse* and *isums* are as straight forward as that for *squares*.

4.3.1 Summary

The sequence of steps that generally form an inductive proof using the substitution axiom as a pure longitudinal ripple are shown in figure 9. The sequence of steps are numbered in relation to the plan iteration and the position of the step within the plan. The steps in the verification plan are numbered 0.n because the verification plan is not actually part of the sequence of synthesis plans but precedes the sequence of plans.

Figure 9: Sequence of steps in a verification proof

0.1 Induction

0.2 Rippling

0.3 Ripple using substitution axiom

0.4 Fertilisation

0.5 Plan residue

0.6 Base case

These steps are illustrated by the general schematic proof in figure 10.

Figure 10: Verification proof

$$\forall y.\forall l.(f\ l\ y = g\ l\ y) \qquad 0.1$$

Step Case

$\forall y.(f\ t\ y = g\ t\ y) \vdash$

$$\forall y.(f\ \boxed{(c\ t)}^{\uparrow}\ \lfloor y \rfloor \quad = \quad g\ \boxed{(c\ t)}^{\uparrow}\ \lfloor y \rfloor) \qquad 0.2$$

$$\text{Ripple}\downarrow \qquad \downarrow\text{Ripple}$$

$$\forall y.(\ \boxed{(c_1\ t\ y)\ \oplus\ (f\ t\ \left\lfloor \boxed{(c_2\ t\ y)}^{\downarrow} \right\rfloor)}^{\uparrow} \quad = \quad \boxed{(c_3\ t\ y)\ \oplus\ (g\ t\ \left\lfloor \boxed{(c_2\ t\ y)}^{\downarrow} \right\rfloor)}^{\uparrow}\ ) \qquad 0.3$$

$$\boxed{\forall y.((c_1\ t\ y) = (c_3\ t\ y)) \wedge \forall y.((f\ t\ \left\lfloor \boxed{(c_2\ t\ y)}^{\downarrow} \right\rfloor) = (g\ t\ \left\lfloor \boxed{(c_2\ t\ y)}^{\downarrow} \right\rfloor))}^{\uparrow} \qquad 0.4$$

$$\forall y.((c_1\ t\ y) \quad = \quad (c_3\ t\ y)) \qquad 0.5$$

Base Case

$$\forall y.(f\ b\ y \quad = \quad g\ b\ y) \qquad 0.6$$

4.4 The Initial Synthesis Plan

The initial synthesis plan is our first attempt to produce a synthesis plan. The synthesis plan follows closely the general pattern used in the verification. The major difference is that we use a metavariable in place of the right hand side of the equation. This metavariable cannot be rippled without instantiating it and if we instantiate it early on in the proof we lose the benefits that middle out reasoning can give us.

To get around this problem we side step the rippling on the right and speculate what its result might be. We then return to fill in the gap by running the proof back after doing the latter stages of the proof. In running the proof back we follow all the steps we have already done backwards to the point at which we used a speculative rewrite. By doing this we propagate the instantiation of the metavariables backwards in the proof to this point. The metavariable instantiations performed during the latter stages of the proof allow us to verify the speculative rewrite we made and complete the proof.

In the synthesis of transformations we construct a conjecture by equating the function that we are attempting to transform and a metavariable whose instantiation we constrain during the proof.

The initial synthesis plan only considered the rippling out case because the target HOFs mostly used by the compiler are map and $foldr$ which, due to the form of recursion they use, both involve only rippling out.

4.4.1 The Plan

Consider:

$$\forall x.(squares\ x = M\ x)$$

Step Case

First of all in the step case we ripple on the left as we did in the verification.

$$\begin{aligned}
&squares\ t{=}M\ t \vdash\\
&\qquad squares\ \boxed{h :: \boxed{t}}^{\uparrow} \quad = \quad M\ \boxed{h :: \boxed{t}}^{\uparrow}\\
&\boxed{(h * h) :: \boxed{(squares\ t)}}^{\uparrow} \quad = \quad M\ \boxed{h :: \boxed{t}}^{\uparrow}
\end{aligned}$$

Now the proof is blocked because there are no wave rule involving metavariables which can be used on the right. To overcome this we speculate what the right hand side might look like if we were able to ripple it. Here we speculate that there is a metavariable in part of the wave structure on the right. This allows us to exploit any further synthesis opportunities that may exist in the corresponding structure on the left by recursively calling the synthesis process. We also speculate that the outer function on the right is identical to the outer function on the left. We do this so that we can use the substitution axiom to setup the synthesis goal for the recursive synthesis. We will relax this condition later on to allow the constructor functions to be equal instead of identical (section 4.5).

$$\boxed{(h * h) :: \boxed{(squares\ t)}}^{\uparrow} \quad = \quad \boxed{(M_1\ h\ t) :: \boxed{(M\ t)}}^{\uparrow}$$

Now the goal is unblocked and the proof can proceed as in the verification case.

$$\boxed{(h * h) = (M_1\ h\ t) \wedge \boxed{(squares\ t) = (M\ t)}}^{\uparrow}$$

$$(h * h) = (M_1\ h\ t)$$

We recursively call the synthesis plan to find a higher-order expression for $(h * h)$. In this example no higher-order expression can be found because $*$ is not recursively defined but in other examples there may be further synthesis to be done. Since we cannot perform synthesis in this case we simply instantiate M_1 to the left hand side $(h * h)$.

After the residue is dealt with we then run the proof backwards to the point at which we speculated the right hand side of the goal in the step case.

$$(h * h) \quad = \quad (h * h)$$

$$\boxed{(h * h) = (h * h) \wedge \boxed{(squares\ t) = (M\ t)}}^{\uparrow}$$

$$\boxed{(h * h) :: \boxed{(squares\ t)}}^{\uparrow} \quad = \quad \boxed{(h * h) :: \boxed{(M\ t)}}^{\uparrow}$$

At this point we run the proof backwards to verify the speculative rewrite. The metavariable M will be constrained to be instantiated to only HOFs. The HOFs that could be used here are $foldr$ and map; others such as $foldl$ and $scan$ are ruled out because their structure does not match the problem. The choice between $foldr$ and map is an arbitrary one since both are applicable. This choice between $foldr$ and map adds a choice point in the proof, we will see later on how how this choice is made (section 4.6). Let us say that M is instantiated to the HOF $foldr$.

$$
\begin{aligned}
\boxed{(h * h) :: \boxed{(squares\ t)}}^{\uparrow} &= foldr\ \lambda y.\lambda x.((x * x) :: y)\ B\ \boxed{h :: \boxed{t}}^{\uparrow} \\
squares\ \boxed{h :: \boxed{t}}^{\uparrow} &= foldr\ \lambda y.\lambda x.((x * x) :: y)\ B\ \boxed{h :: \boxed{t}}^{\uparrow}
\end{aligned}
$$

The step case proof is done and M has been instantiated to the HOF $foldr$. Though the base case argument B still remains uninstantiated.

Base Case

$$
\begin{aligned}
squares\ [] &= foldr\ B\ [] \\
[] &= foldr\ B\ [] \\
[] &= B
\end{aligned}
$$

The base case is planned and the base case metavariable B is instantiated.

4.4.2 Summary

The initial synthesis plan proceeds as follows. The proof is symmetrical with the first half being done forward and the second half being done backwards to instantiate the metavariables. After the step case is complete we use the base case to instantiate any remaining base case metavariables.

The numbering of the steps in the synthesis plan reflect the changes in the plan since the previous iteration. New or altered steps are numbered according to the iteration of the plan (in the case of the initial synthesis plan 1.n) and are shown in bold. Steps which are unchanged retain their numbering from the previous iteration (steps which are unchanged from the verification plan will be numbered 0.n).

The sequence of steps are shown in figure 11.

Figure 11: Sequence of steps typical for a proof using the initial synthesis plan

0.1 Induction

0.2 Concrete rippling

1.1 **Speculative ripple**

0.3 Ripple using substitution axiom

0.4 Fertilisation

1.2 **Recursively synthesise from residue**

1.3 **Run proof back**

1.4 **Verify the speculative rewrite**

0.6 Base case

A step in which we speculate what the result of rippling would be if we had an applicable wave rule(1.1) has been added to deal with the metavariable in the goal. We use the synthesis plan to recursively perform further synthesis on subgoals which may be left in the residue after fertilisation(1.2). Steps are added to run the proof backward(1.3) to the point at which the speculative ripple was made and to use the metavariable instantiation to find or verify a rule which matches the speculative rewrite(1.4). The general schematic proof corresponding to the initial synthesis plan is illustrated in figure 12.

Figure 12: Initial general synthesis proof

$$f\ l = M\ l \qquad 0.1$$

Step Case

$f\ t = M\ t \vdash$

$$
\begin{array}{rcll}
f\ \boxed{(c\ \boxed{t})}^{\uparrow} & = & M\ \boxed{(c\ \boxed{t})}^{\uparrow} & 0.2\&1.1 \\
\text{Ripple}\downarrow & = & \vdots & \\
\boxed{(c_1\ t)\ \oplus\ \boxed{(f\ t)}}^{\uparrow} & = & \boxed{(M_1\ t)\ \oplus\ \boxed{(M\ t)}}^{\uparrow} & 0.3 \\
\multicolumn{3}{c}{\boxed{(c_1\ t) = (M_1\ t) \wedge \boxed{(f\ t) = (M\ t)}}^{\uparrow}} & 0.4 \\
\forall y.((c_1\ t) & = & (M_1\ t)) & 0.5 \\
\forall y.((c_1\ t) & = & (h_1\ t)) & 1.3 \\
\multicolumn{3}{c}{\boxed{(c_1\ t) = (h_1\ t) \wedge \boxed{(f\ t) = (M\ t)}}^{\uparrow}} & 1.3 \\
\boxed{(c_1\ t)\ \oplus\ \boxed{(f\ t)}}^{\uparrow} & = & \boxed{(h_1\ t)\ \oplus\ \boxed{(M\ t)}}^{\uparrow} & 1.4 \\
\text{Ripple back}\downarrow & = & \downarrow\text{Ripple back} & \\
f\ \boxed{(c\ \boxed{t})}^{\uparrow} & = & hof\ f\ \boxed{(c\ \boxed{t})}^{\uparrow} &
\end{array}
$$

Base Case

$$f\ b = hof\ f\ B\ b \qquad 0.6$$

Notes:

0.6 hof stands for the higher-order function that the metavariable has been instantiated to. At the moment this is either map or $foldr$ but this will change as the plan becomes capable of finding other HOFs.

The benefits of rippling versus simply instantiating this schematic proof are discussed in section 5.3.

Advantages/Disadvantages

✓ The synthesis method can find transformations involving the map or $fold$ HOFs.

✓ There is the possibility that if the synthesis plan could be applied recursively then this method would find more than one higher-order expression in the original function.

✗ There may be other possible HOFs that this method misses. The restriction to rippling out eliminates the possibility of finding expressions involving $foldl$ or $scan$.

✗ Constraining the outermost functor of the rewritten metavariable to be identical to that on the left may eliminate some possible solutions.

✗ The process of running the entire proof backwards simply retreads the path that the proof has travelled before. This is very inefficient when all that is really needed is to ripple over the gap in the proof.

4.5 The Lemma

It is very inefficient to run the entire proof backwards when only one tiny part of the proof needs to be considered. To address this problem we decided that the speculative step in the rippling proof could be performed by a speculative lemma which could be tackled at a later stage. This use of a lemma does not really impact on the general overall story at the high-level but rather allows us to encode the hole that we previously filled in by running over the proof backwards. This goal is carried along with the proof so that the speculative rewrite in our reasoning is available to us when we are ready to verify it. The lemma would represent a rewrite of the structure on the right hand side and would be used to help instantiate the metavariable M. The original problem of not being able to ripple the metavariable is solved because the lemma will perform the ripple allowing us to continue with the proof. The speculative lemma will be planned when all the metavariables in the wave structure are instantiated and it will be used to find a HOF instantiation for the original metavariable M.

This process of lemma speculation has been used before in the framework of a critics mechanism(section 2.4.5). The difference in this case is that, because this is a synthesis process, there may be metavariables in the wave-hole as well as in the wavefront. Although this seems different, typically the metavariable in the wave-hole is safe from instantiation because it is removed from the main proof by fertilisation and instantiation of metavariables is prohibited during the rippling process to prevent premature instantiation.

The previous synthesis plan may miss several possible solutions in various examples due to constraining the outermost functor on the right to be identical to that on the left. This eliminates the possibility of finding any HOF that might exist in that function.

To counter this problem the substitution axiom needs to be looked at. The original substitution axiom requires the outer functions to be identical and the application of the axiom eliminates them:

$$(f\ a_1 \cdots a_n) = (f\ M_1 \cdots M_n) \Rightarrow a_1 = M_1 \wedge \cdots \wedge a_n = M_n$$

This also eliminates the possibility that a higher-order expression may be found for the outer function but at the same time the substitution axiom is necessary to set up the goals for any recursive synthesis calls. To allow synthesis of the outer function of the left hand side we must alter the substitution axiom so that they require that the functions be equal rather than identical. We can then use a metavariable in place of the outer function on the right and use it to set up another synthesis goal. The new substitution axiom will be:

$$(f\ a_1 \cdots a_n) =_\sigma (M\ M_1 \cdots M_n) \Rightarrow \forall x_1 \cdots x_n (f\ x_1 \cdots x_n =_\sigma M\ x_1 \cdots x_n) \wedge a_1 =_\sigma M_1 \wedge \cdots \wedge a_n =_\sigma M_n$$

The individual arguments are equated as before but instead of eliminating the function symbols we equate them using universally quantified variables in each argument position. This is like generalising the function away from each argument position. In verification this rule gives rise to the possibility that the result will be a non-theorem. In synthesis However, this means that it may simply eliminate some possible solutions.

In the verification mode the non-identity of the outer functions may simply cause an argument misalignment which prevents the proof from finishing. For example:

If $g = \forall x.\forall y.(f\ y\ x)$

$$f\ a\ b\ =\ g\ b\ a$$

will produce

$$\forall x.\forall y.(f\ x\ y\ =\ g\ x\ y) \wedge a\ =\ b \wedge b\ =\ a$$

The original may be true but the result is false if f and g are not commutative or $a \neq b$.

In the synthesis case However, there are metavariables which can be moulded.

$$f\ a\ b\ =\ M\ M_1\ M_2$$

This means that the problem found in the verification never arises because M_1 can be instantiated to a, M_2 to b and M to f. In the substitution setting the application of the extended substitution axiom merely rules out the possibility of finding g for M instead of creating a theorem which cannot be proven. The elimination of this result is of no consequence because for any g the synthesis will be able to find an f which is identical but has its arguments reversed.

4.5.1 The Plan

We proceed with the proof as before until the rippling process is blocked by the metavariable.

$$squares\ t = M\ t \vdash$$

$$\boxed{(h * h) :: \boxed{(squares\ t)}}^{\uparrow}\ =\ M\ \boxed{h :: \boxed{t}}^{\uparrow}$$

At this point instead of simply deciding what the result of rippling on the right would be we speculate a schematic wave rule to perform the ripple step for us.

$$M\ \boxed{h :: \underline{t}}^{\uparrow} \Rightarrow \boxed{M_2\ h\ t\ (M_1\ h\ t)\ \underline{(M\ t)}}^{\uparrow}$$

We set this wave rule up as a separate goal which will be proven later on. We are effectively carrying the missing part of the proof around with us so that we have easy access to it when the time comes to instantiate M.

The rewrite we speculate is different from the previous one. In the previous plan we decided that the outermost functor was identical to that on the left. We did this because the substitution axiom would fail to apply if the outer functions were not identical. However, with our altered substitution axiom the outer most functor need not be identical and we can use a metavariable instead of the outer function that exists on the left hand side.

$$\boxed{(h * h) :: \underline{(squares\ t)}}^{\uparrow} \quad = \quad \boxed{M_2\ h\ t\ (M_1\ h\ t)\ \underline{(M\ t)}}^{\uparrow}$$

The new substitution axiom applies and we get the result:

$$\boxed{\forall x.\forall y.(x :: y = M_2\ h\ t\ x\ y) \wedge ((h * h) = (M_1\ h\ t)) \wedge \underline{(squares\ t) = (M\ t)}}^{\uparrow}$$

After fertilisation applies we have two goals to which we can apply the synthesis plan:

$$\forall x.\forall y.(x :: y = M_2\ h\ t\ x\ y)$$

and

$$(h * h) = (M_1\ h\ t)$$

In various examples neither, either or both can have higher-order expressions found for them. In this proof no further synthesis is possible so we simply instantiate the two metavariables.

When we have instantiations for M_1 and M_2(in this example $\lambda x.(x*x)$ and $\lambda x.\lambda y.(x :: y)$) and the main step case proof is complete we try to plan the schematic wave rule:

$$M\ \boxed{h :: \boxed{t}}^{\uparrow} = \boxed{(h * h) :: \boxed{(M\ t)}}^{\uparrow}$$

We cannot progress any further without instantiating M so we examine the structure to determine what HOFs are suitable. The wave structure is similar to that of $foldr$ so it is a candidate, also there is a :: as the outer function which means that map is also a candidate.

Let's choose a $foldr$, as before this choice is arbitrary, as our HOF and instantiate M. We will look at the choice of map or $foldr$ later on in chapter 6. This is only a partial instantiation of M as the arguments to the HOF are still represented by metavariables(F and B). F and B are not applied to any arguments because although h and t are in scope here they will not be in the base case and the function F will be available in the base case even if it is not used.

$$foldr\ F\ B\ h :: t = (h * h) :: (foldr\ F\ B\ t)$$

Symbolic evaluation will now apply and we can use the $foldr$ step case rule to rewrite the left hand side.

$$F(foldr\ F\ B\ t) = (h * h) :: (foldr\ F\ B\ t)$$

Identity is now applied and the two sides are made identical by instantiating F to $\lambda x.\lambda y.\lambda z.((y * y) :: z)$.

As before the base case argument B is not instantiated by this process but it will be taken care of by the base case proof as before.

4.5.2 Summary

Although not shown here other steps may be applied before induction. We ripple the concrete structure on the left of the equality. Then we speculate a wave rule to apply on the right hand side. We carry this around with us to enable us to verify it at a later time. We apply substitution axioms to split the step case goal into its component parts. We fertilise and recursively call the synthesis process on each subgoal in the residue. We then plan the speculated wave rule to prove it and to instantiate the original metavariable. Any remaining base case metavariables will be taken care of by the base case proof.

The sequence of steps are shown in figure 13.

Figure 13: Sequence of steps in a typical speculative lemma synthesis proof

0.1 Induction

0.2 Concrete rippling

2.1 **Speculative ripple**

2.2 **Ripple using altered substitution axiom**

0.4 Fertilisation

1.2 Recursively synthesise from residue

2.3 **Plan the speculative wave rule**

0.6 Base case

The speculative ripple step has been altered(2.1) so that instead of simply creating a gap it uses a speculative wave rule to continue the proof. The substitution axiom has also been altered to allow synthesis to be performed on the outer function instead of eliminating it(2.2). The rippling back steps have been replaced by the planning of the speculative wave rule to complete the step case proof(2.3). The general schematic proof corresponding to the speculative lemma synthesis plan is shown in figure 14.

Figure 14: General speculative lemma synthesis proof

$$f\ l \quad = \quad M\ l \qquad 0.1$$

Step Case

$f\ t = M\ t \vdash$

$$f\ \boxed{(c\ \underline{t})}^{\uparrow} \quad = \quad M\ \boxed{(c\ \underline{t})}^{\uparrow}\ ) \qquad 0.2\&2.1$$

$$\text{Ripple}{\downarrow} \quad = \quad {\downarrow}\text{Speculative wave rule}$$

$$\boxed{(c_1\ t)\ \oplus\ \underline{(f\ t)}}^{\uparrow} \quad = \quad \boxed{M_2\ t\ (M_1\ t)\ \underline{(M\ t)}}^{\uparrow}\ ) \qquad 0.3$$

$$\boxed{\forall x.\forall y.((x\ \oplus\ y) = (M_2\ t\ x\ y)) \wedge (c_1\ t) = (M_1\ t) \wedge \underline{(f\ t) = (M\ t)}}^{\uparrow} \qquad 0.4$$

$$\forall x.\forall y.((x\ \oplus\ y) = (M_2\ t\ x\ y)) \wedge (c_1\ t) = (M_1\ t) \qquad 1.2$$

Speculative wave rule

$$M\ \boxed{(c\ \underline{t})}^{\uparrow} \quad = \quad \boxed{h_2\ t\ y\ (h_1\ t\ y)\ \underline{(M\ t)}}^{\uparrow} \qquad 2.3$$

Base Case

$$f\ b\ y{=}hof\ \lambda x.(h_2\ t\ (h_1\ t)\ x)\ (B\ t)\ b) \qquad 0.6$$

Advantages/Disadvantages

✓ Using the schematic wave rule allows us to carry around the hole in the step case proof so that when we are ready to verify the speculative rewrite we have it available and don't have to run backwards through the entire proof.

✓ Using the altered substitution axiom allows us to exploit another site of potential transformation.

✗ The altered substitution axiom brings with it the possibility that it will perform a generalisation to a non-theorem. In the synthesis setting this actually only means that some solutions may be ruled out by its application.

✗ Although the potential for transformation in this plan is greater than previously there are other sites of potential transformation that may have been overlooked. There may be some structure that may be missed just outside the wave hole.

✗ The plan is restricted to rippling out. This restricts the choice of HOF and narrows the range of examples for which we can find higher-order expressions.

4.6 Copycat Rippling

Introducing rippling sideways and hybrid rippling into the plan will increase the number of examples that the plan will work over. Modifying the plan to include rippling sideways and hybrid rippling was straightforward even though the existing plan was designed with rippling out in mind. The only problems that were encountered were in setting up recursive goals using structure that had been rippled into a sink.

In the previous plan two metavariables were introduced at the wave rule speculation stage. These correspond to the two sites of potential transformation that we previously identified; the structure just outside the wave hole and inside the outer function. However, if we consider the possible structure that a goal might have there could be sites of transformation at other places. For example inside a sink, when using rippling sideways or hybrid rippling.

$$\forall y.(M\ \boxed{(c\ t)}^{\uparrow}\ \lfloor y \rfloor) \Rightarrow \boxed{\forall y.(M_2\ t\ y\ (M_1\ t\ y)\ (M_3\ t\ y\ (M\ t\ \left\lfloor \boxed{(M_4\ t\ y)}^{\downarrow} \right\rfloor)))}^{\uparrow}$$

Any of the metavariables M_1, M_2, M_3 and M_4 might be instantiated to identity. On further examination it became obvious that there could be any number of sites contained within each of the four metavariables.

Previously the plan used a speculative wave rule that tried to cover all the possible sites of possible synthesis. After the introduction of rippling sideways and hybrid rippling, if metavariables introduced by the schematic wave rule were not needed they could be instantiated to identity and removed. However, the use of a supposedly all encompassing rule was discarded when it became clear that such a rule would never cover all the possibilities in all cases.

We abandon the concept of an outer function which effectively restricts the structure to having an arity two function as the outermost functor. Instead a wave rule would be constructed that matched the form of the concrete structure on the left hand side of the equality. By doing this we get a one to one relationship between the functions in the concrete structure on the left hand side and the metavariables speculated in the wave rule on the right hand side.

It would be simpler to use a single metavariable and rely on λ*Clam* s higher-order unification to introduce new ones as necessary. However, a combination of the way we restrict metavariables from premature instantiation and restrictions made to the rippling process made in the implementation (section 5) prevent this. Overcoming these implementation problems would allow the use of a single metavariable. In section 4.7 we present the synthesis plan using this single metavariable method.

To enable the plan to exploit all the sites, the substitution axiom may be required to be applied more than once to totally split apart the goal. While doing this the rippling must be preserved so the wave hole should not be subject to these rewrites. However, structure which has been rippled into a sink will be represented on the right by a metavariable. Because the sinks on the left and right will no longer be identical, fertilisation can no longer take place. A way around this problem is to abstract away from the sink argument in the wave hole and construct a subgoal by equating the contents of the sinks. We implemented a similar technique as part of the *Clam* fertilisation critic(section 3.4).

4.6.1 The Plan

We proceed with the proof exactly as before but this time when we speculate the schematic wave rule to fill in the hole in the proof we use the concrete structure on the left hand side to determine the structure of the wave rule.

$squares\ t = M\ t \vdash$

$$\boxed{(h * h) :: \boxed{(squares\ t)}}^{\uparrow} \quad = \quad M\ \boxed{h :: \boxed{t}}^{\uparrow}$$

We speculate the structure of the wave rule by examining the structure of the wave front on the left - $\boxed{(h * h) :: \boxed{(\ldots)}}^{\uparrow}$. We have the functions $*$ and $::$ so the wave rule we speculate has two metavariables, M_1 corresponding to $::$ and M_2 corresponding to $*$, in the positions occupied by those functions - $\boxed{M_1\ h\ t\ (M_2\ h\ t)\ \boxed{(\ldots)}}^{\uparrow}$

$$M\ \boxed{h :: \boxed{t}}^{\uparrow} \Rightarrow \boxed{M_1\ h\ t\ (M_2\ h\ t)\ \boxed{(M\ t)}}^{\uparrow}$$

In the rippling sideways example of the synthesis of the *reverse* function, we have the following wave structure:

$$\forall y.(rev\ \left\lfloor \boxed{(h :: \boxed{y})}^{\downarrow} \right\rfloor\ t {=} M\ \lfloor y \rfloor\ \boxed{h :: \boxed{t}}^{\uparrow}\ )$$

The speculative wave rule that arises from this is determined by the wavefront inside the sink so in addition to the original metavariable M we have one new metavariable M_1 corresponding to the $::$ in the sink:

$$\forall y.(M\ \lfloor y \rfloor\ \boxed{h :: \boxed{t}}^{\uparrow}\ ) \Rightarrow \forall y.(M\ \left\lfloor \boxed{(M_1\ h\ t\ \boxed{y})}^{\downarrow} \right\rfloor\ t)$$

In the hybrid rippling example there are metavariables in both the wavefront and sink positions:

$$\forall y.(\ \boxed{(h + y) :: \boxed{(isums\ \left\lfloor \boxed{(h + \boxed{y})}^{\downarrow} \right\rfloor\ t)}}^{\uparrow}\ {=} M\ \lfloor y \rfloor\ \boxed{h :: \boxed{t}}^{\uparrow}\ )$$

giving three new metavariables

$$\forall y.(M\ \lfloor y \rfloor\ \boxed{h :: t}^{\uparrow}) \Rightarrow \forall y.(\ \boxed{M_1\ h\ t\ y\ (M_2\ h\ t\ y)\ (M\ \left\lfloor \boxed{(M_3\ h\ t\ y)}^{\downarrow} \right\rfloor\ t)}^{\uparrow}\ )$$

Rewriting in the *squares* example with the speculated wave rule we get:

$$\boxed{(h * h) :: (f\ t)}^{\uparrow} \quad = \quad \boxed{M_1\ h\ t\ (M_2\ h\ t)\ (M\ t)}^{\uparrow}$$

We then use substitution axioms repeatedly to split apart the goal.

$$\boxed{\begin{array}{l}(h * h) = (M_1\ h\ t) \wedge \forall x.\forall y.((x :: y) = (M_2\ h\ t\ x\ y)) \wedge \\ \hfill \forall y.((squares\ t) = (M\ t))\end{array}}^{\uparrow}$$

In the rippling sideways case the substitution axiom is required to manipulate the structure within the sink. In the original rippling sideways strategy for verification, the sinks are required to be identical. Several examples were found where fertilisation would be blocked because the sinks were equal but not identical. A critic was suggested to construct a rule which could be used to rewrite one of the sinks to be identical to the other and so unblock the fertilisation step(section 2.4). In the synthesis we cannot rely on the rule to rewrite the sinks because they contain metavariables and cannot be proved without instantiating them. Instead we use the extended substitution axioms to remove the sink structure from the goal and move them into the wavefront leaving the sink variables untouched. This is similar to the fertilisation critic in that the rule we use to rewrite the sinks within the critic is the same as the structure we move into the wavefront. The purpose of the rule in the critic is to make the sinks identical. We do that by using the substitution axiom to remove the extra sink structure from the goal. The correctness is preserved because instead of proving the rule before using it to rewrite the sinks we simply carry the rule around with us delaying its proof until later on.

This process applied to the *reverse* example is as follows:

$$\forall y.(rev \left\lfloor \boxed{(h :: y)}^{\downarrow} \right\rfloor t = M \left\lfloor \boxed{(M_1\ t\ h\ y)}^{\downarrow} \right\rfloor t$$

$$\boxed{\forall y.((h :: y) = (M_1\ t\ h\ y)) \wedge \forall y.(rev\ \lfloor y \rfloor\ t = M\ \lfloor y \rfloor\ t)}^{\uparrow}$$

In the hybrid rippling case both the goals created by the extended substitution axiom and the goal produced by the extraction of sinks must be included in the wavefront. The *isums* example proceeds as follows:

$$\forall y.(\boxed{(h+y) :: (isums \left\lfloor \boxed{(h+y)}^{\downarrow} \right\rfloor t)}^{\uparrow} = \boxed{M_1\ t\ h\ y\ (M_2\ t\ h\ y)\ (M \left\lfloor \boxed{(M_3\ t\ h\ y)}^{\downarrow} \right\rfloor t)}^{\uparrow})$$

$$\boxed{\begin{array}{r} \forall w.\forall x.\forall y.(w :: x = M_1\ t\ h\ y\ w\ x) \wedge \forall y.(h+y = M_2\ t\ h\ y) \wedge \\ \forall y.(h+y = M_3\ t\ h\ y) \wedge \forall y.((isums\ \lfloor y \rfloor\ t) = (M\ \lfloor y \rfloor\ t)) \end{array}}^{\uparrow}$$

The rest of the *isums* plan proceeds as before. We call the synthesis recursively on the goals generated and then plan the schematic wave rule and the base case.

