Automating the synthesis of decision procedures in a constructive metatheory *

Alessandro Armando [a] Jason Gallagher [b] Alan Smaill [b] Alan Bundy [b]

[a] *Dipartimento di Informatica, Sistemistica e Telematica – Università di Genova*
Viale Causa 13, 16145 Genova, Italy
E-mail: armando@dist.unige.it
[b] *Department of Artificial Intelligence – University of Edinburgh*
80 South Bridge – Edinburgh EH1 1HN, Scotland
E-mail: jason@aisb.ed.ac.uk, smaill@aisb.ed.ac.uk, bundy@aisb.ed.ac.uk

We present an approach to the automatic construction of decision procedures, via a detailed example in propositional logic. The approach adapts the methods of proof-planning and the heuristics for induction to a new domain, that of meta-theoretic procedures. This approach starts by providing an alternative characterisation of validity; the proofs of the correctness and completeness of this characterisation, and the existence of a decision procedure, are then amenable to automation in the way we describe. In this paper we identify a set of principled extensions to the heuristics for induction needed to tackle the proof obligations arising in the new problem domain and discuss their integration within the *CIAM-Oyster* system.

1. Introduction

The task of building automated reasoning systems aims for systems with some concise and clear underlying logic, yet supporting powerful inference procedures. Furthermore, we want such procedures to be *correct* so that we do not endanger the system's integrity by their use, and *efficient* so that they can be useful in practice. Finally, we would like to have *automated assistance* in the construction of such procedures, perhaps using the system itself. In this paper, we concentrate on the final aspect, and describe an approach to the automatic construction of decision procedures.

We make use of two logical systems: one is the object logic that we reason about (in this paper, classical propositional logic characterised via truth assignments), and the other is the meta-logic that we reason in (here a constructive type theory). For the latter, we choose a version of Martin-Löf's Type Theory,

* The authors are supported in part by grants EPSRC GR/L/11724, British Council (ROM/889/95/70) and MURST in collaboration with CRUI under the Programme *British-Italian Collaboration in Research and Higher Education*

as developed in [14]. For present purposes, the meta-logic gives us a restricted version of a standard sorted predicate calculus (i.e., an intuitionistic calculus), together with induction principles for inductively defined data-types – e.g., lists, formula trees. The meta-logic is designed in such a way that functional programs can be automatically derived from proofs of statements of an appropriate form; this allows us to regard the synthesis of meta-theoretic procedures as a theorem proving task. If we succeed, we obtain a *verified* procedure for the task at hand, which assures us of its soundness. In this context, finding a decision procedure for a property ϕ of objects of type τ can be achieved by finding a proof in this meta-theory of the statement

$$\forall x : \tau.(\phi(x) \vee \neg\phi(x)) \tag{1}$$

(Note that because of the restrictions to this logic, this statement is *not* trivially true, as it would be in classical logic; indeed if the property ϕ is not decidable, then the statement has no proof in this logic.) The problem then is to find a good way of guiding the search for proofs of statements of this form.

Whether logics are defined by proof systems or semantically, the definition is invariably inductive (cf. Tarski's truth definition), and procedures for reasoning about them are often recursive. So the proofs we need to find in constructive logic involve induction; we follow the promising techniques of proof plans [11], and particularly that of rippling [9] to guide our proof search.

To reduce the overall synthesis task to a set of proof obligations amenable of automated assistance, we adopt the following methodology: *(i) Conjecture an alternative characterisation of validity* (i.e., extend the metatheory with definitions formalising a property P of the formulas of the object language); *(ii) Show that the characterisation is indeed decidable* (i.e., provide a constructive proof of the decidability statement $\forall w.(P(w) \vee \neg P(w))$); *(iii) Extract the program from the decidability proof*; and *(iv) Show that the new characterisation is indeed equivalent to validity.*

The approach is presented by discussing the synthesis of a decision procedure for the biconditional fragment of propositional logic in the *CIAM-Oyster* system. Step *(i)* clearly requires human intervention, and we provide our conjecture manually here. We discuss the automation of points *(ii)* and *(iii)* as carried out in *CIAM-Oyster* and show that a significant degree of automated assistance is achieved. Finally, it is a well known logical result that a program can be extracted from a constructive proof of its specification, and in our implementation, step *(iv)* simply amounts to retrieving this "extract" which is built up automatically by the prover as the proof is constructed.

The contributions of our work are several fold:

- we apply proof-planning and rippling to a new domain, that of meta-theoretic procedures;
- we identify a principled extension to the set of heuristics for induction;

- we advance and discuss solutions to the problem of integrating the proposed extension to the earlier repertoire of heuristics of the *CIAM-Oyster* system;
- we describe in detail the case study so that it can be used as challenge problem for automated theorem provers.

It is worth pointing out that most of the effort in extending the set of available heuristics was confined to the design and implementation of a new proof procedure, and — more importantly — that the proof planning framework allowed us to easily incorporate the new procedure into the system. Moreover the high level of automation achieved in the case study witnesses the strength and generality of the available heuristics for induction.

The paper is organised as follows. Section 2 introduces the proof-planning paradigm. Section 2.1 surveys the repertoire of heuristics for inductive domains available in the *CIAM-Oyster* system; Section 2.2 presents the extension we have incorporated to automate the synthesis of meta-theoretic procedures. Section 3 presents the case study in detail. Section 4 discuss the related work, and finally, in Section 5, we draw some conclusions.

2. Proof planning

The proofs described in this paper have been carried out in *Oyster-CIAM*, a theorem planning and proving system for a higher-order, constructive, typed logic. *Oyster* is a reimplementation of the Nuprl interactive proof editor [14]. *Tactics* are Prolog programs which drive *Oyster* by applying its rules of inference. Tactics raise the level of interaction with the system from the basic inference rules provided by the prover to complex inference steps close to the ones used in informal mathematics. However tactics alone do not provide the degree of flexibility required in practice.

