Automatic Synthesis of Recursive Programs: The Proof-Planning Paradigm

Alessandro Armando
DIST – University of Genova
Via Opera Pia 11A
16145 Genova
Italy
armando@dist.unige.it

Alan Smaill
University of Edinburgh
Dept. of AI, 80 South Bridge
Edinburgh EH1 1HN
Scotland
smaill@dai.ed.ac.uk

Ian Green
University of Edinburgh
Dept. of AI, 80 South Bridge
Edinburgh EH1 1HN
Scotland
I.Green@ed.ac.uk

Abstract

We describe a proof plan that characterises a family of proofs corresponding to the synthesis of recursive functional programs. This plan provides a significant degree of automation in the construction of recursive programs from specifications, together with correctness proofs. This plan makes use of meta-variables to allow successive refinement of the identity of unknowns, and so allows the program and the proof to be developed hand in hand. We illustrate the plan with parts of a substantial example — the synthesis of a unification algorithm.

1 Introduction

Our interest is in the automation of synthesis of recursive functional programs. We show how recent work in *proof plans* [4, 5] provides a means of controlling the synthesis process; together with the use of techniques from implementations of higher-order logic, this allows a clean treatment of the logical context in which program fragments have to be constructed.

We illustrate the strength of our approach by showing that a substantial degree of automation is achieved in a challenge case study: the synthesis of a unification algorithm. We have chosen this example because of its significance (there have been several studies showing the application of formal development techniques to this particular algorithm), and because it represent a challenge case study for automated program synthesis (even the simpler problem of verifying the algorithm is a challenge for state-of-the-art verification systems; as a matter of fact, all the verification proofs we are aware of rely on a great deal of user interaction).

We work in constructive type theory, similar to that of [7]. This formal framework has the advantage of *total correctness* of the synthesised programs, unlike the weaker partial correctness results of other approaches (see section 3). In this setting program synthesis is achieved by proving a specification statement of the form

$$\forall input.\exists output.spec(input, output) \qquad (1)$$

where $input$ is a vector of arguments (each possibly of different type) and $output$ is a result, and $spec$ is a proposition describing the required relation between them. A program meeting this specification can be extracted from its proof via the formulae-as-types principle [14]. This style of program synthesis has been widely advocated [22, 23], and supported in a number of interactive implementations, such as NuPRL [7]. However, there has been less success so far in the automation of this approach; we suggest here that a proof-planning approach can structure the task in such a way as to allow significant automation.

A *proof-plan* is a characterisation of a family of proofs; proofs in this family have a common structure which is used to guide search for proofs in that family. Work in proof planning has been focused primarily on proofs of universally quantified statements in inductively defined domains. However, in the synthesis case, some extensions and generalisations of earlier techniques are required. We have implemented *middle-out reasoning* in order to address a key difficulty in proving theorems such as (1). The basic idea is to represent unknown quantities (`eureka' values) with meta-variables; these variables are gradually refined as subsequent proof-planning takes place (see section 2). Control of this refinement is the crucial factor in the success of such an approach, and the search control offered by proof-planning provides the necessary mechanism to guide the selection of instantiations and proof steps. In addition, we use a representation based on higher order logic programming, where a notion of scope is built in [20]. This allows us to filter out quickly many badly formed candidates.

We liken our approach to that advocated by Gries who wrote [12, p. 5]: "More importantly, the study of program correctness proofs has led to the discovery and elucidation of methods of *developing* programs. Basically, one attempts to develop a program and its proof hand-in-hand, with the proof ideas leading the way!". Our immediate concern is to synthesise programs directly from typically non-executable specifications, rather than to compose pre-existing programs.

We believe the approach provides a promising methodology that has a number of technical, as well as practical, advantages over the many other approaches to automatic program synthesis.

Overview of Paper

In section 2 we describe a proof plan for the synthesis of recursive functional programs. We illustrate aspects of this approach with a running example. In section 3 we discuss related work, and finally we draw some conclusions and mention future work in section 4.