The choice of HOF is restricted by the form of rippling used which corresponds to the form of recursion used in the original function. Rippling in (as in *reverse*) restricts the choice of HOF to *foldl* while hybrid rippling (as in *isums*) restricts the HOF to *scan*. Rippling out restricts the HOF to *map* or *foldr*. We will prioritise the choice of *map* over *foldr* because *map* is the more specific of the two and may have a slightly more efficient implementation. Also if we use the *map*/*fold* equivalence as a transformation rule we can always transform a *map* into a *foldr* while the transformation of *foldr* into *map* is not always possible. We will retain the selection of HOF as a choice point in the proof to enable our synthesis method to find as many HOFs as possible.

This synthesis plan has been implemented in $\lambda Clam$ (see chapter 5). Using this implementation we were able to synthesise transformations from several examples. *map* and *foldr* transformations were synthesised from the *squares* example, *foldl* from the *reverse* example and *scan* from the *isums* example as well as several transformations involving multiple nested HOFs (chapter 6).

4.6.2 Summary

We ripple the concrete structure as much as possible. Then by imitating the concrete structure we speculate a schematic wave rule to rewrite the unknown term structure. We carry this rewrite around with us so we can verify it at a later time. Substitution axioms split the goal into its component parts and enable fertilisation. The residue is dealt with recursively by the synthesis plan to deal with any synthesis subgoals which have been set up. The speculative wave rule is used to instantiate the original metavariable. The base case proof takes care of any remaining metavariables.

The sequence of steps are shown in figure 15.

Figure 15: Sequence of steps in a typical copycat ripple synthesis proof

0.1 Induction

0.2 Concrete rippling

3.1 **Speculative ripple**

3.2 **Ripple using altered substitution axiom**

0.4 Fertilisation

1.2 Recursively synthesise from residue

2.3 Plan the speculative wave rule

0.6 Base case

The speculative ripple step (3.1) was altered so that the speculative wave rule matches the specific arrangement of functions in the goal rather than some supposedly generally applicable rule. Rippling with the substitution axiom was altered (3.2) to apply the substitution axiom repeatedly to the goal in order to fully split it into its component functions. The general schematic proof corresponding to the copycat rippling synthesis plan is shown in figure 16. This synthesis plan has been implemented in $\lambda Clam$.

Figure 16: General copycat ripple synthesis proof

$$\forall y.\forall l.(f\ l\ y \quad = \quad M\ l\ y) \qquad 0.1$$

Step Case

$\forall y.(f\ t\ y = M\ t\ y) \vdash$

$$\forall y.(f\ \boxed{(c\ \underline{t})}^{\uparrow}\ \lfloor y \rfloor \quad = \quad M\ \boxed{(c\ \underline{t})}^{\uparrow}\ \lfloor y \rfloor) \qquad 0.2\&3.1$$

$$\text{Ripple}\downarrow \quad = \quad \downarrow\text{Speculative wave rule}$$

$$\forall y.(\boxed{(c_1\ t\ y)\ \oplus\ \underline{(f\ t\ \left\lfloor \boxed{(c_2\ t\ \underline{y})}^{\downarrow} \right\rfloor)}}^{\uparrow} \quad =$$

$$\boxed{M_2\ t\ y\ (M_1\ t\ y)\ \underline{(M\ t\ \left\lfloor \boxed{(M_3\ t\ \underline{y})}^{\downarrow} \right\rfloor)}}^{\uparrow}\ ) \qquad 3.2$$

$$\boxed{\begin{array}{r}\forall w.\forall x.\forall y.((w\ \oplus\ x) = (M_2\ t\ y\ w\ x)) \wedge \forall y.((c_1\ t\ y) = (M_1\ t\ y)) \wedge \\ \forall y.((c_2\ t\ y) = (M_3\ t\ y)) \wedge \underline{\forall y.((f\ t\ y) = (M\ t\ y))}\end{array}}^{\uparrow} \qquad 0.4$$

$$\begin{array}{r}\forall w.\forall x.\forall y.((w\ \oplus\ x) = (M_2\ t\ y\ w\ x)) \wedge \\ \forall y.((c_1\ t\ y) = (M_1\ t\ y)) \wedge \forall y.((c_2\ t\ y) = (M_3\ t\ y))\end{array} \qquad 1.2$$

Speculative wave rule

$$\forall y.(M\ \boxed{(c\ \underline{t})}^{\uparrow}\ \lfloor y \rfloor) = \forall y.(\boxed{h_2\ t\ y\ (h_1\ t\ y)\ \underline{(M\ t\ \left\lfloor \boxed{(h_3\ t\ \underline{y})}^{\downarrow} \right\rfloor)}}^{\uparrow}\ ) \qquad 2.3$$

Base Case

$$\forall y.(f\ b\ y{=}hof\ \lambda x.(h_2\ t\ y\ (h_1\ t\ y)\ x)\ (B\ t\ y)\ b) \qquad 0.6$$

Advantages/Disadvantages

✓ By using the speculative rewrite to introduce a metavariable for every function in on the left hand side we ensure that we exploit every site of potential synthesis.

✓ By enabling rippling sideways and hybrid rippling we have extended the plan so that now it can handle the higher order functions *foldl* and *scan* as shown by the *reverse* and *isums* examples (results for *reverse* and *isums* are show in appendix D).

- ✗ The choice of HOFs is still restricted to the ones that we know properties of and can tailor our method to find. This plan will only find expressions involving map, $foldr$, $foldl$, or $scan$. Ideally the plan should be able to handle any HOF that is available to it.

- ✗ This plan only handles one step induction. This is fine for map, $foldr$, $foldl$ and $scan$ but other HOFs may use different forms of recursion requiring different induction schemes. For instance zip works over two lists of the same length and requires simultaneous induction.

- ✗ By speculating a new metavariable for every goal we commit ourselves to finding a HOF for the individual component functions of the subgoal. This eliminates the possibility of finding different results that may arise out of synthesis where two functions are composed together.

- ✗ The high level synthesis may take composed goals to synthesise and will not split up goals to find results for the individual functions.

- ✗ If a synthesis goal is planned at the top level the results generated will be different from those generated if it is a subgoal of a larger problem. At the top level composed functions are not split apart(which eliminates some possible results) and at lower levels composed functions are always split apart(which eliminates different possible results.)

- ✗ Speculating a new metavariable for every individual function does not seem natural in the setting of higher-order unification because possible combinations of functions are usually represented by a single metavariable.

4.7 Imitating Rippling

After implementing the previous synthesis plan we found several problems. This led us to develop an improved plan which though not implemented addresses the shortcomings of the previous plan and does so in a much more simple and elegant way. The improved plan is discussed in this section. Implementation of this plan could be done as future work (section 8.4.7).

Most of the disadvantages of the implemented method stem from the multiple metavariables speculated in the schematic wave rule. This along with the splitting up of the goal prevent some possible solutions from being found. Subproblems that might contain composed functions are split up and top level examples which contain composed functions are not split up to find results for the individual functions. To find all possible solutions we must standardise the way that top level and subproblem functions are treated. We decided that instead of imitating the concrete structure on the left hand side of the equation we would instead imitate the rippling process itself. This creates only one metavariable in the wavefront and one in the sink.

This change means that any subproblem involving composed functions will be planned as a top level problem with composed functions. To find solutions for the individual functions under this new scheme we must add a new method for splitting problems involving composed functions into goals involving single functions. By making this a choice point the plan can explore the possible results of both splitting and not splitting composed functions.

The decompose method can be considered to be superfluous because backtracking over the rippling process should find both solutions. However, in the implementation of the previous synthesis plan it was found that unrestricted backtracking over rippling incurred huge performance costs, so much so that the planner would run out of memory before finding a second result. For this reason backtracking over rippling was restricted and the possibility of finding both the split and composed results via rippling was ruled out. By choosing to split (or not) composed functions we make the possibility of finding both solutions explicit in the plan instead of relying on backtracking over rippling to provide them.

The implementation of the previous plan highlighted another problem with the previous technique. The techniques for choosing a HOF required knowledge of the properties of the possible HOFs in order to decide which of them to instantiate the metavariable to. If, instead of simply deciding which HOF to use to instantiate the metavariable, we unify the speculative rewrite and the HOF step case rewrite we can eliminate unsuitable HOFs and instantiate the function argument at the same time. Also by using unification we avoid the need for information about the HOFs and the method can use any HOF available. This also opens up the possibility of finding HOF expressions involving HOFs which require different forms of induction like *zip*

Although the final plan has not been implemented it is an incremental development of the previous plan and we have confidence that it will function correctly. The main differences between the implemented plan and the final plan are the addition of the decompose method, the method for choosing the HOF and the construction of the speculative rewrite. The new decompose method is performed before the existing synthesis and doesn't directly interfere with the synthesis process. It simply increases the number of goals that the synthesis process is applied to. The HOF choice method is a generalising change local to a single method so will not interfere with the current synthesis. The construction of the speculative rewrite reduces the set of subgoals that is created for further synthesis; combined with the decompose rule this allows a greater range of results to be found. The important part, from the point of view of the immediate synthesis, is the part within the wave hole is protected by the skeleton preserving property of rippling.

4.7.1 The Plan

The decomposition method will apply before induction if the problem contains composed functions. We use the following rule to rewrite the problem:

$$\forall l.((f \circ g)\ l = M\ l) \Rightarrow \forall l.(f\ l = M_1\ l) \wedge \forall l.(g\ l = M_2\ l) \wedge (M = (M_1 \circ M_2))$$

In the case of the *squares* example:

$$\forall l.(squares\ l \quad = \quad M\ l)$$

the decomposition cannot apply because *squares* is a single function and not a composition of functions. Had the problem been $squares\ (incs\ l)$[2] then the decomposition rule would apply and produce the goals $\forall l.(squares\ l = M\ l)$ and $\forall l.(incs\ l = M\ l)$

After decomposition the synthesis is as before until the speculation of the schematic wave rule. Instead of speculating a rewrite by imitating the concrete structure on the left hand side we imitate the form of rippling being used. The *squares* example uses rippling out so we speculate a rewrite which has a single new metavariable in the wave front position:

$$M\ \boxed{h :: \boxed{t}}^{\uparrow} = \boxed{M_1\ h\ t\ \boxed{(M\ t)}}^{\uparrow}$$

If rippling sideways is used, as in the *reverse* example, we speculate a rewrite with a single new metavariable in the sink position:

$$\forall y.(M\ \lfloor y \rfloor\ \boxed{h :: \boxed{t}}^{\uparrow} = M\ \left\lfloor \boxed{(M_1\ h\ t\ \boxed{y})}^{\downarrow} \right\rfloor\ t)$$

[2]Where *incs* is a function which increments every element in a list by one.

In the *isums* example there is hybrid rippling so we speculate a rewrite with two new metavariables: one in the wave front and one in the sink:

$$\forall y.(M\ \lfloor y \rfloor\ \boxed{h :: \boxed{t}}^{\uparrow} = \boxed{M_1\ h\ t\ y\ \boxed{(M\ \left\lfloor \boxed{(M_2\ h\ t\ \boxed{y})}^{\downarrow} \right\rfloor\ t)}}^{\uparrow}\ )$$

For other induction schemes the rules may look like:

$$\forall y.(M\ \lfloor y \rfloor\ \boxed{h1 :: \boxed{t1}}^{\uparrow}\ \boxed{h2 :: \boxed{t2}}^{\uparrow} =$$

$$\boxed{M_1\ y\ h1\ t1\ h2\ t2\ \boxed{(M\ \left\lfloor \boxed{(M_2\ h1\ t1\ h2\ t2\ \boxed{y})}^{\downarrow} \right\rfloor\ t1\ t2)}}^{\uparrow}\ )$$

for simultaneous induction and:

$$\forall y.(M\ \lfloor y \rfloor\ \boxed{h1 :: h2 :: \boxed{t}}^{\uparrow} = \boxed{M_1\ h1\ h2\ t\ \boxed{(M\ \left\lfloor \boxed{(M_2\ h1\ h2\ t\ \boxed{y})}^{\downarrow} \right\rfloor\ t)}}^{\uparrow}\ )$$

for two-step induction. These rules show hybrid rippling, rippling out and rippling sideways could also be used in conjunction with these induction schemes.

The rewrite is applied and the substitution axiom is used to split up the goal. In this case instead of using the substitution axiom to totally split apart the goal it is used only to split the wavefront and wave-hole of each side apart:

$$\boxed{(h * h) :: \boxed{(squares\ t)}}^{\uparrow} \quad = \quad \boxed{M_1\ h\ t\ \boxed{(M\ t)}}^{\uparrow}$$

$$\boxed{\forall x.(((h * h) :: x) = (M_1\ h\ t\ x)) \wedge \boxed{(squares\ t) = (M\ t)}}^{\uparrow}$$

In the *reverse* example there is no wavefront and the sink part is split from the rest of the goal to form a wavefront:

$$rev\ \left\lfloor \boxed{(h :: \boxed{y})}^{\downarrow} \right\rfloor\ t \quad = \quad M\ \left\lfloor \boxed{(M_1\ h\ t\ \boxed{y})}^{\downarrow} \right\rfloor\ t$$

$$\boxed{\forall y.((h :: y) = (M_1\ h\ t\ y)) \wedge \boxed{(rev\ t) = (M\ t)}}^{\uparrow}$$

In the *isums* example there are both wavefront and sink parts which are split from the rest of the goal:

$$\boxed{(h+y) :: \boxed{(isums \left\lfloor \boxed{(h+\boxed{y})}^{\downarrow} \right\rfloor t)}}^{\uparrow} = \boxed{M_1\ h\ t\ y\ \boxed{(M \left\lfloor \boxed{(M_2\ h\ t\ \boxed{y})}^{\downarrow} \right\rfloor t)}}^{\uparrow}$$

$$\boxed{\forall x.\forall y.(((h+y) :: x) = (M_1\ h\ t\ y\ x)) \wedge \forall y.((h+y) = (M_2\ h\ t\ y)) \wedge \boxed{\forall y.((isums\ y\ t) = (M\ y\ t))}}^{\uparrow}$$

As before the wave hole is removed by fertilisation and the subproblems in the wavefront are dealt with by recursion.

The speculated wave rule is then unified with any available HOF rewrites. In the *squares* example the speculative wave rule[3] will unify with either the *map* or *foldr* wave rules:

$$\begin{aligned} M\ \boxed{h :: \boxed{t}}^{\uparrow} &= \boxed{(h*h) :: \boxed{(M\ t)}}^{\uparrow} \\ map\ F\ \boxed{h :: \boxed{t}}^{\uparrow} &= \boxed{(F\ h) :: \boxed{(map\ F\ t)}}^{\uparrow} \\ foldr\ F\ B\ \boxed{h :: \boxed{t}}^{\uparrow} &= \boxed{F\ h\ \boxed{(foldr\ F\ B\ t)}}^{\uparrow} \end{aligned}$$

The *reverse* speculative rewrite will unify with the *foldl* wave rule:

$$\begin{aligned} M\ \lfloor y \rfloor\ \boxed{h :: \boxed{t}}^{\uparrow} &= M \left\lfloor \boxed{(h :: \boxed{y})}^{\downarrow} \right\rfloor t \\ foldl\ F\ \lfloor A \rfloor\ \boxed{h :: \boxed{t}}^{\uparrow} &= foldl\ F \left\lfloor \boxed{(F\ \boxed{A}\ h)}^{\downarrow} \right\rfloor t \end{aligned}$$

And the *isums* speculative rewrite will unify with the *scan* wave rule:

$$\begin{aligned} M\ \lfloor y \rfloor\ \boxed{h :: \boxed{t}}^{\uparrow} &= \boxed{(h+y) :: \boxed{(M \left\lfloor \boxed{(h+\boxed{y})}^{\downarrow} \right\rfloor t)}}^{\uparrow} \\ scan\ F\ \lfloor A \rfloor\ \boxed{h :: \boxed{t}}^{\uparrow} &= \boxed{(F\ h\ A) :: \boxed{(scan\ F \left\lfloor \boxed{(F\ h\ \boxed{A})}^{\downarrow} \right\rfloor t)}}^{\uparrow} \end{aligned}$$

This technique allows an any HOF to be used for instance if we extended the plan to use simultaneous induction the *zip* function immediately becomes a candidate HOF[5]:

$$\begin{aligned} M\ \boxed{h_1 :: \boxed{t_1}}^{\uparrow}\ \boxed{h_2 :: \boxed{t_2}}^{\uparrow} &= \boxed{(h_1+h_2) :: \boxed{(M\ t_1\ t_2)}}^{\uparrow} \\ zip\ F\ \boxed{h_1 :: \boxed{t_1}}^{\uparrow}\ \boxed{h_2 :: \boxed{t_2}}^{\uparrow} &= \boxed{(F\ h_1\ h_2) :: \boxed{(zip\ F\ t_1\ t_2)}}^{\uparrow} \end{aligned}$$

[3] $(h*h)$ :: is the instantiation for M_1 from recursive synthesis in the *squares* example. In the *reverse* example h :: is the instantiation for M_1. The *isums* example contains two instantiations, $(h+y)$ :: for M_1 and $h+y$ for M_2[4].

[4] In the *isums* example the instantiations for M_1 and M_2 must be related by $M_1 = (M_2\ ::)$. If this is not the case then the unification will fail because the two Fs in the *scan* wave rule must be instantiated to the same thing.

[5] For illustration we use a function which *zip*s two lists together using $+$.

The unification may succeed in various cases and the order in which the rewrites are tried will determine the order in which the different solutions are found. This ordering may at first be arbitrary but eventually some mechanism could be supplied by which the ordering could be changed. The unification will instantiate the function argument automatically.

Lastly we use the base case, as before, to finish the proof and instantiate any remaining base case variable.

4.7.2 Summary

Before the induction takes place we decompose(or not) goals which consist of two composed functions. We use induction to split the goal into step case and base case goals. We then ripple the concrete structure. By imitating the form of rippling that took place in the concrete structure we speculate a rewrite of the unknown structure. We carry this wave rule around with us and plan it at a later time. We apply substitution axioms to split apart the goal by equating the term structure in the wave fronts, wave holes and sinks on either side. We eliminate the wave hole with fertilisation and recursively apply the synthesis plan to the residue. The speculative wave rule is unified with HOF rewrites to instantiate the original metavariable and the base case is used to instantiate any remaining base case metavariables.

The sequence of steps are shown in figure 17.

Figure 17: Sequence of steps in a typical imitated ripple synthesis proof

4.1 **Decompose**

0.1 Induction

0.2 Concrete rippling

4.2 **Speculative ripple**

4.3 **Ripple using substitution axiom**

0.4 Fertilisation

1.2 Recursively synthesise from residue

4.4 **Unify speculative wave rule with available HOF rewrites**

0.6 Base case

A new decompose step (4.1) has been added which can split apart composed functions in the synthesis goal. The speculative ripple step (4.2) has been altered so that instead of exactly copying the structure of the concrete goal onto the speculative wave rule the form of concrete rippling is used to construct a more general goal. The substitution axiom ripple step (4.3) has been altered to split the goal into wave front, wave hole and sink parts instead of deconstructing it into individual functions. The instantiation of the speculative wave rule step (4.4) has been changed to unify the rule with any available HOF rewrites. The general schematic proof corresponding to the imitated ripple synthesis plan is shown in figure 18.

Figure 18: General imitated ripple synthesis proof

$$\forall l.\forall y.((f \circ g)\ l\ y \quad = \quad M\ l\ y) \qquad 4.1$$

$$\forall l.\forall y.(f\ l\ y = M_f\ l\ y) \quad \wedge \quad \forall l.\forall y.((g\ l\ y = M_g\ l\ y) \qquad 0.1$$

Step Case

$$\forall y.(f\ t\ y = M_f\ t\ y) \vdash$$

$$\forall y.(f\ \boxed{(c\ \underline{t})}^{\uparrow}\ \lfloor y \rfloor \quad = \quad M_f\ \boxed{(c\ \underline{t})}^{\uparrow}\ \lfloor y \rfloor) \qquad 0.2\&4.2$$

$$\text{Ripple}\!\downarrow \quad = \quad \downarrow\!\text{Speculative wave rule}$$

$$\forall y.(\ \boxed{(c_1\ t\ y\ (f\ t \left\lfloor \boxed{(c_2\ t\ \underline{y})}^{\downarrow} \right\rfloor))}^{\uparrow} \quad = \quad \boxed{M_1\ t\ y\ (M_f\ t \left\lfloor \boxed{(M_2\ t\ \underline{y})}^{\downarrow} \right\rfloor)}^{\uparrow}\ ) \qquad 4.3$$

$$\boxed{\forall x.\forall y.((c_1\ t\ y\ x) = (M_1\ t\ y\ x)) \wedge \forall y.((c_2\ t\ y) = (M_2\ t\ y)) \wedge \ \underline{\forall y.((f\ t\ y) = (M_f\ t\ y))}}^{\uparrow} \qquad 0.4$$

$$\forall x.\forall y.(((c_1\ t\ y\ x) = (M_1\ t\ y\ x)) \wedge \forall y.((c_2\ t\ y) = (M_2\ t\ y)) \qquad 1.2$$

Speculative wave rule

$$\forall y.(M\ \boxed{(c\ \underline{t})}^{\uparrow}\ \lfloor y \rfloor) \quad = \quad \forall y.(\ \boxed{h_1\ t\ y\ (M_f\ t \left\lfloor \boxed{(h_2\ t\ \underline{y})}^{\downarrow} \right\rfloor)}^{\uparrow}\ ) \qquad 4.4$$

Base Case

$$\forall y.(f\ b\ y{=}hof\ \lambda x.(h_1\ t\ y\ x)\ (B\ t\ y)\ b) \qquad 0.6$$

Advantages/Disadvantages

✓ Synthesis results are found for both composed and individual functions which increases the number of solutions that may be found.

✓ There is no longer a discrepancy in how the top level and subproblems are treated.

✓ The speculative wave rule has been simplified and metavariables for every single function are no longer needed. This corresponds more closely with the results from higher-order unification.

✓ Unifying the schematic wave rule with HOF rewrites allows any available HOF to be used instead of only those which are specified as candidates.

4.8 Summary

The synthesis plan has developed from a very naïve plan through several stages to much more general, powerful and elegant one. The first two plans were changed and extended as a result of examining their flaws and capabilities independent of a concrete implementation. The penultimate of these plans has been implemented using the $\lambda Clam$ higher-order proof planner. Though the implementation of this plan was done in $\lambda Clam$ implementation should be possible in any higher-order inductive theorem prover. The final plan iteration was produced by addressing the problems that this implementation brought to light when tested on several examples. This final plan was not implemented. The next chapter will describe this implementation of the penultimate plan.

Chapter 5

Implementation

The synthesis plan discussed in section 4.6 was implemented using the λ*Clam* theorem prover.

In this chapter we discuss the workings of λ*Clam* and the changes that were made to enable it to perform the synthesis plan. We look at the changes to the representation of metavariables, the implementation of new methods, and the way in which the methods, new and old, were arranged to perform the plan.

5.1 λ*Clam*

λ*Clam* is a higher-order version of the *Clam* theorem prover (section 2.4.6). λ*Clam* implements proof planning techniques using methods and methodicals to guide proof search through heuristics like rippling.

λ*Clam* uses methodicals to specify which methods shall be applied in a proof attempt and the order in which they should be applied. For instance the top level method(`top_meth`) of λ*Clam* tries to apply various methods(see below) to the problem. At each stage the methods are tried in the same order as defined, when the first fails to apply the next one down is tried; this type of method is called a waterfall because the problem trickles down the list of methods until one applies. This waterfall method is implemented in λ*Clam* by using the `orelse_meth` methodical which takes two methods and tries the first; if it doesn't apply it tries the second. By repeatedly nesting `orelse_meth` methodicals in the second position the waterfall effect is created(see figure 19).

Figure 19: Code for λ*Clam*'s top level waterfall method

```
method_def top_meth
      (complete_meth
        (repeat_meth
          (orelse_meth all2i
          (orelse_meth taut
          (orelse_meth sym_eval
          (orelse_meth existential
          (orelse_meth (repeat_meth generalise)
                       ind_strat
          )))))
        )
      ).
```

In this chapter we will represent λ*Clam*'s code using pseudocode. The pseudocode for λ*Clam*'s top level waterfall method is shown in figure 20.

Figure 20: Pseudocode for λ*Clam*'s top level waterfall method

```
top_meth =
  REPEAT
  (
         all2i
     OR  taut
     OR  sym_eval
     OR  existential
     OR  (REPEAT generalise)
     OR  ind_strat
  )
```

Various methodicals are shown in these methods. **THEN** (`then_meth`) indicates where one method must be applied after another. **OR** (`orelse_meth`) shows where one second one is applied if the first one fails. **REPEAT** (`repeat_meth`) indicates a where the method is applied repeatedly to the goal until it fails. **PAIR** (`pair_meth`) indicates where a method which takes two subgoals and applies the first method to the first and the second method to the second. **MAP** (`map_meth`) shows where the method is applied to all sub goals. **TRY** (`try_meth`) indicates where a method application is attempted but not necessary for the continuation of the proof.

The tautology checking method `taut`, the symbolic evaluation method `sym_eval`, and the induction strategy `ind_strat` are all implemented in a similar way using `then_meth`, `orelse_meth` and `repeat_meth` methodicals.

At various points in a proof a method may create two subproblems instead of just one. The `and` method, for example, will create two separate goals corresponding to the two conjuncts and a plan will have to be found for both. To handle methods with more than one output, the methodicals `map_meth`, and `pair_meth` are provided. `map_meth` will apply a method to every subproblem and `pair_meth` will apply two different methods to the subproblems. The methodical `then_meths` is used with `pair_meth` instead of `then_meth` because a different behaviour is required. `then_meth and (pair_meth meth1 meth2)` will apply `(pair_meth meth1 meth2)` to both of the subgoals from `and` whereas `then_meths and (pair_meth meth1 meth2)` will apply `meth1` to the first and `meth2` to the second.

The two methods in figure 21 show the original structure of λ*Clam*'s induction strategy and step case. The code for these methods can be found in appendix B.1.

Figure 21: Top level induction strategy and step case

Induction:	Step Case:
`ind_strat` =	`step_case` =
`induction` **THEN**	(**REPEAT** `ripple_out_once`) **THEN**
PAIR	(**TRY** `fertilise`)
(**MAP** `id_meth`)	
(**MAP** `step_case`)	

5.2 Changes to λ*Clam*

In trying to do synthesis in λ*Clam* many changes were needed. The easiest way to alter the behaviour of λ*Clam* is to use methodicals to change the way it attempts to apply methods to the goals. These custom methods built from other methods and methodicals can be input into λ*Clam* along with the conjecture so no re-compiling of the planner is necessary.

However, the lower-level manipulation necessary to perform synthesis was not possible by simply changing the composite methods. This meant that changes to the source code for the atomic methods(the method preconditions) needed to be made.

5.2.1 Representing metavariables

Consider the conjecture

$$rowsmult\ k\ l\ =\ M\ k\ l$$

The λ*Clam* planner immediately applied symbolic evaluation and used the identity rewrite rule ($A\ =\ A$) to instantiate M to $rowsmult$. This is not the result we want so the instantiation of M needs to be restricted.

Three ways of restricting the instantiation of M were considered; *representation*, *annotation*, and *restriction*.

Representation

The first option is to explicitly replace metavariables in any conjecture with a new construct and then deal with that representation when attempting to match rewrite rules.

$$rowsmult\ k\ l\ =\ M\ k\ l$$

becomes

$$rowsmult\ k\ l\ =\ meta_M\ k\ l$$

Then when instantiation is deemed necessary $meta_M$ can be transformed back or replaced by a term which has been constructed by setting up a separate unification problem.

Annotation

The next option is to shield the metavariable from instantiation by encasing it within an annotation.

$$rowsmult\ k\ l\ =\ M\ k\ l$$

becomes

$$rowsmult\ k\ l\ =\ meta\ (M\ k\ l)$$

When instantiation is required the annotation can be removed and the metavariable instantiated as normal.

This method is simpler and more elegant than the previous one since it allows the metavariable to remain in the goal unchanged.

Restriction

The third way is to attempt to restrict the use of unification within a rewrite so that a metavariable is left alone for as long as possible. The `flex` predicate in $\lambda Clam/\lambda$Prolog allows us to determine if a term is a metavariable or not and so might be used to eliminate rewrites which instantiate a metavariable.

Metavariables Implemented

The problem with the first method (*representation*) is that it is difficult to keep the metavariables synchronised across all the different subgoals when the actual metavariables are removed from the goal. The introduction of control information into the goal in the second method (*annotation*) is undesirable but it is the most simple and elegant solution. The third method (*restriction*) doesn't alter the goal but rewriting the $\lambda Clam$ code to handle the instantiation of metavariables is difficult and complex to implement, and inefficient.

The decision was made to use *annotation* to restrict the instantiation of metavariables in the goal. In complicated examples However, the metavariable in the goal began to pick up stray variable arguments as a side effect of the $\lambda Clam$ rewriting process. To counter this a hybrid solution was implemented by representing the metavariable in the goal by a function as in *representation*. The actual metavariable is added to the hypotheses to allow the effects of instantiation to be global as in *annotation*.

$$hyp\ 1\ M \vdash$$

$$rowsmult\ k\ l\ =\ (meta2\ 1)\ k\ l$$

hyp is the wrapper function which allows M to be included in the hypotheses. $meta2$ is the representation of the metavariable in the goal, in this case indicating that the metavariable should have arity 2. Metavariables do not have specific arities but the concrete representation of the metavariable in the goal does need to have an arity appropriate to its context. When the meta-representation is replaced by the metavariable it is then free to take on extra arguments as part of its instantiation. The integer argument 1 to both hyp and $meta2$ is the index which distinguishes different metavariables and links the metavariable in the hypothesis with its representation in the goal.

