Continuations of Proof Strategies

Julian Richardson[1] and Alan Smaill[2]

[1] Department of Computing and Electrical Engineering
Heriot-Watt University
Riccarton Campus
Edinburgh EH14 4AS
julianr@cee.hw.ac.uk

[2] Division of Informatics
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
smaill@dai.ed.ac.uk

1 Introduction

Proof planning [1] is a technique for automatically constructing proofs from abstractions of tactics called *methods*. Proof planning has been successfully applied in many domains, for example: mathematical induction [1], higher-order program synthesis [7] and hardware verification [3]. In the first proof planner, *Clam* [2], application of methods during the search for a proof plan was controlled by a simple waterfall (§2.1). In this paper, we describe *methodical* expressions, which we use to define more complex proof search strategies. Methodicals compose methods in the same way that tacticals compose tactics. The work described in this paper has been implemented as part of the λ*Clam* system [11, 10].

We present a meta-interpreter which gives a semantics to methodical expressions. In conjunction with a mechanism for executing methods and a search strategy, this meta-interpreter can be used as a proof planner. This planner does not have any notion of proof state, and is unsuitable for interactive proof.

We present a set of transformation rules which we use to define a new meta-interpreter which does maintain a notion of proof state, is suitable for interactive proof, and we prove that it is equivalent to the first in the sense that it produces the same plans. This meta-interpreter has been used for some time as the basis of the interactive planner in λ*Clam*, although it does not yet treat the whole of the methodical language.

The transformation rules can be interpreted as defining a notion of *continuation* [9] for proof strategies. Previous work on a continuation semantics for Prolog [4] indicates how the meta-interpreter should be extended to cover the more problematical elements of the methodical language.

The definition of methodicals, use of formal proof to verify the equivalence of proof planners, and use of continuations in the context of proof planning are novel.

2 Tactics, Methods, Tacticals and Methodicals

A *tactic* [5] is a procedure which constructs a piece of proof in some logic. The tactic is defined as a composition of primitive rules of inference. For example, a tactic may strip all universal quantifiers from the front of a goal and then apply a lemma. When a tactic is applied to a goal in the proof, one branch is created for each subgoal which has not been proved by the tactic, and the goal is labeled with the tactic, not with the individual proof rules the tactic applied. LCF defines *tacticals* for composing tactics.

In proof planning, we distinguish between the *object-level*, where proofs are constructed using tactics, and the *meta-level* (where the control, search and proof explanation properties are better than at the object-level), where schematic proofs are constructed from abstractions of the object-level tactics called *methods*. Meta-level goals G are generated by the following grammar:

$$G ::= \mathit{emptyGoal} \mid A \mid \mathit{allGoal}\ \lambda v.G[v] \mid G \ast\ast G$$

where meta-level sequents A are abstractions of object-level sequents formed by operations (which we do not here formalise) such as omitting hypotheses, annotating terms, or replacing terms by meta-variables. *allGoal* represents meta-level universal quantification, and $\ast\ast$ represents pairing of meta-level goals.

A methodical expression consists of methods linked by methodicals. Methodical expressions which contain no methodicals are called *atomic methods*, otherwise they are *compound methods*. The following methodicals, and their types, are available:[1]

Methodical	Type	Description
id_meth	meth	Does nothing.
orelse_meth	meth → meth → meth.	First, which failing second
cond_meth	(goal → o) → meth → meth → meth.	If condition holds, then meth1, otherwise meth2
pair_meth	meth → meth → meth.	Pair of methods for pair of goals
try_meth	meth → meth.	Either given method, or id_meth
repeat_meth	meth → meth.	Exhaustive recursion on subgoals – oblige to succeed at least once
then_meth	meth → meth → meth.	Chain second method to all subgoals of first
map_meth	meth → meth.	Map method over compound goal structure
complete_meth	meth → meth.	Only succeed if subgoals all trivial
cut_meth	meth → meth.	Only give first solution

2.1 Hierarchical Proof Planning

Clam employed a two-deep method hierarchy, consisting of methods and *submethods*. *Clam*'s *waterfall* is a list of methods whose applicability is tested whenever a new subgoal is generated. Submethods must be invoked explicitly by methods (or submethods), and cannot appear in the waterfall. The need to explicitly invoke submethods made expressing complex proof strategies declaratively difficult. For example, the control structure of *Clam*'s *step_case* method, depicted in Figure 1, was realised by making explicit calls to the planning apparatus, thereby spoiling any chance of a declarative reading of the method.

