Under consideration for publication in Formal Aspects of Computing

Higher Order Function Synthesis Through Proof Planning

Andrew Cook, Andrew Ireland and Greg Michaelson
Department of Computing and Electrical Engineering
Heriot-Watt University Riccarton, Scotland, EH14 4AS
{ceeajc,air,greg}@cee.hw.ac.uk
0131 451 {4179,3409,3422}

Abstract. The close association between higher order functions (HOFs) and algorithmic skeletons is a promising source of automatic parallelisation of programs. An approach to synthesising HOFs from functional programs through proof planning is presented, and its realisation in $\lambda Clam$ is discussed.

1. Introduction

1.1. 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 multiprocesssing* applied in arbitrary order to local subexpressions. *page 185*

Early work on functional parallelism focussed on reduction of Curry combinators [Tur79], lifted automatically from functional programs, but these proved of too low granularity for efficient parallel evaluation [Sto84]. Somewhat more success has obtained from parallel reduction of super-combinators[Hug84], 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:

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

Correspondence and offprint requests to: Andrew Cook, Andrew Ireland and Greg Michaelson
Department of Computing and Electrical Engineering
Heriot-Watt University Riccarton, Scotland, EH14 4AS
{ceeajc,air,greg}@cee.hw.ac.uk
0131 451 {4179,3409,3422}

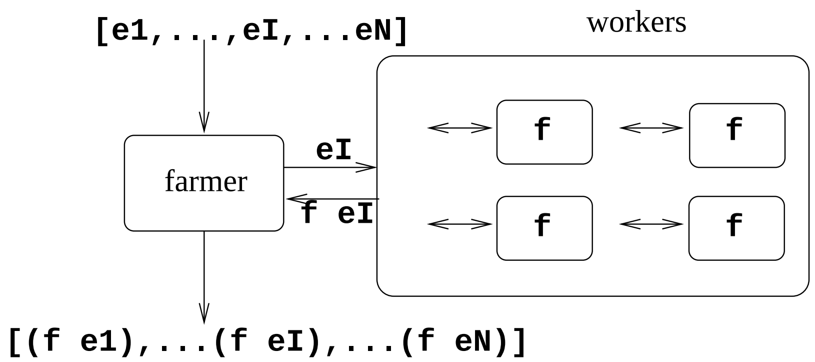

Fig. 1. Process farm

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

HOFs originate in pre-computer mathematical logic, notably for formalisations of notions of computability. Thus, Church's λ calculus[Chu41], the basis 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:

$(f\ \texttt{o}\ g)\ x \Leftrightarrow f\ (g\ x)$

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

$(\texttt{map}\ f)\ \texttt{o}\ (\texttt{map}\ g) \Leftrightarrow \texttt{map}\ (f\ \texttt{o}\ g)$

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

$\texttt{map}\ f \Leftrightarrow \texttt{foldr}\ (\texttt{fn}\ h\ \texttt{=>}\ \texttt{fn}\ t\ \texttt{=>}\ f\ h::t)$

1.2. HOFs and skeletons

We have been investigating an approach to automatic parallelisation of functional programs[MIK97] which draws heavily on Cole's 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 - Figure 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: $f\ x\ y \Leftrightarrow f\ y\ x$
Identity: $f\ x\ b \Rightarrow x$