When a metavariable needs to be instantiated the representation in the goal ($meta21$) is replaced with the metavariable in the hypotheses (M) and rewriting can occur as normal.

5.2.2 Method Changes

Several methods needed to be added to λ*Clam* and other existing ones altered to allow the synthesis to take place.

New methods are introduced for manipulating a meta-representation that prevents the metavariable from being instantiated. The `insmet` method introduces a meta-representation and requires the goal to be existentially quantified over a function. It removes this existential quantification and inserts the meta-representation into the goal as well as adding a new metavariable into the hypotheses. `meta_rm` removes the meta-representation and replaces it with the corresponding metavariable from the hypotheses. There are other methods which add, remove and manipulate the meta-representations as part of the synthesis process.

The existing `induction` method performed the base case before the step case but in the synthesis the step case should be done first so the `induction` method was altered (`sinduction`). Later on it became apparent that in some examples the ability of this method to perform induction over natural numbers as well as lists was interfering with the synthesis process so it was restricted to induction over lists. Currently no synthesis is possible from functions that use recursion over natural numbers. Restricting the synthesis process to induction over lists does no harm because this induction is never used in verification mode. Any function using natural number recursion will trigger a simple instantiation of the corresponding metavariable as opposed to a HOF synthesis.

Two special purpose rippling methods were implemented to help split apart the goal during rippling. `meta_wr_split` rewrites the metavariable part of the goal which will be all of the right hand side of the equation. The structure of the left hand side is copied over to the right but the functions on the left are replaced with metavariables on the right. The `meta_wr_split` method handles all the representational issues with metavariables by inserting the representations and the metavariables in the hypotheses itself. The original right hand side of the equation is equated with the new right hand side and the result is set up as a subgoal.

The `vapourise` method originally started out as the `split` method. The `split` method would split apart a goal by equating corresponding parts on either side of an equality and creating a conjunction of them. The intent was that the `split` would apply repeatedly until the desired result was found. In practice this did not happen. Sometimes `split` would fail to apply in cases where the problem was surrounded by other structure and the necessary subproblems were not formed. Also sometimes the method would apply to the wave hole, so splitting up the occurrence of the induction hypothesis and preventing fertilisation. The `vapourise` method is a more tightly constrained special purpose method than `split`, it applies the rule used in the `split` method recursively over the goal while at the same time avoiding changing the wave hole.

The `i_hof` method chooses the HOF which is to be used for the synthesis. In the subgoal created by `meta_wr_split` there will be the same metavariable on both sides of the equation. This metavariable will be replaced by the HOF chosen by the `i_hof` method. The choice of HOF made by `i_hof` attempting to instantiate the metavariable to a candidate HOF; `i_hof` currently knows about map, $foldr$, $foldl$ and $scan$. This instantiation is restricted by some side conditions. If the desired function has an arity of 1 then only map and $foldr$ are available because they don't use an accumulator otherwise $foldl$ and $scan$ are also available. map and $scan$ require the presence of a $::$ in the goal so this is also a side condition for these two candidates. `i_hof` removes all meta-representations from the goal before it starts so that the representation won't get in the way of the later instantiation of the HOFs arguments.

5.2.3 Structural Changes

The structure of the methodicals in $\lambda Clam$ is relatively simple because it is designed for verification. When we come to do synthesis however, the structure of the method application has to be more tightly constrained to allow useful backtracking. Because we want to find several results from each synthesis attempt the search tree has to be severely pruned in order to reduce the time taken to find subsequent results and to prevent discovery of the same result many times over. This pruning eliminates some results, the *mergesort* example (section 6.3.5) provides evidence of this, but without this pruning the search becomes unfeasibly large to allow backtracking.

In order to ensure that the new methods were applied and at the correct time, the method structure had to be changed. Initially most of the changes were the addition of the new methods (section 5.2.2) to the existing methodical structure. New methods were added to the top level waterfall to perform the tasks of introducing and removing the annotation and selecting the HOF. New rewrite rules, in the form of custom methods, also had to be introduced into the ripple method to allow the rippling process to split the goal as required. This method structure is shown in appendix B.1.

This structure worked on several examples. However, some examples failed, through running out of time or space. When instantiation of metavariables was done out of order a great deal of backtracking was needed. To counter this the methodical structure of the induction method was changed to specify exactly which methods should be done to each subgoal in what sequence.

This is necessary because the partial instantiation of the metavariable in one part of the proof constrains the instantiation in the next. Without a semi-strict ordering the instantiation would occur when the planner doesn't have enough information to make a useful instantiation giving rise to needless backtracking. Unless instantiations of the metavariables in the subproblems are made first then the there is not enough structure for an intelligent higher-order function instantiation to be made. This ordering could be encoded in heuristic preconditions to methods and part of a less restricted method structure. This was not done because the amount of backtracking in the synthesis is an issue and a less restrictive method structure would give rise to extra backtracking. The final method structure can be found in appendix B.2.

Top Level

The method in figure 22 shows the structure of the top level waterfall for the λ*Clam* synthesis with the `isolate` and `remove_meta` methods shown at the side.

Figure 22: Top level waterfall and auxiliary methods for synthesis

Waterfall:

```
top_meth=
  REPEAT
  (
          ins_met
      OR  i_hof
      OR  isolate
      OR  all2i
      OR  taut
      OR  sym_eval
      OR  existential
      OR  (REPEAT generalise)
      OR  syn_strat
      OR  remove_meta
  )
```

```
isolate =
  (TRY REPEAT all_i) THEN
  exp_and

remove_meta =
  (TRY REPEAT all_i) THEN
  meta_rm
```

The top level method was changed very little, though some additions were made to the waterfall method. The methods `isolate` and `remove_meta` are made up of methodicals and restrict the simplification of goals. The method `syn_strat` replaces the induction strategy `ind_strat` with a restricted version to perform synthesis.

`isolate` is a method for splitting apart conjunctions of goals. Firstly any universal quantification at the top level is removed using `all_i` and then the conjunction of goals is split up by repeated application of `and_i`.

`remove_meta` is for removing the meta-representation that appears in the goal. At the top level if the synthesis attempt fails then the metavariable needs to be removed to allow instantiation to take place. The `remove_meta` method removes any quantification at the top level and then uses the `meta_rm` method to replace the meta-representation with the actual metavariable that it represents.

Synthesis Strategy

The diagram in figure 23 shows the structure of the synthesis strategy method which replaces the induction strategy in the original method structure.

Figure 23: The new induction strategy and step case methods for synthesis

Induction:

```
syn_strat=
   sinduction THEN
   (   PAIR
       (   syn_step THEN
           (   PAIR
               (   recurse)
               (   hof)
           )
       )
       (   syn_top)
   )
```

Step Case:

```
syn_step=
   setup_ripple THEN
   recripple THEN
   (
       (
           post_ripple THEN
           (TRY fertilise)
       )
       OR
       id_meth
   )
```

`syn_strat` is very different from and much more complicated than the `ind_strat` method it replaces. `ind_strat` simply applies the induction method and specifies methods for dealing with the step and base cases. The step case method in the synthesis case `syn_step` produces several extra subgoals and the `syn_strat` method also specifies how to deal with them. The `syn_step` method produces a subgoal containing the expressions which may be available for further synthesis and these are dealt with by the `recurse` method. The step-case also produces a goal which represents the schematic wave rule that is used to perform the speculative ripple step, this goal is dealt with by the `hof` method.

The `syn_strat` method uses the `complete_meth` methodical which ensures that the goal is completely planned before moving on. `complete_meth` was used around the step-case to ensure that the recursive synthesis and the choice of HOF is performed before the base case. `complete_meth` was also used to ensure that the recursive synthesis is done before the choice of HOF. This use of `complete_meth` has drawbacks however; because the goal must be completely planned by `syn_strat` the base case and residue goals cannot simply drop out and be dealt with by the `top_level` method they must be explicitly dealt with within the `syn_strat` method by `syn_top`. In future implementations, to avoid the necessity for this rather cumbersome structure, we could implement a custom `complete_meth` which would allow the required goal to drop out of it while insisting that all others are dealt with.

In the verification the base case from the induction is dealt with using `id_meth` which does nothing, The base case is untouched and remains after returning from the `ind_strat` method to be tackled by the top level method. The synthesis instead specifies explicitly the method `syn_top` to be used for planning the base case. `syn_top` is similar to the top level method but is more focused to the context of use.

The `syn_step` method is based on the `step_case` method in the verification. The `step_case` method sets up a rippling goal, performs rippling and removes the rippling annotations and applies fertilisation. The synthesis method is complicated by the fact that the speculative ripple step will produce a subgoal which must be allowed to escape from the method without undergoing the fertilisation step. The rippling in the verification case is taken care of by the repeating the ripple method: in the synthesis, because some ripple methods produce more than one goal, an explicitly recursive ripple method `recripple` is used instead. The rippling methods are shown in figure 24.

Figure 24: Rippling methods for synthesis

Single ripple:

```
ripple_inout=
          vapourise
    OR    ripple_inout_once
    OR    meta_wr_split
```

Recursively ripple:

```
recripple=
   (
            ripple_inout
      OR    (PAIR
               ripple_inout
               id_meth
            )
   ) THEN
   (
            recripple
      OR    id_meth
   )
```

The `recripple` method applies the method `ripple_inout` method and then applies itself to the result. This duplicates the effect of the `repeat_meth` methodical but `recripple` also takes into account the fact that `ripple_inout` may produce two different types of goal, a rippling goal and the speculated lemma produced by the speculative ripple step. The `repeat_meth` methodical cannot be used to achieve this effect because the production of two goals bars further application of the method. `recripple` continues to apply itself to one type of goal while allowing the other to pass through unaltered.

`ripple_inout` applies one of three rippling methods, `vapourise`, the standard ripple `ripple_inout_once`, or `meta_wr_split`. `ripple_inout_once` is sufficient for verification but for the synthesis `vapourise` and `meta_wr_split` are also needed. `meta_wr_split` works only on goals with metavariables so will only apply in the synthesis setting. `vapourise` will generalise the goals with similar structures by equating their various parts and, though probably not particularly useful, might be used in the verification as well as synthesis. `vapourise` will generalise away from every single argument position in the goal. It also sets up equalities of the arguments as components of the new goal instead of removing them.

Other methods used in the synthesis strategy are shown in figure 25. `recurse`

Figure 25: Other synthesis methods

Recursively synthesise:

```
recurse =
    isolate THEN
    (MAP syn_top)
```

HOF choice:

```
hof =
    (TRY REPEAT all_i) THEN
    i_hof THEN
    syn_top
```

simply uses the `isolate` method to split up the goals and `syn_top` to recursively apply the synthesis process to each.

`hof` removes any quantification around the goal using `all_i` and then applies `i_hof` the method for choosing a HOF. The result is then planned using `syn_top` to ensure the resulting metavariables in the goal are instantiated.

5.3 Plan Generality

The implemented plan has several drawbacks which are described in section 4.6; these are mostly concerned with the number of generated results but some specify problems relating to the set of examples the plan will work on.

The induction scheme restriction is the most obvious problem which impairs the generality of the plan. Since the induction scheme is restricted to single step induction the plan can only work on those examples which use the corresponding form of recursion. The induction restriction is matched by the choice of HOF; the choice of HOF is limited to *map*, *foldr*, *foldl*, and *scan* all of which require only single step induction. This restriction on the capabilities of the plan could be addressed in a future implementation.

The class of problem the plan was designed to be able to handle are those closely linked to the compiler's target HOFs (*map* and *fold*). Although the plan has been extended to handle *foldl* and *scan* it is possible that the synthesis plan will break down if the examples become too distant from this set. The application of the methods in the synthesis plan are more tightly constrained than in verification. This is partly to ensure that the many metavariables are instantiated in the right order and partly to simplify backtracking so we can generate alternative results in a reasonable time.

Within the synthesis plan there is some opportunity for other unrelated methods to intervene. The restrictions on the induction plan make it a bad idea for methods to intervene during rippling and before certain metavariables are instantiated. Before the application of induction there is an opportunity for other methods to apply. This is also true in the residue subproblems generated to enable recursive synthesis as well as the top level problem.

The set of examples the plan was designed to handle can be represented by the following schematic equations which correspond to the recursive case of the function:

$$\begin{aligned} func_1\ h :: t &= f\ h\ (func_1\ (g\ h))\ t \\ \forall y.(func_2\ y\ h :: t &= f\ y\ h\ (func_2\ (g\ y\ h)\ t)) \end{aligned}$$

This class of problem includes functions which may require a HOF using two function arguments (f and g) or a form of more general *scan* which does not require a ::. The present plan cannot choose these HOFs. However, the plan is set up to handle this type of problem. The `i_hof` method could be extended with these functions without needing any other alterations to the implementation. For example, there is a form of tail recursive *map*, (*mapl*) which could be added instances of which currently will produce only a $foldl$.

$$
\begin{aligned}
mapl\ f\ l &= mapleft\ f\ []\ l \\
mapleft\ f\ [] &= b \\
mapleft\ f\ b\ h :: t &= mapleft\ f\ b{+\!\!+}[f\ h]\ t
\end{aligned}
$$

Programs which contain a more complicated structure will be problematic for the synthesis plan to handle. If we look at the class of function corresponding to the $filter$ (see figure 2) HOF the presence of an if-then-else with recursive calls on both branches presents the plan with a difficult obstacle requiring a casesplit:

$$func\ h :: t \quad = \quad if\ (c\ h)\ then\ (f\ h\ (func\ t))\ else\ (func\ t)$$

The two recursive calls in the step case may cause the synthesis plan to fail on this type of example. An alternative implementation for the $filter$ type problem is:

$$
\begin{aligned}
func\ h :: t &= ifh\ (c\ h)\ h\ (func\ t) \\
where:\quad ifh\ true\ h\ x &= f\ h\ x \\
ifh\ false\ h\ x &= x
\end{aligned}
$$

Where the repeated recursion is abstracted out. However, this is of little comfort since we cannot rely on the user to be so accommodating as to phrase all the relevant functions in this way and the synthesis plan itself cannot perform this transformation.

The schematic proofs in chapter 4 show how we expect the synthesis proofs to progress. If these schematic proofs were the only route to a successful synthesis we could simply instantiate the schematic proofs with the problem and obtain a result. However, the proof planning framework provides a more powerful and flexible structure for discovering a proof.

Proof planning allows the synthesis to deal with unexpected occurrences. It also extends the class of programs that the synthesis can handle by allowing methods to apply before the synthesis process so that programs with different structure than we might expect could be made amenable to synthesis.

Proof planning also provides an easily extensible framework for the synthesis. For example at the moment the synthesis plan will only do one step induction on lists However, the proof planning framework allows different induction schemes to be used which would allow the synthesis plan to produce results for a larger range of programs and to discover expressions involving a larger number of HOFs.

5.4 Summary

The synthesis plan (section 4.6) was implemented in $\lambda Clam$. Many alterations had to be made to prevent the metavariable from being instantiated too quickly and to guide it towards the desired results. Methods were added for synthesis and the method structuring was changed. Although some of the changes weren't very elegantly implemented the result is a working prototype synthesis system. We will evaluate this prototype in the next chapter.

Chapter 6

Evaluation

6.1 Introduction

We tried the implementation (chapter 5) of the synthesis plan (section 4.6) out on a number of examples. Each example typically yields multiple results and we look at how each of these are reached through the search space. The times taken to discover each result vary widely and we look at the similarities and differences in the times taken and what this tells us about the search space. Lastly we look at an example that throws up some interesting results and suggests the next iteration of the synthesis plan(as described in section 4.7).

6.2 Synthesis Attempts and Arities

In the execution of a synthesis problem we would expect most of the work to be done in the synthesis process. In all the examples there will be times where a synthesis will be attempted but there is no synthesis possible. These occasions may significantly add to the time taken to find results and to backtrack over the proof tree.

The occurrences of these dead-end synthesis attempts are closely linked to the arity of the function that synthesis is being performed on. Each additional argument to a function involved in a synthesis problem adds an extra synthesis attempt to be explored. Each of these additional attempts may yield no result but increase the time it takes to traverse the search space.

The Anatomy of Synthesis Attempts

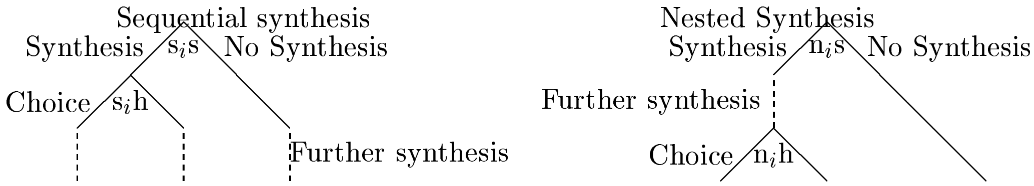
We attempt to generate all possible HOFsyntheses. The results with the most HOFs are generated first because the synthesis is always tried before simple instantiation. The final result is a simple identity and is generated after all the other possible alternatives have been explored. The time for this last result is a good indication of the time taken to traverse the entire search space.

In each synthesis attempt there are two main choice points which affect the result of the synthesis. One is whether to attempt synthesis at all and the other is the choice of HOF that is made at the end. Sometimes there is only one choice of HOF that can be made so it is always chosen.

In sequential synthesis the syntheses are separate so one synthesis is complete before the other is begun. In nested synthesis the inner synthesis is completely done before the previous one is finished. The trees for the two types of synthesis are illustrated in figure 26.

We annotate the nodes of nested and sequential synthesis with s_is or n_is, to indicate the synthesis choice point, and s_ih or n_ih to indicate the HOF choice point. The subscript i is to distinguish between different syntheses.

Figure 26: Synthesis trees for sequential and nested synthesis



We will use this notation when examining the trees of the *squs2d*(section 6.3.3),

$mmult$(section 6.3.4) and $mergesort$(section 6.3.5)

6.3 Example Results

The results for seven examples are given below. The maximum number of HOFs found is given along with the number of results and the time taken to traverse the entire search space. The definitions of the functions and the complete results can be found in appendix D.

Function	HOFs	# Results	Total Time(secs)
rev	1 $foldl$	2	199
$isums$	1 $scan$	2	1527
$sum2d$	2 $foldr$s	3	239
$subset$	2 $foldr$s	3	2221
$mergesort$	2 $foldr$s	4	417
$squs2d$	2 $maps$	7	1201
$mmult$	2 $maps$ + 1 $foldr$	14	10899

For rev and $presums$, synthesis finds either the single HOF or the original function. Note that synthesising $scan$ involves duplicated term structure instantiation and synthesis for both occurrences of $(f\ a\ h)$ in:

$$scan\ f\ a\ [] \ = \ []$$

$$scan\ f\ a\ (h :: t) \ = \ (f\ a\ h) :: (scan\ f\ (f\ a\ h)\ a\ t)$$

incurring additional costs.

For $sum2d$ and $subset$, synthesis finds the top level $foldr$, the top and inner $foldr$ or the original function. However, where $sum2d$ involves an arity 1 function calling an arity 2 function, $subset$ involves nested arity 2 functions, which both have more term structure than $sum2d$s components.

$mergesort$, $sum2d$ and $subset$ all have two $foldr$ sites. Here, $mergesort$ appears anomalous as four solutions are found, compared with only three for $sum2d$ and $subset$. However, $mergesort$ is defined in terms of two composed functions, so an additional result is found for its right-hand side as well as for the original top level call. Despite more solutions being found, synthesis from $mergesort$ is still considerably faster than from $subset$, again because it consists of nested calls to arity 1 functions.

Finally, $squs2d$ and $mmult$ both involve map as primary HOFs, so equivalent $foldr$ solutions are also found. $squs2d$ consists of arity 1 functions whereas $mmult$ is built solely from arity 2 functions.

6.3.1 Benefits of Parallelisation

The benefits of parallelisation for the test examples described here will probably be small. The simple examples involving one HOF for example rev and $isums$ and even those involving two like $sum2d$ and $squs2d$ are unlikely to benefit much from parallelisation because the processing costs are already quite small.

For the larger examples like the matrix multiplication example, $mmult$, the benefits will be greater. The speed up gained by parallelisation of the $mmult$ example is discussed in [MSBK01]. Hamdan also discusses the speed up from parallelisation and compares the effect of different combinations of map and $fold$ for $mmult$ and $mergesort$ in chapter 6 of his thesis [Ham00]. In the $mmult$ example running with the HOFs implemented in parallel the best speed up was found using three nested $fold$s running using 65 processors on a Fujitsu AP1000. The next best was one outer $fold$ but with two inner maps again using 65 processors.

6.3.2 Verifying the Results

For the above results we ran the synthesised expression through the system in verification mode to check that it was a valid transformation. The simpler transformations, involving one HOF, such as rev and $isums$, were very easy to verify with $\lambda Clam$. The more complicated transformations like $sum2d$ and $squs2d$ required a minor alteration to the verification plan to allow $\lambda Clam$ to perform nested induction. The synthesis of the large examples in $\lambda Clam$ was more problematic: both *mergesort* and *mmult* caused the theorem prover to produce an error and exit before the proof was finished. However, when the conjectures were split up, the theorem prover was able to produce proofs for each individual component. This indicates that the system may contain a memory leak or simply runs out of resources when presented with large examples.

In the *mmult* example proofs were found for the following conjectures:

$$\begin{aligned} mmult &= \lambda x.\lambda y.(map\ \lambda z.(rowmult\ z\ (transpose\ y))\ x) \\ rowmult &= \lambda x.\lambda y.(map\ \lambda u.(dotprod\ x\ u)\ y) \\ transpose &= \lambda x.(foldr\ \lambda y.\lambda z.(dist\ y\ z)\ []\ x) \end{aligned}$$

By using the *rowmult* and *transpose* conjectures as rewrite rules the *mmult* conjecture obtain the original result from the *mmult* conjecture:

$$mmult = \lambda x.\lambda y.(map\ \lambda z.(map\ \lambda u.(dotprod\ z\ u)\ (foldr\ \lambda u.\lambda v.(dist\ u\ v)\ []\ y))\ x)$$

So the synthesis result is proven correct though the theorem prover's inability to prove the composed conjecture remains a subject for investigation.

We now consider *squs2d*, *mmult* and *mergesort* in more detail.

6.3.3 Squares 2D

The *squs2d* function squares every element in a 2 dimensional list(a list of lists). It does this by applying *squares* function to every list in the list. The definitions of *squs2d* and *squares* are as follows:

$$squs2d\ [] = [] \tag{36}$$

$$squs2d\ (h :: t) = (squares\ h) :: (squs2d\ t) \tag{37}$$

$$squares\ [] = [] \tag{38}$$

$$squares\ (h :: t) = (h * h) :: (squares\ t) \tag{39}$$

Both $squs2d$ and $squares$ can be represented by maps so we predict that the first result found would contain two nested maps. Subsequent results should contain combinations of maps, $fold$s and the original functions.

Results

The first synthesis result found is:

$$squs2d \Rightarrow \lambda x.(map\ \lambda y.(map\ \lambda z.(z * z)\ y)\ x)$$

This contains two nested maps as predicted. map can also be represented as a $foldr$ and at the different HOF sites there is a choice of whether to perform synthesis at all. This means that many alternative results are found on backtracking.

The table of results for $squs2d$ is below. The two synthesis sites($squs2d$ and $squares$) are represented by the higher order function synthesised or the name of the original function if there was no synthesis. The time for each result is given(in seconds) and the time difference(in seconds) between consecutive results is shown in the rightmost column. The full results can be found in appendix D.

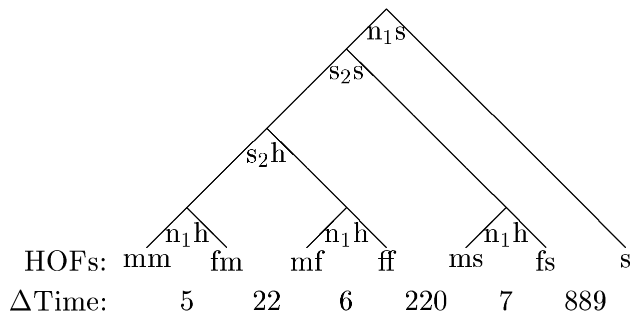
HOFs	Time(secs)	ΔTime(secs)
$(map(map))$	52	-
$(foldr(map))$	57	5
$(map(foldr))$	79	22
$(foldr(foldr))$	85	6
$(map(squares))$	305	220
$(foldr(squares))$	312	7
$squs2d$	1201	889

The last entry doesn't involve any synthesis but simply constructs the transformation rule which substitutes $squs2d$ for itself. This last synthesis takes substantially greater time to be found than the pervious ones and is entirely trivial. This synthesis result cannot be eliminated because it represents the base case of the synthesis recursion. In the previous results the function $squares$ is a result of this trivial synthesis also synthesis is attempted on the function $*$ and the trivial instantiation halts the synthesis recursion. The large time for the last trivial synthesis is the result of backtracking over the large search tree while searching for any other possible result.

Analysis of Results

The diagram in figure 27 illustrates the search space for the *squs2d* example. The choice points are marked with an indication of the function synthesis it belongs to and if it is a synthesis choice or a HOF choice.

Figure 27: The synthesis tree for *squs2d*



In the search tree there are two syntheses, one nested within the other. The outer one is marked n_1 to indicate that another synthesis is nested within it. The inner one is marked s_2 because although it is not done in sequence with another there are no syntheses nested within it.

The synthesis of $squ2d(n_1)$ is attempted first and the synthesis of *squares* is nested within that. The n_1 synthesis choice point is at the root of the tree and n_1 HOF choice point is near the leaves of the tree. Both of the s_2 choice points are in between.

Consecutive Results

If we look at the times for consecutive results we can see that the time taken between results is linked to the traversal of the tree structure.

We see similar results(5, 6 and 7 seconds) for every second value each of which corresponds to the traversal of the n_1h subtree. The traversal of the n_1h subtree requires only the backtracking over a single HOF choice point, $\mathrm{n_1h}$, so we would expect it to be very quick. $\mathrm{n_1h}$ is the choice of *map* or *foldr* for the top level synthesis.

We also see that the further up the tree backtracking takes place, the longer the interval between results. The traversal of the $\mathrm{s_2h}$ subtree takes longer(22 seconds) as the backtracking over two HOF choice points($\mathrm{n_1h}$ and $\mathrm{s_2h}$) is required. The $\mathrm{s_2s}$ subtree takes longer still(220 seconds) because backtracking to a synthesis choice point will take longer. Finally backtracking over the $\mathrm{n_1s}$ takes the longest amount of time(889 seconds) because we backtrack over two synthesis choice points($\mathrm{n_1s}$ and $\mathrm{s_2s}$).

6.3.4 Matrix Multiplication

Matrix multiplication has the potential for the introduction of several HOFs. The definition of the matrix multiplication function *mmult* and its subfunctions are as follows:

$$\begin{aligned}
mmult\ m_1\ m_2 &= (rowsmult\ m_1)\ (transpose\ m_2) && (40)\\
rowsmult\ []\ cols &= [] && (41)\\
rowsmult\ (row :: rows)\ cols &= (rowmult\ row\ cols) :: (rowsmult\ rows\ cols) && (42)\\
rowmult\ row\ [] &= [] && (43)\\
rowmult\ row\ (col :: cols) &= (dotprod\ row\ col) :: (rowmult\ row\ cols) && (44)\\
dotprod\ []\ [] &= 0 && (45)\\
dotprod\ (h_1 :: t_1)\ (h_2 :: t_2) &= (h_1 * h_2) + (dotprod\ t_1\ t_2) && (46)\\
transpose\ [] &= [] && (47)\\
transpose\ (row :: rows) &= dist\ row\ (transpose\ rows) && (48)\\
dist\ []\ l &= [] && (49)\\
dist\ (h_1 :: t_1)\ (h_2 :: t_2) &= (h_1 :: h_2) :: (dist\ t_1\ t_2) && (50)
\end{aligned}$$

$$dist\ (h_1 :: t_1)\ [] \quad = \quad [h_1] :: (dist\ t_1\ []) \tag{51}$$

The functions *dotprod*, and *dist* are not directly involved in the synthesis because HOF expressions for these functions cannot be synthesised. The *dotprod* function requires simultaneous induction which the synthesis process cannot handle at the moment. The *dotprod* function could be represented by a $zip/foldr$ expression:

$$dotprod\ l1\ l2 = foldr\ \lambda x.\lambda y.(x + y)\ []\ (zip\ \lambda x.\lambda y.(x * y))$$

The synthesis of this expression is also problematic since it requires the synthesis of two HOFs from one function. Syntheses of both *dotprod* and *dist* are attempted by the current synthesis implementation in each case no HOF is found and the corresponding metavariables are instantiated to the original functions.

The *mmult* function is defined in terms of the two functions *rowsmult* and *transpose* both of which contain potential HOFs. The *rowsmult* function contains the potential for the introduction of two *map*s; one for *rowsmult* itself and one for *rowmult*, a function it is defined in terms of. *transpose* potentially contains one $foldr$.

The combination of these three potential HOFs gives rise to twelve combinations of results. The result we predict to come first is the synthesis of two *map*s and a $foldr$.

Results

The first synthesis result found is:

$$mmult \Rightarrow \lambda x.\lambda y.(map\ \lambda z.(map\ \lambda u.(dotprod\ z\ u)\ (foldr\ \lambda u.\lambda v.(dist\ u\ v)\ []\ y))\ x)$$

The HOFs we predicted are found immediately. The table of results for the matrix multiplication is below.

The three synthesis sites(the functions *rowsmult*, *rowmult* and *transpose*) are represented on the left by the HOFsynthesised or the name of the original function if there was no synthesis. The time for each result is given(in seconds) and the time difference(in seconds) between consecutive results is shown in the rightmost column. The full results can be found in appendix D.