In λ*Clam*, we use methodicals to declaratively define a compound method which imposes the structure of the step_case strategy:

[1] For reasons of clarity we simplify a little and do not distinguish between *then_meth* and *then_meths*.

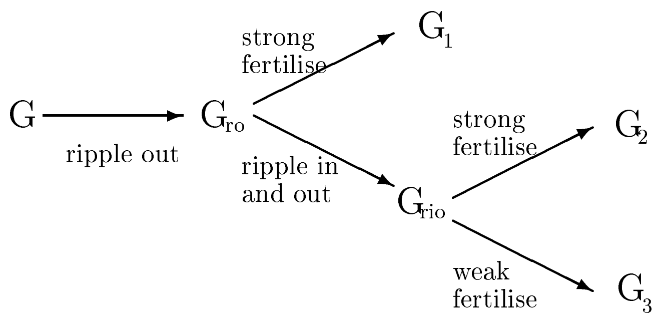

Fig. 1. The structure of *Clam*'s *step_case* method. The result of method application is the generated subgoal with the lowest numerical index.

```
method_def step_case
  (cut_meth (then_meth
               (repeat_meth ripple_out_once)
               (orelse_meth strong_fertilise
                             (then_meth
                                (repeat_meth ripple_inout_once)
                                (orelse_meth strong_fertilise
                                             weak_fertilise))))).
```

Such structures make explicit the control component of the planning strategy, which eases comprehension and may permit some formal reasoning about the planning process.

In λ*Clam*, the simple waterfall of *Clam* is replaced by a compound method *top_meth*. Planning normally starts by trying to expand *top_meth*. This is the default definition, but others are used to create special-purpose proof strategies with more complex forms of top-level control.

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

3 A Meta-Interpreter for Methodicals

We now present rules which define a meta-interpreter for methodicals. We use the notation $M_1 : G \mapsto G'$ to mean that method M_1 applies to goal G resulting in subgoal G' (which could consist of one or more subgoals paired with $**$). The rules below can be interpreted as an inductive definition of the (non-deterministic) $\mapsto$ relation. Methodical expressions are expanded by applying the rules in a goal-directed bottom-to-top way.

$$\frac{}{id\_meth : G \mapsto G} \tag{1}$$

$$\frac{M : G \mapsto G'}{repeat\_meth\ M : G \mapsto G'} \tag{2}$$

$$\frac{M : G \mapsto G'' \qquad repeat\_meth\ M : G'' \mapsto G'}{repeat\_meth\ M : G \mapsto G'} \tag{3}$$

$$\frac{M_1 : G \mapsto G'' \qquad M_2 : G'' \mapsto G'}{then\_meth\ M_1\ M_2 : G \mapsto G'} \tag{4}$$

$$\frac{M_1 : G \mapsto G'}{orelse\_meth\ M_1\ M_2 : G \mapsto G'} \tag{5}$$

$$\frac{M_2 : G \mapsto G'}{orelse\_meth\ M_1\ M_2 : G \mapsto G'} \tag{6}$$

$$\frac{}{try\_meth\ M : G \mapsto G} \tag{7}$$

$$\frac{M : G \mapsto G'}{try\_meth\ M : G \mapsto G'} \tag{8}$$

$$\frac{M_1 : G_1 \mapsto G_1' \qquad M_2 : G_2 \mapsto G_2'}{pair\_meth\ M_1\ M_2 : G_1\ **\ G_2 \mapsto G_1'\ **\ G_2'} \tag{9}$$

$$\frac{M : G \mapsto G'}{map\_meth\ M : G \mapsto G'} \text{if } G \text{ is not a pair} \tag{10}$$

$$\frac{map\_meth\ M : G_1 \mapsto G_1' \qquad map\_meth\ M : G_2 \mapsto G_2'}{map\_meth\ M : G_1\ **\ G_2 \mapsto G_1'\ **\ G_2'} \tag{11}$$

$$\frac{M_1 : G \mapsto G'}{cond\_meth\ C\ M_1\ M_2 : G \mapsto G'} \text{if } C \text{ holds} \tag{12}$$

$$\frac{M_2 : G \mapsto G'}{cond\_meth\ C\ M_1\ M_2 : G \mapsto G'} \text{if } \neg C \text{ holds} \tag{13}$$

$$\frac{M : G \mapsto G'}{complete\_meth\ M : G \mapsto G'} \text{if } G' \text{ trivial} \tag{14}$$

$$\frac{M : G[x/v] \mapsto H[x/v]}{M : allGoal\ \lambda v.G \mapsto allGoal\ \lambda v.H} x \text{ does not appear in } G \text{ or } H \tag{15}$$

We do not here define *cut_meth*, since (as for the Prolog cut, !) its meaning is procedural. We show how to handle *cut_meth* in Section 7.

