DOI 10.1007/s00165-004-0054-5
BCS © 2005
Formal Aspects of Computing (2005) ■: 1–20

Formal Aspects of Computing

Discovering applications of higher order functions through proof planning

A. Cook[1], A. Ireland[1], G. Michaelson[1] and N. Scaife[2]

[1]School of Mathematical and Computer Sciences, Heriot-Watt University Riccarton, Edinburgh, Scotland
Email: {ceeajc, air, greg}@macs.hw.ac.uk, Tel.: +44-131-451 {4179, 3409, 3422}
[2]VERIMAG, Centre Equation 2, Ave de Vignat, 38610 Griers, France
Email: Norman.Scaife@imag.fr, Tel.: +33-4-76-63-48-56

Abstract. The close association between higher order functions (HOFs) and algorithmic skeletons is a promising source of automatic parallelisation of programs. A theorem proving approach to discovering HOFs in functional programs is presented. Our starting point is proof planning, an automated theorem proving technique in which high-level proof plans are used to guide proof search. We use proof planning to identify provably correct transformation rules that introduce HOFs. The approach has been implemented in the $\lambda Clam$ proof planner and tested on a range of examples. The work was conducted within the context of a parallelising compiler for Standard ML.

1. Introduction

1.1. Parallel Functional Programming and Higher Order Functions

Pure functional languages, satisfying the Church-Rosser property of evaluation order independence, have long been proposed as a basis for parallel programming. Thus, Wegner [Weg71] observed in 1971:

> *"Note that [the Church-Rosser theorem] essentially states that lambda expressions can be evaluated by asynchronous multiprocessing applied in arbitrary order to local subexpressions." page 185*

Early work on functional parallelism focused on reduction of Curry combinators [Tur79], originally developed as a succinct acclimatization of computability, which may be lifted automatically from functional programs, but these proved of too fine granularity for efficient parallel evaluation [Sto84]. Somewhat more success was obtained from parallel reduction of super-combinators [Hug84] extracted from programs written in lazy languages such as Haskell, found by lifting maximal free expressions from functional programs, but these are necessarily program specific and hence of unpredictable efficacy for parallelism.

More recently, attention has focused on higher order functions (HOFs) as predictable behaviour, appropriate granularity, general purpose loci of potential parallelism. HOFs are abstractions over functions which may subsequently be specialised with appropriate function arguments for use in specific domains. For example, `map`:

```
fun map f [] = [] |
   map f (h::t) = f h::map f t
fn : ('a -> 'b) -> 'a list -> 'b list
```

applies a function `f` to every element of a list to form a new list:

Correspondence and offprint requests to: Andrew Cook, School of Mathematical and Computer Sciences, Heriot-Watt University, Riccarton, Edinburgh, Scotland, EH14 4AS. Email: ceeajc@macshw.ac.uk

`map` f $[e_1, e_2, \ldots, e_N] \Rightarrow [f\,e_1, f\ e_2, \ldots, f\ e_N]$

For example, `foldr`

```
fun foldr f b [] = b |
   foldr f b (h::t) = f h (foldr f b t)
fn : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b
```

applies a function `f` "between" each element of a list, from left to right:

`foldr f b` $[e_1, e_2, \ldots, e_N] \Rightarrow$ `f` e_1 (`f` e_2 (`f`...(`f` e_N `b`)...))

HOFs originate in pre-computer mathematical logic, notably for formalisations of notions of computability. Thus, Church's λ calculus [Chu41], one of the foundations of contemporary functional languages, is a pure language of function abstraction, and HOFs are implicit in Kleene's recursive function schemes [Kle52]. However, the term HOF is more popularly applied to functions which generalise operations over types, like `map` and `foldr` above, which came to prominence in the late 1950's and early 1960's as people began to explore the relationships between formal computation and practical computing. For example, McCarthy's LISP [McC60] supported mapping primitives which apply functions across lists, as well as a rather clumsy λ calculus-like notation for function abstraction. For example, Gordon [Gor73] cites Barron and Strachey [BS66] as the origin of `lit`, a `foldr` precursor.

HOFs were further popularised by Backus' influential Turing Award paper of 1978 [Bac78]. This introduced the FP approach to program construction, based on higher order combining forms, supported by an algebra for proving properties of composed forms. FP was primarily a language of sequences. Contemporaneously, Bird and collaborators started to explore what was to become the Bird-Meertens Formalism (BMF), a more general algebra of higher order combining forms [Bd97]. Such formalisms provide a strong basis for proving and transforming sequential programs based on HOFs. For example, writing `o` for an infix function composition operator:

`(f o g) x` $\Leftrightarrow$ `f (g x)`

then a composition of `map`s of functions is equivalent to a `map` of the composition of those functions:

`(map f) o (map g)` $\Leftrightarrow$ `map(f o g)`

For example, `map` and `foldr` are related by:

`map` f $\Leftrightarrow$ `foldr (fn` h `=> fn` t `=>` $f\,h$`::`t`) []`

Skillicorn[Ski94] has investigated the construction of accurate cost models for parallel programs using a BMF-based approach. One of our long term aims is to use such cost models to automatically transform programs to optimise useful parallelism located in HOFs.

1.2. HOFs and Skeletons

We have been investigating an approach to automatic parallelisation of functional programs [MIK97] which draws heavily on Cole's foundational characterisation of algorithmic skeletons and their equivalent HOFs [Col89]. An algorithmic skeleton is an abstraction from a pattern of program control, usually parallel control, which may be instantiated with problem specific operations.

For example, `map` may be realised by a skeleton based on a process farm. A farmer communicates with workers which all run the same process `f`. The farmer receives a sequence of values, sends each to a free worker, receives the processed results and assembles them into the correct final sequence – Fig. 1.

For example, a `foldr` with an associative and commutative argument function f, and a base value b which is an identity for f:

Associativity:	$f\ (f\ x\ y)\ z$	$\Leftrightarrow$	$f\ x\ (f\ y\ z)$
Commutivity:	$f\ x\ y$	$\Leftrightarrow$	$f\ x$
Identity:	$f\ x\ b$	$\Rightarrow$	x

may be realised as a divide and conquer skeleton. The divide phase splits a sequence of values and passes them to further divide and conquer stages. For single values e_I and e_{I+1}, $f\,e_I\ e_{I+1}$ is returned to the previous stage's

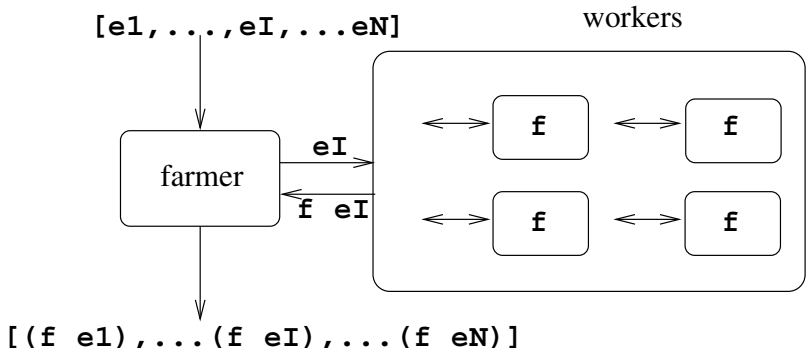

Fig. 1. Process farm

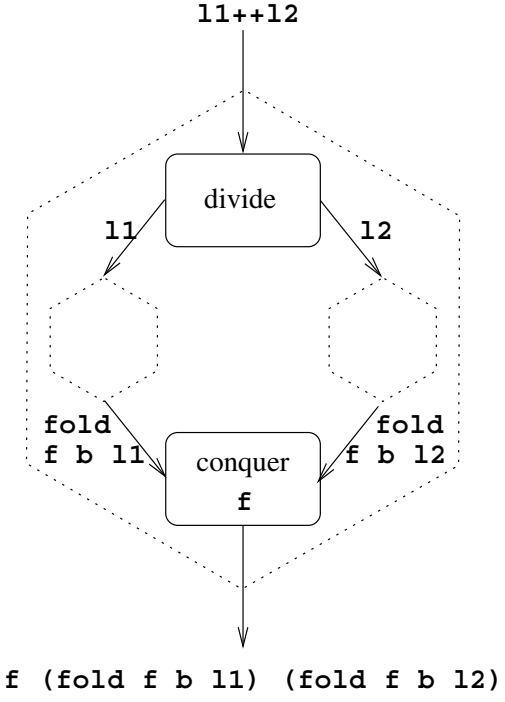

Fig. 2. Divide and conquer

conquer phase, for further combination with f – Fig. 2. Where the function f is not associative or commutative, it may be possible to restructure it in the form:

$$f\ x\ y\ =\ g\ (h\ x)\ y$$

enabling the realisation of `foldr` as a `foldr` of `map`:

`foldr` f `[]` l $\Leftrightarrow$ `foldr` g `[]` `(map` h l`)`

where there may still be farm parallelism from `map`.