2 Proof-Planning

Proof planning was proposed in [4] as a technique for flexibly combining tactics tailored to the goal at hand. The problem of proving a theorem is split into two stages, a planning stage where an appropriate tactic is assembled by the planning engine, and an execution stage where the tactic is executed to obtain a proof.

A *proof plan* is a tactic together with a *method*. A method is a partial specification of the associated tactic. It consists of the *preconditions* under which the tactic is applicable and the *effects* of its application. The application of a method to a given goal amounts to checking the preconditions against the goal and then determining a set of subgoals by computing the effects. It is worth pointing out that a method can promise more than the corresponding tactic can actually deliver, if the method is badly designed. However this only means that the planner can build proof-plans (i.e., tactics) whose execution will eventually fail. Correctness of the final proof is assured by the execution of the tactic, in the normal case of success.

By design then, searching for a proof-plan is easier than searching for a proof in the underlying object theory because search at the planning level is highly constrained. (Of course we cannot plan all theorems of the object theory.) Furthermore, methods take for granted basic (meta-level) operations (e.g., rewriting) which may require non-trivial object-level justification when the corresponding tactic is executed. If a method admits multiple solutions, the search proceeds from the first one, while the remaining ones are generated and used in the event of backtracking.

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 formulas. We consider the following form of rewriting. Let H and K be expressions and Γ a set of formulas, H *rewrites to* K *in context* Γ (in symbols $H \mapsto_\Gamma K$) if there exists a set of rewrite rules justified by the formulas in Γ which allows us to rewrite H to K. Rewrite rules are generated from equalities, equivalences and implications, in such a way that if H occurs in a formula $\phi[H]$ and $H \mapsto_\Gamma K$ then $\Gamma \vdash \phi[H]$ logically follows from $\Gamma \vdash \phi[K]$.

It is often the case that the formula needed to justify a given rewrite is not immediately available in the context but — instead — it is buried somewhere in the goal. To overcome the difficulty we extend the notion of context as follows. If α occurs in $\phi[H]$, we say that α *fertilizes* H *in* $\phi[H]$ if the following fact (called *fertilization condition*) holds:

$$\vdash (\alpha \rightarrow K = H) \rightarrow (\phi[K] \rightarrow \phi[H]).$$

It can be readily shown that if $H \mapsto_{\Gamma \cup \{\alpha\}} K$ and α fertilizes H in $\bigwedge \Gamma \rightarrow \phi[H]$, then $\Gamma \vdash \phi[H]$ logically follows from $\Gamma \vdash \phi[K]$. (Here $\bigwedge \Gamma$ is the conjunction of all the formulas in Γ.) This fact gives us an extended notion of *context-sensitive rewriting*: in order to rewrite H in $\Gamma \vdash \phi[H]$, we can consider as part of the context of rewriting all the sub-formulas of the goal which fertilize H in $\bigwedge \Gamma \rightarrow \phi[H]$.

For an example, consider the goal:

$$\vdash s = t \wedge (P(u) \rightarrow P(s));$$

it is easy to verify that $s = t$ fertilizes H in the goal; therefore it is sound to rewrite using the equality to obtain:

$$\vdash s = t \wedge (P(u) \rightarrow P(t)).$$

This extended notion of fertilization was introduced in [10]. The same work defines a notion of context-sensitive rewriting which is computationally easier but weaker (and therefore insufficient to tackle our case study) than the extended form of rewriting defined above.