Proof planning was proposed in [11] as a technique for flexibly combining tactics tailored to the goal at hand. A *proof plan* is a tactic together with a *method*. A method is a partial specification of the associated tactic. It consists of five slots (see Figure 1): the *name*, the *input* goal[1], the *preconditions* enabling the application of the tactic, the *effects* of applying the tactic, the *output* sub-goals obtained by a successful application of the tactic to the input goal, and the *tactic* itself specified by the method. The application of a method to a given goal amounts to matching the actual goal with the input slot, checking the preconditions and finally then determining the output sub-goals by computing the effects, i.e. it simulates the application of the associated tactic. If a method admits multiple solutions, the search proceeds on the first one, while the remaining ones are generated and used in the event of a backtracking. Methods can be hierarchically

[1] In this paper we consider goals to be sequents of the form $\Gamma \vdash \phi$, where ϕ is a formula and Γ is a sequence of formulae.

```
method(Name,
       Γ ⊢ φ,                          % Input Slot
       [Pre_1,...,Pre_l],              % Preconditions
       [Post_1,...,Post_m],            % Effects
       [Γ_1 ⊢ φ_1, ..., Γ_n ⊢ φ_n],    % Output Slot
       Tactic)
```

Figure 1. The slots of a method

organised by gluing together simpler methods via constructs, e.g. `THEN`, `ORELSE`, and `REPEAT`, called *methodicals*. Let M_1 and M_1 be methods specifying the tactics T_1 and T_2 respectively. (M_1 `THEN` M_2) specifies the sequential composition of T_1 and T_2; (M_1 `ORELSE` M_2) is equivalent to M_1 if M_1 is successful on the input goal, otherwise it is equivalent to M_2; (`REPEAT` M_1) specifies the exhaustive application of T_1.

Theorem proving in *Oyster-CIAM* is a two-stage activity. First *CIAM* tries to build a proof plan for the goal at hand by using plan formation techniques on the current repertoire of methods. Secondly — if the first step succeeds — the tactic associated with the proof plan is executed engendering the desired *Oyster* proof. It is worth pointing out that a method can promise more than what the corresponding tactic can actually do. In particular a method can be incorrect, i.e., it can deliver subgoals whose provability does not imply the provability of the input goal. However this only means that the planner can build proof-plans (i.e., tactics) whose execution will eventually fail or generate a theorem different from the goal proof-planning was applied to. As in every tactic-based theorem prover, it is impossible for a tactic to engender an unprovable sequent (provided that the basic inference rules are correctly implemented).

Proof-planning amounts to trying the available methods according to a specific search strategy keeping track of the successful methods. The proof described in this paper have been carried out using the standard *depth-first* planner available in *CIAM*.

2.1. Proof-planning in inductive domains

CIAM's methods encode a particularly effective set of heuristics for theorem proving on inductive domains. The `base_case` method (Figure 2) encodes the strategy of exhaustively applying the `equal`, `normalize_term`, and `casesplit` submethods to the input goal, using the `elementary` submethod to get rid of "tautological" goals (i.e., goals provable using propositional and simple equational reasoning only). `equal` looks for equalities in hypotheses of the input goal and uses them to rewrite the goal. `normalize_term` rewrites the input goal by using the available definitions as a terminating term rewriting system. (When *CIAM*

```
base_case ≡ REPEAT (elementary ORELSE sym_eval)
 sym_eval ≡ REPEAT (equal ORELSE
                    normalize_term ORELSE
                    casesplit)
```

Figure 2. The base_case method

parses the available definitions it looks for a *recursive path ordering* [17] guaranteeing the termination of the rewriting process carried out by normalize_term.) A *complementary set* is a set of conditional rewrite rules:

$$\begin{gathered} C_1 \rightarrow L \Rightarrow R_1 \\ \vdots \\ C_n \rightarrow L \Rightarrow R_n \end{gathered}$$

where $C_1, \ldots, C_n$ are the conditions of the rules and $(C_1 \vee \ldots \vee C_n)$ is a theorem of the current theory. (In most cases $n = 2$ and C_2 is the negation of C_1.) Whenever a conditional rewrite in a complementary set is applicable to the input goal $\Gamma \vdash \phi$ (i.e. there exists a substitution σ such that $L\sigma$ occurs in the goal) casesplit performs a case split in the proof specifying $C_1\sigma \vdash \phi$, ..., $C_n\sigma \vdash \phi$ in the output slot. (This approach to case splitting is similar to the one advocated in [5].)

The generalise method replaces all the occurrences of a term in the goal by a new variable (provided that certain heuristic criteria are met). It is not unusual in inductive proofs [6] that more general goals turn out to be easier to prove.

The ind_strat method first selects the most promising form of induction via a process called *ripple analysis* [10], then it applies the base_case and the step_case submethods to the base and step cases respectively. The main activ-

```
step_case ≡ REPEAT (wave ORELSE casesplit)
             THEN fertilise
```

Figure 3. The step_case method

ity of step_case is a rewrite process called *rippling* [9] aiming at enabling the application of the induction hypothesis. To illustrate consider the proof of the associativity of $+$ by structural induction on x. The induction hypothesis and conclusion are

$$x + (y + z) = (x + y) + z \tag{2}$$

$$\boxed{s(\underline{x})}^{\uparrow} + (y + z) = (\boxed{s(\underline{x})}^{\uparrow} + y) + z \tag{3}$$

The annotations on the conclusion mark the difference between the conclusion and the hypothesis; these annotations are automatically introduced and used by *CLAM* to guide proof search. The sub-expressions which appear in the induction conclusion, but not in the induction hypothesis, are called *wave-fronts*. The rest of the induction conclusion, i.e. the expression occurring also in the induction hypothesis, is called *skeleton*. As illustrated in Figure 3, the `step_case` method amounts to a repeated application of the `wave` and `casesplit` submethods followed by an invocation to the `fertilise` submethod. The `wave` submethod aims at moving the wave-fronts outwards in the expression tree by using a set of *wave-rules*. Wave-rules are rewrite rules on annotated terms which preserve the skeleton term structure and are guaranteed to make progress towards applying the induction hypothesis. Wave-rules are built automatically by *CLAM* from recursive definitions and from lemmas. For instance, the wave-rule generated from the definition of + is:

$$\boxed{s(\underline{x})}^{\uparrow} + y \Rightarrow \boxed{s(\underline{x + y})}^{\uparrow} \tag{4}$$

Repeated application of wave-rule (4) rewrites (3) as follows:

$$\begin{aligned} \boxed{s(\underline{x + (y + z)})}^{\uparrow} &= \boxed{s(\underline{x + y})}^{\uparrow} + z \\ \boxed{s(\underline{x + (y + z)})}^{\uparrow} &= \boxed{s(\underline{(x + y) + z})}^{\uparrow} \end{aligned} \tag{5}$$

If a wave-front is moved to dominate the left- or right-hand term of the induction conclusion then we say the induction conclusion is *beached*. In equation (5) both wave-fronts are beached. If a wave-front is not beached, but no wave-rule applies to it, then we say the wave-front is *blocked*. For more details, consult [9].

`fertilise` is applied when no further rippling is possible. At this stage in fact it is often possible to use the induction hypothesis to rewrite the conclusion into an identity or a simplified formula. In our example, rewriting (5) by means of (2) we obtain an identity thereby finishing the proof.

Proof-planning has been proved successful on a large set of (mainly equationally presented) verification problems on inductive domains [13]. However the task of synthesising meta-theoretic procedures poses new problems and paves the way for challenge extensions.