These rules are non-deterministic, but when applied in conjunction with a search strategy (for example depth-first search), and a mechanism for interpreting atomic methods, they define a proof planner. Proof planning is applied to a goal G, yielding subgoals G', by expanding the expression $top\_meth : G \mapsto G'$.

4 Representation of Proof State

It is natural to represent the state of a partial proof as a tree, in which nodes are labeled with subgoals, and branching represents conjunction. As a proof proceeds, nodes are added to or deleted from the tree. Presenting such a proof tree to a user provides an excellent form of feedback, allowing the user to comprehend and perhaps influence the course of the proof attempt, as was demonstrated in the *XBarnacle* [8] interactive proof planner (§8).

In λ*Clam*, however, the hierarchical nature of the proof strategy as represented by methodical expressions means that we must also label each node in the partial proof tree with the state of meta-interpretation of the proof strategy. For example, consider the proof constructed by applying a proof strategy *repeat_meth (then_meth* M_1 M_2*)* to a goal G. In (4) below, we see the state of meta-interpretation (proceeding from bottom to top, as in Section 3) up to the point where method M_1 has been successfully applied for the first time. Goals which are yet to be determined are denoted by "??".

$$\dfrac{\dfrac{M_1 : G \mapsto G' \quad M_2 : G' \mapsto ??}{then\_meth\ M_1\ M_2 : G \mapsto ??} \qquad repeat\_meth\ (then\_meth\ M_1\ M_2) : ?? \mapsto ??}{repeat\_meth\ (then\_meth\ M_1\ M_2) : G \mapsto ??}$$

Suppose we wish the pause the proof at this point. What information do we need to record in order that we can return later and correctly continue where we left off? We must record the generated subgoal, G', but we must also record the state of interpretation of *repeat_meth (then_meth* M_1 M_2*)*, specifically that we should next apply M_2 to G', and then either apply *repeat_meth (then_meth* M_1 M_2*)* to the subgoal which arises, or stop. In the planner of Section 3 there is no way to derive or represent this information about how to continue.

In the next section, we give transformation rules which allow us to systematically derive this "continuation" information and represent it as a methodical expression, in this case *then_meth* M_2 *(orelse_meth id_meth (repeat_meth (then_meth* M_1 M_2*)))*.

5 Methodical Transformations and Proof State

We now present rewrite rules which can be used to transform any methodical expression M into an equivalent form $F \oplus L$ such that F is a singleton method (not a compound methodical expression), and application of *then_meth* F L to a goal G produces the same subgoals as applying M to G. This transformation can be neatly defined without reference to the structure of goals, and so the transformation rules are only applied to expressions which are not of the form *pair_meth* M_1 M_2 or *map_meth* M — these are handled by rules (9,10,11). We leave treatment of *cut_meth* (see Section 7) and *cond_meth* as further work.

We transform each methodical expression into an equivalent form $F \oplus L$ using a terminating, non-deterministic rewrite relation $\Rightarrow$ defined as follows:

$$M \Rightarrow M \oplus id\_meth \qquad \text{if } M \text{ is a singleton} \tag{16}$$

$$then\_meth\ M_1\ M_2 \Rightarrow M_1 \oplus M_2 \qquad \text{if } M_1 \text{ is a singleton} \tag{17}$$

$$then\_meth\ M_1\ M_2 \Rightarrow F \oplus (then\_meth\ L\ M_2) \quad \text{where } M_1 \Rightarrow F \oplus L \tag{18}$$

$$repeat\_meth\ M \Rightarrow F \oplus (then\_meth\ L\ (orelse\_meth\ (repeat\_meth\ M)\ id\_meth)) \quad \text{where } M \Rightarrow F \oplus L \tag{19}$$

$$\begin{aligned}
orelse_meth\ M_1\ M_2 &\Rightarrow F \oplus L && \text{where } M_1 \Rightarrow F \oplus L && (20)\\
orelse_meth\ M_1\ M_2 &\Rightarrow F \oplus L && \text{where } M_2 \Rightarrow F \oplus L && (21)\\
try_meth\ M &\Rightarrow F \oplus L && \text{where } M \Rightarrow F \oplus L && (22)\\
try_meth\ M &\Rightarrow id_meth \oplus id_meth && && (23)
\end{aligned}$$