2.1 Proof-Planning Verification Proofs

We now illustrate methods encoding a particularly effective set of heuristics for theorem proving on inductive domains. The **induction** method selects the most promising form of induction via a process called *ripple analysis* [6]. The **elementary** method checks whether the goal can be directly proven via simple propositional and equality reasoning. The **sym-eval** method simplifies the goal via repeated application of symbolic evaluation and routine reasoning. The step case(s) of proofs by inductions are dealt by the

wave and **fertilisation** methods. The **wave** method is a rewrite process called *rippling* [5], aimed at enabling the application of the induction hypothesis. To illustrate rippling, consider the proof of the associativity of composition of substitutions over S-expressions. S-expressions are here defined to be the smallest set containing a finite set of constants and variables and closed under pairing (here denoted by "$\cdot$"). (Here "$\lhd$" and"$\diamond$" represent application and composition of substitution over S-expressions respectively.) The proof is by structural induction, and the step case is as follows:

$$\begin{array}{c} l \lhd ((s_1 \diamond s_2) \diamond s_3) = l \lhd (s_1 \diamond (s_2 \diamond s_3)), \\ \boldsymbol{r \lhd ((s_1 \diamond s_2) \diamond s_3) = r \lhd (s_1 \diamond (s_2 \diamond s_3))} \\ \vdash \\ \boxed{l \cdot \boldsymbol{r}} \lhd ((s_1 \diamond s_2) \diamond s_3) = \boxed{l \cdot \boldsymbol{r}} \lhd (s_1 \diamond (s_2 \diamond s_3)) \end{array} \tag{2}$$

The annotations on the conclusion mark the difference between the conclusion and the hypotheses; the latter are dyed with different colours (light gray and black), while the former is dyed in such a way that *(i)* the sub-expressions occurring only in one induction hypothesis are dyed with the corresponding colour (notice that — as a consequence — the sub-expressions occurring in both induction hypothesis are dyed in dark gray); *(ii)* the sub-expressions which do not occur in any induction hypotheses are highlighted by a gray box (and are called *wave-fronts*). For any colour c, the expression we get by removing from the conclusion all the symbols not dyed with c is the *skeleton* of colour c. The conclusion of (2) has two skeletons:

$$\begin{array}{c} l \lhd ((s_1 \diamond s_2) \diamond s_3) = l \lhd (s_1 \diamond (s_2 \diamond s_3)) \\ \boldsymbol{r \lhd ((s_1 \diamond s_2) \diamond s_3) = r \lhd (s_1 \diamond (s_2 \diamond s_3))} \end{array}$$

Notice that the skeletons coincide with the induction hypotheses. Indeed the initial goal of the step case is always decorated in such a way that the skeletons of the conclusion are identical to the available induction hypotheses. The annotations are automatically put in place by a procedure called *difference matching* [2].

The **wave** method 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 skeletons and—simultaneously—make progress towards the application of the induction hypothesis. Wave-rules are automatically extracted from recursive definitions and lemmas. For instance, the wave-rule generated from the definition of the application of a substitution to an S-expression is:[1]

$$\boxed{L \cdot \boldsymbol{R}} \lhd S \ \overset{=}{\mapsto}\ \boxed{L \lhd S \cdot \boldsymbol{R \lhd S}} \tag{3}$$

[1]We use $\overset{=}{\mapsto}$ for a rewrite based on an equation, and $\overset{\leftrightarrow}{\mapsto}$, $\overset{\leftarrow}{\mapsto}$ for rewrites based on an equivalence and implication respectively.

Two applications of wave-rule (3) rewrite the goal above to:

$$\begin{array}{c} l \lhd ((s_1 \diamond s_2) \diamond s_3) = l \lhd (s_1 \diamond (s_2 \diamond s_3)), \\ \boldsymbol{r \lhd ((s_1 \diamond s_2) \diamond s_3) = r \lhd (s_1 \diamond (s_2 \diamond s_3))} \\ \vdash \\ \boxed{l \lhd ((s_1 \diamond s_2) \diamond s_3) \cdot \boldsymbol{r \lhd ((s_1 \diamond s_2) \diamond s_3)}} \\ = \boxed{l \lhd (s_1 \diamond (s_2 \diamond s_3)) \cdot \boldsymbol{r \lhd (s_1 \diamond (s_2 \diamond s_3))}} \end{array} \tag{4}$$

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 (4) all the 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*.