2.2. Propositional and quantificational reasoning

Since both propositional connectives and quantifiers are systematically used to specify proof procedures (cf. Section 3 and [15]), strategies for propositional

AX	$\dfrac{}{\Gamma \vdash_{\mathrm{AX}} \gamma}\ (ax)$ if $\gamma \in \Gamma$ or $\bot \in \Gamma$	$\dfrac{\Gamma \vdash_{\mathrm{AX}} \gamma}{\Gamma \vdash \gamma}\ (ax\text{-}hyp)$
$\wedge$	$\dfrac{\Gamma, \phi, \psi \vdash \gamma}{\Gamma, \phi \wedge \psi \vdash \gamma}\ (\wedge{:}\mathit{left})$	$\dfrac{\Gamma \vdash \phi \quad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi}\ (\wedge{:}\mathit{right})$
$\vee$	$\dfrac{\Gamma, \phi \vdash \gamma \quad \Gamma, \psi \vdash \gamma}{\Gamma, \phi \vee \psi \vdash \gamma}\ (\vee{:}\mathit{left})$	$\dfrac{\Gamma \vdash \phi}{\Gamma \vdash \phi \vee \psi}\ (\vee{:}\mathit{right}_l)$ $\dfrac{\Gamma \vdash \psi}{\Gamma \vdash \phi \vee \psi}\ (\vee{:}\mathit{right}_r)$
$\rightarrow$	$\dfrac{\Gamma \vdash_{\mathrm{AX}} \phi \quad \Gamma, \psi \vdash \gamma}{\Gamma, \phi \rightarrow \psi \vdash \gamma}\ (\rightarrow{:}\mathit{left}_1)$ $\dfrac{\Gamma, \phi_1 \rightarrow (\phi_2 \rightarrow \psi) \vdash \gamma}{\Gamma, (\phi_1 \wedge \phi_2) \rightarrow \psi \vdash \gamma}\ (\rightarrow{:}\mathit{left}_2)$ $\dfrac{\Gamma, \phi_1 \rightarrow \psi \vdash \gamma \quad \Gamma, \phi_2 \rightarrow \psi \vdash \gamma}{\Gamma, (\phi_1 \vee \phi_2) \rightarrow \psi \vdash \gamma}\ (\rightarrow{:}\mathit{left}_3)$ $\dfrac{\Gamma, \psi \vdash \gamma \quad \Gamma, \phi_2 \rightarrow \psi \vdash \phi_1 \rightarrow \phi_2}{\Gamma, (\phi_1 \rightarrow \phi_2) \rightarrow \psi \vdash \gamma}\ (\rightarrow{:}\mathit{left}_4)$	$\dfrac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \rightarrow \psi}\ (\rightarrow{:}\mathit{right})$

Figure 4. `fol` method: inference rules for propositional reasoning

and quantificational reasoning are to be added to the repertoire of methods presented in Section 2.1 in order to automate the sort of reasoning involved in the synthesis of meta-theoretic procedures. To this end we have designed and implemented a new method, `fol`, encoding a complete procedure for intuitionistic propositional logic (borrowed from [18]) augmented with inference rules for quantificational and equational reasoning.

The propositional component of the `fol` method amounts to a backward application of the inference rules in Figure 4 using a depth-first strategy. With the exception of (ax) and $(ax\text{-}hyp)$, the rules in the left (right) column manipulate formulae of the hypothesis list (conclusion) of the sequent they apply to and we refer collectively to them as "left rules" ("right rules" resp.). While the application of right rules is determined by the main connective of the conclusion, in order to apply a left rule we have to choose the hypothesis which the rule has to be applied to. This introduces a first form of indeterminism. A further form of indeterminism is given by $(\vee{:}\mathit{right}_l)$ and $(\vee{:}\mathit{right}_r)$ as both of them can be applied to any sequent having a disjunction as conclusion. Therefore, with the only exception of ($\vee$:$right_l$) and ($\vee$:$right_r$), right rules are given precedence w.r.t. left rules. This has the effect of delaying the indeterminism introduced by the left rules.

Reasoning about equality is carried out by the rules in Figure 5. (*sym_eval*) reduces the input goal to a set of subgoals by invoking the `sym_eval` method introduced in Section 2.1. Notice that the sequent in the conclusion of (*ax*) and (*refl*), and in the premises of (*ax-hyp*) and ($\rightarrow$ *:left*$_1$) is subscripted by "AX"; this means that only (*ax*) and (*refl*) will be used to relieve the left premise of ($\rightarrow$ *:left*$_1$) and the premise of (*ax-hyp*).

$=$	$\dfrac{}{\Gamma \vdash_{\mathrm{AX}} t = t}$ (*refl*)	$\dfrac{\Gamma_1 \vdash \gamma_1 \cdots \Gamma_n \vdash \gamma_n}{\Gamma \vdash \gamma}$ (*sym_eval*)*

* $\Gamma_1 \vdash \gamma_1, \ldots, \Gamma_n \vdash \gamma_n$ ($n \geq 1$) are the result of applying the `sym_eval` method to $\Gamma \vdash \gamma$.

Figure 5. `fol` method: equality rules

Quantificational reasoning is carried out by backward application of the rules in Figure 6. ($\forall$:*left*) and ($\exists$:*right*) introduce in the premise the matrix of

$\forall$	$\dfrac{\Gamma, \phi[X/x] \vdash \psi}{\Gamma, \forall x.\phi \vdash \psi}$ ($\forall$:*left*)	$\dfrac{\Gamma \vdash \phi[a/x]}{\Gamma \vdash \forall x.\phi}$ ($\forall$:*right*)*
$\exists$	$\dfrac{\Gamma, \phi[a/x] \vdash \psi}{\Gamma, \exists x.\phi \vdash \psi}$ ($\exists$:*left*)*	$\dfrac{\Gamma \vdash \phi[X/x]}{\Gamma \vdash \exists x.\phi}$ ($\exists$:*right*)

***Proviso:** a must not appear in the conclusion.

Figure 6. `fol` method: inference rules for quantification

the selected quantified formula with a fresh metavariable, X, substituted for the quantified variable, x. Metavariables play the role of unspecified terms which are to be determined subsequently. The metavariables introduced by the application of ($\forall$:*left*) and ($\exists$:*right*) must be properly dealt and instantiated by other methods, e.g. `sym_eval`, or by other inference rules, e.g. (*refl*). Notice that the instantiations of the metavariables must be propagated on the different branches of the proof tree in a consistent way. For instance, let us consider the following proof attempt:

$$\dfrac{\dfrac{\dfrac{}{P(X) \vdash P(a)}\ (ax) \text{ if } X = a}{P(X) \vdash P(a) \vee R(b)}\ (\vee{:}right_l) \quad \dfrac{\dfrac{}{R(X) \vdash R(b)}\ (ax) \text{ if } X = b}{R(X) \vdash P(a) \vee R(b)}\ (\vee{:}right_r)}{\dfrac{P(X) \vee R(X) \vdash P(a) \vee R(b)}{\forall x.(P(x) \vee R(x)) \vdash P(a) \vee R(b)}\ (\forall{:}left)}\ (\vee{:}left)$$

Clearly the instantiations generated by the two branches of the derivation are inconsistent since they require X to be a and b at the same time. The fact that CI^AM is implemented in Prolog simplifies the implementation of schematic reasoning. Metavariables are readily implemented by means of Prolog variables. The nice effect of this solution is that the instantiations found in a branch of the proof are automatically propagated throughout the derivation.