The `map`/`foldr` equivalence noted above enables the replacement of a process farm by a divide and conquer network. Hamdan [Ham00] discusses the use of this equivalence, and the special case of `foldr` as `foldr` of `map` mentioned earlier, for skeleton parallelism in his EKTRAN system.

For example, composition may be realised as a process pipeline – Fig. 3.

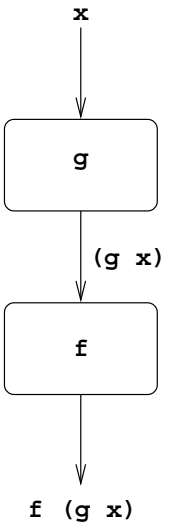

Fig. 3. Pipeline

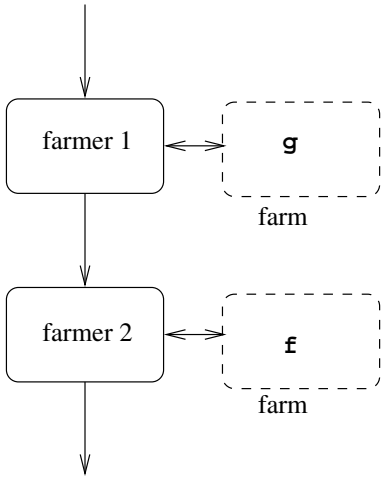

Fig. 4. Pipe of farms

Note that transformations on HOFs have equivalences in restructuring skeletons. For example, the distribution of `map` over compose discussed above is equivalent to restructuring a pipeline of farms (Fig. 4) as a farm of pipelines (Fig. 5). This has been exploited in the parallel implementation of a perspective inversion algorithm for scene/model matching as process farms [WMM+92].

1.3. Example – Matrix Multiplication

Consider matrix multiplication:

$$\begin{bmatrix} x_{11} & \ldots & x_{1n} \\ \ldots & \ldots & \ldots \\ x_{m1} & \ldots & x_{mn} \end{bmatrix} * \begin{bmatrix} y_{11} & \ldots & y_{1m} \\ \ldots & \ldots & \ldots \\ y_{n1} & \ldots & y_{nm} \end{bmatrix} = \begin{bmatrix} z_{11} & \ldots & z_{1m} \\ \ldots & \ldots & \ldots \\ z_{m1} & \ldots & z_{mm} \end{bmatrix}$$

where:

$$z_{ij} = x_{i1} * y_{1j} + x_{i2} * y_{2j} + \ldots + x_{in} * y_{nj}$$

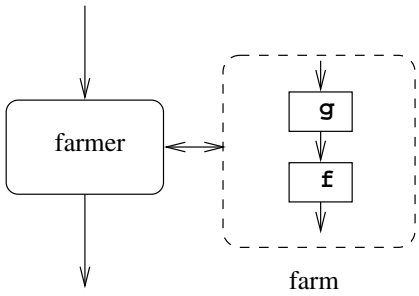

Fig. 5. Farm of pipe

In a pure functional setting, a matrix may be represented as a list of lists of values:

$$[[x_{11}, ..., x_{1n}],\ ...\ [x_{m1}, ..., x_{mn}]]$$

With this representation, multiplication is greatly simplified if the second matrix is transposed to align its columns with the rows of the first matrix:

$$\begin{array}{ll} [[y_{11}, y_{12}, ..., y_{1n}], & [[y_{11}, y_{21}, ..., y_{m1}], \\ [y_{21}, y_{22}, ..., y_{2n}], & [y_{12}, y_{22}, ..., y_{m2}], \\ \quad ... \quad \Longrightarrow & \quad ... \\ [y_{m1}, y_{m2}, ..., y_{mn}]] & [y_{1n}, y_{2n}, ..., y_{mn}]] \end{array}$$

which may be achieved by:

```
fun dist [] [] = [] |
    dist (h1::t1) (h2::t2) = (h1::h2)::dist t1 t2;

fun transpose [] = [] |
    transpose (row::rows) = dist row (transpose rows);
```

Given a row and a transposed column as lists, the dot product may be found by:

```
fun dotprod (h1::t1) (h2::t2) = h1*h2+dotprod t1 t2 |
    dotprod _ _ = 0;
```

One row of the first matrix may then be multiplied by all columns of the second matrix to form the corresponding row of the result matrix:

```
fun rowmult  _ [] = [] |
    rowmult row (col::cols) = dotprod row col::rowmult row cols;
```

Similarly, all rows of the first matrix may be multiplied by all columns of the second matrix to form the whole result matrix:

```
fun rowsmult [] _ = [] |
    rowsmult (row::rows) cols = rowmult row cols::rowsmult rows cols;
```

Thus, to multiply two matrices represented as lists of lists, the second is transposed with `transpose` to enable the multiplication of all rows of the first by all columns of the second with `rowsmult`:

```
fun mmult m1 m2 = rowsmult m1 (transpose m2);
```

The above formulation contains no HOFs. However, `transpose` may be reformulated as a `foldr`:

```
val transpose = foldr dist [];
```

and `rowmult` and `rowsmult` may be reformulated as `maps`:

```
fun rowmult row = map (dotprod row);

fun rowsmult rows cols =
     map (fn row => map (dotprod row) cols) rows;
```

Here we deploy *partial application (η-reduction)* from λ-calculus where $\lambda\, x.f\ x$ is rewritten as f. In an SML function definition, if the function body is a function application whose rightmost actual parameter is the defined function's rightmost formal parameter, then both occurrences of that parameter may be dropped. For example, `fun transpose l = foldr dist [] l` is equivalent to `val transpose = foldr dist []`, where `fun` is actually a shorthand variant of a general `val` definition where an identifier is associated with a function value.

Using the `map/foldr` equivalence discussed above, `rowmult` and `rowsmult` may also be reformulated as `foldrs`:

```
fun rowmult row =
     foldr (fn col => fn cols => dotprod row col::cols) [];

fun rowsmult rows cols =
 foldr (fn row' => fn rows' => rowmult row' cols::rows') [] rows;
```

Thus, `rowsmult` may be realised as a `map` of `map`, a `map` of `foldr`, a `foldr` of `map` or a `foldr` of `foldr`, with equivalent skeleton implementations. The implementation of matrix multiplication for arbitrary length integers by our compiler, through `map/map` realised as nested process farms, is discussed in [MSBK01]. Hamdan [Ham00] explores in considerable depth the choice of sequential or parallel realisations of `map` and `foldr` for the same problem in his EKTRAN system.

1.4. Why Discover HOFs?

While HOFs are powerful abstract programming components, in practice they appear to be under used, and thus opportunities for their parallel exploitation in extant programs are limited. There may be two reasons for this.

First of all, HOFs are seen by students and staff as a relatively hard topic in functional programming courses, just as recursion is often treated as an advanced topic in courses on imperative programming. For example, Paulson [Pau96] makes this explicit in the Preface to his widely used Standard ML text:

> *"The organization reflects my experience with teaching. Higher-order functions appear late, in Chapter 5. They are usually introduced at the very beginning with some contrived example that only confuses students. Higher-order functions are conceptually difficult and require thorough preparation."* page xv

Secondly, HOF use often appears to be forced. Consider the multiplication of all rows by all columns in the matrix multiplication example above. Here, the use of `map` seems relatively uncontroversial, as it is intuitive that this involves multiplying each row by all columns. In contrast, the use of `foldr` seems unnatural as the function argument `f` must take account of the treatment of the other rows as well as of the current row.

Nonetheless, HOF use facilitates skeleton based parallelisation, and apparently unnatural HOF use may aid parallel efficiency. Thus, we deem the discovery of HOFs, in programs that lack them, as eminently worthwhile for the automatic exploitation of parallelism.

1.5. Related Work

There has been considerable research into the exploitation of parallelism through skeletons expressed functionally. For example, Kelly's Caliban[Kel89] added a functional coordination layer to the lazy language Miranda, which enabled abstraction over parallel processing constructs. Here, HOFs proved a natural basis for generic parallel constructs. Similarly, GpH [HRP00] adds *strategies* to Haskell, where sequential processing is complemented by parallel distribution. In GpH, considerable use is made of Haskell's lazy evaluation model which enables parallel invocations to be spread across the graph for a function without forcing evaluation. Again, HOFs prove a natural basis for generic parallel strategies. Eden [PR01], is a Haskell variant with explicit, strict, point to point communication over channels. Once again, skeletons based on HOFs are employed to generalise parallelism.