The **fertilisation** method is applied when no further rippling is possible. At this stage in fact it is often possible to use the induction hypothesis as a rewrite rule to produce an equation between two identical terms. In our example, rewriting (4) by the two induction hypotheses, we obtain an identity thereby finishing the proof. Although the sequence of rewritings is short in this case, in general a long sequence of rewritings can be obtained, guided by the principle of only using wave rules.

Since wave-rules must preserve the skeleton of the annotated goal, the search space induced by rippling is less than that induced by rewriting at the object-level. For any ripple proof, there is a corresponding object-level proof (obtained by removing the annotations), but the converse is not the case since not all object-level rewriting is skeleton-preserving.

Expressions occurring at positions corresponding to universal quantification in hypotheses are annotated as $\lfloor t \rfloor$; this is known as a *sink*. As the universally quantified variables corresponding to sinks may be instantiated to any term, the ripple engine can reduce the difference between the induction conclusion and the induction hypotheses by putting annotated terms into sinks. However, for fertilization to succeed, all the sinks associated with a given universal variable must contain the same term. If this is not the case, a way out of this problem is to look in the context for "wave-front creating" rewrites, i.e., annotated rules of the form:

$$F(Y) \mapsto F(\boxed{G(\boldsymbol{Y})})$$

Wave-front creating rules are used to introduce new wave-fronts that enable further rippling and then bring the missing term structure in the appropriate sinks. (We will see an application of this at the end of §2.3.)

2.2 Proof-Planning Synthesis Proofs

In order to apply the approach above to synthesis proofs, we need to deal with existential quantifiers. The problem of existential witnesses is central to synthesis proofs. Once a witness has been introduced, subsequent proof is a verification that this witness has the required properties. Rather that automate the choice of this `eureka' witness, we introduce a metavariable to stand for the witnessing term. Subsequent planning refines the metavariable into, eventually, an object-level term. We call this simultaneous synthesis and verification *middle-out reasoning*.

A new method called **e-intro** simply supplies a metavariable to stand for a witness, hoping that it will be instantiated in subsequent planning; the dual, **e-elim**, eliminates existentials from the induction hypotheses. These two methods transform synthesis problems into verification problems (containing meta-variables). The verification plan described in [5] becomes a synthesis plan by incorporating these two new methods. This allows *rippling* and the other parts of the plan to proceed as in the verification case.

The plans have the following shape:

```
Verification Plan =
Induction then
  [ sym-eval,
    wave then try fertilisation
  ]

Synthesis Plan =
Induction then
  [ e-intro then sym-eval,
    e-intro then e-elim
    then wave then try fertilisation
  ]
```

The synthesis plan indicates that after using the **induction** method, existential quantifiers are introduced in the base-case(s) prior to symbolic evaluation; in the step-case(s) existentials are introduced in the conclusion and eliminated in the hypotheses, followed by **wave**, and then, if applicable **fertilisation**. To illustrate consider the problem of synthesising a program to find, for any nonempty list s, the last element $last(s)$ of s, and the list $front(s)$ of all but the last element of s. (This example is adapted from [19].) The specification problem is:

$$\forall s{:}T\ list.\exists f{:}T\ list.\exists l{:}T.(s \neq nil \rightarrow s = f@(l :: nil))$$

where :: and @ denote the cons and the concatenation operator, respectively. Let the following wave rules (directly justified by the definition of @ and the substitutivity property of equality) be available:

$$\boxed{E :: \boxed{\boldsymbol{L}}}\ @M \;\overset{\Rightarrow}{\mapsto}\; \boxed{E :: \boxed{\boldsymbol{L@M}}} \qquad (5)$$

$$\boxed{X :: \boxed{\boldsymbol{S_1}}} = \boxed{X :: \boxed{\boldsymbol{S_2}}} \;\overset{\Leftrightarrow}{\mapsto}\; S_1 = S_2 \qquad (6)$$