However the introduction of metavariables complicates the checking of the proviso of the ($\forall$:*left*) and ($\exists$:*right*) rule. Since a metavariable stands for an unspecified term which will be determined after the application of such rules, it is necessary to delay the checking of the proviso to a later stage of the proof (i.e., when the metavariables occurring in the conclusion become instantiated). For instance, consider the following proof attempt (borrowed from [23]):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{R(X,X) \vdash R(a,Y)}\ (ax) \text{ if } X = a \text{ and } Y = a}{\forall x.R(x,x) \vdash R(a,Y)}\ (\forall{:}left)}{\forall x.R(x,x) \vdash \forall z.R(z,Y)}\ (\forall{:}right)}{\forall x.R(x,x) \vdash \exists y.\forall z.R(z,y)}\ (\exists{:}right)$$

If we apply the instantiation suggested by (ax) to the derivation, we discover that the application of ($\forall$:*right*) violates the proviso. To cope with this problem we adopted the solution proposed in [23]: when applying ($\forall$:*right*) and ($\exists$:*left*) the `fol` method uses a term $f(X_1, \ldots, X_n)$ (where $X_1, \ldots, X_n$ are the metavariables occurring in the input goal and f a fresh function symbol) in place of the fresh individual constant a. In this way we are guaranteed that no metavariable in the sequent (i.e. $X_1, \ldots, X_n$) can get subsequently instantiated to $f(X_1, \ldots, X_n)$ (provided that proper unification — i.e. with occur check — is used to carry out the instantiation of metavariables).

A simple form of propositional and quantificational reasoning is also available in another method called `normalize`. `normalize` corresponds to the process of exhaustively applying the ($\forall$:*right*), ($\rightarrow$:*right*), and ($\wedge$:*left*) rules to the input goal and then returning the resulting subgoal. This method makes explicit facts which are otherwise embedded in the propositional structure of the goal and therefore not available to relieve conditions of conditional rewrites. As a consequence `normalize` has the beneficial effect of extending the set of conditional rewrite rules applicable to the goal.

3. The case study

In our case study, we look at the problem of the validity of formulas of propositional logic that only use the connective "$\Leftrightarrow$". The notion of validity is standard: a formula w is valid if and only if it is true with respect to all possible truth-value assignments. Truth-values are represented by the individual constants T and F. Assignments are represented by lists of propositional letters by adopting the convention that a list of propositional letters a represents an assignment α if and only if $\alpha(s) = T$ whenever s is not in a. For the sake of simplicity we drop the type information from the formulae and adopt the convention that p ranges over propositional letters, w over biconditional formulas, a over truth-value assignments, pl over lists of propositional letters and n, m over natural numbers, with subscripts where needed.

The fact that p occurs in w is expressed in the metatheory by $p \in w$ and is formalised as follows (here and in the sequel instead of the definitions we directly present the wave/rewrite-rules as generated by $C\mathit{L}^{\!A}\!M$):

$$p \in p' \Rightarrow p = p' \tag{6}$$

$$p \in \boxed{\underline{w_1} \Leftrightarrow \underline{w_2}}^{\uparrow} \Rightarrow \boxed{\underline{p \in w_1} \vee \underline{p \in w_2}}^{\uparrow} \tag{7}$$

3.1. An alternative characterisation of validity

The biconditional fragment of propositional logic enjoys a very simple and effective decision procedure: *a formula is valid if and only if each propositional letter occurs an even number of times in it.* This can be formally stated as follows:

Theorem 1.

$$\forall w.(\forall p.even(occ(p, w)) \leftrightarrow \forall a.eval(w, a) = T)$$

where $occ(p, w)$ denotes the number of occurrences of p in w; $even(n)$ is true if and only if n is an even natural number and $eval(w, a)$ denotes the truth value of w under a. The rules for $eval$ are:

$$eval(p, nil) \Rightarrow T \tag{8}$$

$$p = p' \rightarrow eval(p, p' :: a) \Rightarrow F \tag{9}$$

$$p \neq p' \rightarrow eval(p, \boxed{p' :: \underline{a}}^{\uparrow}) \Rightarrow eval(p, a) \tag{10}$$

$$eval(\boxed{\underline{w_1} \Leftrightarrow \underline{w_2}}^{\uparrow}, a) \Rightarrow \boxed{equal(\underline{eval(w_1, a)}, \underline{eval(w_2, a)})}^{\uparrow} \tag{11}$$

where $equal(b_1, b_2)$ is a boolean-valued function expressing the equality of its boolean arguments. $\forall p.even(occ(p, w))$ expresses the counting argument above, which we wish to use to establish the validity of w, while $\forall a.eval(w,a) = T$ encodes the usual notion of (classical) propositional validity for w. The rules for *occ* are:

$$p \neq p' \rightarrow occ(p,p') \Rightarrow 0 \tag{12}$$

$$p = p' \rightarrow occ(p,p') \Rightarrow s(0) \tag{13}$$

$$occ(p, \boxed{\underline{w_1} \Leftrightarrow \underline{w_2}}^{\uparrow}) \Rightarrow \boxed{\underline{occ(p,w_1)} + \underline{occ(p,w_2)}}^{\uparrow} \tag{14}$$

In the following sections we show how the proof obligations arising in our case study are successfully (i.e., automatically) theorem planned and proved by the version of *CIAM-Oyster* extended with the `fol` method as illustrated in Section 2.

3.2. Proving the decidability statement

The decidability statement is $\vdash \forall w.\Delta(\forall p.even(occ(p,w)))$, where $\Delta(\alpha)$ abbreviates the formula $(\neg\alpha \vee \alpha)$, and $\neg\alpha$ abbreviates $(\alpha \rightarrow \bot)$.

Thus we are supposing an *even* function given via its recursive definition, and here synthesis should supply an algorithm for checking for any given formula w whether every sentential symbol in w occurs an even number of times or not. It is important to notice that the decidability statement simply specifies such an algorithm and it does not provide in itself any clue on how to carry out such a computation.