In all these approaches, skeletons must be crafted and nominated explicitly by the programmer. However, in general, parallel programming is considerably harder than sequential programming, requiring deep under-

standing of the underlying processor and comunication architecture. We think that parallelism is essentially an implementation technique for gaining performance improvements from programs, and that compilers should extract parallelism from programs automatically. Thus, in our conception, the programmer need not know that HOF use will be the locus of parallelism or that the system may attempt to discover HOFs with useful parallelism in programs that lack them.

2. Proof Planning

To restate the problem, given an arbitrary function we seek to change it to use instances of applications of HOFs with known equivalent parallel skeletons. Our approach is to speculate that there is some instance of a schema for HOF application that corresponds to the original function. We then systematically deploy meaning preserving transformations across the equivalence until the schema is fully instantiated. We view such discovery as an application of theorem proof. Thus, rather than using a transformation or rewrite system, we have chosen to base our approach on tactic-based theorem proving [GMW79].

Designed to support interactive proof, tactic-based reasoning allows primitive proof steps to be packaged-up into programs known as *tactics*. This programming style of proof construction raises the level at which user interaction takes place. However, even working with the best tactic-based theorem provers, a user typically has a strong expectation as to the overall structure a proof will take and knows how to influence the prover's search strategy accordingly. Such skilled users can also be identified by their ability to analyse partial proofs, using the insights gained to again influence the prover's search.

We are aiming at strategies that automate these higher-level theorem proving activities that are typically seen as requiring user interaction. Our starting point is *proof planning* [Bun88], an approach to theorem proving that builds upon tactic-based reasoning. Unlike tactic-based reasoning, proof planning was designed initially to support automation. Starting with a set of general purpose tactics, plan formation techniques are used to construct a customised tactic for a given conjecture. The search for a customised tactic is constrained by a set of *methods*, each of which specifies the applicability of a general purpose tactic. Collectively a set of methods is known as a *proof plan* and defines a family of proofs.

Working with proof plans, instead of primitive proof steps or tactics, brings us closer to the level at which the skilled user works. However, what clearly distinguishes proof planning from other approaches to theorem proving is the support it provides for *middle-out reasoning* [BSH90] and *proof critics* [Ire92, IB96]. Middle-out reasoning is a technique that supports a schematic style of proof construction and which has been applied very successfully to the problem of program synthesis [KBB96, ASG97, SI99], one of the most challenging application areas for theorem proving.

Under-pinned by the notion of a proof plan, middle-out reasoning and proof patching via critics have therefore led to the development of some very powerful strategies. This success can be attributed to the flexibility that the techniques provide in terms of the order in which proofs are constructed, e.g. instead of relying upon the user knowing in advance which lemmas to manually supply to the theorem prover, the proof planning approach enables auxiliary lemmas to be constructed automatically during the course of a proof. Here we use the power of middle-out reasoning to guide the discovery of higher order terms within the context of transformation rules.

3. A Proof Plan for Transformation Verification

We begin by considering the problem of verifying the correctness of transformation rules that introduce HOFs. In particular we describe a proof plan that addresses this verification problem. This verification proof plan provides the foundations for our discovery proof plan that is described in Section 4.

Our application involves reasoning about recursively defined functions, as a cosequence proof by mathematical induction plays a central role. We build upon a proof plan for induction, and in particular a sub-plan called *rippling*. Here we outline the general pattern of rippling and present an example of a ripple guided proof. This paper deals with an application of rippling, for a definitive account the interested reader is directed to [BW96, BSvH$^+$93].

3.1. The General Pattern of Rippling

We are interested in inductive conjectures that take the following general form:

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

Proof by mathematical induction gives rise to base and step case proof obligations. Here we will consider the pattern of proof associated with an arbitrary step case. Assuming an induction on the variable x, then within a step case we will have an induction hypothesis of the form:

$$\forall y'.(f\ t\ y' = g\ t\ y') \tag{1}$$

and an induction conclusion (goal) of the form:

$$\forall y.(f\ (c\ t)\ y = g\ (c\ t)\ y) \tag{2}$$

where $(c\ t)$ denotes the induction term. Note that there exists a syntactic similarity between goal (2) and hypothesis (1). Rippling provides a heuristic for selecting rewrite rules that preserve the syntactic similarity while at the same time making progress to achieving a match between the goal and hypothesis. This involves distinguishing those terms within the goal that differ from the given hypothesis and is achieved by the use of meta-level annotations. In the case of our general step case example, the annotated version of goal (2) takes the form:

$$\forall y \cdot (f\ \boxed{(c\ t)}^{\uparrow}\ \lfloor y \rfloor = g\ \boxed{(c\ t)}^{\uparrow}\ \lfloor y \rfloor) \tag{3}$$

Note that the annotated part of the term, represented by the shading, denotes a mismatch between the goal and the given induction hypothesis. These differences are called *wave-fronts*. The uparrow records the direction in which the wave-front is being moved with respect to the unannotated term structure. A down arrow can also be associated with a wave-front. Directed wave-fronts enable the termination of rippling to be guaranteed, thus eliminating the problem of orienting rewrite rules that occurs within other theorem provers. In addition to wave-fronts, the $\lfloor \ldots \rfloor$ annotation is used to highlight subterms within the goal that correspond to universally quantified variables within the induction hypothesis. These subterms denote positions where wave-fronts can be eliminated via matching with the induction hypothesis. Using annotated goal (3), the general pattern of a ripple proof is shown below:

$$\forall y.(f\ \boxed{(c\ t)}^{\uparrow}\ \lfloor y \rfloor = g\ \boxed{(c\ t)}^{\uparrow}\ \lfloor y \rfloor) \tag{4}$$

$$\forall y.(\boxed{c_1\ t\ y\ (\ f\ t\ \lfloor (c_2\ t\ y) \rfloor\ )}^{\uparrow} = g\ \boxed{(c\ t)}^{\uparrow}\ \lfloor y \rfloor) \tag{5}$$

$$\forall y.(\boxed{c_1\ t\ y\ (\ f\ t\ \lfloor (c_2\ t\ y) \rfloor\ )} = \boxed{c_3\ t\ y\ (g\ t\ \lfloor (c_4\ t\ y) \rfloor)}^{\uparrow})$$

On the left hand side the constructor c has been rippled to become the contexts c_1 and c_2. These contexts denote new positions of the wave-fronts within the goal. The same process happens on the right producing contexts c_3 and c_4. It should be noted that this schematic example shows all the possible wave-front movements.

The manipulation of the annotated goal is achieved through a syntactic class of rewrite rules known as *wave-rules*. Difference identification through annotation, as described above, is used to automatically derive wave-rules from rewrite rules. For example, (5) is obtained from (4) using the following wave-rule[1]:

$$f\ \boxed{(c\ X)}^{\uparrow}\ Y \Rightarrow \boxed{c_1\ X\ Y\ (f\ X\ \boxed{(c_2\ X\ Y)}^{\downarrow})}^{\uparrow} \tag{6}$$

Note that wave-rule (6) contains an inward directed wave-front on the right-hand side while this inward directed wave-front is absent from the right-hand side of (5). This is explained by that fact that as wave-fronts are moved into sink positions differences are eliminated so the wave-front annotation is also eliminated. The final step in the ripple proof given above is achieved by a wave-rule corresponding to substitution, this has the effect of equating the contents of the sinks, i.e.

[1] We use $\Rightarrow$ to denote rewrite rules and $\rightarrow$ to denote logical implication.

$$\forall z.\forall y.((c_1\ t\ y\ z) = (c_3\ t\ y\ z)) \wedge \forall y.((c_2\ t\ y) = (c_4\ t\ y)) \wedge \boxed{\forall y.((f\ t\ y) = (g\ t\ y))}^{\uparrow}$$

Now that the goal is fully rippled we may match the unannotated part of the goal against (1), the induction hypothesis, leaving:

$$\forall z.\forall y.((c_1\ t\ y\ z) = (c_3\ t\ y\ z)) \wedge \forall y.((c_2\ t\ y) = (c_4\ t\ y))$$

What is left is a residue which may require a further induction to prove. The base case is then tackled using the same techniques as the main proof but, usually it requires only symbolic evaluation.