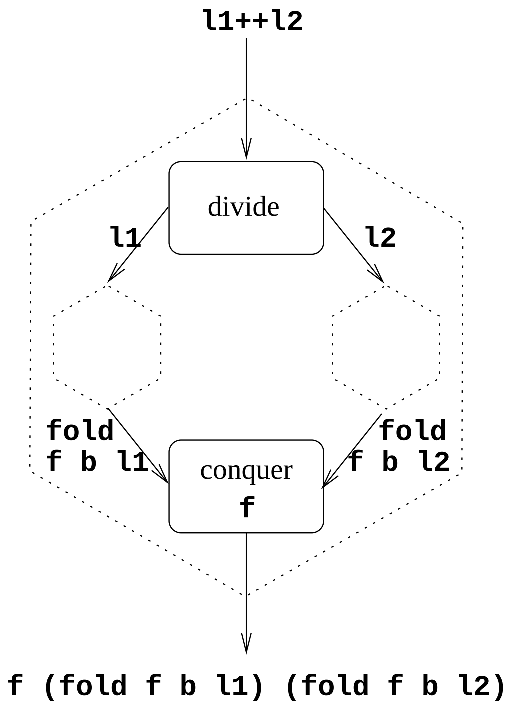

Fig. 2. Divide and conquer

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 conquer phase, for further combination with f - Figure 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 - Figure 3.

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 as a farm of pipelines - Figure 4. 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:

$$\begin{bmatrix} x_{11} & ... & x_{1n} \\ ... & ... & ... \\ x_{m1} & ... & x_{mn} \end{bmatrix} * \begin{bmatrix} y_{11} & ... & y_{1n} \\ ... & ... & ... \\ y_{m1} & ... & y_{mn} \end{bmatrix} = \begin{bmatrix} z_{11} & ... & z_{1n} \\ ... & ... & ... \\ z_{m1} & ... & z_{mn} \end{bmatrix}$$

where:

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

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}]]

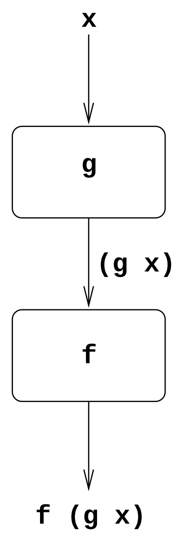

Fig. 3. Pipeline

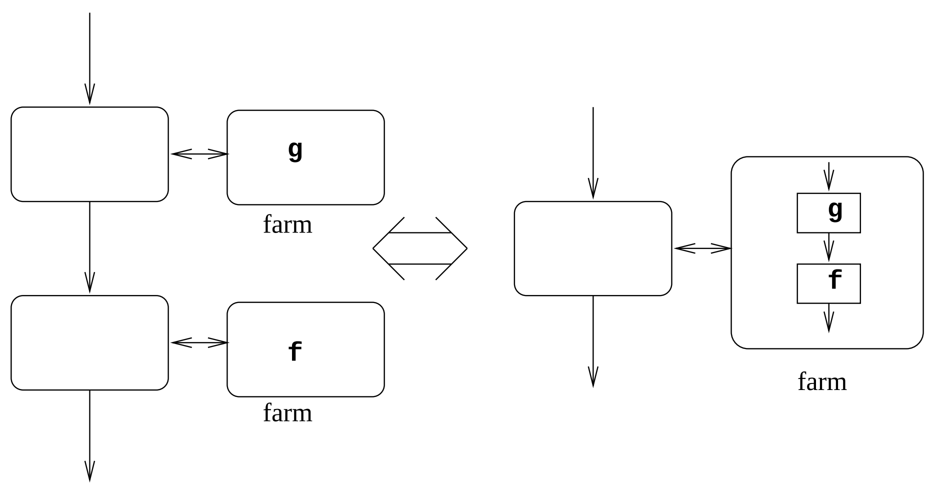

Fig. 4. `(map f) o (map g)` ⇔ `map (f o g)`

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

$$
\begin{array}{lcl}
\texttt{[[}y_{11}\texttt{,}y_{12}\texttt{,}\ldots\texttt{,}y_{1n}\texttt{],} & & \texttt{[[}y_{11}\texttt{,}y_{21}\texttt{,}\ldots\texttt{,}y_{m1}\texttt{],} \\
\ \texttt{[}y_{21}\texttt{,}y_{22}\texttt{,}\ldots\texttt{,}y_{2n}\texttt{],} & & \ \texttt{[}y_{12}\texttt{,}y_{22}\texttt{,}\ldots\texttt{,}y_{m2}\texttt{],} \\
\ \ldots & \to & \ \ldots \\
\ \texttt{[}y_{m1}\texttt{,}y_{m2}\texttt{,}\ldots\texttt{,}y_{mn}\texttt{]]} & & \ \texttt{[}y_{1n}\texttt{,}y_{2n}\texttt{,}\ldots\texttt{,}y_{mn}\texttt{]]}
\end{array}
$$

which may be achieved by:

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

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`:

```
fun transpose l = foldr dist [] l;
```

and `rowmult` and rowsmult may be reformulated as `map`s:

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

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

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

```
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 [MSBK00]. 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 synthesise 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*

Furthermore, 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 synthesis of HOFs, from programs that lack them, as eminently worthwhile for the automatic exploitation of parallelism.

2. Proof planning

We follow the tactic-based approach to 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 analyze 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* [Bun88a], 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 customized tactic for a given conjecture. The search for a customized 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 sythesis of higher order terms within the context of transformation rules.

3. Rippling: A proof plan for rewriting

Our application involves reasoning about recursively defined functions, as a consequence 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 use an example to introduce the main ideas of rippling. Consider the following conjecture:

$$\forall l : list(\tau).\ transpose\ l = foldr\ \lambda x.\lambda y.(dist\ x\ y)\ []\ l \tag{1}$$

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

$$\begin{aligned} foldr\ f\ b\ [] &= b \\ foldr\ f\ b\ (h :: t) &= f\ h\ (foldr\ f\ b\ t) \end{aligned} \tag{2}$$

$$\begin{aligned} transpose\ [] &= [] \\ transpose\ (row :: rows) &= dist\ row\ (transpose\ rows) \\ dist\ []\ l &= [] \\ dist\ (h_1 :: t_1)\ (h_2 :: t_2) &= (h_1 :: h_2) :: (dist\ t_1\ t_2) \\ dist\ (h_1 :: t_1)\ [] &= [h_1] :: (dist\ t_1\ []) \end{aligned} \tag{3}$$

Given the recursive nature of these definitions, a proof of (1) not surprisingly requires proof by mathematical induction. A structural induction on the list l is required, the resulting base case takes the form:

$$transpose\ [] = foldr\ \lambda x.\lambda y.(dist\ x\ y)\ []\ [] \tag{4}$$

Rewriting (4), using definitions (3) and (2) reduces the base case to a trivial identity. In the step case we are given an inductive hypothesis of the form:

$$transpose\ t = foldr\ \lambda x.\lambda y.(dist\ x\ y)\ []\ t \tag{5}$$

while our goal becomes:

$$transpose\ (h :: t) = foldr\ \lambda x.\lambda y.(dist\ x\ y)\ []\ (h :: t) \tag{6}$$

Note that there exists a syntactic similarity between goal (6) and hypothesis (5). 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 example the annotated version of goal (6) takes the form:

$$transpose\ \boxed{(h :: \boxed{t})}^{\uparrow} = foldr\ \lambda x.\lambda y.(dist\ x\ y)\ []\ \boxed{(h :: \boxed{t})}^{\uparrow} \tag{7}$$

Note that the annotated part of the term, represented by the shading, denotes a mismatch (wave-front) between the goal and the given induction hypothesis. 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.

An identical difference identification process is used to compute the applicable rewrite rules at each step within the planning of a proof. To illustrate, the definitions given above provide a set of annotated rewrite rules, known as *wave rules*, that include[1]:

$$transpose\ \boxed{row :: \boxed{rows}}^{\uparrow} \Rightarrow \boxed{dist\ row\ \boxed{transpose\ rows}}^{\uparrow} \tag{8}$$

$$foldr\ f\ b\ \boxed{h :: \boxed{t}}^{\uparrow} \Rightarrow \boxed{f\ h\ \boxed{foldr\ f\ b\ t}}^{\uparrow} \tag{9}$$

The process of deriving wave rules is completely automatic and, as will be illustrated below, is not restricted to controlling the unfolding of definitions. The application of (8) to (7) gives rise to a new goal of the form:

$$\boxed{dist\ h\ \boxed{(transpose\ t)}}^{\uparrow} = foldr\ \lambda x.\lambda y.(dist\ x\ y)\ []\ \boxed{(h :: \boxed{t})}^{\uparrow} \tag{10}$$

Note that wave rule application requires that both object- and meta-level terms match. Rippling on the right-hand side of the (10) goal is achieved using wave rule (9), giving rise to a goal of the form:

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

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