Notice that the inner universal quantification over the sentential symbols is – de facto – bounded to the propositional letters occurring in the formula (since those not occurring occur an even number of times, i.e. 0). Since this pattern of reasoning is common in the synthesis of metatheoretic procedures[2] we focused on the more general result of proving the decidability of $\forall p.\eta(p,w)$ for a generic decidable statement $\eta(p,w)$ under the assumption that $\forall p.(p \notin w \rightarrow \eta(p,w))$, more formally:

Lemma 2.

$$\forall \eta.\ [(\forall p.\forall w.\Delta(\eta(p,w)) \wedge \forall w.\forall p.(p \notin w \rightarrow \eta(p,w))) \rightarrow \forall w.\Delta(\forall p.\eta(p,w))]$$

The following lemma is the key step of the proof of Lemma 2:

Lemma 3.

$$\forall \eta.\ [\forall p.\forall w.\Delta(\eta(p,w)) \rightarrow \forall w.\forall w'.\Delta(\forall p.(p \in w \rightarrow \eta(p,w')))]$$

[2] For instance, the *Affirmative-Negative Rule* of Davis-Putnam procedure [16] states: *"If an atomic formula p occurs in a formula F only affirmatively, or if p occurs only negatively, then all clauses which contain p can be deleted"*. We can use the previous argument to restrict the universal quantification over p to the atomic formulas occurring in the clauses at hand.

Proof of Lemma 2. It suffices to show that $\forall w.\Delta(\forall p.\eta(p,w))$ under the assumptions $\forall p.\forall w.\Delta(\eta(p,w))$ and $\forall w.\forall p.(p \notin w \rightarrow \eta(p,w))$ for a generic binary predicate η of suitable sort. From the first assumption and Lemma 3 it follows $\forall w.\forall w'.\Delta(\forall p.(p \in w \rightarrow \eta(p,w')))$ which can be specialised to $\forall w.\Delta(\forall p.(p \in w \rightarrow \eta(p,w)))$. From this last fact and our second assumption it is easy to conclude $\forall w.\Delta(\forall p.(\Delta(p \in w) \rightarrow \eta(p,w)))$ which can be finally simplified to $\forall w.\Delta(\eta(p,w))$ thanks to the decidability of the "$\in$" relation.

Proof of Lemma 3. The `normalize` method is tried with the effect of applying $(\forall : right)$ and $(\rightarrow: right)$:

$$\forall p.\forall w.\Delta(\eta(p,w)) \vdash \forall w.\forall w'.\Delta(\forall p.(p \in w \rightarrow \eta(p,w')))$$

Structural induction on w is then applied by `ind_strat`. The `base_case` method simplifies (via equation (6)) the base case to:

$$\forall p.\forall w.\Delta(\eta(p,w)) \vdash \forall w'.\Delta(\forall p.(p = p_0 \rightarrow \eta(p,w')))$$

The `fol` method applies $(\forall : right)$, $(\forall : left)$ and $(\vee : left)$ and yields:

$$\eta(P,W) \vdash \Delta(\forall p.(p = p_0 \rightarrow \eta(p,w'))) \tag{15}$$

$$\neg\eta(P,W) \vdash \Delta(\forall p.(p = p_0 \rightarrow \eta(p,w'))) \tag{16}$$

where, for the sake of brevity, only the relevant hypotheses are displayed. Subgoal (15) is readily proven by applying ($\vee$*:right*), ($\forall$*:right*), ($\rightarrow$ *:right*), and (ax) whose effect is also to instantiate P and W to p_0 and w' respectively. More difficult is the proof of (16), which by effect of the propagation of the instantiation is turned into:

$$\neg\eta(p_0,w') \vdash \Delta(\forall p.(p = p_0 \rightarrow \eta(p,w'))) \tag{17}$$

which the `fol` method solves as follows:

$$\begin{aligned}
\neg\eta(p_0,w') \vdash \neg\forall p.(p = p_0 \rightarrow \eta(p,w')) &\text{ by } (\vee\text{:}\mathit{right}_l)\\
\neg\eta(p_0,w'), \forall p.(p = p_0 \rightarrow \eta(p,w')) \vdash \bot &\text{ by } (\rightarrow \text{:}\mathit{right})\\
\neg\eta(p_0,w'), P = p_0 \rightarrow \eta(P,w') \vdash \bot &\text{ by } (\forall\text{:}\mathit{left})\\
\neg\eta(p_0,w'), \eta(p_0,w') \vdash \bot &\text{ by } (\forall\text{:}\mathit{left}) \text{ and } P = p_0\\
\bot, \eta(p_0,w') \vdash \bot &\text{ by } (\forall\text{:}\mathit{left})
\end{aligned}$$

which is readily solved by (ax).

In the step case the goal is

$$\forall w'.\Delta(\forall p.(p \in \boxed{\underline{w_1} \Leftrightarrow \underline{w_2}}^{\uparrow} \rightarrow \eta(p,w')))$$

$$
\begin{array}{l}
dec2(w, w') \equiv \textbf{if } \ atomic(w) \\
\quad \textbf{then if } occurs(w, w') \\
\quad\quad \textbf{then } \pi(w, w') \\
\quad\quad \textbf{else } \mathit{false} \\
\quad \textbf{else } \ \textbf{let } (w_1 \Leftrightarrow w_2) = w \textbf{ in} \\
\quad\quad\quad \textbf{if } dec2(w_1, w') \\
\quad\quad\quad \textbf{then } dec2(w_2, w') \\
\quad\quad\quad \textbf{else } \mathit{false}
\end{array}
$$

Figure 7. Decision Sub-Procedure

Rippling by means of (7) we get

$$\forall w'.\Delta(\forall p.(\boxed{\underline{p \in w_1} \vee \underline{p \in w_2}}^{\uparrow} \rightarrow \eta(p, w')))$$

and further rippling using the following (domain-independent) wave-rules

$$\boxed{\underline{A} \vee \underline{B}}^{\uparrow} \rightarrow C \Rightarrow \boxed{\underline{A \rightarrow C} \wedge \underline{B \rightarrow C}}^{\uparrow}$$

$$\forall x.(\boxed{\underline{A} \wedge \underline{B}}^{\uparrow}) \Rightarrow \boxed{\underline{\forall x.A} \wedge \underline{\forall x.B}}^{\uparrow}$$

$$\Delta(\boxed{\underline{A} \wedge \underline{B}}^{\uparrow}) \Rightarrow \boxed{\underline{\Delta(A)} \wedge \underline{\Delta(B)}}^{\uparrow}$$

yields:

$$\boxed{\underline{\forall w'.\Delta(\forall p.(p \in w_1 \rightarrow \eta(p, w')))} \wedge \underline{\forall w'.\Delta(\forall p.(p \in w_2 \rightarrow \eta(p, w')))}}^{\uparrow}$$

which is easily solved by fertilisation.