3.2. An Example of a Ripple Guided Verification

We now consider the verification of a particular transformation rule, i.e. a transformation that allows us to replace a function *transpose* with an application of an equivalent *foldr*. The associated verification conjecture takes the form:

$$\forall l : list(\tau).\ transpose\ l = foldr\ (\lambda x.\lambda y.(dist\ x\ y))\ []\ l \tag{7}$$

where the functions $transpose$ and $foldr$ are defined as follows:

$$foldr\ f\ b\ [] = b \tag{8}$$
$$foldr\ f\ b\ (h :: t) = f\ h\ (foldr\ f\ b\ t)$$
$$transpose\ [] = [] \tag{9}$$
$$transpose\ (row :: rows) = dist\ row\ (transpose\ rows)$$

Given the recursive nature of these definitions, a proof of (7) not surprisingly requires proof by mathematical induction. A structural induction on the list l is required, the resulting base case takes the form:

$$transpose\ [] = foldr\ (\lambda x.\lambda y.(dist\ x\ y))\ []\ [] \tag{10}$$

Rewriting (10), using definitions (9) and (8) reduces the base case to a trivial identity. In the step case we are given an inductive hypothesis of the form:

$$transpose\ t = foldr\ (\lambda x.\lambda y.(dist\ x\ y))\ []\ t \tag{11}$$

while our goal becomes:

$$transpose\ (h :: t) = foldr\ (\lambda x.\lambda y.(dist\ x\ y))\ []\ (h :: t) \tag{12}$$

Note that there exists the syntactic similarity between goal (12) and hypothesis (11). Using the rippling heuristic, we annotate (12) to give:

$$transpose\ \boxed{(h :: \boxed{t})}^{\uparrow} = foldr\ (\lambda x.\lambda y.(dist\ x\ y))\ []\ \boxed{(h :: \boxed{t})}^{\uparrow} \tag{13}$$

With regards to wave-rules, the above definitions give rise to a set of wave-rules that includes the following:

$$transpose\ \boxed{(X :: \boxed{Y})}^{\uparrow} \Rightarrow \boxed{dist\ X\ (\boxed{transpose\ Y})}^{\uparrow} \tag{14}$$

$$foldr\ F\ X\ \boxed{(Y :: \boxed{Z})}^{\uparrow} \Rightarrow \boxed{F\ Y\ (\boxed{foldr\ F\ X\ Z})}^{\uparrow} \tag{15}$$

The application of (14) to (13) gives rise to a new goal of the form:

$$\boxed{dist\ h\ (\boxed{transpose\ t})}^{\uparrow} = foldr\ (\lambda x.\lambda y.(dist\ x\ y))\ []\ \boxed{(h :: \boxed{t})}^{\uparrow} \tag{16}$$

Rippling on the right-hand side of the (16) goal is achieved using wave-rule (15), giving rise to a goal of the form:

$$\boxed{dist\ h\ (\boxed{transpose\ t})}^{\uparrow} = \boxed{dist\ h\ (\boxed{foldr\ (\lambda x.\lambda y.(dist\ x\ y))\ []\ t})}^{\uparrow} \tag{17}$$

The ripple part of the proof is completed by the application of the following wave-rule[2]:

$$F\ \boxed{Y}^{\uparrow} = F\ \boxed{Z}^{\uparrow} \Rightarrow Y = Z$$

which reduces (17) to:

$$transpose\ t = foldr\ (\lambda x.\lambda y.(dist\ x\ y))\ []\ t$$

The goal now matches (11), the induction hypothesis, so the step case proof is complete.

4. A Proof Plan for Transformation Discovery

We now use the general verification plan presented above to inform the structure of the discovery plan. Below we outline the general discovery plan followed by an example. In the discovery plan we are constructing transformation rules so we want to discover an expression equivalent to the original expression. We begin by considering the general case.

4.1. The General Pattern of Discovery

Given an expression, the aim of the discovery plan is to construct an equivalent expression that involves higher-order functions such as *map* and *foldr*. Our starting point is a discovery conjecture, which has strong similarities to the structure of the verification conjecture used above. However, the discovery conjecture differs in that the right-hand side of the conjecture contains a higher-order meta-variable. The meta-variable is a place-holder for the expression that is to be discovered.

To be more precise, if the expression we want to transform is $f\ x\ y$ then the discovery conjecture takes the form:

$$\forall x.\forall y.(f\ x\ y = M\ x\ y) \tag{18}$$

Note that M denotes a second-order meta-variable which can be instantiated to a function. In proving this schematic conjecture, the constraints of the theorem proving heuristics are used to instantiate M. The plan represents an application of the general strategy of middle-out reasoning mentioned in Section 2. As with the verification counter-part, in planning a proof of (18) we require induction, using rippling in the step case.

Assuming induction on x, then the resulting step case gives rise to an induction hypothesis of the form:

$$\forall y'.\ (f\ t\ y' = M\ t\ y') \tag{19}$$

and a goal of the form:

$$\forall y.\ (f\ (c\ t)\ y = M(c\ t)\ y) \tag{20}$$

Annotating (20) gives rise to a ripple goal of the form:

$$\forall y.\ (f\ \boxed{(c\ \boxed{t})}^{\uparrow}\ \lfloor y \rfloor = M\ \boxed{(c\ \boxed{t})}^{\uparrow}\ \lfloor y \rfloor)$$

Given definitions and properties for the function f, rippling on the left-hand side reduces the goal to give:

$$\forall y.\ \boxed{(c_1\ t\ y\ (\ \boxed{f\ t\ \lfloor (c_2\ t\ y) \rfloor}\ )}^{\uparrow} = M\ \boxed{(c\ \boxed{t})}^{\uparrow}\ \lfloor y \rfloor) \tag{21}$$

At this point the proof is blocked because we do not know what the structure of M is and whether it has any appropriate wave-rules associated with it. To overcome the blockage, we introduce a schematic wave-rule of the form:

$$M\ \boxed{(c\ \boxed{t})}^{\uparrow}\ \lfloor y \rfloor \Rightarrow \boxed{M_1\ t\ y\ (\ \boxed{M\ t\ \lfloor M_2\ t\ y \rfloor}\ )}^{\uparrow} \tag{22}$$

[2] Derived from the substitution law.

Note that schematic wave-rule (22) introduces additional meta-variables, i.e. M_1 and M_2. The construction of the schematic wave-rule is partially determined by the structure preserving property of rippling. Here the term structure being preserved is $(M\ t\ y)$. Additional constraints on the the placement of wave-fronts on the right-hand side of the schematic wave-rule are derived from the application. That is, the position of the wave-fronts on the left-hand side of the blocked goal are used to guide the positioning of wave-fronts on the right-hand side. For example, the left-hand side of (21) contains two wave-fronts, i.e. $c_1\ t\ y\ \ldots^{\uparrow}$ and $(c_2\ t\ \ldots)^{\downarrow}$. Note that the second wave-front is implicit because of the presence of the sink. The positioning of these wave-fronts is used to constrain the positioning of wave-fronts on the right-hand side of (22). That is, we introduce two wave-fronts on the right-hand side of (22), i.e. $M_1\ t\ y\ \ldots^{\uparrow}$ and $(M_2\ t\ \ldots)^{\downarrow}$. Again note that the second wave-front is implicit because of the presence of the sink. The application of (22) to (21) gives:

$$\forall y.\ (c_1\ t\ y\ (f\ t\ \lfloor (c_2\ t\ y) \rfloor)^{\uparrow} = M_1\ t\ y\ (M\ t\ \lfloor (M_2\ t\ y) \rfloor)^{\uparrow})$$

Rippling continues as in the verification counter-part, with the use of a substitution law giving:

$$\forall z. \forall y. ((c_1\ t\ y\ z) = (M_1\ t\ y\ z)) \wedge \forall y. ((c_2\ t\ y) = (M_2\ t\ y)) \wedge \forall y. ((f\ t\ y) = (M\ t\ y))^{\uparrow}$$

We can now match our goal against (19), leaving a residue of the form:

$$\forall z. \forall y.\ ((c_1\ t\ y\ z) = (M_1\ t\ y\ z)) \wedge \forall y.\ ((c_2\ t\ y) = (M_2\ t\ y))$$

Note that each conjunct within the residue takes the form of a discovery conjecture. As a consequence, the discovery plan can be recursively applied to each conjunct in order to instantiate M_1 and M_2. To complete the process of discovery requires us to commit to an instantiation for each schematic wave-rule used during the proof planning. We achieve this by comparing each schematic wave-rule against a set of wave-rules derived from the definitions of the available wave-rules. Once instantiated, the lemma underlying the schematic wave-rule is then proved, typically this involves a further application of the proof plan for induction. Note that the process of instantiating the schematic wave-rules leads to choice, where each choice point gives rise to alternative discovery proofs and thus alternative transformation rules. Examples of this choice are given in the following sections.

