Higher Order Function Synthesis Through Proof Planning

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

Abstract

The close association between higher order functions and algorithmic skeletons is a promising source of automatic parallelisation of programs. An approach to automatically synthesizing higher order functions from functional programs through proof planning is presented. Our work has been conducted within the context of a parallelising compiler for SML, with the objective of exploiting parallelism latent in potential higher order function use in programs.

1 Introduction

Pure functional languages, satisfying the Church-Rosser property of evaluation order independence, have long been proposed as a basis for parallel programming. We have been investigating an approach to automatic parallelisation of functional programs [14] which draws heavily on Cole's characterisation of algorithmic skeletons and their equivalent *Higher Order Functions* (HOFs) [6]. An algorithmic skeleton is an abstraction from a pattern of program control, usually parallel control, which may be instantiated with problem specific operations. For instance, *foldr* may be realised by a "divide and conquer" skeleton based using a binary tree of processors. Many other similar close associations between HOFs and parallel control structures have been identified.

While HOFs are powerful abstract programming components, in practice they appear to be under used, and thus opportunities for their parallel exploitation in 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. Paulson makes this explicit in the Preface to his widely used Standard ML text [16]. Furthermore, HOF use often appears to be forced or unnatural.

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. We adopt a theorem proving approach that supports the dynamic synthesis of transformation rules based upon inductive synthesis proofs.

2 The synthesis problem

To motivate our approach to the synthesis of higher order terms we present an example based upon matrix multiplication. In particular we focus upon *rowsmult*, a function that takes two matrices and multiplies all rows of the first matrix by all columns of the second. Our definition of *rowsmult* is given below:

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

Note that the definition of *rowsmult* does not contain any HOFs, nor do any of the auxiliary definitions. Transformation rules that introduce HOFs are therefore crucial if we are to exploit the skeletons approach to parallelism discussed in §1. In the case of the matrix multiplication example we are interested in the replacement of terms involving *rowsmult* by equivalent higher order terms. One such replacement can be achieved by the following transformation rule:

$$\begin{aligned} &(rowsmult\ X\ Y) \Rightarrow \\ &\quad (foldr\ \lambda z.\lambda u.((rowmult\ z\ Y) :: u)\ [\,]\ X) \end{aligned} \tag{1}$$

The problem that we are concerned with here is how such rules can be automatically synthesized and be guaranteed to be equivalence preserving.

3 Proof planning for synthesis

Our approach builds upon *proof planning* [3], and in particular *middle-out reasoning* [4]. Middle-out reasoning is a technique that supports a schematic style of proof construction and which has been applied to the problems of proof patching [9, 10, 11] and program synthesis [12, 1, 18, 13, 8]. Here we use the power of middle-out reasoning to guide the synthesis of higher order terms within the context of transformation rules.

Our starting point for synthesis is a schematic verification conjecture. In the case of the *rowsmult* synthesis problem outlined in §2, we begin with[1]:

$$\forall t : list(\tau). \forall l : list(\tau).\ rowsmult\ t\ l = M\ t\ l \qquad (2)$$

The basic idea behind our approach is that an instantiation for $\mathcal{M}$is computed as a side-effect to proving (2). Reasoning about transformations such as (1) requires proof by mathematical induction.

We build upon a proof plan for mathematical induction, and in particular a method called *rippling*. Here we only give an outline of rippling, for a definitive account the interested reader is directed to [2, 5]. The application of the proof plan for induction to (2) gives rise to an induction hypothesis of the form[2]:

$$\forall l' : list(\tau).\ rowsmult\ t\ l' = \mathcal{M}\ t\ l' \qquad (3)$$

and an annotated induction conclusion of the form:

$$rowsmult\ \boxed{(h :: \underline{t})}^{\uparrow}\ \lfloor l \rfloor = \mathcal{M}\ \boxed{(h :: \underline{t})}^{\uparrow}\ \lfloor l \rfloor$$

The annotated terms are called *wave-fronts*. Note that wave-fronts are directed, this enables termination of the method to be guaranteed[3] [2]. The additional $\lfloor \ldots \rfloor$ annotations delimit terms within the conclusion that correspond to universally quantified variables in the hypothesis. Moving wave-fronts into such positions provides one way of achieving a match with the hypothesis. A successful application of rippling corresponds to reducing the differences to the point at which a match between conclusion and hypothesis is achieved. The movement of wave-fronts is achieved by annotated rewrite rules known as wave rules. Wave rules are derived automatically from definitions and lemmas and are guaranteed to progress the reduction of differences while preserving the unannotated term structure. The ripple proof for the *rowsmult* example is presented in figure 1. Note that instantiating $\mathcal{M}_1$ to be

$$\lambda w.\lambda x.\lambda y.\lambda z.(rowmult\ w\ x) :: z$$

at step (4) in the proof gives rise to a trivial proof residue. Alternatively, applying the synthesis proof plan recursively to the residue would yield a HOF equivalent to *rowmult*. We return to this choice point in §4. Here we focus on the simple match that provides the following partial instantiation for (5):

$$\mathcal{M}\ \boxed{(X :: \underline{Y})}^{\uparrow}\ Z \Rightarrow \boxed{(rowmult\ X\ Y) :: (\underline{\mathcal{M}\ Y\ Z})}^{\uparrow} \qquad (6)$$

As mentioned above, the instantiation of $\mathcal{M}$ is constrained by the wave rules that are derived from the available higher order function definitions. With in the context of (6), map and *foldr* provide wave rules that will match with (6). Note that *foldr* gives rise to transformation rule (1).