The proof sketched above is automatically found by $C\!L\!AM$ and witnesses the successful integration of the `fol` method with the methods encoding the heuristics for induction. The extract of the proof is the functional program $dec(w) \equiv dec2(w, w)$, where $dec2$ is defined in Figure 7 and $\pi(p, w)$ is a program (extracted from the decidability proof of $\eta(p, w)$) determining whether $\eta(p, w)$ holds.

3.3. Proving the completeness

The completeness statement can be proven via reductio ad absurdum by deriving a contradiction from the assumptions that a generic formula w is both valid and contains a propositional letter p occurring an odd number of times.[3]

[3] The constructive validity of the argument by reductio ad absurdum is here justified by the decidability result proven in Section 3.2.

Under such hypotheses, the following lemma allows us to conclude that $flip(p,a)$ is a falsifying assignment for w. This is clearly in contradiction with the assumed validity of w, and from this we can infer the completeness result.

Lemma 4.

$$\forall p.\forall w.\forall a.(eval(w, flip(p,a)) = nflip(occ(p,w), eval(w,a)))$$

The lemma says that the truth-value of w under the assignment obtained from a by flipping (i.e. changing) the value associated to p (i.e. $flip(p,a)$) is equal to the result of flipping the truth-value of w under a a number of times equal to the number of occurrences of p in w (i.e., $nflip(occ(p,w), eval(w,a))$). The definition of $nflip$ is:

$$nflip(0,b) \Rightarrow b \tag{18}$$

$$nflip(\boxed{s(\underline{n})}^{\uparrow}, b) \Rightarrow \boxed{not(\underline{nflip(n,b)})}^{\uparrow} \tag{19}$$

while $flip$ is defined as follows:

$$flip(p, nil) \Rightarrow p :: nil \tag{20}$$

$$p = p' \rightarrow flip(p, p' :: a) \Rightarrow delete(p,a) \tag{21}$$

$$p \neq p' \rightarrow flip(p, \boxed{p' :: \underline{a}}^{\uparrow}) \Rightarrow \boxed{p' :: \underline{flip(p,a)}}^{\uparrow} \tag{22}$$

where $delete(p,a)$ denotes the result of removing all the occurrences of p from a.

Proof of Lemma 4. Ripple analysis suggests structural induction on w. The base case is

$$\forall a.(eval(p', flip(p,a)) = nflip(occ(p,p'), eval(p',a)))$$

and structural induction on a yields as base and step case

$$eval(p', flip(p, nil)) = nflip(occ(p,p'), eval(p', nil)) \tag{23}$$

and

$$eval(p', flip(p, \boxed{p'' :: \underline{a}}^{\uparrow})) = nflip(occ(p,p'), eval(p', \boxed{p'' :: \underline{a}}^{\uparrow})) \tag{24}$$

respectively. By means of equations (20) and (8) the base case (23) is simplified to:

$$eval(p', p :: nil) = nflip(occ(p,p'), T) \tag{25}$$

Since the left hand sides of (12) and (13) match with the subexpression $occ(p,p')$ of (25), the `casesplit` method suggests to case split over $p = p'$. The resulting cases are then readily solved via symbolic evaluation. The step case (24) is solved by means of rippling, case splitting and two further inductive subproofs.

Going back to the step case of the main goal, the induction conclusion is:

$$eval(\boxed{\underline{w_1} \Leftrightarrow \underline{w_2}}^{\uparrow}, flip(p,a)) = nflip(occ(p, \boxed{\underline{w_1} \Leftrightarrow \underline{w_2}}^{\uparrow}), eval(\boxed{\underline{w_1} \Leftrightarrow \underline{w_2}}^{\uparrow}, a))$$

Application of (11) and (14) yields:

$$\boxed{equal(\underline{eval(w_1, flip(p,a))}, \underline{eval(w_2, flip(p,a))})}^{\uparrow}$$
$$=$$
$$nflip(\boxed{\underline{occ(p,w_1)} + \underline{occ(p,w_2)}}^{\uparrow}, \boxed{equal(\underline{eval(w_1,a)}, \underline{eval(w_2,a)})}^{\uparrow})$$

At this stage no further rippling is possible. The wave front of the left hand side is beached, while both wave fronts in the right hand side are blocked. Fertilisation is however applicable, yielding:

$$\boxed{equal(\underline{nflip(occ(p,w_1), eval(w_1,a))}, \underline{nflip(occ(p,w_2), eval(w_2,a))})}^{\uparrow}$$
$$=$$
$$nflip(\boxed{\underline{occ(p,w_1)} + \underline{occ(p,w_2)}}^{\uparrow}, \boxed{equal(\underline{eval(w_1,a)}, \underline{eval(w_2,a)})}^{\uparrow})$$

A repeated application of the `generalise` method replaces each occurrence of $occ(p,w_1)$, $occ(p,w_2)$, $eval(w_1,a)$ and $eval(w_2,a)$ with new variables m_1, m_2, b_1 and b_2 respectively:

$$equal(nflip(m_1,b_1), nflip(m_2,b_2)) = nflip(m_1+m_2, equal(b_1,b_2)) \qquad (26)$$

The generalisation has the beneficial effect of enabling a further application of the induction method. The induction suggested by ripple analysis is over m_1. The `base_case` method simplifies the base case to:

$$equal(b_1, nflip(m_2,b_2)) = nflip(m_2, equal(b_1,b_2))$$

The induction method is applicable once more and induction over m_2 is attempted. The base case is again proven by `base_case`, so we focus on the step case. The induction conclusion is:

$$equal(b_1, nflip(\boxed{s(\underline{m_2})}^{\uparrow}, b_2)) = nflip(\boxed{s(\underline{m_2})}^{\uparrow}, equal(b_1,b_2))$$

By rippling via (19) and the following wave rule

$$equal(\boxed{not(\underline{b_1})}^{\uparrow}, b_2) \Rightarrow \boxed{not(\underline{equal(b_1,b_2)})}^{\uparrow}$$

we get the fully rippled goal:

$$\boxed{not(\underline{equal(b_1, nflip(m_2, b_2))})}^{\uparrow} = \boxed{not(\underline{nflip(m_2, equal(b_1, b_2))})}^{\uparrow}$$

which is easily solved by `fertilise`.

The induction conclusion in the step case of the proof of (26) is:

$$equal(nflip(\boxed{s(\underline{m_1})}^{\uparrow}, b_1), nflip(m_2, b_2)) = nflip(\boxed{s(\underline{m_1})}^{\uparrow} + m_2, equal(b_1, b_2))$$

Rippling by means of rules (19) and (4) yields:

$$equal(\boxed{not(\underline{nflip(m_1, b_1)})}^{\uparrow}, nflip(m_2, b_2)) = nflip(\boxed{s(\underline{m_1 + m_2})}^{\uparrow}, equal(b_1, b_2))$$

and further rippling by rules (27) and (19) yields the fully rippled goal

$$\boxed{not(\underline{equal(nflip(m_1, b_1), nflip(m_2, b_2))})}^{\uparrow} = \boxed{not(\underline{nflip(m_1 + m_2, equal(b_1, b_2))})}^{\uparrow}$$

which is also easily solved via fertilisation.