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

which reduces (11) to:

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

The goal now matches (5), the induction hypothesis, so the step case proof is complete. For a definitive account of rippling the interested reader is directed to [BW96, BSvH+93].

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

[2] Derived from the substitution law.

4. A plan for synthesising HOFs

We will use a general verification plan to inform the structure of the synthesis plan.

4.1. The general verification plan

In the verification we start out with a conjecture which takes the following general form:

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

Assuming an induction on the variable x. The step case proceeds via rippling.

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

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

On the left hand side the c on has been rippled out to become c_1 and rippled into the sink variable y to become c_2. The same process happens on the right producing c_3 and c_4. If rippling in does not take place then c_2 and c_4, the term strucure in the sinks will be empty and if rippling out does not take place then c_1 and c_3, the wave terms, will be empty.

We may split the goal apart at this point using substitution axioms to isolate the wave, hole and sink parts.

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

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

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

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.

4.2. The general synthesis plan

In the synthesis plan we are synthesising transformation rules so we want to synthesise an expression equivalent to the the original expression. If the expression we want to transform is $f\ x\ y$ then we set up the following conjecture with a higher-order meta-variable on the right hand side which will be instantiated during the course of the proof.

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

As in the verification proof, we use induction and the step case uses rippling.

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

$$\begin{aligned}
\forall y.(f\ \boxed{(c\ t)}^{\uparrow}\ \lfloor y \rfloor &= M\ \boxed{(c\ t)}^{\uparrow}\ \lfloor y \rfloor) \\
\forall y.(\boxed{c_1\ t\ y\ (f\ t\ \lfloor (c_2\ t\ y) \rfloor)}^{\uparrow} &= M\ \boxed{(c\ t)}^{\uparrow}\ \lfloor y \rfloor)
\end{aligned}$$

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. So we speculate a wave rule which might apply the right hand side:

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

This schematic wave rule contains uninstantiated meta-variables and we must keep it so that later we can show that it is a valid rewrite once all the meta-variables have been instantiated. In doing this we bridge the gap in the proof created in this speculative step.

The proof continues in the same manner as the verification proof.

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

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

Fertilisation applies as before and we are left with a residue:

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

This residue contains two expressions which have the same structure as our original synthesis problem so we can recursively apply the synthesis process to instantiate M_1 and M_2.

Now the synthesis differs from the verification because the schematic wave rule we used to unblock rippling must be instantiated. The meta-variables M_1 and M_2 have been instantiated to h_1 and h_2 where $c_1 = h_1$ and $c_2 = h_2$.

$$(M\ \boxed{(M\ \underline{t})}^{\uparrow}\ \lfloor y \rfloor) \Rightarrow \boxed{h_1\ t\ y\ \underline{(M\ t\ \lfloor (h_2\ t\ y)\rfloor)}}^{\uparrow}$$

An instantiation for the meta-variable M is now chosen by comparing the speculative wave rule to wave rules for available higher-order functions. The structure of h_1 and h_2 informs the choice of higher-order function.

After the higher-order function is chosen the schematic wave rule can be proven like any other goal and the function argument of the chosen higher-order function will be instantiated as a result.

The base case of the induction will then be taken care of using symbolic evaluation. If the chosen higher-order function has a base case argument then this may not be instantiated in the proof of the schematic wave rule. In this case the base case proof will instantiate this value.