4.2. An Example of Transformation Discovery

We now apply our discovery plan to the matrix multiplication example with the aim of constructing a combination of higher-order functions equivalent to the `rowsmult`, `rowmult` and `transpose` functions. We start by identifying the expressions equivalent to `rowsmult`. The equational definition required for the identification task are as follows:

$$\begin{aligned}
rowsmult\ []\ cs &= [] \\
rowsmult\ (r :: rs)\ cs &= (rowmult\ r\ cs) :: (rowsmult\ rs\ cs) \\
rowsmult\ r\ []\ &= [] \\
rowsmult\ r\ (c :: cs) &= (dotprod\ r\ c) :: (rowmult\ r\ cs)
\end{aligned}$$

The equational definitions associated with our set of candidate higher-order functions are as follows:

$$\begin{aligned}
map\ f\ [] &= [] \\
map\ f\ h :: t &= (f\ h) :: (map\ f\ t) \\
foldr\ f\ b\ [] &= b \\
foldr\ f\ b\ h :: t &= (f\ h\ (foldr\ f\ b\ t)) \\
foldl\ f\ a\ [] &= a \\
foldl\ f\ a\ h :: t &= (foldl\ f\ (f\ a\ h)\ t) \\
scan\ f\ a\ [] &= [] \\
scan\ f\ a\ h :: t &= (f\ a\ h) :: (scan\ f\ (f\ a\ h)\ t)
\end{aligned}$$

Our initial discovery conjecture for `rows\-mult` takes the form:

$$rowsmult\ k\ l = M\ k\ l \tag{23}$$

Proof by induction on k gives rise to a base case of the form:

$$rowsmult\ []\ l = M\ []\ l$$

In the step case we have an induction hypothesis of the form:

$$rowsmult\ t\ l = M\ t\ l \tag{24}$$

and an (annotated) goal of the form:

$$rowsmult\ (h :: t)^{\uparrow}\ \lfloor l \rfloor = M\ (h :: t)^{\uparrow}\ \lfloor l \rfloor \tag{25}$$

Note that we tackle step cases before base cases because step cases typically provide more constraints for instantiating M. A ripple guided proof of (25) proceeds as previously described:

$$\begin{aligned} rowsmult\ (h :: t)^{\uparrow}\ \lfloor l \rfloor &= M\ (h :: t)^{\uparrow}\ \lfloor l \rfloor \\ (rowmult\ h\ \lfloor l \rfloor) :: (rowsmult\ t\ \lfloor l \rfloor)^{\uparrow} &= M\ (h :: t)^{\uparrow}\ \lfloor l \rfloor \\ (rowmult\ h\ l) :: (rowsmult\ t\ \lfloor l \rfloor)^{\uparrow} &= M_1\ h\ l\ t\ (M\ t\ \lfloor l \rfloor)^{\uparrow} \end{aligned} \tag{26}$$

Note that the schematic wave-rule used to ripple the right-hand side of the goal takes the form:

$$M\ (h :: t)^{\uparrow}\ \lfloor l \rfloor \Rightarrow M_1\ h\ l\ t\ (M\ t\ \lfloor l \rfloor)^{\uparrow} \tag{27}$$

Further rippling of (26) using substitution gives:

$$\forall x.((rowmult\ h\ l) :: x = M_1\ h\ l\ t\ x) \wedge (rowsmult\ t\ \lfloor l \rfloor) = (M\ t\ \lfloor l \rfloor)^{\uparrow}$$

A match is now possible with the induction hypothesis (24) leaving a residue of the form:

$$\forall x.\forall l.\forall h.((rowmult\ h\ l) :: x = M_1\ h\ l\ t\ x)$$

A recursive application of the discovery plan to this goal will result in the instantiation of M_1. Instantiation for M_1 may simply be the left hand side of the equality in the residue or the discovery may generate an expression involving map or $foldr$. Let us consider the expression involving map, i.e.

$$M_1 = (\lambda u.\lambda v.\lambda w.\lambda z.(map\ \lambda x.(dotprod\ u\ x)\ v) :: z) \tag{28}$$

With this instantiation for M_1, the lemma associated with wave-rule schema (27) becomes:

$$M\ (h :: t)^{\uparrow}\ \lfloor l \rfloor \Rightarrow (map\ \lambda x.(dotprod\ h\ x)\ l) :: (M\ t\ \lfloor l \rfloor)^{\uparrow}$$

In order to complete the discovery, an instantiation for M is required. As previously described, this instantiation is determined by considering the wave-rules that arise from the available higher-order function definitions. The selection criteria are as follows:

1. All wave-fronts on the left-hand side of the wave-rule schema must match with wave-fronts on the left-hand side of the candidate wave-rule.
2. The positioning of wave-fronts on the right-hand side of the wave-rule schema must correspond to the positioning of wave-fronts on the right-hand side of the candidate wave-rule.

To illustrate, consider *map*, *foldr*, *foldl* and *scan* as defined earlier. Applying the above selection criteria with respect to (28) gives the following results[3]:

$$map\ F\ (H :: T)^{\uparrow} \Rightarrow (F\ H) :: (map\ F\ T)^{\uparrow} \quad \checkmark$$

$$foldr\ F\ B\ (H :: T)^{\uparrow} \Rightarrow F\ H\ (foldr\ F\ B\ T)^{\uparrow} \quad \checkmark$$

$$foldl\ F\ \lfloor A \rfloor\ (H :: T)^{\uparrow} \Rightarrow foldl\ F\ \lfloor (F\ H\ A) \rfloor\ T \quad \not\checkmark$$

$$scan\ F\ \lfloor A \rfloor\ (H :: T)^{\uparrow} \Rightarrow (F\ H\ A) :: (scan\ F\ \lfloor (F\ H\ A\ H) \rfloor\ T)^{\uparrow} \quad \not\checkmark$$

- ✓ *map* passes the selection criteria because the left-hand side wave-fronts match and the right-hand side wave-fronts correspond.
- ✓ *foldr* passes the selection criteria because the left-hand side wave-fronts match and the right-hand side wave-fronts correspond.
- ✗ *foldl* fails the selection criteria because although the left-hand side wave-fronts match, the wave-fronts on the right-hand side do not correspond, i.e. the right-hand side of the candidate wave-rule contains a inward directed wave-front while the schematic wave-rule contains an upward directed wave-front.
- ✗ *scan* fails the selection criteria because although the left-hand side wave-fronts match, the wave-fronts on the right-hand side do not correspond, i.e. the right-hand side of the candidate wave-rule contains both inward and outward directed wave-fronts while schematic wave-rule only contains an outward directed wave-front.

We will choose *foldr* though *map* is also a possibility. The schematic wave-rule and the higher-order function wave-rule are unified and the result is:

$$foldr\ f_1\ (B\ l)\ (h :: t)^{\uparrow} \Rightarrow f_1\ h\ (foldr\ f_1\ (B\ l)\ t)^{\uparrow}$$

where $f_1 = \lambda x.\lambda y.((map\ \lambda z.(dotprod\ x\ z)\ l) :: y)$. Note that the meta-variable B is still to be instantiated at this point but there is not enough information to instantiate it in the step case proof. Instead the instantiation of B (which is the base case argument to the higher-order function) must be solved in the base case of the original induction. To illustrate we revisit the base case goal:

$$rowsmult\ []\ l\ =\ foldr\ (\lambda x.\lambda y.map\ (\lambda z.dotprod\ x\ z)\ l :: y)\ (B\ l)\ [] \tag{29}$$

Rewriting (29) using the non-recursive definitions for *rowsmult* and *foldr* equations yields:

$$[]\ =\ B\ l$$

Now the base case variable B can be instantiated to $(\lambda z.[])$ which completes the discovery proof. The final instantiation for M is as follows:

$$(\lambda v.\lambda w.foldr\ (\lambda x.\lambda y.map\ \lambda z.dotprod\ x\ z)\ w) :: y)\ [])\ v\ w$$

Applying this instantiation to (23), the discovery conjecture, gives:

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

5. Implementation

5.1. clam and λ clam

The original proof planning proposal was in [Bun88] as automatic search control for the NuPRL system [CAB+86]. NuPRL was designed for program synthesis, based on Martin-Löf Intuitionistic Type theory to provide a constructive logic framework. In NuPRL, proof plans are represented by LCF [GMW79] style tactics.

Subsequently, a light weight version of NuPRL, called *oyster*[Hor88], was implemented in. In *oyster*, the search for a proof must be guided externally by interaction with a human or another Prolog program. Thus, the

[3] We use a ✓ to denote success and a ✗ to denote failure.

meta-level proof planner *clam* [BvHHS90] is built on top of *oyster* and provides tactics which guide the proof search. Such tactics are specified by methods through heuristic preconditions what guide the search for a proof. Once created, a proof plan is translated into an object-level proof, using the corresponding tactics to guide the *oyster* proof system.