Despite the complexity of the proof (6 inductive subproofs, 4 generalisations are needed), no search is carried by the planner, i.e., no backtracking occurs. This fact witnesses the strength of the heuristics for induction (rippling in particular) available in *CLAM-Oyster*.

3.4. Proving the correctness

The proof of the correctness statement requires two lemmas (Lemma 5 and Lemma 6).

Lemma 5.

$$\forall w. \forall a. (eval(w, a) = eval(w, flipl(a, nil)))$$

where $flipl(pl, a)$ denotes the assignment obtained from a by changing the truth-values associated with the propositional letters in pl and nil denotes the empty list and hence represents the assignment associating T to every propositional letter. The lemma says that the assignment obtained from nil by flipping the truth values associated to the propositional letters in a is *equivalent* to a itself. Here follow the rules for $flipl$:

$$flipl(nil, a) \Rightarrow a \tag{27}$$

$$\neg member(p, pl) \rightarrow flipl(\boxed{p :: \underline{pl}}^{\uparrow}, a) \Rightarrow \boxed{flip(p, \underline{flipl(pl, a)})}^{\uparrow} \tag{28}$$

$$member(p, pl) \rightarrow flipl(\boxed{p :: \underline{pl}}^{\uparrow}, a) \Rightarrow flipl(pl, a)) \tag{29}$$

Proof of Lemma 5. Ripple analysis suggests structural induction on w. Since the step case is easily solved by the `step_case` method, here we focus on the base case:

$$\forall a.(eval(p, a) = eval(p, flipl(a, nil)))$$

A further induction on a yields the following two subgoals:

$$eval(p, nil) = eval(p, flipl(nil, nil)) \tag{30}$$

$$eval(p, \boxed{p' :: \underline{a}}^{\uparrow}) = eval(p, flipl(\boxed{p' :: \underline{a}}^{\uparrow}, nil)) \tag{31}$$

(30) is readily solved by `base_case`. (31) is proved by `step_case` in the following way. `casesplit` carries out a double case split over $member(p', a)$ and $p = p'$ as suggested by equations (28), (29), (9), and (10). This leaves us with four subgoals. In the cases $\{p \neq p', member(p', a)\}$ and $\{p \neq p', \neg member(p', a)\}$ the goal is first rippled by a repeated application of `wave` and then solved by `fertilise`. More interesting are the remaining cases $\{p = p', member(p', a)\}$ and $\{p = p', \neg member(p', a)\}$. In the first case, the `wave` submethod rewrites the goal to:

$$eval(p, \boxed{p' :: \underline{a}}^{\uparrow}) = eval(p, flipl(a, nil))$$

`fertilise` uses the induction hypothesis to rewrite the conclusion into:

$$eval(p, p' :: a) = eval(p, a)$$

The `base_case` method applies (9) and simplifies the goal to:

$$F = eval(p, a)$$

The `ind_strat` method "lifts" the hypothesis $member(p, a)$ into the conclusion and then proves the resulting goal by structural induction on a. [4]

In the other case, i.e. when $p = p'$ and $\neg member(p', a)$, `wave` first applies (29):

$$eval(p, \boxed{p' :: \underline{a}}^{\uparrow}) = eval(p, \boxed{flip(p', \underline{flipl(a, nil)})}^{\uparrow})$$

[4] The capability of carrying out ripple analysis on equivalent goals obtained by *lifting* into the conclusion heuristically chosen hypotheses was a missing feature of rippling analysis. Its introduction has been motivated by the present case study.

and then uses Lemma 4 as the following wave-rule:

$$eval(w, \boxed{flip(p, \underline{a})}^{\uparrow}) \Rightarrow \boxed{nflip(occ(p, w), \underline{eval(w, a)})}^{\uparrow} \tag{32}$$

to rewrite the goal to:

$$eval(p, \boxed{p' :: \underline{a}}^{\uparrow}) = \boxed{nflip(occ(p, p'), \underline{eval(p, flipl(a, nil))})}^{\uparrow}$$

`fertilise` then turns the goal into:

$$eval(p, p' :: a) = nflip(occ(p, p'), eval(p, a))$$

The goal is then simplified by `base_case` which, by applying (9), (13), (19), and (18) yields:

$$F = non(eval(p, a))$$

Similarly to the previous case, `ind_strat` first lifts the hypothesis $\neg member(p, a)$ into the conclusion and then proves the resulting goal by structural induction on a.

Lemma 6.

$$\forall w.(\forall p.even(occ(p, w)) \rightarrow \forall pl.\forall a.eval(w, a) = eval(w, flipl(pl, a)))$$

This lemma says that if w is a formula in which all the propositional letters occur an even number of times then the truth value of w w.r.t. a given assignment a is equal to the truth value of w w.r.t. the assignment obtained from a by flipping the truth values associated to the letters in any given list pl.

Proof of lemma 6. The `normalize` method applies ($\forall$: $right$) and ($\rightarrow$: $right$) yielding:

$$\forall p.even(occ(p, w)) \vdash \forall pl.\forall a.eval(w, a) = eval(w, flipl(pl, a))$$

Ripple analysis then suggests structural induction on pl. The base case is

$$eval(w, a) = eval(w, flipl(nil, a))$$

which is readily solved by rewriting with (27). The step case is

$$eval(w, a) = eval(w, flipl(\boxed{p :: \underline{pl}}^{\uparrow}, a))$$

Equations (28) and (29) suggest to case split over $member(p, pl)$. If $member(p, pl)$, then we can apply (29), obtaining:

$$eval(w, a) = eval(w, flipl(pl, a))$$

which is solved by fertilisation. If $\neg member(p, pl)$, then we can apply (28):

$$\forall p.even(occ(p, w)) \vdash eval(w, a) = eval(w, \boxed{flip(p, \underline{flipl(pl, a)})}^{\uparrow})$$

and further rippling by means of (32) yields:

$$\forall p.even(occ(p, w)) \vdash eval(w, a) = \boxed{nflip(occ(p, w), \underline{eval(w, flipl(pl, a))})}^{\uparrow}$$

The `fertilise` submethod then applies and yields

$$\forall p.even(occ(p, w)) \vdash eval(w, a) = nflip(occ(p, w), eval(w, a))$$

The `base_case` method then applies the following rewrite:

$$even(n) \rightarrow nflip(n, b) \Rightarrow b$$

thereby turning the goal into an identity.