4.3. Example - HOF synthesis from matrix multiplication

We can use the plan on the matrix multiplication example to find combinations of higher-order functions equivalent to the `rowsmult`, `rowmult` and `transpose` functions. We will look at synthesising the expressions equivalent to `rowsmult`. Equations needed in the synthesis of the `rowsmult` expression are as follows:

$$rowsmult\ []\ cs = [] \quad (12)$$

$$rowsmult\ (r :: rs)\ cs = (rowmult\ r\ cs) :: (rowsmult\ rs\ cs) \quad (13)$$

$$rowsmult\ r\ [] = [] \quad (14)$$

$$rowsmult\ r\ (c :: cs) = (dotprod\ r\ c) :: (rowmult\ r\ cs) \quad (15)$$

The equations for the candidate higher-order functions are:

$$map\ f\ [] = [] \quad (16)$$

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

$$foldr\ f\ b\ [] = b \quad (18)$$

$$foldr\ f\ b\ h :: t = (f\ h\ (foldr\ f\ b\ t)) \quad (19)$$

$$foldl\ f\ a\ [] = a \quad (20)$$

$$foldl\ f\ a\ h :: t = (foldl\ f\ (f\ a\ h)\ t) \tag{21}$$
$$scan\ f\ a\ [\,] = [\,] \tag{22}$$
$$scan\ f\ a\ h :: t = (f\ a\ h) :: (scan\ f\ (f\ a\ h)\ t) \tag{23}$$

We first construct a conjecture by equating the function, **rowsmult**, with and unknown meta-variable M.

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

Induction is applied.
Base case

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

Step case

$$rowsmult\ t\ l = M\ t\ l \vdash$$
$$rowsmult\ (\ h :: t\ )\ \lfloor l \rfloor \quad = \quad M\ (\ h :: t\ )\ \lfloor l \rfloor$$

The step case is tackled first because M must be instantiated before the base case can be attempted. We use rippling to guide the rewriting of the step case.

$$(rowmult\ h\ \lfloor l \rfloor) :: (rowsmult\ t\ \lfloor l \rfloor) \quad = \quad M\ (h :: t)\ \lfloor l \rfloor$$

On the right hand side of the equation rippling is blocked so we perform a speculative ripple step using the schematic wave rule:

$$M\ (h :: t)\ \lfloor l \rfloor \Rightarrow M_1\ h\ l\ t\ (M\ t\ \lfloor l \rfloor)$$

This wave rule imitates the type of rippling that took place on the left.

$$(rowmult\ h\ l) :: (rowsmult\ t\ \lfloor l \rfloor) \quad = \quad M_1\ h\ l\ t\ (M\ t\ \lfloor l \rfloor)$$

We now split the equation using substitution axiom. This merges the wave fronts so that the induction hypothesis is contained whole within the wave front.

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

The wave hole $(rowsmult\ t\ l) = (M\ t\ l)$ can be replaced by $true$ using the induction hypothesis. The wave front residue can now be the subject of further synthesis attempts.

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

Recursive application of synthesis 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 synthesis may find an expression involving map or $foldr$. Let us consider the expression involving $map(M_1 \ = \ \lambda u.\lambda v.\lambda w.((map\ \lambda x.(dotprod\ u\ x)\ v) :: w))$.
Once M_1 has been fully instantiated the sub-problem created by the meta-ripple is tackled.

$$M\ (h :: t)\ l \ = \ (map\ \lambda x.(dotprod\ h\ x)\ l) :: (M\ t\ l)$$

higher-order function is chosen for M,
At this point an instantiation for the meta-variable M is chosen by comparing the speculative wave rule to an actual wave rule for a higher order function. If we consider map, $foldr$, $foldl$ and $scan$, we have:

$$\begin{array}{rcll}
map\ f\ \boxed{(h :: \underline{t})}^{\uparrow} & = & \boxed{(f\ h) :: \underline{(map\ f\ t)}}^{\uparrow} & \checkmark \\
foldr\ f\ b\ \boxed{(h :: \underline{t})}^{\uparrow} & = & \boxed{f\ h\ \underline{(foldr\ f\ b\ t)}}^{\uparrow} & \checkmark \\
foldl\ f\ \lfloor a \rfloor\ \boxed{(h :: \underline{t})}^{\uparrow} & = & foldl\ f\ \lfloor (f\ h\ a) \rfloor\ t & \times \\
scan\ f\ \lfloor a \rfloor\ \boxed{(h :: \underline{t})}^{\uparrow} & = & \boxed{(f\ h\ a) :: \underline{(scan\ f\ \lfloor (f\ h\ a\ h) \rfloor\ t)}}^{\uparrow} & \times
\end{array}$$

✓ *map* might be chosen since there is a :: as the outermost functor on the right and there is no sink term.

✓ *foldr* might be chosen because there is no sink term.

✗ *foldl* will not be chosen because there is a sink term but no wave term in the *foldl* wave rule whereas the speculative wave rule contains a wave term but no sink term.