4 Implementation and results

Our synthesis proof plan was developed within λ*Clam* [17], a proof planner written in λProlog [15], 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.

There are a number of choice points within our synthesis proof plan. This is illustrated in table 1 where the complete set of solutions generated by our synthesis plan for the *rowsmult* problem are presented[4]. 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 $scan$. Table 2 gives for each example, the potential HOFs, the number of results found with the synthesis prototype and the overall time to find all results. For rev and $presums$, synthesis finds either the single HOF or the original function. For $sum2d$ and $subset$, synthesis finds the top level *foldr*, the top and inner *foldr* or the original function. However, where $sum2d$ involves an arity 1 function calling an arity 2 function, $subset$ involves nested arity 2 functions, which both have more term structure than $sum2d$s components. $mergesort$, $sum2d$ and $subset$ all have two *foldr* sites. Here, $mergesort$ appears anomalous as four solutions are found, compared with only three for $sum2d$ and $subset$. However, $mergesort$ is defined in terms of two composed functions, so an additional result is found for its right-hand side as well as for the original top-level call. Despite more solutions being found, synthesis from

[1] Note that we adopt the convention of using upper case letters to denote first order variables and upper case calligraphic letters to denote higher order meta-variables.

[2] We focus here only on the step case proof obligation.

[3] Guarantee does not hold within the context of middle-out reasoning.

[4] Our experiments were carried using Sun UltraSPARC 10.

Step case proof:

$$
\begin{aligned}
rowsmult\ \boxed{(h :: \underline{t})}^{\uparrow}\ \lfloor l \rfloor &= \mathcal{M}\ \boxed{(h :: \underline{t})}^{\uparrow}\ \lfloor l \rfloor \\
\boxed{(rowmult\ h\ l) :: (\underline{rowsmult\ t\ \lfloor l \rfloor})}^{\uparrow} &= \boxed{\mathcal{M}_1\ h\ l\ t\ (\underline{\mathcal{M}\ t\ \lfloor l \rfloor})}^{\uparrow} \\
\boxed{\forall x : list(\tau).\ ((rowmult\ h\ l) :: x = \mathcal{M}_1\ h\ l\ t\ x)\ \wedge\ \underline{(rowsmult\ t\ \lfloor l \rfloor) = (\mathcal{M}\ t\ \lfloor l \rfloor)}}^{\uparrow} &
\end{aligned} \tag{4}
$$

Wave rules:

$$
\begin{aligned}
rowsmult\ \boxed{(X :: \underline{Y})}^{\uparrow}\ Z &\Rightarrow \boxed{(rowmult\ X\ Y) :: (\underline{rowsmult\ Y\ Z})}^{\uparrow} \\
\mathcal{M}\ \boxed{(X :: \underline{Y})}^{\uparrow}\ Z &\Rightarrow \boxed{\mathcal{M}_1\ X\ Y\ Z\ (\underline{\mathcal{M}\ Y\ Z})}^{\uparrow} \\
\boxed{\mathcal{M}_3\ (\underline{X})}^{\uparrow} = \boxed{\mathcal{M}_4\ (\underline{Y})}^{\uparrow} &\Rightarrow \boxed{(\mathcal{M}_3\ Z\ = \mathcal{M}_4\ Z)\ \wedge\ (\underline{X = Y})}^{\uparrow}
\end{aligned} \tag{5}
$$

Figure 1. Synthesis proof for *rowsmult*

Result	Solution	Time (secs)
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 1. Results for *rowsmult* problem

Function	HOFs	Results	Total Time (secs)
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

Table 2. Results for corpus

$mergesort$ 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. A more detailed discussion of our results can be found in [7].

5 Conclusion

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 λProlog 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 synthesized HOFs cannot 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 analyze the times taken for the different syntheses in terms of the structural complexities of the source programs. Here, it might be useful to characterize 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 was 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.

References

[1] A. Armando, A. Smaill, and I. 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.

[2] D. Basin and T. Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.

[3] 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.

[4] A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L. Clarke, editor, *Proceedings of UK IT 90*, pages 221–6. IEE, 1990.

[5] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993.

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

[7] A. Cook. *Using Proof in Transformation Synthesis for Automatic Parallelisation*. PhD thesis, Heriot-Watt University, 2001.

[8] P. Flener, K.-K. Lau, M. Ornaghi, and J. Richardson. An abstract formalisation of correct schemas for program synthesis. *Journal of Symbolic Computation*, 30(1):93–127, 2000.

[9] 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.

[10] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996.

[11] A. Ireland and J. Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):65–97, Feb. 2001.

[12] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for synthesis and induction. *Journal of Automated Reasoning*, 16(1–2):113–145, 1996.

[13] D. Lacey, J. D. C. Richardson, and A. Smaill. Logic program synthesis in a higher order setting. In *Proceedings of the First International Conference on Computational Logic (CL2000)*, volume 1861 of *Lecture Notes in AI*, pages 912–925. Springer Verlag, 2000.

[14] 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.

[15] D. Miller and G. Nadathur. An overview of λProlog. In R. Bowen, K. & Kowalski, editor, *Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic Programming*. MIT Press, 1988.

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

[17] J. Richardson, A. Smaill, and I. Green. System description: proof planning in higher-order logic with lambdaclam. In C. Kirchner and H. Kirchner, editors, *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, Lindau, Germany, July 1998.

[18] 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.