HOFs	Time(secs)	ΔTime(secs)
$(map\,(map)(foldr))$	103	-
$(foldr(map)(foldr))$	113	10
$(map\,(foldr)(foldr))$	157	44
$(foldr(foldr)(foldr))$	167	10
$(map\,(rowmult)(foldr))$	435	268
$(foldr(rowmult)(foldr))$	445	10
$(map\,(map)(transpose))$	894	449
$(foldr(map)(transpose))$	904	10
$(map\,(foldr)(transpose))$	943	39
$(foldr(foldr)(transpose))$	953	10
$(map\,(rowmult)(transpose))$	1216	263
$(foldr(rowmult)(transpose))$	1226	10
$(rowsmult\ (transpose))$	10894	9668
$mmult$	10899	5

The last two entries don't involve any synthesis but simply construct the transformation rules which substitute *mmult* for its definition and itself.

$$mmult \Rightarrow \lambda x.\lambda y.(rowsmult\ x\ (transpose\ y))$$

$$mmult \Rightarrow \lambda x.\lambda y.(mmult\ x\ y)$$

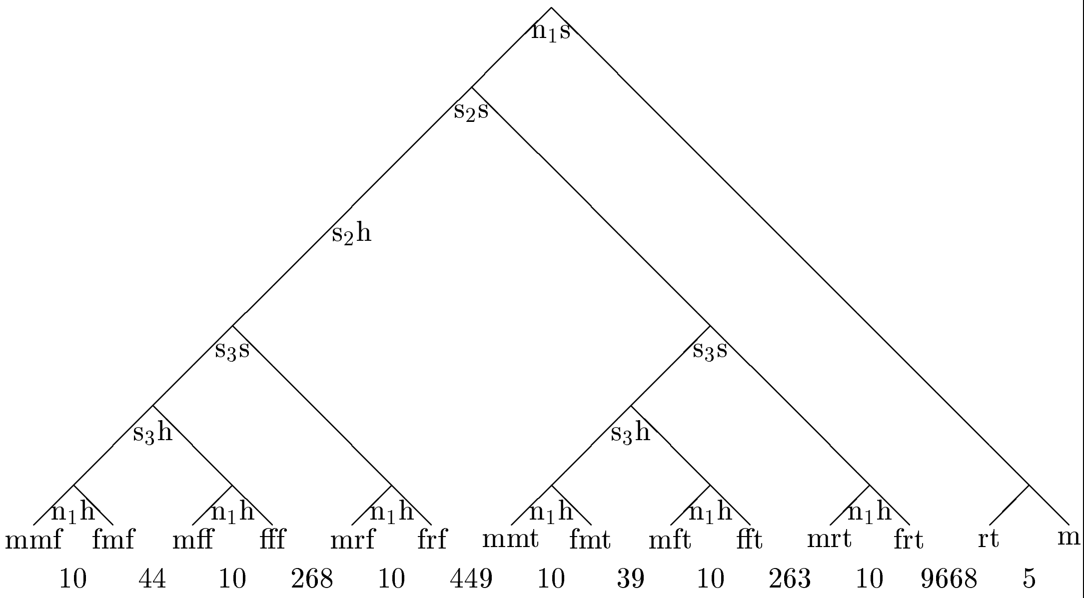
Analysis of Results

The diagram in figure 28 illustrates the search space for the *mmult* example. The choice points are marked with an indication of the function synthesis it belongs to, the arrangement and if it is a synthesis choice or a HOF choice.

There are two different arrangements of synthesis attempts in the search tree, sequential and nested.

The synthesis of *rowsmult*(n_1) is attempted first. The synthesis of *transpose*(s_2) is nested within that because the transpose function occupies an argument position within *rowsmult*. The *rowmult*(s_3) synthesis is completely separate from the *transpose*

Figure 28: The synthesis tree for *mmult*



synthesis so they are performed sequentially, However, *rowmult* is a subfunction of *rowsmult* so its synthesis is also nested within that of *rowsmult*.

We can see the HOF choice points for the first(n_1h) and third(s_3h) syntheses near the leaves of the tree. The choice at these points is between the HOFs *map* and *foldr*. The HOF choice point for the second(s_3h) synthesis is closer to the root of the tree. There is actually no choice to be made here because *foldr* is the only HOF that can be used.

Consecutive Results

If we look at the differences between the times for consecutive results we can clearly see that the time taken between results is linked to the traversal of the tree structure.

Every second value we get for the time difference is 10 seconds. This corresponds to backtracking over a single HOF choice point, n_1h, in the the search tree. This choice point is the choice of *map* or *foldr* for the top level synthesis.

The traversal of the s_3h subtrees produces results of 44 and 39 seconds. These very similar values correspond to the backtracking over two HOF choice points, n_1h and s_3h.

We can also see the same sort of similarities between the values 268 and 263 seconds which both correspond to the traversal of s_3s subtrees. In this subtree backtracking takes place over the two HOF choice points and a synthesis choice point.

Comparing *mmult* and *squs2d*

The s_3s subtree in the *mmult* example and the s_2s subtree in the *squs2d* example both contain the synthesis and HOF choice points for a *map* synthesis and the HOF choice point for another *map*. If we compare the results we can see that there are similarities between the two examples.

In both *mmult* and *squs2d* we see very short times for the traversal of the n_1h subtree(about 6 seconds for *squs2d* and 10 seconds for *mmult*). The traversal of the s_ih[1] subtrees are also similar in each example(22 seconds in *squs2d* and about 40 seconds in *mmult*). Finally we see the same sort of similarity in the s_is subtrees(220 seconds in the *squs2d* and about 265 seconds for *mmult*).

6.3.5 Mergesort

The *mergesort* function is defined in terms of *mergeall* and *sortall*:

$$mergesort\ x = mergeall\ (sortall\ x) \tag{52}$$

$$mergeall\ [] = [] \tag{53}$$

$$mergeall\ (h :: t) = merge\ h\ (mergeall\ t) \tag{54}$$

$$sortall\ [] = [] \tag{55}$$

$$sortall\ (h :: t) = (sort\ h) :: (sortall\ t) \tag{56}$$

$$sort\ [] = [] \tag{57}$$

$$sort\ (h :: t) = insert\ h\ (sort\ t) \tag{58}$$

[1] $_i$ is 3 in the *mmult* example but 2 in the *squs2d* example.

Synthesis attempts are also made on all the functions used by mergesort such as $merge$ and $insert$ but the synthesis process was unable to find a HOF containing expression for these and the corresponding metavariables were instantiated to the original function.

$mergeall$ contains one possible $foldr$ where as $sortall$ contains the possibility of one map in itself and one $foldr$ in the subfunction $sort$. The result we predict for this example contains two $foldr$s and a map.

Results

The first result we obtain is:

$$\lambda x.(foldr\ \lambda y.\lambda z.(merge\ (foldr\ \lambda u.\lambda v.(insert\ u\ v)\ []\ y)\ z)\ []\ x)$$

This result is different from our predicted result. We expect the first result to give us three HOFs, one map and two $foldr$s, However, we see only two $foldr$s.

Comparing the first and second results provides a clue to what is happening in this example:

$$\begin{array}{lll}
\lambda x.(foldr\ \lambda y.\lambda z.(merge\ (& foldr\ \lambda u.\lambda v.(insert\ u\ v)\ [] & y)\ z)\ []\ x) \\
\lambda x.(foldr\ \lambda y.\lambda z.(merge\ (& sortm & y)\ z)\ []\ x)
\end{array}$$

We can see that the second $foldr$ in the first example represents the $sort$ function.

We predicted a map and a $foldr$ for the $sortall$ and $mergeall$ functions respectively instead all we see is a single $foldr$. The second result again gives us a clue to what this represents.

$$\lambda x.(foldr\ \lambda y.\lambda z.(merge\ (sortm\ y)\ z)\ []\ x)$$

We can see that the first HOF in the first result is a $foldr$ with the composition of $merge$ and $sort$ as its argument function. The synthesis of the $sortall$ and $mergeall$ functions has become intertwined and produced a single $foldr$.

What has happened is that the synthesis process has unexpectedly exploited a distributivity property. Normally backtracking would eventually find the unfused version later on but, the rigid structure we put in place to allow multiple results to be found in a reasonable time prevents this from happening (see chapter 5).

$$\forall l.\forall b.(foldr\ \lambda x.\lambda y.(f\ x\ y)\ b\ (map\ \lambda x.(g\ x)\ l) = foldr\ \lambda x.\lambda y.(f\ (g\ x)\ y)\ b\ l)$$

This distributivity property is described in more detail in section 3.3.2. We can see that the $foldr$ function argument on the right hand side:

$$\lambda x.\lambda y.(f\ (g\ x)\ y)$$

is similar in structure to the $foldr$ function argument in the $mergesort$ result:

$$\lambda y.\lambda z.(merge\ (sortm\ y)\ z)$$

So the synthesis process produces a result for $mergeall$ and $sortall$ functions combined which has the $sort$ function as a subfunction. The times for each result is given(in seconds) and the time difference(in seconds) between consecutive results is shown in the rightmost column.

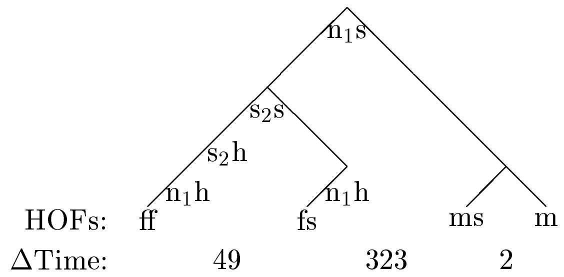
HOFs	Time(secs)	ΔTime(secs)
$(foldr(\quad foldr \quad))$	53	-
$(foldr(\quad sort \quad))$	102	49
$(mergeall(\ sortall))$	415	323
$mergesort$	417	2

As with the $mmult$ example the last two entries don't involve any synthesis but simply construct the transformation rules which substitute $mergesort$ for its definition and itself.

$$mergesort \Rightarrow \lambda x.(mergeall\ (sortall\ x))$$

$$mergesort \Rightarrow \lambda x.(mergesort\ x)$$

Figure 29: The synthesis tree for *mergesort*



Analysis of Results

The diagram in figure 29 illustrates the search space for the *mergesort* example.

The merged synthesis of *mergeall* and *sortall*(n_1) is attempted first but the synthesis of the subfunction *sort*(s_2) is nested within that. Both the merged synthesis and the *sort* can only be represented by the HOF *foldr* so there is no choice to be made at either the n_1h and the s_2h HOF choice points.

The times for the *mergesort* example do not really tell us much because there are so few of them. It is difficult to compare them with *mmult* because *mergesort* contains arity one functions and *mmult* contains arity two functions.

We can see that the backtracking over two HOF choice points and one synthesis choice point takes significantly less time than backtracking over a single HOF and two synthesis choice points. The actual times 49 and 323 seconds seem similar to results found for the *mmult* (44 and 268 seconds) example. However, upon closer examination we can see that these results relate to different positions in the search. The 49 second result for *mergesort* is in a similar position to the 268 second result for *mmult*. This does further indicate that arity two functions take longer than similar arity one functions. The 323 second result has no similar position in the *mmult* example because there are two sequential inner syntheses in *mmult* and only one in *mergesort*.

Revising the Synthesis Plan

The combined result obtained is valid and it is interesting that the synthesis process will find this type of result. Unfortunately the steps that produced this result also prevented the synthesis process from finding the predicted result.

The results from the *mergesort* example informed the development of the synthesis plan prompting the final revision described in section 4.7. The *mergesort* problem also brought to light a flaw in the existing synthesis plan which was an inconsistency between the way top level and nested goals were treated. In nested goals composed functions were always split up into separate functions whereas the top level goals were never split. To allow both types of results to be discovered and to treat top level goals to be treated the same the following changes were made.

- Addition of a method to split apart and treat composed functions as separate problems.

- Change to the method which constructs nested subproblems to allow the synthesis of composed functions.

The new synthesis plan introduces a new kind of choice point into the search tree. When presented with a composition there is a choice of whether to split it up or not. If it is not split then the synthesis proceeds as normal merging the syntheses if they can be merged or performing them nested if not. If it is split then the separate problems will be done sequentially.

6.4 Summary

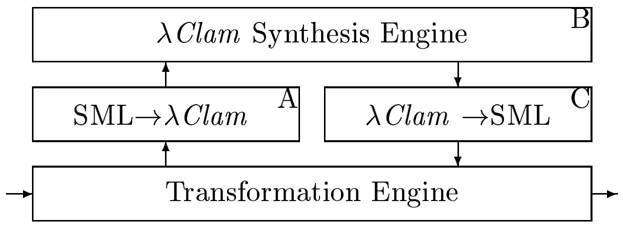
The implemented plan has been shown to find several results for several example including real world implementations of algorithms (*mmult* (section 6.3.4) and *mergesort* (section 6.3.5)). This evaluation brings to light some problems with the implementation as it stands and these have suggested a possible further iteration for the suggested new plan described in section 4.7. The transforms found by the implementation show that HOFs can be found in programs. However, to translate these HOFs into parallel behaviour the transformations need to be translated back into SML to make them available to the compiler. The next chapter details the translation modules meant to link the synthesis system with the parallelising compiler.

Chapter 7

System Integration

The transformation engine in the compiler manipulates Standard ML(SML) to produce a program optimised for a parallel architecture. It does this by using algorithmic skeletons to create efficient parallel implementations for various applications of HOFs such as *map* and *foldr*. There may be sites where the program has not been written using these HOFs but where equivalent program structure which does use HOFs can be synthesised. This synthesis can be performed by the synthesis engine in $\lambda Clam$. (Figure 30B)

Figure 30: The Transformation System



For the synthesis to take place the program structure must be converted from SML into the syntax of $\lambda Clam$ so that the synthesis engine can work on it. Also once a transformation rule has been synthesised it must be translated from the $\lambda Clam$ syntax back into SML. This allows it to be used by the transformation engine to change the program into something which may be implemented more efficiently using algorithmic skeletons.

7.1 Notations

A subset of SML was assumed as the input notation. This was converted into a representation that can be read into λ*Clam* using its in built library mechanism. Although there are differences between these two representations the match is close enough that the differences can be either ignored or overcome.

7.1.1 SML subset

The SML used for the translation is a subset of the syntax of the bare language found in [HMM86]. The constructs that are not included in our SML subset are exception handlers, declaration sequences, and layered patterns(see Figure 31).

There are certain base constructs which are assumed by the translator to be defined in the input SML. These include the definitions of lists and constructors, and operations on lists, the definitions of booleans as well as the definition of the standard types and operations on them.

7.1.2 λ*Clam* Notation

The λ*Clam* syntax which is the target for the translation of SML is the syntax of λ*Clam*'s library files. However, the translation from λ*Clam* to SML uses λ*Clam*'s internal notation as source and λ*Clam* itself is used as the translation engine.

Library Files

The library files are split into five main parts, type declarations, equations, usage, conjectures and methods. The last two are not output by the translator as it stands. Extra methods are not required and have no counterpart in the SML syntax. Conjectures are

Figure 31: The syntax of the bare SML subset

exp Expressions

aexp::=	**var**	(variable)
	con	(constructor)
	{ **lab**1=**exp**1,___, **lab**n=**exp**n }	(record,n≥0)
	(**exp**)	
exp::=	**aexp**	(atomic)
	exp aexp	(application)
	exp : **ty**	(constraint)
	fn **match**	(function)
match::=	**rule**1 | ___ | **rule**n	(n≥1)
rule::=	**pat** =>**exp**	

dec Declarations

dec::=	val **vb**	(values)
	type **tb**	(types)

pat Patterns

apat::=	_	(wildcard)
	var	(variable)
	con	(constant)
	{ **lab**1=**pat**1, ___ , **lab**n=**pat**n [...] }	(record,n≥0)
	(**pat**)	
pat::=	**pat**	(atomic)
	con apat	(construction)
	pat : **ty**	(constraint)

vb Value Bindings

vb::=	**pat** = **exp**	(simple)
	vb1 and ___ and **vb**n	(multiple,n≥2)
	rec **vb**	

ty Types

ty::=	**tyvar**	(type variable)
	[**ty_seq**]**tycon**	(type constn)
	{ **lab**1:**ty**1, ___ , **lab**n:**ty**n }	(record type,n≥0)
	ty ->**ty**'	(function type)
	(**ty**)	

seq Sequences

ty_seq::=	**ty**	
	(**ty**1,___,**ty**n)	(n≥1)
tyvar_seq::=	**tyvar**	
	(**tyvar**1,___,**tyvar**n)	(n≥1)

not output because, as it stands, the input SML syntax is not annotated with information about where synthesis should take place. It is possible to construct synthesis conjectures from SML syntax. However, without any indication of where in a program synthesis should take place it is unclear what conjectures should be constructed. Information about which functions should be synthesised might take the form of the annotation of the SML syntax or by some separate indication.

Conjectures could be formed for every function in a program and synthesis performed on each in turn. This is inefficient because some functions may call on other functions and the synthesis will automatically plan these functions as part of the synthesis of the parent function; if every function was planned there would be many repeated synthesis attempts. There is another possibility, a conjecture could be formed from the top level SML program expression. This expression is currently not translated but translating it into a conjecture would allow the synthesis of every function that is actually used in the program while ignoring those which may remain unused.

For each function in the input program there should be a type declaration, equations and usage instructions in the library file output. The type declaration merely informs λ*Clam* of the type of the function. For example:

```
define squares: (olist A) -> (olist A).
```

This definition simply links the name of the function (`squares`) with its type (`(olist A) -> (olist A)`).

The equations are more complicated as they list the cases of the function to allow λ*Clam* to use them as rewrite rules:

```
equations:
  squares1 equiv (squares onil) => onil.
  squares2 equiv (squares (ocons H T))
    => (ocons (times H H) (squares T)).
endequations.
```

The syntax of these equations is very similar to λ*Clam*'s internal representation described in the next section. The polarity of these equations is `equiv` other polarities are `lefttoright` or `righttoleft` but the translator will only generate `equiv` because we only deal with equivalences.

The usage instructions are there to tell λ*Clam* which equations it can use for symbolic evaluation. The following:

```
sym_eval: squares1, squares2.
```

instructs λ*Clam* that it can use both the `squares1` and `squares2` equations in symbolic evaluation.

Internal λ*Clam* Notation

The internal representation of terms within λ*Clam* is similar to the representation of the syntax within equations in library files. The main difference is that there may be quantification and λ-expressions present in the internal representation where there are none in the equations in the library files.

The structure of the equations in the library files is first an identifier followed by its polarity. `righttoleft` is used if the rewrite can only be applied from right to left, `lefttoright` for vice-versa and `equiv` if it can be used either way. Following this is the left and right hand sides of the equation separated by a `=>`.

The structure of the right and left sides of the equation are in λ*Clam*'s internal representation. Capital letters represent metavariables and can be thought of as being similar to variables in patterns or expressions in the SML syntax. The names of functions specified in the equations are either standard functions declared within λ*Clam* itself or functions which have been introduced in a library file by a type definition.

λ-expressions and quantification are represented by the `\` notation used in λProlog.

The translation of specific constructs to and from the SML and the λ*Clam* representation will be given in the sections on the individual translators.

7.1.3 Differences between SML and λ*Clam*

The representations of programs in SML and λ*Clam* are very similar but there are some significant differences which hinder the translation from one to the other.

The main differences are in the type systems of the two. λ*Clam* library files require that functions be explicitly typed whereas in SML the types may be worked out from the context.

λ*Clam* uses a constructive type system where the datatypes such as lists and natural numbers are defined recursively. Lists are defined as an empty list or an element attached to another list. Natural numbers are defined as zero or the successor of a natural number.

SML uses similar definitions for lists and other datatypes. However, SML also supports integers rather than natural numbers; these can have a negative value where natural numbers cannot.

This discrepancy creates some difficulties in the transformation between λ*Clam* syntax and SML. These have mostly been sidestepped by the transformation process. Integer values in the SML program are assumed to be positive by the transformer and converted into natural numbers. Small integers, zero, one, and two are converted into their corresponding natural number representation, 0, $s(0)$, and $s(s(0))$. However, it is infeasible to replace all integers by their natural number equivalent because the terms structure quickly becomes too large to handle, so large numbers are replaced by natural number variables.

7.2 SML Parser/Type Checker

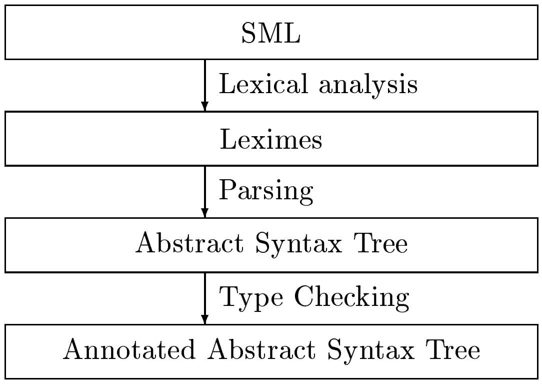
In order to translate SML into λ*Clam* syntax we need to parse the SML and produce an intermediate abstract syntax tree representation. Because the λ*Clam* syntax requires that the types be explicit, each branch of the abstract syntax tree must be annotated with its type. This annotated abstract syntax tree can then be used to output λ*Clam* syntax.

The SML to λ*Clam* component was written in the C programming language [KR88] because there are existing packages, like lex, which can be used to perform parts of the parsing process instead of having to implement the system from the ground up.

7.2.1 SML→Abstract Syntax Tree

The SML parser which translate the bare SML into the abstract syntax tree representation can be broken down into three parts, the lexical analyser, the parser, and the type checker(Figure 32).

Figure 32: SML Parser/Type Checker



Lexical Analysis

To perform the lexical analysis phase of the SML translation we used the lexical analyser lex. Lex takes as input a file defining the types of leximes that exist in the input language. It compiles this into a lexical analysis module in C which can be included into a C program. The lexical analysis module takes the file containing the program and breaks it down into a list of leximes which may be accessed by means of some predefined procedures.

The lexime definitions for SML were compiled by examining the syntax of the core SML as defined in [HMM86].

Parsing

The parsing part of the SML translator was written in C using the lexical analysis module constructed by lex. Each lexime is tagged as being of a certain type by the lexical analyser and the parser checks that the combination of leximes is valid in the core SML syntax [HMM86]. If so, the parser constructs the abstract syntax tree representation of the program as it goes along. Although we could have used a DCG to construct a parser for the SML subset we felt that this would be less efficient.

Type Checking

The type checking is done as a separate phase in the parser. Once the abstract syntax tree has been constructed the type checker will work over it annotating each node with the appropriate type. The type checking algorithm we used was Milner's $\mathcal{W}$ algorithm as described in [FH88]. We could have preprocessed the SML using another SML type checker to generate the types but this would have duplicated the parsing work. Also there is a possibility that λProlog could have been used to deduce the types but we chose to use C to simplify the translator/λ*Clam* interface.

7.2.2 Abstract Syntax Tree→λ*Clam*

The second phase of the translator is to translate the abstract syntax tree into the λ*Clam* syntax. Because the abstract syntax tree contains much more explicit structure than the source SML the output of λ*Clam* syntax is much easier than the parsing of SML.

The output of SML is done by a display module in the translator which will traverse the abstract syntax tree and output the relevant type definitions and rewrite rules for any functions to a file which can be read into λ*Clam*. The display module also has several other output modes for debugging including core SML output, extended SML output and a graphical representation of the abstract syntax tree.

Because the λ*Clam* library file is split into three separate sections the translator first outputs to three temporary files with type definitions, equations and symbolic evaluation declarations before combining them into a single λ*Clam* library file.

The rules for the creation of λ*Clam* equations from the abstract syntax tree representation are shown below.

7.2.3 Translation Rules

Expressions

Figure 33: Translation rules for SML expressions

{**lab**1=**exp**1,___,**lab**n=**exp**n} ⇒**exp**1 ___ **exp**n

(**exp**) ⇒(**exp**)

(fn true=>true | false=>**exp**')(**exp**) ⇒(oor **exp exp**')

(fn true=>**exp**' | false=>false)(**exp**) ⇒(oand **exp exp**')

(fn true=>**exp**1 | false=>**exp**2)(**exp**) ⇒(if **exp exp**1 **exp**2)

+{**lab**1 = **exp**, **lab**2 = 1} ⇒(s **exp**)

exp aexp ⇒(**exp aexp**)

exp:**ty** ⇒**exp**

let **dec** in **exp** end ⇒**dec exp**

fn **match** ⇒**match**

rule1 | ___ | **rule**n ⇒*name*1 equiv **rule**1
⋮
*name*n equiv **rule**n

pat => **exp** ⇒(*name* **pat**) => **exp**.

Figure 33 displays the rules for translating SML expressions into λ*Clam* notation. In the translation of rules the *name* has been recorded in the enclosing declaration. If there is no enclosing declaration the rule is not translated.

Let expressions are only translated if they occur at the top level. A let expression appearing within the body of a function will not be translated because in λ*Clam* library files all declarations must be global. Thus there is no equivalent λ*Clam* representation for a let expression appearing within the body of a function. At the top level the expression part of a let expression is not useful to λ*Clam* unless it contains further declarations. Let expressions nested within the expression part of a top level let expression will be translated but otherwise the expression part of a let expression is not translated.

We could possibly use the top level let expression to construct a conjecture so we can use this to perform synthesis over the entire program. The feasibility of this depends on the complexity of this expression; if the term structure is too large then neither synthesis plan nor the translator may be able to handle it. There will also be functions called on concrete arguments, if we wish to perform synthesis it is useful to be able to abstract away from these arguments. The generalise method in the synthesis plan may be able to perform this abstraction but it might also be useful to have the translators perform this abstraction as part of the conjecture forming process.

Declarations

Figure 34: Translation rules for SML declarations

val **vb**$\Rightarrow$**vb**

Figure 34 displays the rules for translating SML declarations into λ*Clam* notation. Type and datatype declarations appearing in the input are not translated because λ*Clam* cannot dynamically parse new types. Types other than the standard types must be coded into the source files and λ*Clam* recompiled before they be used. Local declarations are not translated for the same reasons as let expressions must be used at the top level; declarations in λ*Clam* must be global and local expressions introduce the possibility of name clashes and rules applying outside their intended scope.

Patterns

Figure 35: Translation rules for SML patterns

_	⇒_
{**lab**1=**pat**1,___,**lab**n=**pat**n}	⇒**pat**1 ___ **pat**n
(**pat**)	⇒(**pat**)
con apat	⇒(**con apat**)
pat : **ty**	⇒**pat**

Figure 35 displays the rules for translating SML patterns into λ*Clam* notation. Type constraints in patterns are not translated because the constraint is taken care of at the type checking stage. At the translation stage the abstract syntax tree has already been annotated with types and the information contained in the type constraint is subsumed by the type annotations.

Bindings

Figure 36 displays the rules for translating SML bindings into λ*Clam* notation. It is in the bindings that the main structure of the λ*Clam* library file is constructed. Firstly the pattern on the left of the binding is recorded in as the name of the function; if the pattern is not an identifier then the binding will not be translated.

Then type annotation attached to the abstract syntax tree is used to determine the type of the identifier. Together with the name this type information is used to construct a λ*Clam* type definition which is output to a file containing all such definitions.

The expression part of the binding will contain the rules that define a function.

Figure 36: Translation rules for SML bindings

pat = **exp**	⇒*name* = **pat** *defines:* define **name**:*deduced type.* *equations:* equations: **exp** *or* **name**1 equiv **name** => **exp**. endequations. *symevals:* symeval: **name**1,___,**name**n.
vb1 and ___ and **vb**n	⇒**vb**1 ___ **vb**n
rec **vb**	⇒**vb**

This expression is translated and the result output to a file which already contains the start of an equation block. At the end of the translation of the entire tree the equation block will be closed completing this file.

If the expression is a function expression then the rules contained in the expression will be used to form the equations. If however the expression is a simple value or non-function expression then the recorded name will be used to form a single equation from it.

The number of equations is recorded and after the expression has been translated, this information together with the name will be used to output a `symeval` declaration to allow all the translated equations to be used by λ*Clam* for symbolic evaluation.

Types

Figure 37 displays the rules for translating SML types into λ*Clam* notation. Record types, which are used in SML for functions which take tuple arguments are translated into curried function types in λ*Clam*. Certain standard functions in SML which use tuple types correspond to equivalent functions in SML which use curried types. For instance:

Figure 37: Translation rules for SML types

ty_seq tycon	⇒(**tycon ty_seq**)
{**lab**1 : **ty**1,___, **lab**n : **ty**n}	⇒**ty**1 ->__ ->**ty**n
ty -> **ty**'	⇒(**ty**) -> **ty**'
(**ty**)	⇒(**ty**)
(**ty**1,___,**ty**n)	⇒**ty**1 ___ **ty**n

SML- +:{#1:int, #2:int} ->int

λ*Clam*- plus:nat ->nat ->nat

Also because the λ*Clam* syntax representation doesn't seem to have a tuple type the decision was made to translate all tuple types in SML to the equivalent curried function type in λ*Clam*. This conversion from the tuple to curried form should have no impact on correctness. In the original program tupled functions will be fully applied to values or variables or be completely unapplied. A tupled function differs from a curried one only when an attempt is made to partially apply it; in the unapplied and fully applied situation the two forms will behave the same. A tupled function in the original program must either be unapplied or fully applied so the transformed function being fit back into the program must also be unapplied or fully applied so will exhibit the same behaviour whether it has been translated into curried form or not.

The final entry in the table is the translation rule for type sequences used in polymorphic types like the list type.

Values

Figure 38 displays the rules for translating SML values into λ*Clam* notation. The small integers 0 and 1 are translated into their equivalent λ*Clam* syntax zero and (s zero). However due to the size of the term structure, higher numbers are translated into s of

Figure 38: Translation rules for SML values

0	⇒zero
1	⇒(s zero)
>1	⇒(s *variable*)

some variable. This allows a reasonable amount of manipulation to be done with these numbers without producing huge term structure.