✗ *scan* will not be chosen because there it requires there to be a sink term in the speculative wave rule. This sink term is also required to be the same as the part of the wave term which is the head argument position of the ::.

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:

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

$$\text{where } f_1 = \lambda x.\lambda y.((map\ \lambda z.(dotprod\ x\ z)\ l) :: y)$$

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.

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

Rewriting with the base case *rowsmult* and *foldr* equations yields:

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

Now the base case variable B can be instantiated to [] and the proof finished.
The final instantiation for M is as follows:

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

5. Implementing the HOF synthesis plan with $\lambda Clam$

5.1. *Clam* & λ*Clam*

The original idea for rippling was presented in [Bun88b] 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, NuPRL has been reimplemented in the logic programming language Prologand renamed *Oyster*[Hor88]. In *Oyster*, the search for a proof must be guided externally by interaction with a human or another Prologprogram. Thus, the meta-level proof planner *Clam* [BvHHS90] is built on top of *Oyster* and provides tactics which guide the proof search. Such tactics are described by methods which specify their pre- and post-conditions, and encode the heuristics necessary for the automated search for a proof plan. A meta-level proof plan, once created, may be 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+91]. Next, synthesis and middle-out reasoning capabilities were introduced[KBB93]. Most recently, a critics mechanism has been provided[IB95] 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, λ*Clam* is a higher-order version of *Clam* written in λProlog the higher-order 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 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 λ*Clam* system enabled the close integration of middle-out reasoning and proof planning[RSG98].

5.2. λ*Clam* and HOF Synthesis

In trying to synthesise HOFs with λ*Clam* a number of changes were needed. λ*Clam* uses methodicals to specify which methods shall be applied in a proof attempt and the order in which they should be applied. Such methodicals provide a flexible basis for modifying proof planning behaviour. In particular, custom methods built from methodicals can be input into λ*Clam* along with a conjecture so no re-compiling of the planner is necessary.

However, the necessary lower-level manipulation to perform synthesis 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.

For example, given:

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

the λ*Clam* planner immediately applied symbolic evaluation and used the identity rewrite rule($A\ =\ A$) to instantiate M back to the original to $rowsmult$. Thus, the instantiation of M needs to be restricted.

To control instantiation, a meta-variable may now be represented in a goal by a place holder function. The actual meta-variable is added to a list of meta variables, which is carried around with the goal to allow the effects of instantiation to be global. When a meta-variable needs to be instantiated, the representation is replaced with the meta-variable in the list and rewriting can occur as normal.

For example, in:

$$\begin{array}{c}[(1\ M),\ldots]\\ rowsmult\ k\ l\ =\ (meta2\ 1)\ k\ l\end{array}$$

$meta2$ is the representation of the meta-variable in the goal, indicating here that it should have arity 2. The integer argument 1 to $meta2$ is the index into the meta-variable list, linking the representation in the goal to the actual meta-variable, M, stored in the list.

A number of other modifications were required to enable HOF synthesis in λ*Clam*:

- The methodical structure controling sequencing was strengthened to ensure that the partial instantiation of a meta-variable in one part of a proof constrains subsequent instantiation.
- New methods were introduced for manipulating the meta-annotation 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 HOF to be used for the synthesis.

6. Results

The result for the $rowsmult$ example 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 synthesis prototype and forcing it to backtrack gives several other results:

Result	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