The state of a partial proof is represented in a proof tree whose nodes are labelled by triples $\langle G, M, X\rangle$ of a subgoal G, the atomic method M applied at the node to generate its children (if it is not a leaf node), and the methodical expression X to apply to G in order to generate its subgoals. Planning proceeds at a node $\langle G, M, X\rangle$ by transforming X into the form $F \oplus L$, applying the atomic method F and extending the proof tree with the new node(s), labelling the new node(s) with the continuation L. We can then pause for interaction with a user or with proof critics. Proof planning can then be continued at some arbitrary (user-selected) node by applying the continuation associated with that node.

6 Equivalence of the Two Planners

We now prove that a planner which applies the transformation rules to transform each methodical into the form $F \oplus L$ before applying the meta-interpreter of section 3 is equivalent to a planner which uses only the meta-interpreter.

First some preliminaries. Two methodical expressions are equivalent if they have the same effect on goals when the standard meta-interpreter is applied:

$$M_1 \equiv M_2 \leftrightarrow (\forall G, G' \,.\, M_1 : G \mapsto G' \leftrightarrow M_2 : G \mapsto G')$$

We need two important properties of *then_meth*:

$$\begin{aligned}
then_meth\ (then_meth\ M_1\ M_2)\ M_3 &\equiv then_meth\ M_1\ (then_meth\ M_2\ M_3) && (24)\\
F \oplus L &\equiv then_meth\ F\ L \quad \text{if } F \text{ is atomic} && (25)
\end{aligned}$$

The first of these follows easily from the definition of *then_meth* in section 3 as long as the derivations of M_1, M_2 and M_3 are independent, which is the case in the absence of side-effects. The second is definitional.

Given a methodical expression M define the *size* of M, $|M|$ to be the number of (occurrences of) methodicals contained in M. If M is atomic then $|M| = 0$.

We first prove that every methodical expression can be transformed into the required form $F \oplus L$, and then that transformation does not change the interpretation of the methodical expression according to the rule of Section 3.

Theorem 1. *Every methodical expression can be transformed*

$$\forall M. \exists F, L. M \Rightarrow F \oplus L$$

Proof. The proof is by induction on the structure of M. We give the proof for the case where M = *repeat_meth M'*. The other cases are similar.

Base case: M atomic. Then by (16), $M \Rightarrow M \oplus id\_meth$.

Step case: by the induction hypothesis, assume $\exists F', L' \,.\, M' \Rightarrow F' \oplus L'$.

Then *repeat_meth M'*$\Rightarrow$ $F' \oplus (then\_meth\ L'\ (orelse\_meth\ (repeat\_meth\ M)\ id\_meth))$ by (19), i.e. we have produced witnesses for F and L as required.

Theorem 2. *Transformation does not alter the interpretation of methodical expressions.*

$$\forall M, F, L \forall G, G' . M \Rightarrow F \oplus L \rightarrow [M : G \mapsto G' \leftrightarrow F \oplus L : G \mapsto G']$$

Proof. Again, the proof is by induction on the size of M.

Base case: If M is atomic then there is nothing to prove, since clearly $M \Rightarrow M \oplus id\_meth$ and $M : G \mapsto G' \leftrightarrow M \oplus id\_meth : G \mapsto G'$.

Step case: Assume the result for all methodical expressions of size $k < n$. We now prove the result for methodical expressions of size n.

We here prove the result for *M=repeat_meth M'* and *M=orelse_meth* M_1 M_2. The proof is similar for the other cases.