Identifiers

Figure 39: Translation rules for SML declarations

int⇒nat	+⇒plus	-⇒minus	∗⇒times	
s⇒S				
list⇒olist	nil⇒onil	[]⇒onil	::⇒ocons	
bool⇒boolean	false⇒ofalse	true⇒otrue	or⇒oor	and⇒oand
<⇒olt	>⇒ogt	<=⇒olte	>=⇒ogte	=⇒oeq
map⇒lmap	foldr⇒lfoldr	foldl⇒lfoldl	filter⇒lfilter	scan⇒lscan

declared identifier⇒*identifier*

undeclared identifier⇒*capitalised identifier*

Figure 39 displays the rules for translating SML identifiers into $\lambda Clam$ notation. s is a standard function in $\lambda Clam$ but doesn't have an equivalent in SML; the translation rule for s is present because some variable which happens to be called s in the SML input should not be naively translated into s in $\lambda Clam$ where it is a standard function.

The type boolean in the target $\lambda Clam$ syntax is non-standard and is added to make the $\lambda Clam$ types closer to those in SML. All the functions defined on the new $\lambda Clam$ boolean type (ofalse-oeq) have been added to $\lambda Clam$ to match the standard operations in SML.

The HOFs (lmap-lscan) are also additions to the $\lambda Clam$ syntax to allow synthesis of expressions involving them.

7.2.4 Example

To illustrate the translation we will look at the matrix multiplication example. The bare SML source defining the functions used in the example is as follows:

```
let
  val
  rec dist=fn a1 => fn a2 =>
    (fn {#1 = nil, #2 = _} => nil |
        {#1 = :: {#1 = h1, #2 = t1}, #2 = :: {#1 = h2, #2 = t2}}
          => :: {#1 = :: {#1 = h1, #2 = h2}, #2 = dist t1 t2} |
        {#1 = :: {#1 = h1, #2 = t1}, #2 = nil}
          => :: {#1 = :: {#1 = h1, #2 = nil}, #2 = dist t1 nil}
    ) {#1 = a1, #2 = a2} and
  rec transpose=fn a1 =>
    (fn nil => nil |
        :: {#1 = row, #2 = rows} => dist row (transpose rows)
    ) a1 and
  rec dotprod=fn a1 => fn a2 =>
    (fn {#1 = :: {#1 = h1, #2 = t1}, #2 = :: {#1 = h2, #2 = t2}}
          => + {#1 = * {#1 = h1, #2 = h2}, #2 = dotprod t1 t2} |
        {#1 = _, #2 = _} => 0
    ) {#1 = a1, #2 = a2} and
  rec rowmult=fn a1 => fn a2 =>
    (fn {#1 = _, #2 = nil} => nil |
        {#1 = row, #2 = :: {#1 = col, #2 = cols}}
          => :: {#1 = dotprod row col, #2 = rowmult row cols}
    ) {#1 = a1, #2 = a2} and
  rec rowsmult=fn a1 => fn a2 =>
    (fn {#1 = nil, #2 = _} => nil |
        {#1 = :: {#1 = row, #2 = rows}, #2 = cols}
          => :: {#1 = rowmult row cols, #2 = rowsmult rows cols}
            ) {#1 = a1, #2 = a2} and
  rec mmult=fn a1 => fn a2 =>
    (fn {#1 = m1, #2 = m2}
          => rowsmult m1 (transpose m2)) {#1 = a1, #2 = a2}
in
  true
end
```

The $\lambda Clam$ library file resulting from the translation of this source by the translator is as follows:

```
define dist: (olist A) -> (olist (olist A)) -> (olist (olist A)).
define transpose: (olist (olist A)) -> (olist (olist A)).
define dotprod: (olist nat) -> (olist nat) -> nat.
define rowmult: (olist nat) -> (olist (olist nat)) -> (olist nat).
define rowsmult: (olist (olist nat)) -> (olist (olist nat)) -> (olist (olist nat)).
define mmult: (olist (olist nat)) -> (olist (olist nat)) -> (olist (olist nat)).
equations:
```

```
  dist1 equiv (dist onil _) => onil.
  dist2 equiv (dist (ocons H1 T1) (ocons H2 T2))
    => (ocons (ocons H1 H2) ((dist T1) T2)).
  dist3 equiv (dist (ocons H1 T1) onil)
    => (ocons (ocons H1 onil) ((dist T1) onil)).
  transpose1 equiv (transpose onil) => onil.
  transpose2 equiv (transpose (ocons ROW ROWS))
    => ((dist ROW) (transpose ROWS)).
  dotprod1 equiv (dotprod (ocons H1 T1) (ocons H2 T2))
    => (plus (times H1 H2) ((dotprod T1) T2)).
  dotprod2 equiv (dotprod _ _) => zero.
  rowmult1 equiv (rowmult _ onil) => onil.
  rowmult2 equiv (rowmult ROW (ocons COL COLS))
    => (ocons ((dotprod ROW) COL) ((rowmult ROW) COLS)).
  rowsmult1 equiv (rowsmult onil _) => onil.
  rowsmult2 equiv (rowsmult (ocons ROW ROWS) COLS)
    => (ocons ((rowmult ROW) COLS) ((rowsmult ROWS) COLS)).
  mmult1 equiv (mmult M1 M2) => ((rowsmult M1) (transpose M2)).
endequations.
sym_eval: dist1, dist2, dist3.
sym_eval: transpose1, transpose2.
sym_eval: dotprod1, dotprod2.
sym_eval: rowmult1, rowmult2.
sym_eval: rowsmult1, rowsmult2.
sym_eval: mmult1.

end.
```

7.3 λ*Clam* SML Output

Once a transformation rule has been synthesised then it must be translated into a format that the transformer can understand. Since the transformer manipulates SML syntax the result of the synthesis should be translated into SML.

λ*Clam* has built into it the ability to display conjectures and logical expressions in a variety of different ways. By extending this feature of λ*Clam* we avoid having to parse the λ*Clam* output because we can use λ*Clam*'s structured internal representation to output SML syntax. So instead of implementing a separate component to translate λ*Clam* to SML we use λ*Clam* itself to output the result in the desired syntax.

The λ*Clam* output is, at the moment, restricted to output to the terminal. To enable the transformer to pick up the synthesised transformations the output should be to a file but this functionality has not yet been implemented.

The rules for the output of SML from λ*Clam* are shown below.

7.3.1 Translation Rules

The translation rules from λ*Clam* syntax to SML are much simpler than the other way around. All that needs to be translated is the result rather than the rules and equations for every non-standard function involved in the problem.

Functions & Arguments

Figure 40: Translation rules for λ*Clam* functions

func	⇒**func**
x\ (**func**)	⇒(fn **x** => **func**)
x\ (**y**\ (**func**))	⇒(fn **x** => **fn y** => **func**)
x\ (**y**\ (**z**\ (**func**)))	⇒(fn **x** => **fn y** => **fn z** => **func**)
func arg	⇒(**func arg**)
infix_func arg1 arg	⇒(**arg1 infix_func arg2**)
func arg1 arg2	⇒(**func arg1 arg2**)
func arg1 arg2 arg3	⇒(**func arg1 arg2 arg3**)

Figure 40 displays translation rules for λ*Clam* functions and arguments. Due to internal efficiency considerations λ*Clam* only handles functions of up to arity 3. For the same reasons the translator contains separate translation rules for arity 0, 1, 2 and 3 functions.

Identifiers

Figure 41 displays translation rules for λ*Clam* identifiers.

λ*Clam* contains two separate boolean types: a formula type and an object boolean type. The formula type is present to allow λ*Clam* to reason about formulae. Unfortunately this type cannot be reconciled with the object types of functions that it reasons about. For this reason a boolean type was added to λ*Clam* to allow it to perform

Figure 41: Translation rules for λ*Clam* identifiers

zero⇒0	s **val**⇒**val**+1	plus⇒+	minus⇒-	times⇒*	
onil⇒[]	ocons⇒::				
otrue⇒true	ofalse⇒false	oor⇒or	oand⇒and		
olt⇒<	ogt⇒>	olte⇒<=	ogte⇒>=	oeq⇒=	eq⇒=
lmap⇒map	lfoldr⇒foldr	lfoldl⇒foldl	lfilter⇒filter	lscan⇒scan	
user_defined_object **name**⇒**name**					

synthesis on functions which take boolean arguments.

λ*Clam* has built into it operators for the formula type but new operators and literals for the new boolean type had to be added; the new boolean operators are prefixed by an 'o' like the predefined list constructors ocons and onil. So 'oeq' is the equals operator for the boolean type while 'eq' is the inbuilt equals operator for λ*Clam*'s formula type.

7.3.2 Example

We again use the matrix multiplication example to illustrate the translation of the result from the λ*Clam* synthesis to SML. The result for the instantiation of the metavariable expressed in λ*Clam* syntax is:

```
X\ Y\
  (lmap Z\
  (lmap U\ (dotprod Z U)
    (lfoldr U\ V\ (dist U V) onil Y)) X)
```

This is translated by the λ*Clam* translation rules into the bare SML syntax:

```
(fn x => fn y =>
  (map (fn z =>
  (map (fn u => (dotprod z u))
    (foldr (fn u => fn v => (dist u v)) [] y))) x))
```

Chapter 8

Conclusions & Future Work

8.1 Summary

Chapter 1 outlined the aims of our research and provided an indication of the context in which the work was done.

Chapter 2 provided an overview of the theorem proving, program transformation and synthesis technologies that we used in our work. We looked at transformation and synthesis techniques in general and presented an overview of theorem proving techniques, in particular proof planning and the rippling heuristic.

Chapter 3 documented our work on verification. We implemented a critic to unblock the fertilisation step of a ripple proof and we proved various properties of HOFs using *Clam* and λ*Clam*.

Chapter 4 recorded the iterations that the plan went through, from the initial ideas to the final revision of the plan.

Chapter 5 described the implementation of the penultimate synthesis plan in the framework of the λ*Clam* theorem prover.

Chapter 6 concerned the evaluation of the results obtained from the λ*Clam* implementation.

Chapter 7 deals with the implementation of translators that allow SML syntax to be used by the λ*Clam* theorem prover and the synthesised results to be translated back into SML code.

8.2 Contributions

8.2.1 HOF Verification

We used the *Clam*, and later the $\lambda Clam$, theorem provers to verify properties such as the $map/fold$ equivalence and distributivity properties of map and $fold$ over $compose$ as well as verification of various functions and their expression using HOFs.

8.2.2 Fertilisation Critic

We implemented a critic within the framework of the *Clam* proof planning theorem prover. There are certain cases where the fertilisation step of a ripple proof attempt in *Clam* can become blocked. Our critic can unblock these proof attempts by constructing, planning and applying a rewrite step.

Armando [ASG97] developed a sink-equalisation method. However, this method relies on simple rewriting to make the sinks equal. The sink equalisation method first attempts to use the rules in the rewrite system to make the sinks equal if this fails then the rewrite system is extended by adding rewrite rules generated from subformulae of the goal. The reliance on the rewrite system to make the sinks equal sets this method apart from our critic implementation. We use the proof system to prove the two sinks equal and the proof of the equality may involve induction instead of simple rewriting.

8.2.3 Synthesis Plan

The penultimate revision of the synthesis plan (section 4.6) is implemented and has been successfully tested on several examples to produce multiple higher-order expressions for each example. The final iteration of the synthesis plan (section 4.7) addresses several issues, such as the extension to any higher-order function and synthesis of composed functions, to produce a more powerful and elegant plan than the one implemented.

Our synthesis work is built on Kraan's [KBB96] logic program synthesis work (see section 2.2.3). Lacey [LRS00] extended Kraan's work by transplanting it into the higher-order setting of λ*Clam*. Both Kraan and Lacey present a general approach to the synthesis of programs. Our work is more specialised and focused on the discovery of HOFs.

Kraan and Lacey's synthesis from an unexecutable specification gives rise to the problem of how to choose a suitable induction scheme for the program. In our work the specification of the program is much more concrete, an executable ML program, and will already contain a suitable recursive template for our synthesis. However, the criteria for a suitable result for us are very narrow compared to Kraan and Lacey's work. We desire our result to contain a HOF while they require only an executable program.

The thrust of Kraan and Lacey's synthesis work is to specialise a general unexecutable specification into a concrete implementation. However, our goal is slightly different. We generalise a specific concrete implementation by expressing it as a specialisation of a more general algorithm (a HOF).

8.2.4 Implementation

The penultimate synthesis plan is implemented in λ*Clam*. This implementation has been tested on several examples with good results. For each example the λ*Clam* synthesis engine can generate multiple possible transformations. The implementation has found as many as 14 different results for one example.

8.2.5 Evaluation

Several examples were evaluated. The times taken for results to be obtained are compared against the search tree for the example. From the times and the structure of the results we were able to deduce the general structure of the search tree for each example and expose inefficiencies in the synthesis plan.

8.2.6 Translators

The SML to λ*Clam* translator parses and type checks the SML syntax producing an abstract syntax tree which is annotated with types. This tree structure is then translated into a λ*Clam* library file containing all the information relevant to the functions defined in the SML. The nature of the λ*Clam* theorem prover allowed the λ*Clam* to SML translator to be written in to the synthesis system. The internal λ*Clam* structure is portrayed in the output as SML syntax.

8.3 Critique

8.3.1 Critic

Although the fertilise critic, implemented in *Clam*, works on the examples we used to test it, it has a number of problems. Some arise because the implementation is not as neat as it could be for efficiency considerations. However, others are more serious and affect the theorem prover's ability to prove certain theorems when the critic is in place.

The preconditions for the fertilise method had to be changed to allow the critic to detect partial failure. The single original precondition (`ind_hyp_match`) had to be split into two where success of the first and failure of the second would trigger the activation of the fertilise critic. The first precondition (`all_waves_sunk`) was easy to test for but the second was not. The `ind_hyp_match` precondition encompasses both of the required preconditions and was implemented in an efficient way. Splitting this precondition into two completely separate preconditions would have meant adding significant complexity to the implementation. To address the efficiency considerations it was decided that the two new preconditions would be the first precondition, `all_waves_sunk`, combined with the original precondition, `ind_hyp_match`. Although the original precondition subsumes the first, the first is needed to allow the system to detect partial failure and the implementation of a separate second precondition which did not subsume the first would have added significant inefficiency to the system.

Since the fertilise critic is applicable only to fertilisation in the transverse rippling case there arose the problem of how to prevent the critic from firing when the rippling was entirely longitudinal. This was addressed by using two cases in the fertilise method; one for the transverse and one for the longitudinal case. However, this is a clumsy solution and it would be more elegant for a single method to apply in both cases.

The critic is unable to handle non-identical sinks in the hybrid rippling case. The first precondition of the critic requires that all wave fronts are within sinks so it rules out this case because in hybrid rippling some wave fronts are rippled out. There is the possibility that the waves rippled into the sinks could manifest the problem that the critic was implemented to solve.

Finally in splitting the fertilise method into two cases for longitudinal and transverse rippling the hybrid case was ignored. This means that the critic implementation actually breaks the fertilisation method in cases involving hybrid rippling.

8.3.2 Verification

The verification of several properties was performed by the theorem prover. However, the proofs of these properties(map/fold equivalence, map/compose distributivity) are relatively simple and there wasn't really any doubt that the theorem prover would be able to produce the proofs. The verification of possible user defined functions expressed using HOFs and the map/fold equivalence was useful in informing the development of the synthesis plan. Also the proof of the map/compose distributivity property helped the identification of a similar property involving $foldr$ as well as map when it arose in the $mergesort$ example (section 6.3.5).

8.3.3 Implementation

The implementation of the penultimate synthesis plan was done largely without consideration of the method precondition structure. The preconditions to each new method are taken up by a large block of code which is relatively hard to understand. Teasing the specific logical preconditions out of the methods is difficult because working with λProlog's higher-order unification is often unintuitive and makes the code very messy. Unexpected instantiations often needed to be weeded out increasing the length of the code and reducing its readability. Naive implementation of preconditions resulted in the code for different preconditions being blurred together.

An attempt was made to implement the final synthesis plan without HOF matching and without extending its induction schemes. This wasn't completed because the higher order unification(HOU) in λProlog presents a barrier to the elegant implementation of a part of the plan. The `decompose` method would not function correctly when two composed functions were to be synthesised together. The instantiation of one metavariable would restrict the instantiation of the other when we desired the synthesis of each to be performed independently. This problem is not insurmountable but the implementation would involve using complex, inefficient code to fight against λProlog's HOU instead of using it to our advantage.

The method which makes the choice of HOF uses hard coded functions and knowledge about them to make the decision. This restricts the synthesis to functions which it knows about instead of being able to choose between arbitrary HOFs.

8.3.4 Evaluation

The evaluation of the synthesis results provides some insights into the behaviour of the synthesis system. However, the exact behaviour of the system cannot fully be explained with the results we obtained. The number and sequence of results found by the synthesis affects the time taken to find each one. The arities of the functions involved also affects these times. However, there may be other factors present in the examples we used which obscure the results.

The testing was done on a relatively small number of very different examples which makes the comparison of the results difficult. Further testing of the synthesis system on groups of similar examples might help reveal the cause(s) of the time differences.

8.3.5 Translators

The SML to $\lambda Clam$ translator will translate chunks of the syntax of our chosen SML subset into a form that is recognisable by $\lambda Clam$. The efficiency of the parsing/type checking is not optimal. When presented with increasingly large examples of SML the translation time becomes very long and the program runs out of memory very quickly.

The $\lambda Clam$ to SML translator does not experience the same problems. It is a simple set of clauses as synthesised results tend to be much smaller and simpler than the original SML.

Although the translators have been implemented they have not yet been fully integrated into the system. For the SML to $\lambda Clam$ translator it is necessary to manually run the translator on the input and collect its output for use in $\lambda Clam$. For the $\lambda Clam$ portrayal system the translation is done automatically but the output must be manually collected for use in the compiler.

8.4 Future Work

8.4.1 Verification

The verification of nested HOF transformation rules proved problematic within $\lambda Clam$ due to the way that the theorem prover deals with induction and quantification. A temporary fix to this problem was implemented but $\lambda Clam$ was still unable to prove HOF conjectures requiring more than two nested inductions. This problem should be addressed within the $\lambda Clam$ proof planner itself.

8.4.2 Critic

The work on the synthesis plan involving equating sinks suggests a different approach to the problem that the fertilise critic addresses. Instead of using a critic to rewrite sinks and unblock fertilisation we can use a method to abstract away from the contents of the sinks and set up a separate subgoal.

If the two sinks are not identical then fertilisation cannot take place. However, we can remove the content of the sinks using the following rule:

$$\forall x. \cdots \forall y.(f\ \lfloor g\ x \rfloor \cdots y = h\ \lfloor k\ x \rfloor \cdots y) \Rightarrow$$
$$\boxed{\forall x. \cdots \forall y.(f\ \lfloor x \rfloor \cdots y = h\ \lfloor x \rfloor \cdots y) \wedge \forall x.(g\ x = k\ x)}^{\uparrow}$$

where x does not in the other arguments of f or h

This is an extension of the substitution axiom. The sink contents are removed from the induction hypothesis allowing fertilisation to take place. The content of the sinks is set up as a separate equality which drops out of the ripple proof as a residue to be proven afterwards.

A method based on this rewrite could be implemented and used as a last resort when the fertilisation step becomes blocked. This method should avoid the problems inherent in the critic implementation because the existing fertilise method would not need to be altered to accommodate a critic.

8.4.3 Plan

A possible extension to the synthesis plan might be to allow it to perform synthesis of accumulator style HOFs from functions which do not use accumulators. Although in the context of the compiler this would not produce many advantages it is an interesting goal in itself. The implementation of recursive functions using accumulators is useful because they may be converted to use iterative loops which are more efficient.

8.4.4 Implementation

The proposed synthesis plan needs to be implemented and tested. This could be accomplished by making some small changes to the metavariable representation system, rewriting the HOF choice method and updating other methods.

Another approach is to re-implement the final synthesis plan from scratch. This would involve implementing new methods and re-implementing the metavariable representation system. A rewrite of the metavariable representation system could help avoid the obstacle to the use of the `decompose` method when we have two composed functions.

The HOF rewrite method undergoes large changes between the implemented and proposed synthesis plans and so would have to be completely reimplemented for both approaches.

8.4.5 Evaluation

The evaluation of the synthesis plan should be extended by testing the current implementation with a larger set of similar examples so that trends and unusual results can be more easily identified. In particular larger examples with more potential HOFs and different examples with equivalent HOFs should be used.

8.4.6 Translators

The translators should be more closely integrated into the main project. This would entail the SML transformer activating the SML→λ*Clam* translator to translate SML to be manipulated and output the result into the λ*Clam* library. The transformer component of the main project would activate λ*Clam* to perform the synthesis. The SML portrayal system in λ*Clam* would need to be extended to output the results to a file; the transformer component could then access these as possible transformation rules.

The SML to λ*Clam* translator is inefficient and cannot handle large examples. To allow the synthesis to work on larger examples the translator should be altered or rewritten.

Automatic generation of synthesis conjectures should be added to the capabilities of the translator to enable the programs to be fed to the synthesis engine without need for human input.

8.4.7 Future work

Interaction between the compiler and λ*Clam* would allow the synthesis process to be constrained and guided by performance information. The synthesis could then choose the optimal solution instead of having to waste time finding all the possible alternatives.

The examples highlighted some functions that could be represented by a combination of HOFs. Recognising combinations of HOFs is much more difficult than finding a single one but the synthesis might be extended in this respect.

There is a possibility that the synthesis techniques could be used to synthesise HOFs as opposed to HOF expressions. By recognising recursion patterns the synthesis process might synthesise custom HOFs on demand.

The synthesis process could be extended to synthesise higher-order expressions from functions which manipulate trees, sets and other datatypes. There is even the possibility that synthesis might be performed on functions that manipulate user defined datatypes.

Appendix A

Rules

Substitution rule[1]

$$(f\ a_1 \cdots a_n) =_\sigma (f\ M_1 \cdots M_n) \Rightarrow a_1 =_\sigma M_1 \wedge \cdots \wedge a_n =_\sigma M_n$$

Extended Substitution rule

$$\begin{aligned}&(f\ a_1 \cdots a_n) =_\sigma (M\ M_1 \cdots M_n) \Rightarrow \\ &\qquad \forall x_1 \cdots x_n (f\ x_1 \cdots x_n =_\sigma M\ x_1 \cdots x_n) \wedge a_1 =_\sigma M_1 \wedge \cdots \wedge a_n =_\sigma M_n\end{aligned}$$

Decomposition rule

$$(A \circ B) =_\sigma M \Rightarrow A =_\sigma M_1 \wedge B =_\sigma M_2 \wedge M = (M_1 \circ M_2)$$

Speculative ripple hybrid

$$\begin{aligned}&\forall x.\forall y.(\boxed{g\ \boxed{(f\ x\ \lfloor(h\ y)\rfloor)}}^{\uparrow} =_\sigma M\ \boxed{(c\ \boxed{x})}^{\uparrow}\ y) \Rightarrow \\ &\qquad \forall x.\forall y.(\boxed{g\ \boxed{(f\ x\ \lfloor(h\ y)\rfloor)}}^{\uparrow} =_\sigma \boxed{M_1\ \boxed{(M\ x\ \lfloor(M_2\ y)\rfloor)}}^{\uparrow})\end{aligned}$$

Splitting ripple

$$\begin{aligned}&\forall x.\forall y.(\boxed{g\ \boxed{(f\ x\ \lfloor(h\ y)\rfloor)}}^{\uparrow} =_\sigma \boxed{M_1\ \boxed{(M\ x\ \lfloor(M_2\ y)\rfloor)}}^{\uparrow}) \Rightarrow \\ &\qquad \boxed{\forall x.(g\ x =_\sigma M_1\ x) \wedge \forall x.(h\ x =_\sigma M_2\ x) \wedge \boxed{\forall x.(f\ x =_\sigma M\ x)}}^{\uparrow}\end{aligned}$$

[1] $=_\sigma$ denotes synthesis.

Appendix B

Methodicals

B.1 Original Composite Method Structure

```
method_def top_meth
  (complete_meth
    (repeat_meth
      (orelse_meth all2i
      (orelse_meth taut
    (orelse_meth sym_eval
    (orelse_meth existential
    (orelse_meth (repeat_meth generalise)
      ind_strat
    )))))
  )
  ).

method_def sym_eval
  (cut_meth
    (then_meth
      (sym_eval_rewrites L)
      (repeat_meth
      (orelse_meth
        (rewrite_with Rs L)
    redundant
      )
      )
    )
  ).

method_def ripples
  (cut_meth
    (then_meth set_up_ripple
    (then_meth (repeat_meth ripple_inout_once)
    post_ripple))).

method_def step_case
  (cut_meth
```

```
    (then_meth set_up_ripple
  (then_meth (repeat_meth ripple_out_once)
  (then_meth post_ripple
      (try_meth fertilise)
  )))
  ).

method_def fertilise
  (cut_meth
    (orelse_meth
    fertilisation_strong
    fertilisation_weak ) ).

method_def ind_strat
  (then_meths induction
    (pair_meth (map_meth id_meth) (map_meth step_case) )).

method_def exi_ind_strat
  (then_meths induction_exi
    (pair_meth sym_eval (rippling_bound 2))).

method_def taut
  (complete_meth
    (repeat_meth
    (orelse_meth axiom
    (orelse_meth false_e
    (orelse_meth imp_i
    (orelse_meth all_i
    (orelse_meth all2i
    (orelse_meth and_i
    (orelse_meth and_e
    (orelse_meth or_e
    (orelse_meth imp_e1
    (orelse_meth imp_e2
    (orelse_meth imp_e3
    (orelse_meth imp_e4
    (orelse_meth or_il or_ir))))))))))))))).
```

B.2 Synthesis Composite Method Structure

The method definitions `top_meth` and `fertilise` replace those in the original method structure(section B.1). The other methods, marked `(user\_method "...")`, are new methods. Other methods referred to but not defined here are either atomic or the same as defined in the original method structure(section B.1).

```
method_def (user_method "top_meth")
  (complete_meth
    (repeat_meth
    (orelse_meth insmet
    (orelse_meth i_hof
    (orelse_meth (user_method "isolate")
    (orelse_meth (cut_meth all2i)
    (orelse_meth (cut_meth taut)
    (orelse_meth (cut_meth sym_eval)
    (orelse_meth (cut_meth existential)
    (orelse_meth (repeat_meth generalise)
    (orelse_meth (user_method "syn_strat")
                 (user_method "remove_meta")
    )))))))))
    )
  ).

method_def fertilise
  (cut_meth
  (orelse_meth
    fertilisation_strong
    fertilisation_weak
  )
  ).

method_def sym_eval
  (cut_meth
      (repeat_meth
        (orelse_meth
          (rewrite_with_list Rs)
          redundant
        )
      )
  ).

method_def (user_method "syn_top")
  (repeat_meth
    (orelse_meth (cut_meth taut)
    (orelse_meth sym_eval
    (orelse_meth (user_method "syn_strat")
                 (user_method "remove_meta")
    )))
  ).
method_def fertilise
   ( cut_meth
    ( orelse_meth
      fertilisation_strong
      fertilisation_weak
    )
```

```
  ).

method_def (user_method "remove_meta")
  (then_meth
    (cut_meth (try_meth (repeat_meth all_i)))
    meta_rm
  ).

method_def (user_method "syn_strat")
  (then_meths synthesis
    (pair_meth
      (complete_meth
        (then_meths
          (user_method "syn_step")
          (pair_meth
            (complete_meth
              (user_method "recurse")
            )
            (complete_meth
              (user_method "hof")
            )
          )
        )
      )
      (complete_meth (cut_meth (user_method "syn_top")))
    )
  ).

method_def (user_method "syn_step")
    (cut_meth
      (then_meth set_up_ripple
      (then_meth (user_method "recripple")
        (orelse_meth
          (then_meth
            post_ripple
            (try_meth fertilise)
          )
          id_meth
        )
      ))
    ).

method_def (user_method "recripple")
    (cut_meth
      (then_meths
        (orelse_meth
          (user_method "ripple_inout")
          (pair_meth (user_method "ripple_inout") id_meth)
        )
```

```
        (orelse_meth (user_method "recripple")
          id_meth
        )
      )
    ).

method_def (user_method "ripple_inout")
    (cut_meth
      (orelse_meth vapourise
       (orelse_meth ripple_inout_once
        meta_wr_split
      ))
    )
    .

method_def (user_method "recurse")
  (then_meth (user_method "isolate")
    (map_meth
      (user_method "syn_top")
    )
  ).

method_def (user_method "isolate")
  (cut_meth
    (then_meth
      (try_meth (repeat_meth all_i))
      (user_method "exp_and")
    )
  ).

method_def (user_method "exp_and")
  (then_meth
      and_i
    (map_meth
  (orelse_meth (user_method "exp_and") id_meth)
    )
  ).

method_def (user_method "remove_meta")
  (then_meth
    (try_meth (repeat_meth all_i))
    meta_rm
  ).

method_def (user_method "hof")
  (then_meth (try_meth (repeat_meth all_i))
  (then_meth i_hof
            (user_method "syn_top")
  )).
```