Note that there are significant time differences between finding groups of related syntheses. These arise because synthesis 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 synthesis 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.

The synthesis has been used to find results for several further examples involving the HOFs map, $foldr$, $foldl$, and $scan$. The following table gives for each example, the potential HOFs, the number of results found with the synthesis prototype and the overall time to find all results. The examples are defined in the Appendix.

Function	HOFs	Results	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 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 occurences of (fah) in:

$scanfa[] = []$
$scanfa(h :: t) = (fah) :: (scanf(fah)at)$

, 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 anomolous 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. 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. Conclusions

We have shown that proof planning is an appropriate technique for discovering HOFs in programs that lack them and for exploring the space of possible HOFs. While we have demonstrated our approach on relatively small examples, we are encouraged that it appears to scale well with the number of permissible HOFs. It will be interesting to apply our current synthesis technology and HOF selection to progressively larger programs.

Our work has been conducted in the context of a parallelising compiler for SML, with the objective of exploiting parallelism latent in potential HOF use in programs. Although automatic translators between Core SML and Lambda Prolog have been constructed, at present, HOF synthesis is independent of the compiler. It will be interesting to link HOF synthesis more closely with parallelising compilation. Initially, the compiler might invoke synthesis if no HOFs are present in the original program, to find a first set of HOFs. Subsequently, the compiler could re-invoke synthesis, to transform the program, if profiling and analysis suggest that present or synthesised HOFs cannot not deliver exploitable parallelism. At simplest, this might be one-shot, blind synthesis to return a next alternative. In the longer term, it would be more effective for the compiler to provide profile and performance indicators to constrain synthesis. Such constraints are likely to prove very important as the number of permitted HOFs increases, to manage exponential growth in the space of possible syntheses.

Finally, it would be interesting to analyse the times taken for the different syntheses in terms of the structural complexities of the source programs. Here, it might be useful to characterise patterns of nesting and composition in HOF use. These might be combined with analyses of synthesis behaviour for each HOF from a program consisting essentially of that HOF, to try and establish a predictive model for HOF synthesis.

Acknowledgments

This work is supported by UK EPSRC grant GR/L42889, "Parallelising compilation of Standard ML through prototype instrumentation and transformation".

We are pleased 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.

A. Examples & Results

The definitions for the example functions and their equivalent synthesised result are as follows.

- *reverse*

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

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

- *presums*

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

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

- *sum2d*

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

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

- *subset*

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

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

- $mergesort$

$$\begin{aligned} mergesort\ x &= mergeall\ (sortmall\ x) \\ mergeall\ [] &= [] \\ mergeall\ (h :: t) &= merge\ h\ (mergeall\ t) \\ sortmall\ [] &= [] \\ sortmall\ (h :: t) &= (sortm\ h) :: (sortmall\ t) \\ sortm\ [] &= [] \\ sortm\ (h :: t) &= insertm\ h\ (sortm\ t) \end{aligned}$$

$$mergesort = \lambda x.(foldr\ \lambda y.\lambda z.(merge\ y\ z)\ []\ (map\ \lambda y.(foldr\ \lambda z.\lambda u.(insertm\ z\ u)\ []\ y)\ x))$$

- $squs2d$

$$\begin{aligned} squs2d\ [] &= [] \\ squs2d\ (h :: t) &= (squares\ h) :: (squs2d\ t) \\ squares\ [] &= [] \\ squares\ (h :: t) &= (h \times h) :: (squares\ t) \end{aligned}$$