The **induction** method suggests structural induction on s. The step case is:

$$\exists f{:}T\ list.\exists l{:}T.(s \neq nil \rightarrow s = f@(l :: nil)) \vdash$$
$$\exists f{:}T\ list.\exists l{:}T.(\boxed{x :: \boxed{\boldsymbol{s}}} \neq nil \rightarrow \boxed{x :: \boxed{\boldsymbol{s}}} = f@(l :: nil))$$

e-intro (**e-elim**) removes the existential quantifiers in the conclusion (hypothesis), replacing f and l with metavariables F and L (constants f_c and l_c), respectively:

$$s \neq nil \rightarrow s = f_c@(l_c :: nil)$$
$$\vdash \boxed{x :: \boxed{\boldsymbol{s}}} \neq nil \rightarrow \boxed{x :: \boxed{\boldsymbol{s}}} = F@(L :: nil)$$

Let us focus on the case $s \neq nil$. Using (5), the **wave** method instantiates F to $\boxed{E :: \boxed{\boldsymbol{F'}}}$ and rewrites the conclusion to:

$$s \neq nil \rightarrow s = f_c@(l_c :: nil)$$
$$\vdash \boxed{x :: \boxed{\boldsymbol{s}}} \neq nil \rightarrow \boxed{x :: \boxed{\boldsymbol{s}}} = \boxed{X :: \boxed{\boldsymbol{F'@(L :: nil)}}}$$

Wave rule (6) now suggests to instantiate E to x and rewrites the conclusion of the goal to:

$$s \neq nil \rightarrow s = f_c@(l_c :: nil)$$
$$\vdash \boxed{x :: \boxed{\boldsymbol{s}}} \neq nil \rightarrow s = F'@(L :: nil)$$

Finally the **fertilization** method solves the goal by instantiating F' to f_c and L to l_c. At the end of the proof F and L are thus instantiated to $x :: f_c$ and l_c, respectively. The program fragments extracted from the above proof are:

if $s \neq nil$ **then** $\quad front(x :: s) = x :: front(s)$

if $s \neq nil$ **then** $\quad last(x :: s) = last(s)$

Middle out reasoning is still potentially very explosive, since many rewrites are potentially applicable in the presence of meta-variables. However, the restriction of rewriting in induction proofs according to the **wave** method restricts considerably possible instantiations.

2.3 Synthesis of a Unification Algorithm

We follow the formalisation of [18], and represent terms as S-expressions. The reader should consult [18] for more detail on this formalisation, and the unification problem in general. The aim here is to synthesise an algorithm which, given two terms, determines whether there is a unifying substitution for the variables in the terms, returning such a substitution in this case, and otherwise returns a token indicating that no such substitution exists.

The relevant syntactic notions used in the specification are defined in type theory, so as to satisfy appropriate properties. In particular, in addition to the wave rule (3), we have the following two wave-rules concerning the composition of substitutions:

$$E \lhd \boxed{\boxed{S_l} \diamond S_r} \;\overset{\Rightarrow}{\mapsto}\; \boxed{\boxed{E \lhd S_l} \lhd S_r} \qquad (7)$$

$$\lfloor \boldsymbol{E} \rfloor \lhd \boxed{S_l \diamond \boldsymbol{S_r}} \;\overset{\Rightarrow}{\mapsto}\; \lfloor \boldsymbol{E} \lhd \boldsymbol{S_l} \rfloor \lhd \boldsymbol{S_r} \qquad (8)$$

(These rules are identical at the object level; the two possible annotations here indicate that there are two roles that the rewrites can play in the course of inductive proof.)

The top-level goal in this case is thus (universally quantified in e_1, e_2):

$$\begin{array}{c}
\exists s. \left(\begin{array}{c} e_1 \lhd s = e_2 \lhd s \\ \wedge \\ \forall s'.(e_1 \lhd s' = e_2 \lhd s' \to \forall e.(e \lhd s \lhd s' = e \lhd s')) \end{array} \right) \\
\vee \\
\forall s.(e_1 \lhd s \neq e_2 \lhd s)
\end{array}$$

In a type theory formulation, a proof of such a disjunction yields an algorithm determining which disjunct holds, and for the left disjunction returning a witnessing unifier, corresponding to the existential quantifier.