Appendix C

Methods

C.1 Atomic Methods

```
method1 insmet
  (seqGoal (H >>> (exists Type P)))
  (insert_metahyp 1 N (fun_type_wr M) H Hn)
  true
  (  (seqGoal (Hn >>> (P (meta N))))
  )
    (idPlan).

method1 insmet
  (seqGoal (H >>> (existsF1 P)))
  (insert_metahyp 1 N (fun_type_wr M) H Hn)
  true
  (  (seqGoal (Hn >>> (P (meta1 N))))
  )
    (idPlan).

method1 insmet
  (seqGoal (H >>> (existsF2 P)))
  (insert_metahyp 1 N (fun_type_wr M) H Hn)
  true
  (  (seqGoal (Hn >>> (P (meta2 N))))
  )
    (idPlan).

method1 insmet
  (seqGoal (H >>> (existsF3 P)))
  (insert_metahyp 1 N (fun_type_wr M) H Hn)
  true
  (  (seqGoal (Hn >>> (P (meta3 N))))
  )
    (idPlan).

method1  synthesis (seqGoal (H >>> G))
  (
```

```
  G = (forall Ta Ga),
  (  Ta = (olist _)
  ;
    (Ga = (El\ (forall Tb (Gb El))),
    (  Tb = (olist _)
    ;
      (Gb = (El\ Em\ (forall Tc (Gc El Em))),
      Tc = (olist _))
    ))
  ),
  all_ripple_analysis 1 G Suggestion _Flaws _Unflaws,
    ntrace RIPPLE_TRACE "Suggestion = %t\n" ((ftrm Suggestion)::nil),
    nth N (Subst @@ (_Ant @@ _Conseq)) Suggestion,
    ntrace RIPPLE_TRACE "Subst = %t, N=%t\n" ((ftrm Subst)::(ftrm N)::nil),
    not(Subst = empty),!,
    applicable_induction_scheme _ Subst (seqGoal (H >>> G)) N Subgls,
    Subgls = (Base ** Step),
    Subgoals = (Step ** Base),
  Step = (allGoal _ Ex\ (allGoal _ Wy\ (seqGoal ((Hst Ex Wy) >>> (Gst Ex Wy))))),
  can_synth (Gst wild_card wild_card) H,!,
    ntrace RIPPLE_TRACE "\n****INDUCTION\nSubgoals = %t\n*****\n\n" ((ftrm Subgoals)::n
  )
  true
  Subgoals  (tac (induction_tac N) ).

method1 meta_wr_split (ripGoal (H >>> G) Skel E)
  (
    G = (eq L R),
    Skel = (eq Ls Rs),
    (
      R = (meta _);
      R = ((meta1 _) _);
      R = ((meta2 _) _ _);
      R = ((meta3 _) _ _ _)
    ),
    L = (Lw Ls),
    Rr = (Lw Rs),
    metaise Rr H Rn Hn,
    memb_and_rest Skel Hn Hs,
    Gn = (eq L Rn),
    Gs = (eq R Rn),
    embeds EL (fun_type_wr Ls) (fun_type_wr L),
    embeds EL (fun_type_wr Rs) (fun_type_wr Rn),
    embeds E1 (fun_type_wr Skel) (fun_type_wr Gn)
  )
    true
  (
    (ripGoal (Hn >>> Gn) Skel E1)
    ** (seqGoal (Hs >>> Gs))
```

```
  )
  (tac meta_wr_split).
method1 meta_wr_split (ripGoal (H >>> G) Skel E)
  (
    G = (forall _ Ex\ (eq (L Ex) (R Ex))),
    Skel = (eq Ls Rs),
    (
      (R Ex) = (meta _);
      (R Ex) = ((meta1 _) _);
      (R Ex) = ((meta2 _) _ _);
      (R Ex) = ((meta3 _) _ _ _)
    ),
    Lwc = (L wild_card),
    contains_not_wc Ls Lsw,
    Lsm = (Lsw M),
    Lwc = (Lw Lsm),
    not ((Lw wild_card) = (Lw Lsm)),
    contains_not_wc Rs Rsw,
    Rsm = (Rsw M2),
    M2 = M,
    contains_not_wc (Lw Rsm) Rr,
    metaise (Rr wild_card) H Rnw Hn,
    contains_not_wc Rnw Rn,
    Hyp = (forall T Ex\ (Yp Ex)),
    memb_and_rest Hyp Hn Hr,
    Gn = (forall T Ex\ (eq (L Ex) (Rn Ex))),
    embeds EL (fun_type_wr Ls) (fun_type_wr (L Ex)),
    embeds EL (fun_type_wr Rs) (fun_type_wr Rn),
    embeds E1 (fun_type_wr Skel) (fun_type_wr Gn)
  )
    true
  (
    (ripGoal (Hn >>> Gn) Skel E1)
    ** (seqGoal (Hr >>> (forall T Ex\ (eq (R Ex) (Rn Ex)))))
  )
  (tac meta_wr_split).

method1 vapourise (ripGoal (H >>> (eq L R)) Skel E)
  (
    T = (eq L R),
    vapourise_goal H T Vapour,
    not(T = Vapour),
    embeds E1 (fun_type_wr Skel) (fun_type_wr Vapour)
  )
  true
  (ripGoal (H >>> Vapour) Skel E1)
  (tac vapourise).
method1 vapourise (ripGoal (H >>> G) Skel E)
  (
```

```
    G = (forall T Ex\ (eq (L Ex) (R Ex))),
    G = (forall T Gt),
    Gt = (Ex\ (eq (L Ex) (R Ex))),
    vapourise_goal H (Gt wild_card) Vapw,
    Vapw = (Vapour wild_card),
    not(Gt = Vapour),
    not((Vapour Ex) = (_ Th),
      not(flex Th),
      Th = wild_card),
    Qvap = (forall T Ex\ (Vapour Ex)),
    embeds E1 (fun_type_wr Skel) (fun_type_wr Qvap)
  )
  true
  (ripGoal (H >>> Qvap) Skel E1)
  (tac vapourise).

method1 meta_rm (seqGoal (H >>> (eq L R)))
  (
    not (flex R),
    R = (meta P),
    not(L = (meta P)),
      member (meta_hyp P (fun_type_wr M)) H
  )
  true
  (seqGoal (H >>> (eq L M)))
  (idPlan).
method1 meta_rm (seqGoal (H >>> (eq L R)))
  (
    not (flex R),
    R = ((meta1 P) Q),
    not(L = ((meta1 P) Q)),
    member (meta_hyp P (fun_type_wr M)) H
  )
  true
  (seqGoal (H >>> (eq L (M Q))))
  (idPlan).
method1 meta_rm (seqGoal (H >>> (eq L R)))
  (
    not (flex R),
    R = ((meta2 P) Q S),
    not(L = ((meta2 P) Q S)),
    member (meta_hyp P (fun_type_wr M)) H
  )
  true
  (seqGoal (H >>> (eq L (M Q S))))
  (idPlan).
method1 meta_rm (seqGoal (H >>> (eq L R)))
  (
    not (flex R),
```

```
    R = ((meta3 P) Q S T),
    not(L = ((meta3 P) Q S T)),
    member (meta_hyp P (fun_type_wr M)) H
  )
  true
  (seqGoal (H >>> (eq L (M Q S T))))
  (idPlan).

method1 i_hof (seqGoal (H >>> (eq L R)))
  (
    L = ((meta1 N) (ocons Hd Tl)),
    R = (Ro ((meta1 N) Tl)),
    not((Ro ecks) = R),
    member (meta_hyp N (fun_type_wr M)) H,
    remove_meta (fun_type_wr L) (fun_type_wr Nl) H,
    remove_meta (fun_type_wr R) (fun_type_wr Nr) H,
    !,
printf "\nTrying to find a HOF for M_~(s)p\n" [~N],
    (
      (
           Nr = (ocons _He _Tr),
printf "Trying map for M_~(s)p.\n" [~N],
        (M (ocons Hd Tl)) = (lmap _Mf (ocons Hd Tl))
      );
      (
printf "Trying foldr for M_~(s)p.\n" [~N],
        (M (ocons Hd Tl)) = (lfoldr _Ff _Fb (ocons Hd Tl))
      );
      (
printf "No HOF for M_~(s)p.\n" [~N],
        fail
      )
    )
  )
  true
  (seqGoal (H >>> (eq Nl Nr))) (idPlan).
method1 i_hof (seqGoal (H >>> (eq L R)))
  (
    L = ((meta2 N) Aa Ab),
    R = (Ro ((meta2 N) Ra Rb)),
    not((Ro ecks) = R),
    (  (Aa = (ocons Hd Tl));
      (Ab = (ocons Hd Tl))
    ),
    (  (  Ra = Tl,
        Rb = Si
      );(  Rb = Tl,
        Ra = Si
      )
```

```
    ),
    remove_meta (fun_type_wr Si) (fun_type_wr Ns) H,
    member (meta_hyp N (fun_type_wr M)) H,
    remove_meta (fun_type_wr L) (fun_type_wr Nl) H,
    remove_meta (fun_type_wr R) (fun_type_wr Nr) H,
    !,
printf "\nTrying to find a HOF for M_~(s)p\n" [~N],
    (
      (
        Nr = (ocons _He _Tr),
printf "Trying map for M_~(s)p.\n" [~N],
        (M Aa Ab) = (lmap _Mf (ocons Hd Tl))
      );
      (
printf "Trying foldr for M_~(s)p.\n" [~N],
    (M Aa Ab) = (lfoldr _Ff _Fb (ocons Hd Tl))
      );
      (
        not(flex Ns),
        Nr = (ocons Ns _Tr),
printf "Trying scan for M_~(s)p.\n" [~N],
        (M Aa Ab) = (lscan _Sf _Sb (ocons Hd Tl))
      );
      (
printf "Trying foldl for M_~(s)p.\n" [~N],
        (M Aa Ab) = (lfoldl _Lf _Lb (ocons Hd Tl))
      );
      (
printf "No HOF for M_~(s)p.\n" [~N],
        fail
      )
    )
  )
    true
  (seqGoal (H >>> (eq Nl Nr))) (idPlan).
```

C.2 New Predicates

```
insert_metahyp Inum Onum M Ihyps Ohyps :-
  (
    (
      member (meta_hyp Inum _) Ihyps,!,
      Nnum is (Inum + 1),
      insert_metahyp Nnum Onum M Ihyps Ohyps,!
    )
    ;
    (
      memb_and_rest (meta_hyp Inum M) Ohyps Ihyps,
```

```
      Onum is Inum
    )
  ).

can_synth (eq L R) Hyps :-
  L = (El (ocons H T)),
  not(L = (El wild_card)),
  R = (Ar (ocons H T)),
  (
    (R = (meta N));
    (R = (meta1 N _));
    (R = (meta2 N _ _));
    (R = (meta3 N _ _ _))
  ),
  not(R = (Ar wild_card)).
can_synth (forall _ Ex\ (eq (L Ex) (R Ex))) Hyps :-
  L = (El (ocons H T)),
  not(L = (El wild_card)),
  R = (Ar (ocons H T)),
  (
    (R = (Wy\ (meta N)));
    (R = (Wy\ (meta1 N (_ Wy))));
    (R = (Wy\ (meta2 N (_ Wy) (_ Wy))));
    (R = (Wy\ (meta3 N (_ Wy) (_ Wy) (_ Wy))))
  ),
  not(R = (Ar wild_card)).

vapourise_goal Hyps H H :-
  H = (eq L R),
  skel_hyp H Hyps.
vapourise_goal Hyps C C :-
  C = (eq L R),
  not(R = (meta _);
    R = ((meta1 _) _);
    R = ((meta2 _) _ _);
    R = ((meta3 _) _ _ _)),!.
vapourise_goal Hyps G Vap :-
  G = (eq Gl Gr),
  Hy = (forall T F),
  member Hy Hyps,
  Fs = (F wild_card),
  Fs = (eq (Fl wild_card) (Fr wild_card)),
  Gl = (Fl (Sl wild_card)),
  Gr = (Fr (Sr wild_card)),
  Qh = (forall T x\ (eq (Fl x) (Fr x))),
  Qs = (forall T x\ (eq (Sl x) (Sr x))),
  Vap = (and Qh Qs).
vapourise_goal Hyps G Vap :-
  G = (eq L R),
```

```
  func_args (fun_type_wr L) 1 (fun_type_wr Lf)
    ((fun_type_wr La)::nil),
  func_args (fun_type_wr R) 1 (fun_type_wr Rf)
    ((fun_type_wr Ra)::nil),
  env_otype (env_otype_wr Lf) Hyps Tf,
  Tf = (Ta arrow _),
  env_otype (env_otype_wr La) Hyps Ta,
  vapourise_goal Hyps (eq La Ra) Qa,
  Qf = (forall Ta x\ (eq (Lf x) (Rf x))),
  Vap = (and Qa Qf),!.
vapourise_goal Hyps G Vap :-
  G = (eq L R),
  func_args (fun_type_wr L) 2 (fun_type_wr Lf)
    ((fun_type_wr La)::(fun_type_wr Lb)::nil),
  func_args (fun_type_wr R) 2 (fun_type_wr Rf)
    ((fun_type_wr Ra)::(fun_type_wr Rb)::nil),
  env_otype (env_otype_wr Lf) Hyps Tf,
  Tf = (Ta arrow Tb arrow _),
  env_otype (env_otype_wr La) Hyps Ta,
  env_otype (env_otype_wr Lb) Hyps Tb,
  vapourise_goal Hyps (eq La Ra) Qa,
  vapourise_goal Hyps (eq Lb Rb) Qb,
  Qf = (forall Ta x\ (forall Tb y\ (eq (Lf x y) (Rf x y)))),
  Vap = (and (and Qb Qa) Qf),!.
vapourise_goal Hyps G Vap :-
  G = (eq L R),
  func_args (fun_type_wr L) 3 (fun_type_wr Lf)
    ((fun_type_wr La)::(fun_type_wr Lb)::(fun_type_wr Lc)::nil),
  func_args (fun_type_wr R) 3 (fun_type_wr Rf)
    ((fun_type_wr Ra)::(fun_type_wr Rb)::(fun_type_wr Rc)::nil),
  env_otype (env_otype_wr Lf) Hyps Tf,
  Tf = (Ta arrow Tb arrow Tc arrow _),
  env_otype (env_otype_wr La) Hyps Ta,
  env_otype (env_otype_wr Lb) Hyps Tb,
  env_otype (env_otype_wr Lc) Hyps Tc,
  vapourise_goal Hyps (eq La Ra) Qa,
  vapourise_goal Hyps (eq Lb Rb) Qb,
  vapourise_goal Hyps (eq Lc Rc) Qc,
  Qf = (forall Ta x\ (forall Tb y\ (forall Tc z\ (eq (Lf x y z) (Rf x y z))))),
  Vap = (and (and (and Qc Qb) Qa) Qf),!.
vapourise_goal Hyps G (eq L R) :-
  G = (eq L R),
  (
    func_args (fun_type_wr L) 0 (fun_type_wr Lf)
      ((fun_type_wr La)::(fun_type_wr Lb)::(fun_type_wr Lc)::nil),
    func_args (fun_type_wr R) 0 (fun_type_wr Rf)
      ((fun_type_wr Ra)::(fun_type_wr Rb)::(fun_type_wr Rc)::nil)
  ;
    member (L$_) Hyps,
```

```
    member (R$_) Hyps
  ),!.

metaise Ze Hy Ze Hy :-
  (
    arity 0 (fun_type_wr Ze);
    member (Ze$_) Hy;
    Ze = wild_card
  ).
metaise T Hy ((meta1 N) Ma) Hn :-
  func_args (fun_type_wr T) 1 _
    ((fun_type_wr Ta)::nil),
  (  (T = ((meta1 N) _),!, H0 = Hy);
    insert_metahyp 1 N (fun_type_wr Mf) Hy H0
  ),
  metaise Ta H0 Ma Hn.
metaise T Hy ((meta2 N) Ma Mb) Hn :-
  func_args (fun_type_wr T) 2 _
    ((fun_type_wr Ta)::(fun_type_wr Tb)::nil),
  (  (T = ((meta2 N) _ _),!, H0 = Hy);
    insert_metahyp 1 N (fun_type_wr Mf) Hy H0
  ),
  metaise Ta H0 Ma H1,
  metaise Tb H1 Mb Hn.
metaise T Hy ((meta3 N) Ma Mb Mc) Hn :-
  func_args (fun_type_wr T) 3 _
    ((fun_type_wr Ta)::(fun_type_wr Tb)::(fun_type_wr Tc)::nil),
  (  (T = ((meta3 N) _ _ _),!, H0 = Hy);
    insert_metahyp 1 N (fun_type_wr Mf) Hy H0
  ),
  metaise Ta H0 Ma H1,
  metaise Tb H1 Mb H2,
  metaise Tc H2 Mc Hn.

meta_right Alist Right H Hn :-
  Alist = (nil),
  insert_metahyp 1 N (fun_type_wr _) H Hn,
  Right = ((meta1 N) ecks).
meta_right Alist Right H Hn :-
  Alist = (A1::nil),
  insert_metahyp 1 N (fun_type_wr _) H Hn,
  Right = ((meta2 N) A1 ecks).
meta_right Alist Right H Hn :-
  Alist = (A1::A2::nil),
  insert_metahyp 1 N (fun_type_wr _) H Hn,
  Right = ((meta3 N) A1 A2 ecks).
meta_right Alist Right H Hn :-
  Alist = (A1::A2::A3::nil),
  insert_metahyp 1 N (fun_type_wr _) H Hn,
```

```
  Right = ((meta4 N) A1 A2 A3 ecks).

meta_right_s Alist Right H Hn :-
  Alist = (nil),
  insert_metahyp 1 N (fun_type_wr _) H Hn,
  Right = (meta N).
meta_right_s Alist Right H Hn :-
  Alist = (A1::nil),
  insert_metahyp 1 N (fun_type_wr _) H Hn,
  Right = ((meta1 N) A1).
meta_right_s Alist Right H Hn :-
  Alist = (A1::A2::nil),
  insert_metahyp 1 N (fun_type_wr _) H Hn,
  Right = ((meta2 N) A1 A2).
meta_right_s Alist Right H Hn :-
  Alist = (A1::A2::A3::nil),
  insert_metahyp 1 N (fun_type_wr _) H Hn,
  Right = ((meta3 N) A1 A2 A3).

remove_meta G NG Hyps :-
  func_args G 1 Gf (Ga::nil),
  (
    (  Gf = (fun_type_wr (meta1 Fn)),
      member (meta_hyp Fn (fun_type_wr M)) Hyps,!
    );
    (  Gf = (fun_type_wr M) )
  ),
  remove_meta Ga (fun_type_wr Na) Hyps,
  NG = (fun_type_wr (M Na)).
remove_meta G NG Hyps :-
  func_args G 2 Gf (Ga::Gb::nil),
  (
    (  Gf = (fun_type_wr (meta2 Fn)),
      member (meta_hyp Fn (fun_type_wr M)) Hyps,!
    );
    (  Gf = (fun_type_wr M) )
  ),
  remove_meta Ga (fun_type_wr Na) Hyps,
  remove_meta Gb (fun_type_wr Nb) Hyps,
  NG = (fun_type_wr (M Na Nb)).
remove_meta G NG Hyps :-
  func_args G 3 Gf (Ga::Gb::Gc::nil),
  (
    (  Gf = (fun_type_wr (meta3 Fn)),
      member (meta_hyp Fn (fun_type_wr M)) Hyps,!
    );
    (  Gf = (fun_type_wr M) )
  ),
  remove_meta Ga (fun_type_wr Na) Hyps,
```

```
  remove_meta Gb (fun_type_wr Nb) Hyps,
  remove_meta Gc (fun_type_wr Nc) Hyps,
  NG = (fun_type_wr (M Na Nb Nc)).
remove_meta G NG Hyps :-
  Gf = G,
  (
    (  Gf = (fun_type_wr (meta Fn)),
      member (meta_hyp Fn (fun_type_wr M)) Hyps,!
    );
    (  Gf = (fun_type_wr M) )
  ),
  NG = (fun_type_wr M).

skel_hyp Skel Hyps :-
  Hy = (forall T F),
  member Hy Hyps,
  (F wild_card) = Skel.
skel_hyp Skel Hyps :- member Skel Hyps.
contains_flex (fun_type_wr Term) :-  flex Term,!.
contains_flex (fun_type_wr Term) :-
  func_args (fun_type_wr Term) N _ Terl,
  list_contains_flex Terl,!.
contains_flex (fun_type_wr Term) :-
  func_args (fun_type_wr (Term wild_card)) N _ Terl,
  list_contains_flex Terl,!.
contains_flex (fun_type_wr Term) :-
  func_args (fun_type_wr (Term wild_card wild_card)) N _ Terl,
  list_contains_flex Terl,!.
contains_flex (fun_type_wr Term) :-
  func_args (fun_type_wr (Term wild_card wild_card wild_card)) N _ Terl,
  list_contains_flex Terl,!.

list_contains_flex nil :- fail.
list_contains_flex (Arg::Args) :-
  (contains_flex Arg;list_contains_flex Args).
contains_not_wc Skel Skelf :-
  Skel = (Skelf wild_card),
  Skelm = (Skelf _),
  not (Skelm = (_ Wc),
    not (flex Wc),
    Wc = wild_card).
containo_not_wc Skel Skelf :-
  Skel = (Skelf wild_card),
  Skelm = (Skelf _),
  not (Skelm = (_ Wc),
    not (flex Wc),
    Wc = wild_card).
```

Appendix D

Examples & Results

D.1 *squs2d* & *squares*

$$\begin{aligned} squs2d\ [] &= [] & (59)\\ squs2d\ (h :: t) &= (squares\ h) :: (squs2d\ t) & (60)\\ squares\ [] &= [] & (61)\\ squares\ (h :: t) &= (h \times h) :: (squares\ t) & (62) \end{aligned}$$

$$squs2d = \lambda x.(map\ \lambda y.(map\ \lambda z.(z * z)\ y)\ x)$$

Example	Results	Time/s
squs2d	$\lambda x.(map\ \lambda y.(map\ \lambda z.(z * z)\ y)\ x)$	52
	$\lambda x.(foldr\ \lambda y.\lambda z.((map\ \lambda u.(u * u)\ y) :: z)\ []\ x)$	57
	$\lambda x.(map\ \lambda y.(foldr\ \lambda z.\lambda u.((z * z) :: u)\ []\ y)\ x)$	79
	$\lambda x.(foldr\ \lambda y.\lambda z.((foldr\ \lambda u.\lambda v.((u * u) :: v)\ []\ y) :: z)\ []\ x)$	85
	$\lambda x.(map\ \lambda y.(squares\ y)\ x)$	305
	$\lambda x.(foldr\ \lambda y.\lambda z.((squares\ y) :: z)\ []\ x)$	312
	$\lambda x.(squs2d\ x)$	1201

$$squares = \lambda x.(map\ \lambda z.(z * z)\ x)$$

Example	Results	Time/s
squares	$\lambda x.(map\ \lambda y.(y * y)\ x)$	26
	$\lambda x.(foldr\ \lambda y.\lambda z.((y * y) :: z)\ []\ x)$	31
	$\lambda x.(squares\ x)$	223

D.2 *sum2d* & *sumb*

$$\begin{aligned} sum2d\ [] &= 0 & (63)\\ sum2d\ (h :: t) &= sumb\ (sum2d\ t)\ h & (64)\\ sumb\ b\ [] &= b & (65)\\ sumb\ b\ (h :: t) &= h + (sumb\ b\ t) & (66) \end{aligned}$$

$$sum2d = \lambda x.(foldr\ \lambda y.\lambda z.(foldr\ \lambda u.\lambda v.(u + v)\ z\ y)\ 0\ x)$$

Example	Results	Time/s
$sum2d$	$\lambda x.(foldr\ \lambda y.\lambda z.(foldr\ \lambda u.\lambda v.(u+v)\ z\ y)\ 0\ x)$	39
	$\lambda x.(foldr\ \lambda y.\lambda z.(sumb\ z\ y)\ 0\ x)$	73
	$\lambda x.(sum2d\ x)$	239

$$sumb = \lambda x.\lambda y.(foldr\ \lambda u.\lambda v.(u+v)\ x\ y)$$

Example	Results	Time/s
$sumb$	$\lambda x.\lambda y.(foldr\ \lambda z.\lambda u.(z+u)\ x\ y)$	24
	$\lambda x.\lambda y.(sumb\ x\ y)$	51

D.3 $isums$

$$isums\ b\ [] = [] \tag{67}$$
$$isums\ b\ (h :: t) = (h+b) :: (isums\ (h+b)\ t) \tag{68}$$

$$isums = \lambda x.\lambda y.(scan\ \lambda z.\lambda u.(z+u)\ x\ y)$$

Example	Results	Time/s
$isums$	$\lambda x.\lambda y.(scan\ \lambda z.\lambda u.(z+u)\ x\ y)$	63
	$\lambda x.\lambda y.(scan\ \lambda z.\lambda u.(z+u)\ x\ y)$	502
	$\lambda x.\lambda y.(isums\ x\ y)$	1527

D.4 $subset$ & mem

$$subset\ []\ x = \top \tag{69}$$
$$subset\ (h :: t)\ x = (mem\ h\ x) \wedge (subset\ t\ x) \tag{70}$$
$$mem\ x\ [] = \bot \tag{71}$$
$$mem\ x\ (h :: t) = (x = h) \vee (mem\ x\ t) \tag{72}$$

$$subset = \lambda x.\lambda y.(foldr\ \lambda z.\lambda u.((foldr\ \lambda v.\lambda w.((z=v) \vee w)\ \bot\ y) \wedge u)\ \top\ x)$$

Example	Results	Time/s
$subset$	$\lambda x.\lambda y.(foldr\ \lambda z.\lambda u.($ $(foldr\ \lambda v.\lambda w.((z=v) \vee w)\ \bot\ y) \wedge u)\ \top\ x)$	63
	$\lambda x.\lambda y.(foldr\ \lambda z.\lambda u.((mem\ z\ y) \wedge u)\ \top\ x)$	282
	$\lambda x.\lambda y.(subset\ x\ y)$	2221

$$mem = \lambda x.\lambda y.(foldr\ \lambda v.\lambda w.((x=v) \vee w)\ \bot\ y)$$

Example	Results	Time/s
mem	$\lambda x.\lambda y.(foldr\ \lambda z.\lambda u.((x=z) \vee u)\ \bot\ y)$	33
	$\lambda x.\lambda y.(mem\ x\ y)$	220

D.5 *mmult*, *rowsmult*, & *transpose*

$$
\begin{aligned}
mmult\ m_1\ m_2 &= (rowsmult\ m_1)\ (transpose\ m_2) && (73)\\
rowsmult\ []\ cols &= [] && (74)\\
rowsmult\ (row :: rows)\ cols &= (rowmult\ row\ cols) :: (rowsmult\ rows\ cols) && (75)\\
rowmult\ row\ [] &= [] && (76)\\
rowmult\ row\ (col :: cols) &= (dotprod\ row\ col) :: (rowmult\ row\ cols) && (77)\\
transpose\ [] &= [] && (78)\\
transpose\ (row :: rows) &= dist\ row\ (transpose\ rows) && (79)
\end{aligned}
$$

$$mmult = \lambda x.\lambda y.(map\ \lambda z.(map\ \lambda u.(dotprod\ z\ u)\ (foldr\ \lambda u.\lambda v.(dist\ u\ v)\ []\ y))\ x)$$

Example	Results	Time/s
mmult	$\lambda x.\lambda y.(map\ \lambda z.(map\ \lambda u.(dotprod\ z\ u)$ $(foldr\ \lambda u.\lambda v.(dist\ u\ v)\ []\ y))\ x)$	103
	$\lambda x.\lambda y.(foldr\ \lambda z.\lambda u.((map\ \lambda v.(dotprod\ z\ v)$ $(foldr\ \lambda v.\lambda w.(dist\ v\ w)\ []\ y)) :: u)\ []\ x)$	113
	$\lambda x.\lambda y.(map\ \lambda z.(foldr\ \lambda u.\lambda v.((dotprod\ z\ u) :: v)\ []$ $(foldr\ \lambda u.\lambda v.(dist\ u\ v)\ []\ y))\ x)$	157
	$\lambda x.\lambda y.(foldr\ \lambda z.\lambda u.((foldr\ \lambda v.\lambda w.((dotprod\ z\ v) :: w)\ []$ $(foldr\ \lambda v.\lambda w.(dist\ v\ w)\ []\ y)) :: u)\ []\ x)$	168
	$\lambda x.\lambda y.(map\ \lambda z.(rowmult\ z\ (foldr\ \lambda u.\lambda v.(dist\ u\ v)\ []\ y))\ x)$	436
	$\lambda x.\lambda y.(foldr\ \lambda z.\lambda u.((rowmult\ z$ $(foldr\ \lambda v.\lambda w.(dist\ v\ w)\ []\ y)) :: u)\ []\ x)$	445
	$\lambda x.\lambda y.(map\ \lambda z.(map\ \lambda u.(dotprod\ z\ u)\ (transpose\ y))\ x)$	895
	$\lambda x.\lambda y.(foldr\ \lambda z.\lambda u.$ $((map\ \lambda v.(dotprod\ z\ v)\ (transpose\ y)) :: u)\ []\ x)$	905
	$\lambda x.\lambda y.(map\ \lambda z.$ $(foldr\ \lambda u.\lambda v.(9(dotprod\ z\ u) :: v)\ []\ (transpose\ y))\ x)$	944
	$\lambda x.\lambda y.(foldr\ \lambda z.\lambda u.($ $(foldr\ \lambda v.\lambda w.((dotprod\ z\ v) :: w)\ []\ (transpose\ y)) :: u)\ []\ x)$	954
	$\lambda x.\lambda y.(map\ \lambda z.(rowmult\ z\ (transpose\ y))\ x)$	1217
	$\lambda x.\lambda y.(foldr\ \lambda z.\lambda u.((rowmult\ z\ (transpose\ y)) :: u)\ []\ x)$	1226
	$\lambda x.\lambda y.(rowsmult\ x\ (transpose\ y))$	10894
	$\lambda x.\lambda y.(mmult\ x\ y)$	10899

$$rowsmult = \lambda x.\lambda y.(map\ \lambda z.(map\ \lambda u.(dotprod\ z\ u)\ y)\ x)$$