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

- $mmult$

$$\begin{aligned} mmult\ m_1\ m_2 &= (rowsmult\ m_1)\ (transpose\ m_2) \\ rowsmult\ []\ cols &= [] \\ rowsmult\ (row :: rows)\ cols &= (rowmult\ row\ cols) :: (rowsmult\ rows\ cols) \\ rowmult\ row\ [] &= [] \\ rowmult\ row\ (col :: cols) &= (dotprod\ row\ col) :: (rowmult\ row\ cols) \\ dotprod\ []\ [] &= 0 \\ dotprod\ (h_1 :: t_1)\ (h_2 :: t_2) &= (h_1 \times h_2) + ((dotprod\ t_1)\ t_2) \\ transpose\ [] &= [] \\ transpose\ (row :: rows) &= dist\ row\ (transpose\ rows) \end{aligned}$$

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

References

[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.

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

[Bd97] R. Bird and Oege de Moor. *Algebra of Programming*. Prentice-Hall, 1997.

[BS66] D. W. Barron and C. Strachey. Chapter 3: Programming. In L. Fox, editor, *Advances in Programming and Non-numerical Computation*. Pergammon, 1966.

[BSH90] A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L.H. Clarke, editor, *Proceedings of UK IT 90*, pages 221–6. IEE, 1990. Also available from Edinburgh as DAI Research Paper 448.

[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*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[Bun88a] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th 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.

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

[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.

[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.

[Chu41] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[Col89] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman, 1989.

[GMW79] 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.

[Gor73] M. J. C. Gordon. An Investigation of *Lit*. Technical Report Memorandum: MIP-R-101, School of Artificial Intelligence, University of Edinburgh, 1973.

[Ham00] M. Hamdan. *A Combinational Framework for Parallel Programming using Algorithmic Skeletons*. PhD thesis, Department of Computing and Electrical Engineering, Heriot-Watt University, 2000.

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

[Hug84] R. J. M. Hughes. *The Design and Implementation of Programming Languages*. PhD thesis, University of Oxford, 1984.

[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).

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

[Ire92] A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proofs. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag, 1992. Also available from Edinburgh as DAI Research Paper 592.

[KBB93] 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.

[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.

[Kle52] S. C. Kleene. *Introduction to Metamathematics*. North-Holand, 1952.

[McC60] J. McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine: Part 1. *Communications of the ACM*, 3:184–195, April 1960.

[MIK97] 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. School of Mathematical and Computer Sciences, University of St Andrews, September 1997.

[MSBK00] G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested Algorithmic Skeletons from Higher Order Functions. *Parallel Algorithms and Applications: special issue on High Level Models and Languages for Parallel Processing*, September 2000.

[Pau96] L. C. Paulson. *ML for the Working Programmer*. CUP, 2nd edition, 1996.

[RSG98] J.D.C Richardson, A. Smaill, and Ian Green. System description: proof planning in higher-order logic with lambda-clam. 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.

[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.

[Sto84] W. Stoye. Director Strings on SKIP. In L. Augustsson, J. Hughes, T. Johnsson, and K. Karlson, editors, *Proceedings of the Workshop on Implementation of Functional Languages*, Report 17, pages 347–356. Programming Methodology Group, University of Göteborg and Chalmers University of Technology, February 1984.

[Tur79] D. A. Turner. A New Implementation Technique for Applicative Languages. *Software Practice and Experience*, 9:31–49, 1979.

[Weg71] P. Wegner. *Programming Languages, Information Structures and Machine Organisation*. McGraw-Hill, 1971.

[WMM+92] A. Wallace, G. Michaelson, P. McAndrew, K. Waugh, and W. Austin. 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, February 1992.