Initially, *clam* methods were implemented to use rippling out to guide inductive proof [BvHS88]. These methods were later extended to include multi-wave rippling, rippling in, rippling sideways, destructor style rippling, and existential rippling [BSvH$^+$93]. Next, synthesis and middle-out reasoning capabilities were introduced [KBB93]. Most recently, a critics mechanism has been provided [IB96] to make productive use of the occasional failure of rippling.

Many of the tasks *clam* is used for are naturally higher-order. Synthesis and verification techniques, such as critics, rely on the use of middle-out reasoning and higher-order unification. In *clam* these techniques were later extensions and are not naturally integrated into the theorem proving framework.

clam's successor, $\lambda clam$ is a higher-order version of *clam* written in λProlog [MN88] the higher-order extension of Prolog. In particular, we used the TERZO λProlog [Wic] version of $\lambda clam$. 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 enables higher-order unification. λProlog is based on hereditary Harrop formulas instead of the Horn clauses of Prolog. The higher-order unification built in to the $\lambda clam$ system enabled the close integration of middle-out reasoning and proof planning [RSG98].

5.2. λ and Higher-order Function Discovery

In discovering HOFs within $\lambda clam$, a number of changes were needed. $\lambda clam$ uses *methodicals* to specify which methods shall be applied in a proof attempt and the order in which they should be applied. Methodicals are method combinators and are analogous to LCF style tacticals [GMW79]. Methodicals provide a flexible basis for modifying proof planning behaviour. In particular, methods built from methodicals can be input into $\lambda clam$ along with a conjecture so no re-compiling of the planner is necessary.

However, the necessary lower-level manipulation to perform HOF discovery could not be achieved by solely changing methodicals. This required changes to the source code for the methods, in particular to strengthen constraints on meta-variable instantiation to avoid immediate selection of a component of the original conjecture.

A number of other modifications were required to support HOF discovery in $\lambda clam$:

- The methodical structure controlling sequencing was strengthened to ensure that the partial instantiation of a meta-variable in one part of a proof constrains subsequent instantiations.
- New methods were introduced for manipulating the meta-annotations that prevents a meta-variable from being instantiated.
- A special purpose method was implemented to split apart the goal during rippling. This was required to generate new sub-goals corresponding to potential nested HOFs within a goal.
- A new method was provided to choose the target HOF for potential discovery.

6. Results

The result for the matrix multiplication example presented above was:

$$rowsmult \;=\; (\lambda v.\lambda w.foldr\ (\lambda x.\lambda y.map\ (\lambda z.(dotprod\ x\ z)\ w) :: y)\ [])\ v\ w$$

However, running the matrix multiplication example through the discovery proof plan and forcing it to backtrack gives several other solutions. These solutions together with timings are presented in Table 1.

Note that there are significant time differences between finding groups of related solutions. These arise because discovery is guided to initial loci of HOFs. However, once a locus is exhausted, backtracking search over considerable term structure may be required to reach a new locus.

Note that the order in which the proof planning occurs leads to the original conjecture being discovered after all other possibilities have been exhausted. Hence, the apparently substantial time taken to discover $(\lambda x.\lambda y.rowsmult\ x\ y)$ is actually a good indication of complete search space traversal to failure.

Table 1. Solutions discovered for the matrix multiplication example

No.	Solution	Time (s)
1	$(\lambda x.\lambda y.map\ (\lambda z.map\ (\lambda u.dotprod\ z\ u)\ y)\ x)$	73
2	$(\lambda x.\lambda y.foldr\ (\lambda z.\lambda u.map\ (\lambda v.(dotprod\ z\ v)\ y) :: u)\ []\ x)$	83
3	$(\lambda x.\lambda y.map\ (\lambda z.foldr\ (\lambda u.\lambda v.(dotprod\ z\ u) :: v)\ []\ y)\ x)$	122
4	$(\lambda x.\lambda y.foldr\ (\lambda z.\lambda u.$ $(foldr\ (\lambda v.\lambda w.(dotprod\ z\ v) :: w)\ []\ y) :: u)\ []\ x)$	132
5	$(\lambda x.\lambda y.map\ (\lambda z.rowmult\ z\ y)\ x)$	401
6	$(\lambda x.\lambda y.foldr\ (\lambda z.\lambda u.(rowmult\ z\ y) :: u)\ []\ x)$	411
7	$(\lambda x.\lambda y.rowsmult\ x\ y)$	2771

Table 2. HOF discovery results

Function	HOFs	Solutions	Total Time (s)
rev	1 *foldl*	2	200
presums	1 *scan*	2	1527
sum2d	2 *foldr*s	3	238
subset	2 *foldr*s	3	2221
mergesort	2 *foldr*s	4	417
squs2d	2 *map*s	7	1201
mmult	2 *map*s + 1 *foldr*	14	10899

The discovery plan has been used to find solutions for several other examples involving the HOFs *map*, *foldr*, *foldl*, and *scan*. These results are presented in Table 2. For each example we give the prototype function, the potential HOFs, the number of solutions discovered, and the overall time to find all results. The definitions related to these examples are given in the Appendix.

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

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

This incurs additional costs. For *sum2d* and *subset*, the plan 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. Note that *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, the discovery of HOFs in *mergesort* is still considerably faster than for *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. Note that *mmult* is the full matrix multiplication example presented in Section 1.3, whereas *rowsmult*, discussed above, does not include the *transpose* stage.

7. PMLS Integration

Our work has been conducted in the context of the PMLS parallelising compiler for Standard ML (SML). PMLS has the objective of exploiting parallelism latent in potential HOF use in programs. We have evaluated the 12 derived versions of the matrix multiplication example by:

- translating the $\lambda Clam$ to SML;
- compiling the translated SML to native code via PMLS;
- running the native code on an IBM SP2 multi-processor architecture.

The parallel runtimes on 1, 2, 4 and 8 processors are shown in Table 3, for a 50×50 matrix. The annotations *M* and *F* indicate which `map`s and `fold`s are evaluated using parallel skeletons. The times are typical of the behaviour

Table 3. Parallel runtimes for nested implementations.

Version	HOFs	R_1	R_2	R_4	R_8
1	MMF	0.483	0.595	15.69	7.954
2	FMF	0.472	0.886	11.51	5.759
3	MFF	0.481	0.600	8.655	4.602
4	FFF	0.473	0.882	8.844	5.108
5	MF	0.455	0.578	1.724	0.535
6	FF	0.449	0.875	2.336	0.651
7	MM	0.167	0.267	1.684	0.738
8	FM	0.185	0.840	4.449	2.035
9	MF	0.164	0.270	1.389	0.699
10	FF	0.167	0.598	1.952	0.799
11	M	0.147	0.310	0.127	0.090
12	F	0.137	0.590	0.167	0.112

of our skeletons when instantiated with functions with little or no work relative to communication. The outer `map` skeleton shows lower overheads than a `fold` in an equivalent position as seen by comparing Versions 7 with 8 or 11 with 12. Here nesting is of little use since the overheads are high with respect to the sequential runtimes.

For this application the single non-nested `map` gives the best actual performance. This has been borne out by other work on matrix multiplication [SMH01]. Furthermore, these runtimes broadly correspond to the predictions from the PMLS profiler. This work is discussed in detail in Michaelson and Scaife [MS03].

Although automatic translators between Core SML and $\lambda Prolog$ have been constructed, at present, HOF discovery is independent of the compiler. It will be interesting to link HOF discovery more closely with parallelising compilation.

8. Related Work

A number of systems realise parallelism in HOFs through skeletons. Examples include compilers for the imperative skeleton language P3L [Pel98] and the divide and conquer Haskell subset HDC [HL00], and the Skipper framework for OCaml augmented with skeletons [SCD01]. In such systems, however, the user must explicitly invoke a HOF or equivalent construct for it to be implemented as a skeleton.

There has been comparatively little research into the use of HOFs for automatic parallelisation. The FAN skeleton framework for parallel programming by transformation [AGLP01] was contemporaneous with the work described here. FAN is based on a coordination notation for second order functions over arrays and scalers. Each function has an associated cost equation. Starting with a naive program, transformation rules are applied to try and minimise the overall program cost. Subsequently, FAN was implemented in the META tool [Ald02].

FAN/META differs from our approach in two important respects. First of all, the system is interactive and offers transformation options to the user, where our approach is fully automatic. Secondly, for FAN/META a program must contain explicit HOFs as the initial loci of transformation. In contrast, our system will automatically discover initial HOFs.

9. Discussion

We have shown that, in principle, proof planning is an appropriate technique for discovering HOFs in programs that lack them and for exploring the space of possible HOFs. However, discovery takes unacceptably long times for even relatively small programs.