Example	Results	Time/s
rowsmult	$\lambda x.\lambda y.(map\ \lambda z.(map\ \lambda u.(dotprod\ z\ u)\ y)\ x)$	70
	$\lambda x.\lambda y.(foldr\ \lambda z.\lambda u.((map\ \lambda v.(dotprod\ z\ v)\ y) :: u)\ []\ x)$	79
	$\lambda x.\lambda y.(map\ \lambda z.(foldr\ \lambda u.\lambda v.((dotprod\ z\ u) :: v)\ []\ y)\ x)$	116
	$\lambda x.\lambda y.(foldr\ \lambda z.\lambda u.($ $(foldr\ \lambda v.\lambda w.((dotprod\ z\ v) :: w)\ []\ y) :: u)\ []\ x)$	125
	$\lambda x.\lambda y.(map\ \lambda z.(rowmult\ z\ y)\ x)$	380
	$\lambda x.\lambda y.(foldr\ \lambda z.\lambda u.((rowmult\ z\ y) :: u)\ []\ x)$	389
	$\lambda x.\lambda y.(rowsmult\ x\ y)$	2632

$$transpose = \lambda x.(foldr\ \lambda u.\lambda v.(dist\ u\ v)\ []\ x)$$

Example	Results	Time/s
$transpose$	$\lambda x.(foldr\ \lambda y.\lambda z.(dist\ y\ z)\ []\ x)$	20
	$\lambda x.(transpose\ x)$	50

D.6 *mergesort*

$$\begin{aligned} mergesort\ x &= mergeall\ (sortmall\ x) && (80)\\ mergeall\ [] &= [] && (81)\\ mergeall\ (h :: t) &= merge\ h\ (mergeall\ t) && (82)\\ sortmall\ [] &= [] && (83)\\ sortmall\ (h :: t) &= (sortm\ h) :: (sortmall\ t) && (84)\\ sortm\ [] &= [] && (85)\\ sortm\ (h :: t) &= insertm\ h\ (sortm\ t) && (86) \end{aligned}$$

$$mergesort = \lambda x.(foldr\ \lambda y.\lambda z.(merge\ y\ z)\ []\ (map\ \lambda y.(foldr\ \lambda z.\lambda u.(insertm\ z\ u)\ []\ y)\ x))$$

Example	Results	Time/s
$mergesort$	$\lambda x.(foldr\ \lambda y.\lambda z.(merge$ $(foldr\ \lambda u.\lambda v.(insertm\ u\ v)\ []\ y)\ z)\ []\ x)$	53
	$\lambda x.(foldr\ \lambda y.\lambda z.(merge\ (sortm\ y)\ z)\ []\ x)$	102
	$\lambda x.(mergeall\ (sortmall\ x))$	415
	$\lambda x.(mergesort\ x)$	417

D.7 *rev*

$$\begin{aligned} rev\ b\ [] &= b && (87)\\ rev\ b\ (h :: t) &= rev\ (h :: b)\ t && (88) \end{aligned}$$

$$rev = \lambda x.\lambda y.(foldl\ \lambda z.\lambda u.(u :: z)\ x\ y)$$

Example	Results	Time/s
rev	$\lambda x.\lambda y.(foldl\ \lambda z.\lambda u.(u :: z)\ x\ y)$	101
	$\lambda x.\lambda y.(foldl\ \lambda z.\lambda u.(u :: z)\ x\ y)$	184
	$\lambda x.\lambda y.(rev\ x\ y)$	199

Appendix E

Proofs

E.1 Synthesis

The $squs2d$ example yeilds:

$$squs2d = \lambda x.(map\ \lambda y.(map\ \lambda z.(z * z)\ y)\ x)$$

as the first result. The proof plan generated by the synthesis plan is as follows.

```
(thenPlan
 (thenPlan
  (tac (induction_tac 1))
  (andPlan
   (allPlan X\ [X/U_1] (allPlan Y\ [Y/U_2]
     (thenPlan
      (thenPlan
       (thenPlan
        (rewr_plan squs2d2)
        (thenPlan
         (tac meta_wr_split)
         (thenPlan
          (andPlan (tac vapourise) idPlan)
          (thenPlan
           (andPlan (rewr_plan assAnd2) idPlan)
           (thenPlan
            (andPlan (rewr_plan assAnd1) idPlan)
            (andPlan (rewr_plan assAnd2) idPlan)
       )))))
       (andPlan (rewr_hyp (eq (squs2d U_1) (meta1 1 U_1))) idPlan)
      )
      (andPlan
       (thenPlan
        (thenPlan (trule and_i)
         (andPlan idPlan
          (thenPlan
           (trule and_i)
           (andPlan
            (thenPlan
             (trule and_i)
             (andPlan idPlan idPlan)
            ) idPlan
        ))))
        (andPlan
         (thenPlan
          (rewr_plan idty)
          (trule axiom)
         )
         (andPlan
          (andPlan
           (thenPlan
            (rewr_plan idty)
            (trule axiom)
           )
           (thenPlan
            (thenPlan
             (tac (induction_tac 1))
             (andPlan
              (allPlan Z\ (allPlan U\
                (thenPlan
```

```
    (thenPlan
     (thenPlan
      (rewr_plan squares2)
      (thenPlan
       (tac meta_wr_split)
       (thenPlan
        (andPlan (tac vapourise) idPlan)
        (thenPlan
         (andPlan (rewr_plan assAnd2) idPlan)
         (thenPlan
          (andPlan (rewr_plan assAnd1) idPlan)
          (andPlan (rewr_plan assAnd2) idPlan)
     )))))
     (andPlan (rewr_hyp (eq (squares Z) (meta1 3 Z))) idPlan)
    )
    (andPlan
     (thenPlan
      (thenPlan
       (trule and_i)
       (andPlan idPlan
        (thenPlan
         (trule and_i)
         (andPlan
          (thenPlan
           (trule and_i)
           (andPlan
            (thenPlan
             (trule and_i)
             (andPlan idPlan idPlan)
            )
            idPlan
          ))
          idPlan
      ))))
      (andPlan
       (thenPlan
        (rewr_plan idty)
        (trule axiom)
       )
       (andPlan
        (andPlan
         (andPlan
          (thenPlan
           (rewr_plan idty)
           (trule axiom)
          )
          (thenPlan
           (rewr_plan idty)
           (trule axiom)
          )
         )
         (thenPlan
          (thenPlan
           (thenPlan
            (trule all_i)
            (allPlan V\
             (thenPlan
              (trule all_i)
              (allPlan W\ idPlan)
           )))
           (allPlan V\ (allPlan W\ idPlan))
          )
          (allPlan V\ (allPlan W\
            (thenPlan
             (rewr_plan idty)
             (trule axiom)
        )))))
        (thenPlan
         (thenPlan
          (thenPlan
           (trule all_i)
           (allPlan V\
            (thenPlan
             (trule all_i)
             (allPlan W\ idPlan)
          )))
          (allPlan V\ (allPlan W\ idPlan))
         )
         (allPlan V\ (allPlan W\
           (thenPlan
            (rewr_plan idty)
            (trule axiom)
     )))))))
     (thenPlan
      (thenPlan
       (rewr_plan lmap2)
       (rewr_plan idty)
      )
      (trule axiom)
 )))))
 (thenPlan
  (rewr_plan squares1)
  (thenPlan
   (thenPlan
```

```
                 (rewr_plan lmap1)
                 (rewr_plan idty)
                )
                (trule axiom)
            ))))
            (andPlan
             (allPlan Z\ (allPlan U\
               (andPlan
                (andPlan no_op
                 (andPlan
                  (andPlan
                   (andPlan no_op no_op)
                   (allPlan V\ (allPlan W\ no_op))
                  )
                  (allPlan V\ (allPlan W\ no_op)
                ))) no_op
             ))) no_op
          )))
          (thenPlan
           (thenPlan
            (thenPlan
             (trule all_i)
             (allPlan Z\
              (thenPlan
               (trule all_i)
               (allPlan U\ idPlan)
            )))
            (allPlan Z\ (allPlan U\ idPlan)
           ))
           (allPlan Z\ (allPlan U\
             (thenPlan
              (rewr_plan idty)
              (trule axiom)
       )))))))
       (thenPlan
        (thenPlan
         (rewr_plan lmap2)
         (rewr_plan idty)
        )
        (trule axiom)
   )))))
   (thenPlan
    (rewr_plan squs2d1)
    (thenPlan
     (thenPlan
      (rewr_plan lmap1)
      (rewr_plan idty)
     )
     (trule axiom)
 ))))
 (andPlan
  (allPlan X\ [X/U_1] (allPlan Y\ [Y/U_2]
    (andPlan
     (andPlan no_op
      (andPlan
       (andPlan no_op
        (andPlan
         (allPlan Z\ [Z/U_3] (allPlan U\ [U/U_4]
           (andPlan
            (andPlan no_op
             (andPlan
              (andPlan
               (andPlan no_op no_op)
               (allPlan V\ [V/U_5](allPlan W\ [W/U_6] no_op))
              )
              (allPlan V\ [V/U_5] (allPlan W\ [W/U_6] no_op))
            )) no_op
         ))) no_op
       ))
       (allPlan Z\ [Z/U_3] (allPlan U\ [U/U_4] no_op))
     )) no_op
  ))) no_op
))
```

E.2 Verification

The first result the *squs2d* example yeilds is:

$$squs2d = \lambda x.(map\ \lambda y.(map\ \lambda z.(z * z)\ y)\ x)$$

If we take this result and use the proof planner in verification mode we obtain the following proof plan.

```
(thenPlan
 (thenPlan
  (tac (induction_tac 1))
  (andPlan idPlan
   (allPlan X\ [X/U_1] (allPlan Y\ [Y/U_2]
     (thenPlan
      (thenPlan
       (rewr_plan squs2d2)
       (thenPlan
        (rewr_plan lmap2)
        (rewr_plan (user_rewr "consplit"))
      ))
      (rewr_hyp (eq (squs2d U_1) (lmap Z\ (lmap U\ (times U U) Z) U_1)))
 )))))
 (andPlan
  (thenPlan
   (thenPlan
    (tac redundant)
    (thenPlan
     (rewr_plan squs2d1)
     (thenPlan
      (rewr_plan lmap1)
      (rewr_plan idty)
   )))
   (trule axiom)
  )
  (allPlan X\ [X/U_1] (allPlan Y\ [Y/U_2]
    (thenPlan
     (thenPlan
      (tac redundant)
      (rewr_plan idty)
     )
     (thenPlan
      (trule and_i)
      (andPlan
       (thenPlan
        (thenPlan
         (thenPlan
          (tac (alle))
          (tac (induction_tac 1))
         )
         (andPlan idPlan
          (allPlan Z\ (allPlan U\
            (thenPlan
             (thenPlan
              (rewr_plan squares2)
              (thenPlan
               (rewr_plan lmap2)
               (rewr_plan (user_rewr  "consplit"))
              )
              (rewr_hyp (eq (squares  Z) (lmap V\ (times V V) Z)))
         )))))
         (andPlan
          (thenPlan
           (thenPlan
            (tac redundant)
            (thenPlan
             (rewr_plan squares1)
             (thenPlan
              (rewr_plan lmap1)
              (rewr_plan idty)
           )))
           (trule axiom)
          )
          (allPlan Z\ (allPlan U\
            (thenPlan
             (thenPlan
              (tac redundant)
              (thenPlan
               (rewr_plan idty)
               (rewr_plan idty)
             ))
             (thenPlan
              (thenPlan
               (trule and_i)
               (andPlan (trule axiom) (trule  axiom))
              )
              (andPlan no_op no_op)
        ))))))
        (trule axiom)
))))))))
```

Appendix F

Translators

F.1 SML to $\lambda Clam$

The following code is the abstract syntax tree to $\lambda Clam$ translation code. The code for parsing SML syntax into an abstract syntax tree representation is not included because of space considerations.

```
void wfun(AST t)
{
  AST t1;
  int n;

  n=0;
  t1 = ASTclimb(t, 'r');
  while(ASTglance(t1)==EXPFN)/* While a function */
  {                        /* Climb up*/
    t1=ASTclimb(ASTclimb(ASTclimb(t1, 'l'), 'l'), 'r');
    n++;
  }
    /* Getting the function match list */
  t1 = ASTclimb(ASTclimb(ASTclimb(t1, 'l'), 'l'), 'l');
  PCtree(t1);
}

int cfun(AST t)
{
  AST t1;
  int n;
  LIST ns;
  LIST end;
  AST this;

  if(ASTglance(t)==VBPE &&          /* If it is a simple value binding */
    ASTglance(ASTclimb(t, 'r'))==EXPFN &&  /* And the left is a function */
```

```
  ASTglance(ASTclimb(t, 'l'))==ID)    /* And the right is an identifier */
{
  n=0;                    /* The number of functions starts at one */
  ns = LISTnew(0);              /* The list of functions */
  end = ns;                  /* Point at the end */
  t1 = ASTclimb(t, 'r');            /* The function */
  while(ASTglance(t1)==EXPFN &&                   /* While a function */
    ASTglance(ASTclimb(t1, 'l'))==MATCH &&               /* And a match */
    ASTglance(ASTclimb(ASTclimb(t1, 'l'), 'l'))==RULE &&      /* And a rule */
    ASTglance(ASTclimb(ASTclimb(ASTclimb(t1, 'l'), 'l'), 'l'))==ID) /* And an id */
  {/* add id to list, increment the number of functions, and climb up*/
    end=LISTadd(end, ASTbarkstr(
                 ASTclimb(ASTclimb(ASTclimb(t1, 'l'), 'l'), 'l')), 0);
    end=LISTnext(end);
    n++;
    t1=ASTclimb(ASTclimb(ASTclimb(t1, 'l'), 'l'), 'r');
  }  /* Right so now we have
      the number of functions,
      a list of their names and
      a pointer to the body */
  if(!(ASTglance(t1)==EXPAP &&
    ASTglance(ASTclimb(t1, 'l'))==EXPBR &&
    ASTglance(ASTclimb(ASTclimb(t1, 'l'), 'l'))==EXPFN))
    return(0);
  else
  {
    this = ASTclimb(t1, 'r');
    while(this!=NULL &&  /* Checking that all the elements of ns appear */
      ASTglance(this)==EXPRL &&
      ASTglance(ASTclimb(ASTclimb(this, 'l'), 'r'))==ID)
    {
      if(LISTempty(ns)||
        strcmp(ASTbarkstr(ASTclimb(ASTclimb(this,
                         'l'), 'r')), LISTentry_val(ns)))
        return(0);
      else
      {
        LISTremove(ns);
        this = ASTclimb(this, 'r');
      }
    }
    if(n==1 && ASTglance(this)==ID &&
          !strcmp(ASTbarkstr(this), LISTentry_val(ns)))
    {
      LISTremove(ns);
    }
    if(!LISTempty(ns))
    {  LISTdel(ns);return(0);}
    LISTdel(ns);
```

```
      wfun(t);
      return(1);
    }
  }
  return(0);
}

char *clam_ids(char *id)
{
  int i;
  int tynum;
  char *cid;
  char *incid;

  if(id[0]=='\'')
  {
    if(tyke>25){printf("variable name space overflow\n");}
    if(islower(id[1]))
      incid = &id[2];
    else
      incid = &id[1];
    sscanf(incid, "%d", &tynum);
    cid = malloc(2*sizeof(char));
    for(i=0;i<26;i++)
    {
      if(tynum==ints[i])
      {
        sprintf(cid, "%c", lets[i]);
        return(cid);
      }
    }/* If not found */
    sprintf(cid, "%c", 'A'+tyke);
    sscanf(incid, "%d", &ints[tyke]);
    sscanf(cid, "%c", &lets[tyke]);
    tyke++;
    return(cid);
  }
  SWITCH("int")
    return("nat");
  SWITCH("list")
    return("olist");
  SWITCH("nil")
    return("onil");
  SWITCH("[]")
    return("onil");
  SWITCH("::")
    return("ocons");
  SWITCH("bool")
    return("boolean");
```

```
  SWITCH("false")
    return("ofalse");
  SWITCH("true")
    return("otrue");
  SWITCH("<")
    return("olt");
  SWITCH(">")
    return("ogt");
  SWITCH("<=")
    return("olte");
  SWITCH(">=")
    return("ogte");
  SWITCH("or")
    return("oor");
  SWITCH("and")
    return("oand");
  SWITCH("=")
    return("oeq");
  SWITCH("s")
    return("S");
  SWITCH("+")
    return("plus");
  SWITCH("-")
    return("minus");
  SWITCH("*")
    return("times");
  SWITCH("map")
    return("lmap");
  SWITCH("foldr")
    return("lfoldr");
  SWITCH("foldl")
    return("lfoldl");
  SWITCH("filter")
    return("lfilter");
  SWITCH("scan")
    return("lscan");
  {                  /* If id not standard */
    if(!LISTempty(LISTsearch(names, id)))  /* If id found */
      return(id);                /* return it */
    else                  /* If id not found */
    {
      cid = malloc(20*sizeof(char));      /* convert */
      cid=strcpy(cid,id);
      for(i=0;(i<20&&cid[i]!='\0');i++)    /* all letters to */
        cid[i]=toupper(cid[i]);        /* upper case letters */
      return(cid);            /* And return that */
    }
  }
}
```

```
void PCtree(AST t)
{
   AST t1;
   AST t2;
   AST t3;
  int i;

  if(t == NULL)
    return;

  switch(ASTglance(t))
  {
  case EXP :             /* expression has only a left side */
    PCtree(ASTclimb(t, 'l'));
    break;
  case EXPRL :           /* record list({left entry right[,next]}) */
    PCtree(ASTclimb(t, 'l'));  /* Print entry */
    t1 = ASTclimb(t, 'r');
    while(t1!=NULL && ASTglance(t1)==EXPRL)  /* while another entry */
    {
      PRINT(" ");
      PCtree(ASTclimb(t1, 'l'));/* Print next entry */
      t1 = ASTclimb(t1, 'r');
    }
    break;
  case EXPRE :           /* record entry (left id = right expr) */
    PCtree(ASTclimb(t, 'r'));
    break;
  case EXPBR :           /* bracketed expression */
    if(ASTglance(ASTclimb(t, 'l'))==EXPAP)
      PCtree(ASTclimb(t, 'l'));  /* Print expression */
    else
    {
      PRINT("(");
      PCtree(ASTclimb(t, 'l'));  /* Print expression */
      PRINT(")");
    }
    break;
  case EXPAP :           /* expression application */
    PRINT("(");
    if(ASTglance(ASTclimb(t, 'l'))==EXPBR &&
      ASTglance(ASTclimb(ASTclimb(t, 'l'), 'l'))==EXPFN)
    {                /* Function expression */
      if(ASTglance(ASTclimb(t, 'l'))==EXPBR)
        t1 = ASTclimb(ASTclimb(ASTclimb(t, 'l'), 'l'), 'l');
      else
        t1 = ASTclimb(ASTclimb(t, 'l'), 'l');
      if(boolmatch(t1))
```

```
    {                 /* True or false match */
      t2 = ASTclimb(ASTclimb(t1, 'l'), 'r');
      t3 = ASTclimb(ASTclimb(ASTclimb(t1, 'r'), 'l'), 'r');
      if(ASTglance(t2)==ID && !strcmp(ASTbarkstr(t2),"true"))
      {                 /* true => true match */
        PRINT("or ");
        PCtree(ASTclimb(t, 'r'));
        PRINT(" ");
        PCtree(t3);
      }else if(ASTglance(t3)==ID && !strcmp(ASTbarkstr(t3),"false"))
      {                 /* false => false match */
        PRINT("and ");
        PCtree(ASTclimb(t, 'r'));
        PRINT(" ");
        PCtree(t2);
      }else
      {                 /* If then else */
        PRINT("if ");
        PCtree(ASTclimb(t, 'r'));
        PRINT(" ");
        PCtree(t2);
        PRINT(" ");
        PCtree(t3);
      }
    }
    else
    {                 /* Case match */
    }
  }
  else
  {                 /* Normal application */
    i = 1;
    t1 = ASTclimb(t, 'l');
    if(ASTglance(t1)==ID && !strcmp(ASTbarkstr(t1),"+"))/* checking for +1 */
    {  t1 = ASTclimb(ASTclimb(ASTclimb(ASTclimb(t,'r'),'r'),'l'),'r');
      if(ASTglance(t1)==INTGR && ASTbarkint(t1)==1)
      {
        PRINT("s ");
        PCtree(ASTclimb(ASTclimb(t,'r'),'l'));
        i = 0;
      }
    }
    if(i)
    {
      PCtree(ASTclimb(t, 'l'));
      PRINT(" ");
      PCtree(ASTclimb(t, 'r'));
    }
  }
```

```
  PRINT(")");
  break;
case EXPTY :           /* expression constraint */
  PCtree(ASTclimb(t, 'l'));
  break;
case EXPLT :           /* let expression */
  PCtree(ASTclimb(t, 'l'));
  t1 = ASTclimb(t, 'r');
  if(ASTglance(t1)==EXPBR)
    t1 =ASTclimb(t1, 'l');
  if(ASTglance(t1)==EXPLT)
    PCtree(t1);
  break;
case EXPFN :           /* function expression */
  PCtree(ASTclimb(t, 'l'));
  break;
case MATCH :           /* match */
  PRINT3("\t%s%d equiv ",name,counter);
  PCtree(ASTclimb(t, 'l'));  /* Print rule */
  PRINT("\n");
  t1 = ASTclimb(t, 'r');
  while(t1!=NULL && ASTglance(t1)==MATCH)  /* while another entry */
  {
    counter++;           /* Increment the counter */
    PRINT3("\t%s%d equiv ",name,counter);
    PCtree(ASTclimb(t1, 'l'));  /* Print entry */
    PRINT("\n");
    t1 = ASTclimb(t1, 'r');
  }
  break;
case RULE :           /* rule */
  PRINT2("(%s ",name);
  PCtree(ASTclimb(t, 'l'));    /* Print pattern */
  PRINT(") => ");
  PCtree(ASTclimb(t, 'r'));    /* Print expression */
  PRINT(".");
  break;
case DEC :             /* declaration */
  PCtree(ASTclimb(t, 'l'));
  break;
case DECVL :           /* value declaration */
  PCtree(ASTclimb(t, 'l'));
  break;
case DECTY :           /* type declaration */
  break;
case DECDT :           /* datatype declaration */
  break;
case DECLO :           /* local decalaration */
  PCtree(ASTclimb(t, 'l'));
```

```
  PCtree(ASTclimb(t, 'r'));
  break;
case PAT :               /* pattern */
  PCtree(ASTclimb(t, 'l'));
  break;
case PATUN :             /* underbar */
  PRINT("_");
  break;
case PATRL :             /* pattern record */
  /* Add argument list */
  PCtree(ASTclimb(t, 'l'));    /* Print entry */
  t1 = ASTclimb(t, 'r');
  while(t1!=NULL && ASTglance(t1)==PATRL)/* while another entry */
  {
    PRINT(" ");
    PCtree(ASTclimb(t1, 'l'));/* Print entry */
    t1 = ASTclimb(t1, 'r');
  }
  break;
case PATRC :             /* pattern record flexible */
  break;
case PATRE :             /* pattern record entry */
  t1 = ASTclimb(t, 'l');
  if(ASTglance(t1)==PATRC)
    PCtree(t1);
  else
  {
    PCtree(ASTclimb(t, 'r'));
  }
  break;
case PATBR :             /* bracketed pattern */
  PRINT("(");
  PCtree(ASTclimb(t, 'l'));
  PRINT(")");
  break;
case PATCN :             /* constructor pattern */
  PRINT("(");
  PCtree(ASTclimb(t, 'l'));      /* Constructor */
  PRINT(" ");
  PCtree(ASTclimb(t, 'r'));      /* Record list */
  PRINT(")");
  break;
case PATTY :             /* pattern constraint */
  PCtree(ASTclimb(t, 'l'));      /* Constructor */
  break;
case VB :               /* value binding */
  PCtree(ASTclimb(t, 'l'));
  break;
case VBPE :             /* simple value binding */
```

```
  name = ASTbarkstr(ASTclimb(t, 'l'));
  names = LISTadd(names, name, 0);
if(temp){  file = defile;}
else{    file = opendest(name);}
  SHOW2("\tDoing %s definitions\n",name);
  PRINT2("define %s: ",name);      /* Do type definitions */
  tyke = 0;               /* Set type counter */
  ints = malloc(26*sizeof(int));    /* Set int checklist */
  lets = malloc(26*sizeof(char));    /* Set letter checklist */
  PCtree(ASTtype_is(t));        /* Doing type definition */
  free(ints);              /* Reset int checklist */
  free(lets);              /* Reset letter checklist */
  tyke = 0;               /* Reset type counter */
  PRINT(".\n");             /* Stop doing type definitions */
  SHOW2("\tDoing %s equations\n",name);
if(temp){  file = eqfile;}else
  PRINT("equations:\n");        /* Do equations */
  counter = 1;          /* Set counter */
  if(ASTglance(ASTclimb(t, 'r'))==EXPTY||
    ASTglance(ASTclimb(t, 'r'))==EXPBR)
    t1 = ASTclimb(ASTclimb(t, 'r'), 'l');
  else
    t1 = ASTclimb(t, 'r');
  if(ASTglance(t1)==EXPFN)      /* If it is a function expression */
  {  if(!cfun(t))
      PCtree(ASTclimb(t, 'r'));    /* Doing equations */
  }
  else                /* If just a value */
  {                    /* Doing equation */
    PRINT3("\t%s%d equiv ",name,counter);
    PRINT2("%s  => ",name);
    PCtree(ASTclimb(t, 'r'));      /* Print expression */
    PRINT(".\n");
  }
if(!temp)PRINT("endequations.\n");      /* Stop doing equations */
if(temp){  file = syfile;}
  PRINT("sym_eval: ");        /* Do symevals */
  SHOW2("\tDoing %s evals\n",name);
  PRINT2("%s1",name);          /* Do first symeval */
  for(i=2;i<=counter;i++)        /* Do subsequent symevals */
  {
    PRINT3(", %s%d",name,i);
  }
  PRINT(".\n");             /* Stop doing symevals */
  counter = 0;          /* Reset counter */
if(!temp){
  PRINT("end.\n\n");
  SHOW2("Closing destination file(%s)\n",name);
  if(path!=NULL)
```

```
    fclose(file);
}
  file = stdout;
  break;
case VBLIS :             /* value binding list */
  PCtree(ASTclimb(t, 'l'));
  t1 = ASTclimb(t, 'r');
  while(t1!=NULL && ASTglance(t1)==VBLIS)
  {
    PCtree(ASTclimb(t1, 'l'));
    t1 = ASTclimb(t1, 'r');
  }
  break;
case VBREC :             /* rec value binding */
  PCtree(ASTclimb(t, 'l'));
  break;
case TB :               /* type binding */
  PCtree(ASTclimb(t, 'l'));
  break;
case TBSIV :             /* simple type binding (with tyvar) */
  break;
case TBSIM :             /* simple type binding */
  break;
case TBLIS :             /* type binding list */
  break;
case DB :               /* datatype binding */
  PCtree(ASTclimb(t, 'l'));
  break;
case DBSIV :             /* simple datatype binding(wi' tyvar) */
  break;
case DBSIM :             /* simple datatype binding */
  break;
case DBLIS :             /* datatype binding list */
  break;
case CONL :             /* construct list */
  break;
case CON :               /* construct */
  break;
case TY :               /* type */
  PCtree(ASTclimb(t, 'l'));
  break;
case TYCON :             /* type constructor */
  PRINT("(");
  PCtree(ASTclimb(t, 'r'));      /* Constructor */
  t1 = ASTclimb(t, 'l');
  if(t1!=NULL)
  {
    PRINT(" ");
    PCtree(t1);            /* Type list */
```

```
  }
  PRINT(")");
  break;
case TYRL :              /* type record list */
  PCtree(ASTclimb(t, 'l'));
  t1 = ASTclimb(t, 'r');
  while(t1!=NULL && ASTglance(t1)==TYRL)
  {
    PRINT(" -> ");
    PCtree(ASTclimb(t1, 'l'));
    t1 = ASTclimb(t1, 'r');
  }
  break;
case TYRE :              /* type record entry */
  if(ASTglance(ASTclimb(t, 'r'))==TYFUN)
  {
    PRINT("(");
    PCtree(ASTclimb(t, 'r'));
    PRINT(")");
  }else
    PCtree(ASTclimb(t, 'r'));
  break;
case TYFUN :              /* function type */
  if(ASTglance(ASTclimb(t, 'l'))==TYFUN)
  {
    PRINT("(");
    PCtree(ASTclimb(t, 'l'));
    PRINT(")");
  }else
    PCtree(ASTclimb(t, 'l'));
  PRINT(" -> ");
  PCtree(ASTclimb(t, 'r'));
  break;
case TYBR :              /* bracketed type */
  PRINT("(");
  PCtree(ASTclimb(t, 'l'));
  PRINT(")");
  break;
case TYSEQ :             /* type sequence */
  t1 = ASTclimb(t, 'r');
  if(t1!=NULL) PRINT("(");
  PCtree(ASTclimb(t, 'l'));
  while(t1!=NULL)
  {
    PRINT(" ");
    PCtree(ASTclimb(t1, 'l'));
    t1 = ASTclimb(t1, 'r');
    if(t1==NULL) PRINT(")");
  }
```

```
    break;
  case TYVRS :              /* type variable sequence */
    t1 = ASTclimb(t, 'r');
    if(t1!=NULL || ASTglance(t1)==TYVRS) PRINT("(");
    PCtree(ASTclimb(t, 'l'));
    while(t1!=NULL && ASTglance(t1)==TYVRS)
    {
      PRINT(" ");
      PCtree(ASTclimb(t1, 'l'));
      t1 = ASTclimb(t1, 'r');
      if(t1==NULL || ASTglance(t1)!=TYVRS) PRINT(")");
    }
    break;
  case INTGR :            /* integer */
    i = ASTbarkint(t);
    switch(i)
    {
      case 0 :PRINT("zero");break;
      case 1 :PRINT("(s zero)");break;
      default : PRINT2("(s I%d)",ASTbarkint(t)-1);
    }
    break;
  case FLOAT :            /* real */
    PRINT2("F%f",ASTbarkrel(t));
    break;
  case STRNG :            /* string */
    PRINT2("S%s",ASTbarkstr(t));
    break;
  case ID :             /* identifier */
    PRINT2("%s",clam_ids(ASTbarkstr(t)));
    break;
  case ERROR :           /* ERROR */
    PRINT2("%s",ASTfeel(t));
    break;
  default :
    printf("Error, slippery bark!?\n");
    exit(0);
  }

  /*PRINT2("%s\n",ASTfeel(t));*/
  if(ASTtype_is(t)!=NULL)
  {
  }
}

static char *PCfist(char *fn)
{
  char *str;
  FILE *fp;
```

```
  char c;

  str = (char *)calloc(1000,sizeof(char));
  strcpy(str,"\0");

  fp = fopen(fn, "r");
  if(fp == NULL){  printf("Can't open %s.\n",fn);exit(-1);}

  c = fgetc(fp);
  while(c!=EOF)
  {
    strncat(str,&c,1);
    c = fgetc(fp);
  }

  fclose(fp);
  remove(fn);

  return(str);
}

static void PCinit(AST t)
{
  char *str;
  int nf;

  counter = 0;
  name = NULL;
  file = stdout;
  names = LISTnew(0);

  if(temp)
  {
    nf = creat("-defines", 0644);/* Create destination file */
    if(nf == -1){  printf("Could not create destination defile.");exit(-1);}
    close(nf);
    nf = creat("-equations", 0644);/* Create destination file */
    if(nf == -1){  printf("Could not create destination eqfile.");exit(-1);}
    close(nf);
    nf = creat("-symevals", 0644);/* Create destination file */
    if(nf == -1){  printf("Could not create destination syfile.");exit(-1);}
    close(nf);
    defile = fopen("-defines", "a"); /* Guard against no file. */
    if(defile == NULL){  printf("Can't open defile.\n");exit(-1);}
    eqfile = fopen("-equations", "a");
    if(eqfile == NULL){  printf("Can't open eqfile.\n");exit(-1);}
    syfile = fopen("-symevals", "a");
    if(syfile == NULL){  printf("Can't open syfile.\n");exit(-1);}
  }
```

```
  PCtree(t);

  if(temp)
  {
    fclose(defile);
    fclose(eqfile);
    fclose(syfile);
    str = PCfist("-defines");
    printf("%s", str);free(str);
    str = PCfist("-equations");
    printf("equations:\n%sendequations.\n", str);free(str);
    str = PCfist("-symevals");
    printf("%s\nend.", str);free(str);
  }

  LISTdel(names);
}
```