Given the above lemmas, the proof of correctness proceeds as follows. Let w be a biconditional formula such that $\forall p.even(occ(p, w))$. We must prove that $eval(w, a) = T$ for a generic assignment a. By Lemma 5 $eval(w, a)$ is equal to $eval(w, flipl(a, nil))$. Since every propositional letter occurs an even number of times in w, by Lemma 6 $eval(w, flipl(a, nil))$ is equal to $eval(w, nil)$ and (since nil enjoys the property of satisfying every formula of the fragment) we can conclude $eval(w, a) = T$.

Again, the proof-plans for Lemma 5 and Lemma 6 outlined above are automatically found by *CIAM*. The automated assistance provided by *Clam-Oyster* in this case study (compared with other state-of-the-art theorem provers for induction) is significant. For instance, NQTHM [8] requires several auxiliary lemmas (and hence considerably more user intervention) to prove the same facts.

4. Related work

Little attention has been devoted to the automation of metatheoretic reasoning though it has been recognised as a main obstacle in the way of metatheoretic extensibility. To this extent, the work most closely related to ours is [7]. However Boyer & Moore's system [6,8] allows only the verification of user-defined procedures – even the simple algorithm synthesised here would have to be supplied explicitly before it could be reasoned about. This is because their use of classical quantifier-free logic prevents the specification and hence the automation of the synthesis of theorem proving procedures, whereas this is the main objective of our work.

[15] shows how the adoption of a constructive metatheory allows one to regard theorem proving algorithms as implementations of theorems about proofs

and formulas. The paper gives evidence of the practical viability of the proposal by presenting the synthesis of a tableau decision procedure for a propositional calculus and a matching algorithm. Our work is to be understood in the setting of Constable and Howe's proposal. However our emphasis is on the automation of the synthesis process, which is essential to the success of the programme.

Both [2] and [22] advocate the use of an explicit metatheory as a framework for theorem proving architectures. The former proposes a metatheory built out of a constructive type theory augmented with higher-order abstract data types. The latter describes an experiment in using FS_0 [19] as a theoretical framework for encoding and reasoning about logics. But again, the automation of the metatheoretic reasoning is not addressed.

The use of a declarative metatheory capable of expressing and reasoning about proof procedures is proposed in [20,1]. The emphasis is however on establishing a tight correspondence between the metatheory and the implementation code and the automation of the synthesis activity is not addressed.

Work on program synthesis is clearly related to ours. The *Deductive Tableau System* [21] is an environment for interactive program development. The synthesis activity amounts to proving the specification statement, and the system incrementally builds the program as the proof progresses. Some degree of automation has been achieved in the SNARK system [24], but in an application domain completely different from ours. [3,4] describe a system based on a special method of deductive program synthesis, where several strategies and heuristics provide a significant degree of automation. However, the system does not provide the guarantee of correctness required in the setting of metatheoretic extensibility.

5. Conclusions

We have described the automation of meta-theoretic reasoning within a constructive meta-theory. The adoption of a constructive meta-theory enables us to extract programs from proofs of theorems stating the decidability of certain properties. We have identified a set of principled extensions to the heuristics for induction needed to tackle the proof obligations arising in the new problem domain and we have discussed solutions to the problem of incorporating the proposed extensions into the *CIAM-Oyster* system. While this involved some extensions to the available heuristics, the basic inductive plan and associated heuristics were successful in this new domain. Since syntax is usually defined inductively, and properties of the syntax are defined by corresponding forms of recursion, the heuristics devised in the setting of inductive theorem proving suitably extended in the way we have described provide a significant degree of automation in the setting of meta-theoretic reasoning. The synthesis of the decision procedure for the biconditional fragment described in this paper gives empirical support to this claim.

References

[1] A. Armando, A. Cimatti, and L. Viganò. Building and executing proof strategies in a formal metatheory. In *AI*IA 1993, 3rd Conference of the Italian Association for Artificial Intelligence*, volume 728 of *Lecture Notes in Artificial Intelligence*. Springer Verla, 1993.

[2] D. Basin and R. Constable. Metalogical frameworks. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, Cambridge, 1993. Cambridge University Press.

[3] S. Biundo. A synthesis system mechanizing proofs by induction. In *Proc. 1986 European Conf. on Artificial Intelligence*, pages 69–78, 1986.

[4] S. Biundo. Automated synthesis of recursive algorithms as a theorem proving tool. In Y. Kodratoff, editor, *Eighth European Conference on Artificial Intelligence*, pages 553–8. Pitman, 1988.

[5] A. Bouhoula and M. Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14(2):189–235, 1995.

[6] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, 1979. ACM monograph series.

[7] R. S. Boyer and J S. Moore. Metafunctions. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*, pages 103–184. Academic Press, 1981.

[8] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*. Academic Press, 1988. Perspectives in Computing, Vol 23.

[9] 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.

[10] A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill, and A. Stevens. A rational reconstruction and extension of recursion analysis. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 359–365. Morgan Kaufmann, 1989. Also available from Edinburgh as DAI Research Paper 419.

[11] Alan 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.

[12] Alan Bundy, editor. *12th Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 814, Nancy, France, 1994. Springer-Verlag.

[13] Alan Bundy, Frank van Harmelen, Jane Hesketh, and Alan 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.

[14] R. L. Constable, S. F. Allen, H. M. Bromley, et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

[15] R. L. Constable and D. J. Howe. Implementing metamathematics as an approach to automatic theorem proving. In R. B. Banerji, editor, *Formal Techniques in Artificial Intelligence: A Sourcebook*, pages 45–76, Amsterdam, 1990. North Holland.

[16] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. Association for Computing Machinery*, 7:201–215, 1960.

[17] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, March 1982.

[18] Roy Dyckhoff. Contraction-free sequent calculi for intuitionistic logic. *Journal of Symbolic Logic*, 57:795–807, 1992.

[19] S. Feferman. Finitary inductively presented logics. In *Logic Colloquium '88*, pages 191–220, Amsterdam, 1989. North-Holland.

[20] F. Giunchiglia and P. Traverso. A metatheory of a mechanized object theory. *Artificial Intelligence*, 80:197–241, 1996. Also available as IRST-Technical Report 9211-24, IRST, Trento, Italy, 1992.

[21] Z. Manna and R. J. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[22] Seán Matthews, Alan Smaill, and David Basin. Experience with FS_0 as a framework theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 61–82, Cambridge, 1993. Cambridge University Press.

[23] Lawrence C. Paulson. Designing a theorem prover. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 415–475. Oxford University Press, 1992.

[24] Mark Stickel, Richard Waldinger, Michael Lowry, Thomas Pressburger, and Ian Underwood. Deductive composition of astronomical software from subroutine libraries. In Bundy [12], pages 341–355.