- ***M=repeat_meth M'***:
  Assume *repeat_meth* $M' \Rightarrow F \oplus L$ (by Theorem 1, such a decomposition exists). From rule (19) of the definition of $\Rightarrow$, $M' \Rightarrow F \oplus L'$ for some L' and $L = (then\_meth\ L'\ (orelse\_meth\ (repeat\_meth\ M')\ id\_meth))$. Since $|M'| < |repeat\_meth\ M'|$, we can apply the induction hypothesis and assume (IH) that $M : G \mapsto G' \leftrightarrow F \oplus L' : G \mapsto G'$. Then

$$\begin{array}{c}
F \oplus then_meth\ L'\ (orelse_meth\ (repeat_meth\ M')\, id_meth) : G \mapsto G' \\
\leftrightarrow \text{By (25,24)} \\
then_meth\ (then_meth\ F\ L')\ (orelse_meth\ (repeat_meth\ M')\, id_meth) : G \mapsto G' \\
\leftrightarrow \text{By (4)} \\
then_meth\ F\ L' : G \mapsto H \ \bigwedge\ orelse_meth\ (repeat_meth\ M')\, id_meth) : H \mapsto G' \\
\leftrightarrow \text{By IH} \\
M' : G \mapsto H \ \bigwedge\ orelse_meth\ (repeat_meth\ M')\, id_meth) : H \mapsto G' \\
\leftrightarrow \text{By (5,6)} \\
(M' : G \mapsto H \ \bigwedge\ repeat_meth\ M' : H \mapsto G') \ \bigvee\ M' : G \mapsto G' \\
\leftrightarrow \text{By (3,2)} \\
repeat_meth\ M' : G \mapsto G'
\end{array}$$

- ***M=orelse_meth*** M_1 M_2:
  Suppose that $M_1 \Rightarrow F_1 \oplus L_1$ and $M_2 \Rightarrow F_2 \oplus L_2$ (again, Theorem 1 guarantees that such a decomposition exists). Then by the induction hypothesis (IH): $M_i : G \mapsto G' \leftrightarrow F_i \oplus L_i : G \mapsto G'$.
  By (20, 21), *orelse_meth* $M_1\ M_2 \Rightarrow F \oplus L \leftrightarrow F = F_1 \wedge L = L_1 \bigvee F = F_2 \wedge L = L_2$.
  Consider the case where $F = F_i$ and $L = L_i$:

$$\begin{array}{cl}
F_i \oplus L_i : G \mapsto G' & \\
\leftrightarrow & \text{By IH} \\
M_i : G \mapsto G' & \\
\leftrightarrow & \text{By (5 if } i = 1 \text{ or 6 if } i = 2) \\
orelse_meth\ M_1\ M_2 : G \mapsto G' &
\end{array}$$

7 Developing the Notion of Continuation

Continuations have been used in computer science for, amongst other things, providing a semantics for programming languages. We now outline how previous work described in [4] on a continuation semantics for Prolog can be exploited to both describe and extend the implementation of our planner.

[4] defines a language $\mathcal{B}$ which describes the control element of Prolog, treating conjunction, disjunction, predicate definition and cut. There are no terms or unification on terms; instead, predicates can perform primitive actions. Execution of a program induces a certain sequence of primitive actions. Certain features of $\mathcal{B}$ and their treatment with continuations can be usefully adapted to extend our meta-interpreter:

- Disjunction is handled in [4] by the concept of a *frame*. Execution of a disjunction introduces two frames, each of which models one of the possible subsequent execution paths. If execution fails in one of these frames then it is deleted and execution switches to the other frame.
  Similarly, in our meta-interpreter interpretation of *orelse_meth* results in the creation of two frames, each modelling one of the two possible subsequent interpretation paths. More precisely, let $\mu(G, M; E, T)$ denote that applying the methodical expression M and then applying the continuation methodical expression E. Let frames be separated by a colon, so $\mu(S : T)$ denotes the result of interpreting two alternatives, S and T. Then $\mu((\mathit{orelse\_meth}\ M_1\ M_2; E) : S) = \mu((M_1; E) : (M_2; E) : S)$.
  We anticipate that explicitly modelling disjunction in this way will be useful for proof critics (§8). Clearly it also introduces space inefficiency, since it results in copying the continuation E. This can be addressed by distinguishing two different forms of *orelse_meth*: one which is interpreted explicitly using frames, and a second which is interpreted implicitly using λPrologbacktracking.
- In [4], when a cut is encountered in the body of a procedure call, the frames which represent the alternatives which will be employed if the current frame fails are replaced by the frames which represented the alternatives when the procedure call was initiated.
  Analogously, whenever the body M of a methodical expression *cut_meth* M has been successfully concluded, the frames immediately subsequent to the cut should be replaced by those immediately preceding it. More precisely, $\mu((\mathit{cut\_meth}\, M; E) : S) = \mu((M; !S; E) : S)$ where $\mu((!S; E) : T) = \mu(E) : S$.