F.2 λ*Clam* to SML

The `mice` portrayal predicate below will display terms in SML syntax. There are additional predicates not included `lice` and `nice` which will display terms in LaTeX and in a simple pretty printed form. `lice` and `nice` also use the `nice_term` and `show_term` predicates but have their own symbol displaying predicates `show_symb_l` and `show_symb_n` instead of `show_symb_m`.

```
mice Term Str :- nice_term Term 0 Str 2.

nice_term (fun_type_wr Term) En Str F :-
  flex Term,
  (
    F is 1,show_symb_l (fun_type_wr Term) Sym;
    F is 2,show_symb_m (fun_type_wr Term) Sym;
    show_symb_o (fun_type_wr Term) Sym
  )
  ,!.
nice_term (fun_type_wr Term) En Str F :-
  not(flex Term),
  (
    Term = (ot Num);
    Term = ofalse;
    Term = otrue;
```

```
    Term = onil;
    Term = zero
  ) ,
  show_term (fun_type_wr Term) nil En Str F
  ,!.
nice_term (fun_type_wr Term) En Str F :-
  func_args (fun_type_wr (Term)) N Terf Terl,
  show_term Terf Terl En Str F
  ,!.
nice_term (fun_type_wr Term) En Str F :-
  En1 is En+1,
  func_args (fun_type_wr (Term (ot En1))) N Terf Terl,
  show_term Terf Terl En1 Ters F,
  (
    (F is 1,
      show_symb_l (fun_type_wr (ot En1)) Sym,
      concat_string ["\\lambda ",Sym,".",Ters] Str
    );
    (F is 2,
      show_symb_m (fun_type_wr (ot En1)) Sym,
      concat_string ["(fn ",Sym," => ",Ters,")"] Str
    );
    (
      show_symb_o (fun_type_wr (ot En1)) Sym,
      concat_string ["\\",Sym,Ters] Str
    )
  )
  ,!.
nice_term (fun_type_wr Term) En Str F :-
  En1 is En+1,
  En2 is En1+1,
  func_args (fun_type_wr (Term (ot En1) (ot En2))) N Terf Terl,
  show_term Terf Terl En2 Ters F,
  (
    (F is 1,
      show_symb_l (fun_type_wr (ot En1)) Syma,
      show_symb_l (fun_type_wr (ot En2)) Symb,
      concat_string ["\\lambda ",Syma,".\\lambda ",Symb,".",Ters] Str
    );
    (F is 2,
      show_symb_m (fun_type_wr (ot En1)) Syma,
      show_symb_m (fun_type_wr (ot En2)) Symb,
      concat_string ["(fn ",Syma," => fn ",Symb," => ",Ters,")"] Str
    );
    (
      show_symb_o (fun_type_wr (ot En1)) Syma,
      show_symb_o (fun_type_wr (ot En2)) Symb,
      concat_string ["\\",Syma,"\\",Symb,Ters] Str
    )
```

```
  )
  ,!.
nice_term (fun_type_wr Term) En Str F :-
  En1 is En+1,
  En2 is En1+1,
  En3 is En2+1,
  show_term Terf Terl En3 Ters F,
  func_args (fun_type_wr (Term (ot En1) (ot En2) (ot En3))) N Terf Terl,
  (
    (F is 1,
      show_symb_l (fun_type_wr (ot En1)) Syma,
      show_symb_l (fun_type_wr (ot En2)) Symb,
      show_symb_l (fun_type_wr (ot En3)) Symc,
      concat_string ["\\lambda ",Syma,".\\lambda ",Symb,
                     ".\\lambda ",Symc,".",Ters] Str
    );
    (F is 2,
      show_symb_m (fun_type_wr (ot En1)) Syma,
      show_symb_m (fun_type_wr (ot En2)) Symb,
      show_symb_m (fun_type_wr (ot En3)) Symc,
      concat_string ["(fn ",Syma," => fn ",Symb,
                     " => fn ",Symc," => ",Ters,")"] Str
    );
    (
      show_symb_o (fun_type_wr (ot En1)) Syma,
      show_symb_o (fun_type_wr (ot En2)) Symb,
      show_symb_o (fun_type_wr (ot En3)) Symc,
      concat_string ["\\",Syma,"\\",Symb,"\\",Symc,Ters] Str
    )
  )
  ,!.

show_term Terf nil N Str F :-
  (
    (F is 1,show_symb_l Terf Str);
    (F is 2,show_symb_m Terf Str);
    (show_symb_o Terf Str)
  ),!.
show_term Terf (Tera::nil) N Str F :-
  (
    (F is 1,show_symb_l Terf Sym);
    (F is 2,show_symb_m Terf Sym);
    (show_symb_o Terf Sym)
  ),
  nice_term Tera N Teras F,
  (
    (F is 1,
    concat_string ["(",Sym,"\\ ",Teras,")"] Str);
    concat_string ["(",Sym," ",Teras,")"] Str
```

```
  )
  ,!.
show_term Terf (Tera::Terb::nil) N Str F :-
  (
    (F is 1,show_symb_l Terf Sym);
    (F is 2,show_symb_m Terf Sym);
    (show_symb_o Terf Sym)
  ),
  nice_term Tera N Teras F,
  nice_term Terb N Terbs F,
  (
    (
    infix_function Terf,
    concat_string ["(",Teras,Sym,Terbs,")"] Str
    );(
    F is 1,
    concat_string ["(",Sym,"\\ ",Teras,"\\ ",Terbs,")"] Str
    );
    concat_string ["(",Sym," ",Teras," ",Terbs,")"] Str
  ),!.
show_term Terf (Tera::Terb::Terc::nil) N Str F :-
  (
    (F is 1,show_symb_l Terf Sym);
    (F is 2,show_symb_m Terf Sym);
    (show_symb_o Terf Sym)
  ),
  nice_term Tera N Teras F,
  nice_term Terb N Terbs F,
  nice_term Terc N Tercs F,
  (
    (F is 1,
    concat_string ["(",Sym,"\\ ",Teras,"\\ ",Terbs,"\\ ",Tercs,")"] Str);
    concat_string ["(",Sym," ",Teras," ",Terbs," ",Tercs,")"] Str
  )
  ,!.

show_symb_m (fun_type_wr Meta)  Str :- flex Meta, sprintf Str "Meta" [].
show_symb_m (fun_type_wr (ot 1))  "x".
show_symb_m (fun_type_wr (ot 2))  "y".
show_symb_m (fun_type_wr (ot 3))  "z".
show_symb_m (fun_type_wr (ot 4))  "u".
show_symb_m (fun_type_wr (ot 5))  "v".
show_symb_m (fun_type_wr (ot 6))  "w".
show_symb_m (fun_type_wr (ot 7))  "a".
show_symb_m (fun_type_wr (ot 8))  "b".
show_symb_m (fun_type_wr (ot 9))  "c".
show_symb_m (fun_type_wr (ot 10))  "d".
show_symb_m (fun_type_wr zero)    "0".
show_symb_m (fun_type_wr eq)    " = ".
```

```
show_symb_m (fun_type_wr times)    "*".
show_symb_m (fun_type_wr plus)    "+".
show_symb_m (fun_type_wr ocons)    "::".
show_symb_m (fun_type_wr onil)    "[]".
show_symb_m (fun_type_wr lmap)    "map".
show_symb_m (fun_type_wr lfoldr)  "foldr".
show_symb_m (fun_type_wr lscan)    "scan".
show_symb_m (fun_type_wr oor)    " or ".
show_symb_m (fun_type_wr oand)    " and ".
show_symb_m (fun_type_wr oeq)    "=".
show_symb_m (fun_type_wr ogt)    ">".
show_symb_m (fun_type_wr olt)    "<".
show_symb_m (fun_type_wr ogte)    ">=".
show_symb_m (fun_type_wr olte)    "<=".
show_symb_m (fun_type_wr ofalse)  "false".
show_symb_m (fun_type_wr otrue)    "true".
show_symb_m (fun_type_wr User)    Str :-
  user_defined_object (fun_type_wr User) Str.
show_symb_m (fun_type_wr Terf)    Str :-
  sprintf Str "~(s)p" [~Terf].
```

Bibliography

[AAA+95] Shail Aditya, Arvind, Lennart Augustsson, Jan-Willem Maessen, and Rishiyur S. Nikhil. Semantics of pH: A parallel dialect of Haskell. In Paul Hudak, editor, *Proc. Haskell Workshop, La Jolla, CA USA*, YALEU/DCS/RR-1075, pages 35–49, June 1995.

[ABC+88] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617–640, 1988.

[Abr94] S. Abramsky. Proofs as processes. *Theoretical Computer Science*, 135(1):5–9, December 1994.

[ASG97] Alessandro Armando, Alan Smaill, and Ian Green. Automatic synthesis of recursive programs: The proof-planning paradigm. In *12th IEEE International Automated Software Engineering Conference*, pages 2–9, Lake Tahoe, Nevada, USA, 1997.

[Bac88] R. C. Backhouse. An exploration of the Bird-Meertens formalism. Computing Science Notes CS 8810., Department of Mathematics and Computing Science, University of Groningen, Groningen, NL, 1988.

[Bar77] D. Barstow. A knowledge-based system for automatic program construction. In Raj Reddy, editor, *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, pages 382–388, Cambridge, MA, August 1977. William Kaufmann.

[Bar79] David R. Barstow. The roles of knowledge and deduction in program synthesis. In *Proceedings of the Sixth International Joint Conference on Artificial Intelligence*, pages 37–42, Tokyo, Japan, 1979.

[Bas98] David Basin. Logical framework based program development. *ACM Computing Surveys*, 30(3es):1–4, 1998.

[BBKM93] David Basin, Alan Bundy, Ina Kraan, and Sean Matthews. A framework for program development based on schematic proof. In *Proceedings of the 7th International Workshop on Software Specification and Design (IWSSD-93)*, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-93-231 and Edinburgh DAI Research Report 654.

[BD77] 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.

[Ben92] Dan Benanav. Recognizing unnecessary clauses in resolution based systems. *Journal of Automated Reasoning*, 9(1):43–76, August 1992.

[BHHW86] S. Biundo, B. Hummel, D. Hutter, and C. Walther. The Karlsruhe induction theorem proving system. In Joerg Siekmann, editor, *8th International Conference on Automated Deduction*, pages 672–674. Springer-Verlag, 1986. Springer Lecture Notes in Computer Science No. 230.

[Bir80] R. S. Bird. Tabulation techniques for recursive programs. *ACM Computing Surveys*, 12(4):403–417, December 1980.

[Bir87] Richard S Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987.

[BKR92] A. Bouhoula, E. Kounalis, and M. Rusinowitch. SPIKE: an automatic theorem prover. In *Proceedings of LPAR '92*, number 624 in LNAI. Springer-Verlag, July 1992.

[Ble92] Guy E. Blelloch. NESL: A nested data-parallel language. Technical Report CS-92-103, Carnegie Mellon University, School of Computer Science, January 1992.

[BM73] R. S. Boyer and J S. Moore. Proving theorems about LISP functions. In N. Nilsson, editor, *Proceedings of the third IJCAI*, pages 486–493. International Joint Conference on Artificial Intelligence, August 1973. Also available from Edinburgh as DCL memo No. 60.

[BM88] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, 1988. Perspectives in Computing, Vol 23.

[BMPP89] F. L. Bauer, B. Moeller, H. Partsch, and P. Pepper. Formal program construction by transformations-computer-aided, intuition-guided programming. *IEEE Transactions on Software Engineering*, 15(2):165–180, February 1989.

[BMS80] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An Experimental Applicative Language. In *Conference Record of the 1980 LISP Conference*, pages 136–143, Stanford University, Stanford, California, August 25–27, 1980. ACM Press.

[BSvH+91] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. Research Paper 567, Dept. of Artificial Intelligence, University of Edinburgh, 1991. In the Journal of Artificial Intelligence.

[BSvH+93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, August 1993.

[Bun83] Alan Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, 1983.

[Bun88a] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[Bun88b] Alan Bundy. The use of explicit plans to guide inductive proofs. Research Paper 349, Dept. of Artificial Intelligence, University of Edinburgh, 1988. Short version published in the proceedings of CADE-9.

[Bun99] A. Bundy. A survey of automated deduction, 1999.

[Bus90] David Busvine. Transformation techniques for functional programs. Technical Report 3, Department of Computer Science, Heriot-Watt University, 1990.

[BvHHS90a] 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.

[BvHHS90b] Alan Bundy, Frank van Harmelen, Christian Horn, and Alan Smaill. The oyster-clam system. In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *LNAI*, pages 647–648, Kaiserslautern, FRG, July 1990. Springer Verlag.

[BvHS88] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. Research Paper 413, Dept. of Artificial Intelligence, University of Edinburgh, 1988. Appeared in Journal of Automated Reasoning, 7, 1991.

[BW88] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice-Hall International Series in Computer Science. Prentice-Hall International, 1988.

[BW96] David Basin and Toby Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

[CKM+92] Dan Craigen, Sentot Kromodimoeljo, Irwin Meisels, Bill Pase, and Mark Saaltink. Eves system description. In *Conference on Automated Deduction*, pages 771–775, 1992.

[Coh83] Norman H. Cohen. Eliminating redundant recursive calls. *ACM Transactions on Programming Languages and Systems*, 5(3):265–299, July 1983.

[Col89] Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, London, 1989.

[Con83] Robert L. Constable. Programs as proofs. *Information Processing Letters*, 16(3):105–112, 1983.

[Dar77] R. M. Burstal & John Darlington. A transformation system for developing recursive programs. *Journal of the association for Computing Machinery*, 24(1):44–67, 1977.

[Dar78] J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978.

[Dar81] John Darlington. An experimental program transformation and synthesis system. *Artificial Intelligence*, 16(1):1–46, 1981.

[Der85] Nachum Dershowitz. Synthetic programming. *Artificial Intelligence*, 25(3):323–373, 1985.

[DL94] Y. Deville and K-K. Lau. Logic program synthesis. *Journal of Logic Programming*, 19/20:321–350, 1994.

[eee+92] Paul Hudak [editor], Simon Peyton Jones [editor], Philip Wadler [editor], Brian Boutel, Jon Fairbairn, Joeseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the Programming Language Haskell — A Non-strict, Purely Functional Language. Technical Report 1.2, Yale University, New Haven, Conneticut, U.S.A. and University of Glasgow, Glasgow, U.K., 1992.

[Fea82] Martin S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1 – 20, 1982.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, Wokingham, 1988.

[F.L85] H. Wössner F.L. Bauer, R. Bergahmmer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samelson, M. Wirsing. *The Munich project CIP: volume 1: the wide spectrum language CIP-L*, volume 183 of *LNCS*. Springer-Verlag, 1985.

[FP87] F.L. Bauer, H. Ehler, A. Horsch, B. Möller, H. Partsch, O. Paukner and P. Pepper. *The Munich project CIP. Volume II : the program transformation system CIP-S*, volume 292 of *LNCS*. Springer-Verlag, 1987.

[GBG99] J. Gow, A. Bundy, and I. Green. Extensions to the estimation calculus. Technical Report DAI RP 953, Institute of Representation and Reasoning, Division of Informatics, University of Edinburgh, 1999.

[GL98] S. Gorlatch and C. Lengauer. (De)composition rules for parallel scan and reduction. In *Third Working Conference on Massively Parallel Programming Models*, pages 23–32. IEEE Computer Society, 1998.

[GMW79a] M. Gordon, R. Milner, and C. Wadsworth. Edinburgh LCF. *Lecture Notes in Computer Science*, 78, 1979.

[GMW79b] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Gor88] M. J. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer, 1988.

[Gre99] C. Grelck. Shared memory multiprocessor support for SAC. *Lecture Notes in Computer Science*, 1595:38–53, 1999.

[Ham00] M. Hamdan. *A combinational framework for parallel programming using algorithmic skeletons*. Ph.D. thesis, Dept of Computing and Electrical Engineering, Heriot-Watt University, U.K., January 2000.

[Han94] Chris Hankin. *Lambda Calculi: A Guide for Computer Scientists*, volume 3 of *Graduate Texts in Computer Science*. Clarendon Press, Oxford, 1994.

[Hes91] Jane Thurmann Hesketh. *Using Middle-Out Reasoning to Guide Inductive Theorem Proving*. PhD thesis, University of Edinburgh, 1991.

[Hil76] J. Hilden. Elimination of recursive calls using a small table of "randomly" selected function values. *bit*, 16(1):60–73, 1976.

[HL78] Gérard P. Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11(1):31–55, December 1978.

[HMM86] R. Harper, D. MacQueen, and R. Milner. Standard ML. Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, March 1986.

[Hog81] C. J. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(2):372–392, April 1981.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[IB95] A. Ireland and A. Bundy. Productive use of failure in inductive proof. Research Paper 716, Dept. of Artificial Intelligence, University of Edinburgh, 1995. In JAR 16 (1-2).

[INM84] INMOS, Ltd. *The occam programming manual*. Prentice-Hall International, Englewood Cliffs, 1984.

[Ire92] Andrew Ireland. The use of planning critics in mechanizing inductive proofs. In A. Voronkov, editor, *Proceedings of the International Conference on Logic Programming and Automated Reasoning (LPAR'92)*, volume 624 of *LNAI*, pages 178–189, St. Petersburg, Russia, July 1992. Springer Verlag.

[Ire96] Andrew Ireland. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, March 1996.

[KB70] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297. Pergamon Press, 1970.

[KB81] Elaine Kant and David R. Barstow. The refinement paradigm: The interaction of coding and efficiency knowledge in program synthesis. *IEEE Transactions on Software Engineering*, 7(5):458–471, September 1981. Special Section on Programming Environments.

[KB90] B. Krieg-Brückner. PROgram development by SPECification and TRAnsformation. *Technique et Science Informatiques. Special Issue on Advanced Software Engineering in ESPRIT*, 1990.

[KBB93a] I. Kraan, D. Basin, and A. Bundy. Logic program synthesis via proof planning. In K. K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, pages 1–14. Springer-Verlag, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-92-244 and Edinburgh DAI Research Report 603.

[KBB93b] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for logic program synthesis. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*. MIT Press, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-93-214 and Edinburgh DAI Research Report 638.

[KBB96] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for synthesis and induction. *Journal of Automated Reasoning*, 16(1–2):113–145, 1996. Also available from Edinburgh as DAI Research Paper 729.

[KNZ91] D. Kapur, P. Narendran, and H. Zhang. Automating inductionless induction by test sets. *Journal of Symbolic Computation*, 11:83–111, 1991.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Upper Saddle River, NJ 07458, USA, second edition, 1988.

[Kra94] I. Kraan. *Proof Planning for Logic Program Synthesis*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1994.

[KZ95] D. Kapur and H. Zhang. An overview of rewrite rule laboratory (RRL). *J. of Computer Mathematics with Applications*, 29(2):91–114, 1995.

[Lan66] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[LdM01] David Lacey and Oege de Moor. Imperative program transformation by rewriting. *Lecture Notes in Computer Science*, 2027:52–68, 2001.

[LKKB99] C. Lueth, H. Tej Kolyang, and B. Krieg-Brueckner. TAS and IsaWin: Tools for transformational program development and theorem proving. *Lecture Notes in Computer Science*, 1577:239–244, 1999.

[LP90] K. K. Lau and S. D. Prestwich. Top-down synthesis of recursive logic procedures from first-order logic specifications. In Peter Warren, David H.D.; Szerdei, editor, *Proceedings of the 7th International Conference on Logic Programming (ICLP '90)*, pages 667–684, Jerusalem, June 1990. MIT Press.

[LP92] Shie-Jue Lee and David A. Plaisted. Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning*, 9(1):25–42, August 1992.

[LRS00] D. Lacey, J. D. C. Richardson, and A. Smaill. Logic program synthesis in a higher order setting. In J. Lloyd, V. Dahl, U. Furbachand, and P.J. Stuckey, editors, *Proceedings of the First International Conference on Computational Logic (CL2000)*, volume Vol 1861 of *LNCS*, pages 87–100. Springer Verlag, 2000.

[Mad92] P. Madden. Automatic Program Optimization Through Proof Transformation. In D. Kapur, editor, *CADE11*, pages 446–461. Springer Verlag, 1992. Lecture Notes in Computer Science No. 607. Also available from Edinburgh as DAI Research Paper 604.

[MBS99] P. Madden, A. Bundy, and A. Smaill. Recursive program optimization through inductive synthesis proof transformation. *Journal of Automated Reasoning*, 22(1):65–115, January 1999.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, April 1960.

[Mes93] Message Passing Interface Forum. MPI: A message passing interface. In *Proc. Supercomputing '93*, pages 878–883. IEEE Computer Society, 1993.

[Mic68] Donald Michie. "Memo" functions and machine learning. *Nature*, pages 19–22, April 1968.

[Mic93] Gregory Michaelson. *Interpreter prototypes from formal language definitions.* PhD thesis, Heriot-Watt University, April 1993.

[MIP97] G. Michaelson, A. Ireland, and P.King. Towards a skeleton based parallelising compiler for SML. In C. Clack, T. Davie, and K.Hammond, editors, *Proceedings of 9th International Workshop on Implementation of Functional Languages*, pages 539–546, University of St Andrews, September 1997.

[ML79] 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.

[MN88] D. Miller and G. Nadathur. An overview of λProlog. In R. Bowen, K. & Kowalski, editor, *Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic Programming.* MIT Press, 1988.

[MSA$^+$83] J. R. McGraw, S. K. Skedzcelewski, S. Allan, D. Grit, R. Oldehoeft, J. R. W. Glauert, I. Dobes, and P. Hohensee. *SISAL: Streams and Iteration in a Single-Assignment Language.* Department of Computer Science, University of Manchester, 1983.

[MSBK01] G. Michaelson, N. Scaife, P Bristow, and P King. Nested algorithmic skeletons from higher order functions. *Parallel Algorithms and Applications*, 16:181–206, 2001.

[MTH90] Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[MW79] Z. Manna and R.J. Waldinger. Synthesis: Dreams ⇒ Programs. *Transactions on Software Engineering*, 5(4):294–328, 1979. Also: Stanford University, Department of Computer Science, Research Report No. STAN-CS-77-630, also numbered as Artificial Intelligence Laboratory, Memo nr. AIM-302.

[MW80] Z. Manna and R.J. Waldinger. A deductive approach to program synthesis. *Journal of Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[OR93] E.-R. Olderog and S. Rössig. A case study in transformational design of concurrent systems. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT'93: Theory and Practice of Software Development*, volume 668, pages 90–104. Springer-Verlag, 1993.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS : An integrated approach to specification and verification. Tech report, SRI International, 1992.

[Pau83] Lawrence Paulson. Tactics and tacticals in Cambridge LCF. Technical report, University of Cambridge Computer Laboratory, 1983.

[Pau86] L.C. Paulson. Natural deduction as higher order resolution. *Journal of Logic Programming*, 3:237–28, 1986.

[Pey87] S. L. Peyton Jones. GRIP: A parallel graph reduction machine. In Giles Kahn, editor, *Functional Programming Languages and Computer Architecture*. Springer Verlag, September 1987.

[PK82] Robert Paige and Shaye Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454, July 1982.

[Plo70] G. D. Plotkin. A note on inductive generalization. *Machine Intelligence, Edinburgh Univ. Press*, 5:153–163, 1970. Edinburgh Univ. Press, Edinburgh.

[Plo71] G. D. Plotkin. A further note on inductive generalization. *Machine Intelligence, Edinburgh Univ. Press*, 6:101–124, 1971. Edinburgh Univ. Press, Edinburgh.

[Plo72] G. Plotkin. Building in equational theories. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 73–90, Edinburgh, Scotland, 1972. Edinburgh University Press.

[PP96] Alberto Pettorossi and Maurizio Proietti. Rules and strategies for transforming functional and logic programs. *ACM Computing Surveys*, 28(2):360–414, June 1996.

[PS83] H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Computing Surveys*, 15(3):199–236, September 1983.

[Ric95] Julian D.C. Richardson. *The Use of Proof Plans for Transformation of Functional Programs by Changes of Data Type*. PhD thesis, University of Edinburgh, 1995.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[RSG98] J.D.C Richardson, A. Smaill, and Ian Green. System description: proof planning in higher-order logic with lambdaclam. In Claude Kirchner and Hélène Kirchner, editors, *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, Lindau, Germany, July 1998.

[RW69] G. Robinson and L. T. Wos. Paramodulation and first-order theorem proving. In B. Meltzer and D. Mitchie, editors, *Machine Intelligence 4*, pages 135–150. Edinburgh University Press, 1969.

[SA92] Rolf Socher-Ambrosius. How to avoid the derivation of redundant clauses in reasoning systems. *Journal of Automated Reasoning*, 9(1):77–97, August 1992.

[SBMK98] N. Scaife, P. Bristow, G. Michaelson, and P. King. Engineering a parallel compiler for SML. *Proceedings of 10th International Workshop on Implementation of Functional Languages*, pages 213–226, September 1998.

[SG95] Alan Smaill and Ian Green. Automating the synthesis of functional programs. Research paper 777, Dept. of Artificial Intelligence, University of Edinburgh, 1995.

[Sha96] N. Shankar. Steps towards mechanizing program transformations using PVS. *Science of Computer Programming*, 26(1–3):33–57, 1996.

[SI99] J. Stark and A. Ireland. Towards automatic imperative program synthesis through proof planning. In *The* 14^{th} *IEEE International Conference on Automated Software Engineering*, pages 44–51. IEEE Computer Society, 1999.

[Ski95] David B. Skillicorn. *Foundations of parallel programming*. Number 6 in International series on parallel computation. Cambridge University Press, 1995.

[Smi90] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

[ST98] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[Sta00] J. Stark. *Proof Planning for Imperative Program Development*. PhD thesis, Heriot-Watt University, 2000.

[Sti81] M.E. Stickel. A unification algorithm for associative-commutative functions. *Journal of the Association for Computing Machinery*, 28(3):423–434, 1981.

[Tur75] D. A. Turner. *SASL Language Manual*. Univ. of St. Andrews, CS Dept., St. Andrews, 1975.

[Tur85] David A. Turner. Miranda: A non-strict functional language with polymorphic types. In J. Jouannaud, editor, *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architectures, Nancy, France*, volume 201, pages 1–16. Springer-Verlag, New York, NY, 1985.

[Ull94] Jeffrey D. Ullman. *Elements of ML programming*. Prentice Hall International Editions, 1994.

[Wad88] P. Wadler. Deforestration:transforming programs to eliminate trees. Technical report, University of Glasgow (UK), January 1988.

[Wal93] C. Walther. Combining induction axioms by machine. In *Proceedings of IJCAI-93*, pages 95–101. International Joint Conference on Artificial Intelligence, 1993.

[Wal94] C. Walther. Mathematical induction. In C. J. Hogger D. M. Gabbay and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 2, pages 122–227. Oxford University Press, Oxford, 1994.

[Wan80] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, January 1980.

[Weg76] B. Wegbreit. Goal-directed program transformation. *IEEE Transactions on Software Engineering*, 2(2):69–80, 1976.

[WL69] R. J. Waldinger and R. C. T. Lee. PROW: A step toward automatic program writing. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, pages 241–252, Washington, D. C., U. S. A., 1969.

[WRCS67] Lawrence Wos, George A. Robinson, Daniel F. Carson, and Leon Shalla. The concept of demodulation in theorem proving. *Journal of the ACM*, 14(4):698–709, October 1967.