We indicate how the methods above apply in this example by describing some steps of the planning process, in particular from the step case of the inductive proof.

Suppose that we have two induction hypotheses, labelled l and r corresponding to taking $(l_1 \cdot r_1)$ and $(l_2 \cdot r_2)$ for e_1 and e_2, respectively. The goal is now annotated as follows (where S is a meta-variable):

$$\begin{array}{c}
\left(\begin{array}{c} \boxed{\lfloor l_1 \rfloor \cdot \lfloor \boldsymbol{r_1} \rfloor} \lhd \boxed{S(s_l, \boldsymbol{s_r})} = \boxed{\lfloor l_2 \rfloor \cdot \lfloor \boldsymbol{r_2} \rfloor} \lhd \boxed{S(s_l, \boldsymbol{s_r})} \\ \wedge \\ \forall s' (\boxed{\lfloor l_1 \rfloor \cdot \lfloor \boldsymbol{r_1} \rfloor} \lhd s' = \boxed{\lfloor l_2 \rfloor \cdot \lfloor \boldsymbol{r_2} \rfloor} \lhd s' \\ \to \boxed{S(s_l, \boldsymbol{s_r})} \diamond s' = s')) \end{array} \right) \\
\vee \\
\forall s. \boxed{\lfloor l_1 \rfloor \cdot \lfloor \boldsymbol{r_1} \rfloor} \lhd s \neq \boxed{\lfloor l_2 \rfloor \cdot \lfloor \boldsymbol{r_2} \rfloor} \lhd s
\end{array}$$

By applying wave rules (3), (7) and the following wave-rules

$$\begin{array}{rcl}
\boxed{L_1 \cdot \boldsymbol{R_1}} = \boxed{L_2 \cdot \boldsymbol{R_2}} & \overset{\leftarrow}{\mapsto} & \boxed{L_1 = L_2 \wedge \boldsymbol{R_1 = R_2}} \\
\boxed{S_l \diamond \boldsymbol{S_r}} \lhd S = S & \overset{\leftarrow}{\mapsto} & \boxed{S_l \lhd S = S \wedge \boldsymbol{S_r \lhd S = S}} \\
\boxed{E_1 \lhd \boldsymbol{S}} = \boxed{E_2 \lhd \boldsymbol{S}} & \overset{\leftarrow}{\mapsto} & E_1 = E_2
\end{array}$$

the S gets instantiated to $\boxed{S_l \diamond \boldsymbol{S_r}}$ and we get:

$$\begin{array}{c}
\left(\begin{array}{c} \boxed{\lfloor l_1 \rfloor \lhd s_l = \lfloor l_2 \rfloor \lhd s_l \wedge} \\ \boxed{\lfloor \boldsymbol{r_1 \lhd s_l} \rfloor \lhd \boldsymbol{s_r} = \lfloor \boldsymbol{r_2 \lhd s_l} \rfloor \lhd \boldsymbol{s_r}} \\ \wedge \\ \forall s'.(\boxed{\lfloor l_1 \rfloor \lhd s' = \lfloor l_2 \rfloor \lhd s' \wedge \lfloor \boldsymbol{r_1} \rfloor \lhd \boldsymbol{s'} = \lfloor \boldsymbol{r_2} \rfloor \lhd \boldsymbol{s'}} \\ \to \boxed{s_l \diamond s' = s' \wedge \boldsymbol{s_r \diamond s' = s'}})) \end{array} \right) \\
\vee \\
\forall s. \neg(\boxed{\lfloor l_1 \rfloor \lhd s = \lfloor l_2 \rfloor \lhd s \wedge \lfloor \boldsymbol{r_1} \rfloor \lhd \boldsymbol{s} = \lfloor \boldsymbol{r_2} \rfloor \lhd \boldsymbol{s}})
\end{array}$$