Our system represents a first attempt at HOF discovery and as such is somewhat inefficient. As mentioned in Section 5.1, the version of $\lambda clam$ that was used within our system was written in TERZO [Wic], one of the early implementations of λProlog. TERZO is an interpreter and is relatively inefficient in terms of both time and space. For example, TERZO does not use tail recursive optimisation. Because of these inefficiencies, the developers of $\lambda clam$ switched to PROLOG/MALI [BR] and then to the TEYJUS [Nad] implementation of λProlog. PROLOG/MALI and TEYJUS provide compilers for λProlog and both are an order of magnitude faster than the TERZO interpreter. The timescales of our project, however, did not allow us to exploit the benefits of either the PROLOG/MALI or TEYJUS versions of $\lambda clam$. More recently, $\lambda clam$ has been superseded by ISAPLANNER [DF03], a proof planner for the

ISABELLE [Pau94] proof development system. The support for higher-order meta variables provided by ISABELLE makes ISAPLANNER the ideal framework for extending our work. Switching to ISAPLANNER would also address the fundamental issue of efficiency associated with $\lambda clam$, e.g. in [DF03], a theorem drawn from arithmetic took $\lambda clam$ 5 min to prove while the same theorem in ISAPLANNER required only 0.9 s.

As well as inefficiencies in the underlying implementation of λProlog, inefficiencies can also be attributed in part the simple structure of the $\lambda clam$ proof planner. The planner used was based on full backtracking where the same discovery sequences may be needlessly repeated. This might be alleviated by the introduction of intermediate caching of results.

Furthermore, as noted above the plan is exhaustive in seeking all possible combinations of HOFs. The results from matrix multiplication above show that all potentially useful HOFs were discovered in the first 411 s (15%), with a further 2360 s (85%) spent in fruitlessly searching for other possibilities. Similarly, the results from [MS03] for a simple program to square all elements of a list of list of integers show that six useful HOF combinations were discovered in the first 312 s (26%) with a further 889 s (74%) spent without any further discovery. This suggests that it would be worth exploring heuristics to either predict the likely number of potential HOF combinations, based on the number of HOFs found initially, or to terminate the search after a given period without further discovery.

In the longer term, we think that close cooperation between the discovery plan and compilation might further improve performance, in particular using profiling to guide discovery, analogous to the use of costs to guide transformation in FAN/META, and in a proposed extension to HDC [HL01].

Rather than running the plan before compilation to find all possible HOFs, the compiler might invoke the plan if no HOFs are present in an original program, to find a first set of HOFs. Subsequently, the compiler might re-invoke the discovery plan, to transform the program, if profiling and analysis suggest that present or discovered HOFs cannot deliver exploitable parallelism. In turn, the plan would use profiling and analysis information from the compiler to constrain the choice of alternatives.

Finally, it would be valuable to analyse the times taken for HOF discovery for a large set of representative source programs of varying structural complexities. Here, characterisation of patterns of discovered HOF nesting and composition might be combined with analyses of the observed behaviours of individual HOFs, to try and establish a predictive model for optimal HOF discovery for parallel implementation.

Acknowledgments

This work was supported by UK EPSRC grant GR/L42889, "Parallelising compilation of Standard ML through prototype instrumentation and transformation". We wish to thank Julian Richardson for development of and support for $\lambda Clam$, and our colleagues in the Heriot-Watt University Dependable Systems Group and the University of Edinburgh Mathematical Reasoning Group for much fruitful discussion. We also wish to thank JAIST, Japan for access to their IBM SP2.

Finally we wish to thank the members of our project review panel for their support and advice: Alan Bundy, Dave Berry, Murray Cole and Kevin Mitchell.

References

[AGLP01] Aldinucci M, Gorlatch S, Lengauer C, Pelegatti S (2001) Towards parallel programming by transformation: the FAN skeleton framework. Parallel Algorithms Appl. 16(2–3):87–122

[Ald02] Aldinucci M (2002) Automatic program transformation: the META tool for skeleton-based languages. In: Gorlatch S., Lengauer C, (eds) Constructive methods for parallel programming, volume 10 of Advances in Computation: theory and practice. NOVA Science

[ASG97] Alessandro Armando, Alan Smaill, Ian Green (1997) Automatic synthesis of recursive programs: the proof-planning paradigm. In: 12th IEEE international automated software engineering conference, Lake Tahoe, Nevada, pp. 2–9

[Bac78] Backus, J, (1978) Can Programming be Liberated from the Von Neumann Style? Commun. ACM 21(8):287–307

[Bd97] Bird R, Oege de Moor (1997) Algebra of Programming. Prentice-Hall, Englewood Chitts

[BR] Brisset P, Ridoux O. Prolog/Mali. `ftp.irisa.fr:/local/pm/`

[BS66] Barron DW, Strachey C (1996) Chapter 3: programming. In: Fox L, (ed) Advances in Programming and Non-numerical Computation. Pergammon, New York

[BSH90] Bundy A, Smaill A, Hesketh J (1990) Turning eureka steps into calculations in automatic program synthesis. In: Clarke. SLH., (ed). Proceedings of UK IT 90, pp. 221–6. IEE Also available from Edinburgh as DAI Research Paper 448

[BSvH+93] Bundy A, Stevens A, van Harmelen F, Ireland A, Smaill A (1993) Rippling: a heuristic for guiding inductive proofs. Artificial Intelligence, 62:185–253. Also available from Edinburgh as DAI Research Paper No. 567

[Bun88] Bundy A (1998) The use of explicit plans to guide inductive proofs. In: Lusk R, Overbeek R, (eds) 9th Conference on Automated Deduction, Springer Berlin Heidelberg New York, pp. 111-120. Longer version available from Edinburgh as DAI Research Paper No. 349

[BvHHS90] Bundy A, van Harmelen F, Horn C, Smaill A (1990) The Oyster-Clam system. Research Paper 507, Department of Artificial Intelligence, University of Edinburgh Appeared in the proceedings of CADE-10

[BvHS88] Bundy A, van Harmelen F, Hesketh J, Smaill A (1991) Experiments with proof plans for induction. Research Paper 413, Department of Artificial Intelligence, University of Edinburgh. Appeared in Journal of Automated Reasoning

[BW96] David Basin, Toby Walsh (1996) A calculus for and termination of rippling. J. Automated Reasoning 16(1–2):147–180

[CAB+86] Constable RL, Allen SF, Bromley HM, et al. (1986) Implementing Mathematics with the Nuprl Proof Development System. Prentice Hall, Englewood Clitts

[Chu41] Church A (1941) The calculi of lambda conversion. Princeton University Press, New Jersey

[Col89] Cole MI, (1989) Algorithmic skeletons: structured management of parallel computation. Pitman, London

[DF03] Dixon L, Fleuriot JD (2003) IsaPlanner: a prototype proof planner in Isabelle. In: Proceedings of CADE'03, LNCS, pp. 279–283

[GMW79] Gordon MJ, Milner AJ, Wadsworth CP (1979) Edinburgh LCF - A mechanised logic of computation, volume 78 of Lecture Notes in Computer Science. Springer-Verlag, Berlin Heidelberg New York

[Gor73] Gordon MJC, (1973) An Investigation of *Lit*. In: Technical report memorandum: MIP-R-101, School of Artificial Intelligence, University of Edinburgh

[Ham00] Hamdan H (2000) A combinational framework for parallel programming using algorithmic skeletons. PhD Thesis, Department of Computing and Electrical Engineering, Heriot-Watt University

[HL00] Herrmann CA, Lengauer C (2000) HDC: a higher-order langauge for divide conquer. Parallel Processing Letters, 10(2–3):239–250

[HL01] Herrmann CA, Lengauer C (2001) A transformational approach which combines size inference and program optimisation. In: Taha W, (ed) Semantics, applications, and implementation of program generation, number 2196 in LNCS, Springer, Berlin Heidelberg New York, pp 199–218

[Hor88] Horn C (1988) The Nurprl proof development system. Working paper 214, Department of Artificial Intelligence, University of Edinburgh The Edinburgh version of Nurprl has been renamed Oyster

[HRP00] Hammond K, Rebon-Portillo A (2000) HaskSkell: algorithmic skeletons in haskell. In: Proceedings of 11th international workshop on implementation of functional languages (IFL'99), number 1868 in LNCS. Springer S., Berlin Heidelberg New York

[Hug84] Hughes RJM (1984) The Design and implementation of programming languages. PhD Thesis, University of Oxford

[IB96] Ireland A, Bundy A (1996) Productive use of failure in inductive proof. J. Automated Reasoning, 16(1–2):79–111 Also available as DAI Research Paper No 716, Department of Artificial Intelligence, Edinburgh