8 Applications

The definition of proof strategies using methodical expressions, and the continuation - based proof planner described in this paper have been implemented in λ*Clam*. The planner provides a global view of the state of a partial proof which supports a prototype implementation of proof critics [6]. Describing the critics mechanism is outside the scope of this paper.

A user interface has been developed for λ*Clam*, based on the XBarnacle [8] user interface for *Clam*. The combined system is called xλ*Clam*. The user interface itself is written in Tcl/Tk. It runs as a Unix process, and it communicates with λ*Clam*, which is running in a separate process, via a socket interface. The Tcl/Tk process keeps a copy of the partial proof state, while the proof plan data structure and planner described in this paper enable λ*Clam* to also keep a copy of the proof plan tree. Whenever λ*Clam* makes a change to the proof plan tree, this is communicated to the Tcl/Tk process, and conversely, instructions from the user to modify the proof plan tree are passed from the Tcl/Tk process to the $\lambda Clam$ process. The use of continuations in the proof planner makes the planning process interruptible and places it under control of the user — the user may pause the proof, run the proof planner in non-stop mode, force the proof planner to backtrack at a node, change the current planning node, or load and edit theory files.

9 Further Work

We mention here a few of the many directions for future research:

- We have demonstrated how to define a proof planner for a persistent proof plan data structure in an entirely declarative way. How to optimise this proof planner must be the subject of future work.
- The continuation semantics outlined in Section 7 must be fleshed out. In [4], Prolog programs generate *sequences* of primitive actions, whereas we would prefer to generate *trees* of primitive actions. We should also incorporate failure continuations into our formalism.
- A continuation-based treatment of the complete methodical language, and implementation of *cut_meth* and *orelse_meth* as described in Section 7 will also be the subject of further work.
- The use of continuations suggests extension of the methodical language with constructs (such as exceptions) found in programming languages.

10 Conclusions

In this paper we have outlined how proof planning strategies can be defined using methodicals. We defined a meta-interpreter for methodical expressions which can be naturally interpreted as a proof planner. There is, however, no notion of proof state, so interpretation of methodical expressions cannot be paused.

We presented a set of transformation rules which we used to define a new meta-interpreter which is equivalent to the first (in the sense that it produces the same plans) and which does maintain a notion of proof state, and is suitable for interactive proof. This meta-interpreter has been used for some time as the basis of the interactive planner in $\lambda Clam$.

Finally, we showed how previous work on a continuation semantics for Prolog [4] can provide us with appropriate representations and algorithms for handling *cut_meth* and *orelse_meth*.

11 Acknowledgements

We would like to thank the following for their invaluable comments: Ian Green (who first suggested the use of the term “continuation” in this context), the anonymous referees, members of the *Dream* group, Greg Michaelson, and the participants of the CIAO 2001 workshop. This work was partially funded by EPSRC grant GR/L11724.

References

1. A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.
2. A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.
3. Francisco Cantu, Alan Bundy, Alan Smaill, and David Basin. Experiments in automating hardware verification using inductive proof planning. In M. Srivas and A. Camilleri, editors, *Proceedings of the Formal Methods for Computer-Aided Design Conference*, number 1166 in Lecture Notes in Computer Science, pages 94–108. Springer-Verlag, 1996.
4. Arie de Bruin and Erik P. de Vink. Continuation semantics for PROLOG with cut. In *TAP-SOFT, Vol.1*, pages 178–192, 1989.
5. 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.
6. 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.
7. 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).* 2000.
8. H. Lowe and D. Duncan. XBarnacle: Making theorem provers more accessible. In William McCune, editor, *14th International Conference on Automated Deduction*, pages 404–408. Springer-Verlag, 1997.
9. J. C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation: An International Journal*, 6:233–247, 1993.
10. J. D. C. Richardson, L. A. Dennis, J. Gow, and M. Jackson. User/programmer manual for the λ*Clam* proof planner, v.2.0.0. Technical Paper TBA, Division of Informatics, 2000.
11. J.D.C Richardson, A. Smaill, and Ian Green. System description: proof planning in higher-order logic with λClam. 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.