[Ire92] Ireland A, (1992) the use of planning critics in mechanizing inductive proofs. In: Voronkov A (ed) International conference on logic programming and automated reasoning – LPAR 92, St. Petersburg, Lecture Notes in Artificial Intelligence No. 624, pp 178–189. Springer-Verlag, Berlin Heidelberg New York Also available from Edinburgh as DAI Research Paper 592

[KBB93] Kraan I, Basin D, Bundy A, (1993) Logic program synthesis via proof planning. In: Lau KK, Clement T, (eds) Logic program synthesis and transformation, pp 1–14. Springer, Berlin Heidelberg New York Also available as Max-Planck-Institut für Informatik Report MPI-I-92-244 and Edinburgh DAI Research Report 603

[KBB96] Kraan I, Basin D, Bundy A (1996) Middle-out reasoning for synthesis and induction. J. Automated Reasoning, 16(1–2):113–145 Also available from Edinburgh as DAI Research Paper 729

[Kel89] Kelly P (1989) Functional programming for loosely coupled microprocessors. Pitman, London

[Kle52] Kleene SC (1952) Introduction to metamathematics. North-Holand

[McC60] McCarthy J (1960) Recursive functions of symbolic expressions and their computation by machine: Part 1. Communications of the ACM, 3:184–195

[MIK97] Michaelson G, Ireland A, King P (1997) Towards a skeleton based parallelising compiler for SML. In: Clack C, Davie T, Hammond K (eds) Proceedings of 9th international workshop on implementation of functional languages, School of Mathematical and Computer Sciences, University of St Andrews, pp. 539–546

[MN88] Miller D, Nadathur G (1998) An overview of λProlog. In: Bowen R, Kowalski K (ed). Proceedings of the fifth international logic programming conference/ fifth symposium on logic programming. MIT Press, Cambridge

[MS03] Michaelson G, Scaife N (2000) Skeleton realisations from functional prototypes. In: Rabhi F, Gorlatch S, (eds) Patterns and skeletons for parallel and distributed computing, chapter 5. Springer, Berlin Heidelberg New York

[MSBK01] Michaelson G, Scaife N, Bristow P, King P (2001) Nested Algorithmic skeletons from higher order functions. Parallel algorithms and applications: special issue on high level models and languages for parallel processing

[Nad] Nadathur G, Teyjus lambda Prolog. `http://teyjus.cs.umn.edu`

[Pau94] Paulson LC (1994) Isabelle: a generic theorem prover. Springer, Berlin Heidelberg New York

[Pau96] Paulson LC (1996) ML for the working programmer. CUP, 2nd edn

[Pel98] Pelagatti S (1998) Structured development of parallel programs. Taylor and Francis

[PR01] Pena R, Rubio F, (2001) Parellel functional programming at two levels of abstraction. In: Proceedings of 3rd international conference on principles and practice of declarative programming (PPDP'01). ACM Press, New York

[RSG98] Richardson JDC, Smaill A, Ian Green (1998) System description: proof planning in higher-order logic with lambdaclam. In: Claude Kirchner Hélène Kirchner (eds) 15th international conference on automated deduction, vol 1421 of Lecture Notes in Artificial Intelligence, Lindau, Germany

[SCD01] Sérot J, Coudarcher R, Dérutin JP (2001) Implementation of a skeleton-based parallel programming environment supporting arbitrary nesting. In: Mueller F (ed). 6th International Workshop on High-level parallel programming models and supportive environments, volume 2026 of *LNCS*, Springer, Berlin Heidelberg New York

[SI99] Stark J, Ireland A (1999) Towards automatic imperative program synthesis through proof planning. In: The 14^{th} IEEE international conference on automated software engineering, IEEE Computer Society, pp 49–51

[Ski94] Skillicorne D (1994) Foundations of parallel programming. CUP

[SMH01] Scaife N, Michaelson G, Horiguchi S (2001) Comparative Cross-platform performance results from a parallelising SML compiler. In: Arts T, Mohnen M (eds) Proceedings of 13th international workshop on the implementation of functional languages, number 2312 in LNCS. Springer, Berlin Heidelberg New York

[Sto84] Stoye W (1984) Director strings on SKIP. In: Augustsson L, Hughes J, Johnsson T, Karlson K (eds) Proceedings of the workshop on implementation of functional languages, Report 17, 347–356. Programming methodology group, University of Göteborg and Chalmers University of Technology

[Tur79] Turner DA (1984) A new implementation technique for applicative languages. Software pract. experience 9:31–49

[Weg71] Wegner P (1979) programming languages, information structures and machine organisation. McGraw-Hill, New York

[Wic] Wickline P Terzo lambda Prolog. `ftp.cis.upenn.edu/pub/Terzo`

[WMM+92] Wallace A, Michaelson G, McAndrew P, Waugh K, Austin W (1992) Dynamic control and prototyping of parallel vision algorithms. IEEE Computer: special issue on Parallel Processing for Computer Vision and Image Understanding 25(2):43–53

Appendix A. Examples and Results

The definitions for the example functions and their equivalent solution are presented below.

A.1. *reverse*

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

$$rev = (\lambda x.\lambda y.lfoldl\ (\lambda z.\lambda u.u :: z)\ x\ y)$$

A.2. *presums*

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

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

A.3. *sum2d*

$$\begin{aligned} sum2d\ [] &= 0 \\ sum2d\ (h :: t) &= sumb\ (sum2d\ t)\ h \\ sumb\ b\ [] &= b \\ sumb\ b\ (h :: t) &= h + (sumb\ b\ t) \end{aligned}$$

$$sum2d = (\lambda x.foldr\ (\lambda y.\lambda z.foldr\ (\lambda u.\lambda v.u + v)\ z\ y)\ 0\ x)$$

A.4. *subset*

$$\begin{aligned} subset\ []\ x &\leftrightarrow true \\ subset\ (h :: t)\ x &\leftrightarrow (mem\ h\ x) \wedge (subset\ t\ x) \\ mem\ x\ [] &\leftrightarrow false \\ mem\ x\ (h :: t) &\leftrightarrow (x = h) \vee (mem\ x\ t) \end{aligned}$$

$$subset = (\lambda x.\lambda y.foldr\ (\lambda z.\lambda u.(foldr\ (\lambda v.\lambda w.(z = v) \vee w)\ false\ y) \wedge u)\ true\ x)$$

A.5. *mergesort*

$$
\begin{aligned}
mergesort\ x &= mergeall\ (sortmall\ x)\\
mergeall\ [] &= []\\
mergeall\ (h :: t) &= merge\ h\ (mergeall\ t)\\
sortmall\ [] &= []\\
sortmall\ (h :: t) &= (sortm\ h) :: (sortmall\ t)\\
sortm\ [] &= []\\
sortm\ (h :: t) &= insertm\ h\ (sortm\ t)\\
mergesort &= (\lambda x.foldr\ (\lambda y.\lambda z.merge\ y\ z)\ []\ (map\ (\lambda y.foldr\ (\lambda z.\lambda u.insertm\ z\ u)\ []\ y)\ x))
\end{aligned}
$$

A.6. *squs2d*

$$
\begin{aligned}
squs2d\ [] &= []\\
squs2d\ (h :: t) &= (squares\ h) :: (squs2d\ t)\\
squares\ [] &= []\\
squares\ (h :: t) &= (h \times h) :: (squares\ t)\\
squs2d &= (\lambda x.map\ (\lambda y.map\ \lambda z.(z * z)\ y)\ x)
\end{aligned}
$$

A.7. *mmult*

$$
\begin{aligned}
mmult\ m_1\ m_2 &= (rowsmult\ m_1)\ (transpose\ m_2)\\
rowsmult\ []\ cols &= []\\
rowsmult\ (row :: rows)\ cols &= (rowmult\ row\ cols) :: (rowsmult\ rows\ cols)\\
rowmult\ row\ [] &= []\\
rowmult\ row\ (col :: cols) &= (dotprod\ row\ col) :: (rowmult\ row\ cols)\\
dotprod\ []\ [] &= 0\\
dotprod\ (h_1 :: t_1)\ (h_2 :: t_2) &= (h_1 \times h_2) + ((dotprod\ t_1)\ t_2)\\
transpose\ [] &= []\\
transpose\ (row :: rows) &= dist\ row\ (transpose\ rows)\\
mmult &= (\lambda x.\lambda y.map\ (\lambda z.map\ (\lambda u.dotprod\ z\ u)\ (foldr\ (\lambda u.\lambda v.dist\ u\ v)\ []\ y))\ x)
\end{aligned}
$$

Received December 2003
Revised september 2004
Accepted November 2004 by A. E. Abdallah, P. Y. A. Ryan, S. A. Schneider and D. J. Cooke
Published online ■■■