The following domain-independent wave-rules:

$$\begin{array}{rcl}
C \to \boxed{L \wedge \boldsymbol{R}} & \overset{\leftarrow}{\mapsto} & \boxed{C \to L \wedge \boldsymbol{C \to R}} \\
\boxed{L_1 \wedge \boldsymbol{R_1}} \wedge \boxed{L_2 \wedge \boldsymbol{R_2}} & \overset{\leftarrow}{\mapsto} & \boxed{L_1 \wedge L_2 \wedge \boldsymbol{R_1 \wedge R_2}} \\
\neg \boxed{A \wedge \boldsymbol{B}} & \overset{\leftarrow}{\mapsto} & \boxed{\neg A \vee \boldsymbol{\neg B}} \\
\forall x. \boxed{A \wedge \boldsymbol{B}} & \overset{\leftarrow}{\mapsto} & \boxed{\forall x. A \wedge \boldsymbol{\forall x. B}} \\
\boxed{L \wedge \boldsymbol{R}} \vee C & \overset{\leftarrow}{\mapsto} & \boxed{L \vee C \wedge \boldsymbol{R \vee C}}
\end{array}$$

are then used to rewrite the goal to:

$$\boxed{\begin{array}{c}
\left(\left(\begin{array}{c} \lfloor l_1 \rfloor \lhd s_l = \lfloor l_2 \rfloor \lhd s_l \\ \wedge \\ \forall s'.(\boxed{\lfloor l_1 \rfloor \lhd s' = \lfloor l_2 \rfloor \lhd s' \wedge r_1 \lhd s' = r_2 \lhd s'} \\ \to s_l \diamond s' = s') \end{array} \right) \vee \forall s. \boxed{\lfloor l_1 \rfloor \lhd s \neq \lfloor l_2 \rfloor \lhd s \vee r_1 \lhd s \neq r_2 \lhd s} \right) \\
\wedge \\
\left(\left(\begin{array}{c} \lfloor r_1 \lhd s_l \rfloor \lhd s_r = \lfloor r_2 \lhd s_l \rfloor \lhd s_r \\ \wedge \\ \forall s'.(\boxed{l_1 \lhd s' = l_2 \lhd s' \wedge \lfloor \boldsymbol{r_1} \rfloor \lhd \boldsymbol{s'} = \lfloor \boldsymbol{r_2} \rfloor \lhd \boldsymbol{s'}} \\ \to s_r \diamond s' = s') \end{array} \right) \vee \boldsymbol{\forall s.} \boxed{l_1 \lhd s \neq l_2 \lhd s \vee \lfloor \boldsymbol{r_1} \rfloor \lhd \boldsymbol{s} \neq \lfloor \boldsymbol{r_2} \rfloor \lhd \boldsymbol{s}} \right)
\end{array}}$$

Now the outer wave-front has beached, while the inner ones are blocked. However the major problem here is the fact that in the right conjunct different occurrences of the same sink have been instantiated with different terms (namely $\lfloor r_1 \lhd s_l \rfloor$ and $\lfloor r_1 \rfloor$). To overcome this difficulty we resort to the first induction hypothesis, i.e.:

$$\begin{array}{c}
\left(\left(\begin{array}{c} l_1 \lhd s_l = l_2 \lhd s_l \\ \wedge \\ \forall s'. \left(l_1 \lhd s' = l_2 \lhd s' \to \underbrace{s_l \diamond s' = s'} \right) \end{array} \right) \right) \\
\vee \\
\forall s.(l_1 \lhd s \neq l_2 \lhd s)
\end{array}$$

from which the under-braced equation permits use of the wave-front creating rule, $s' \overset{\leftarrow}{\mapsto} \boxed{s_l \diamond \boldsymbol{s'}}$; this rule is an example of a context-sensitive rule as described earlier. Application of this rule gives:

$$\begin{array}{c}
\left(\begin{array}{c}
\lfloor r_1 \lhd s_l \rfloor \lhd s_r = \lfloor r_2 \lhd s_l \rfloor \lhd s_r \\
\wedge \\
\forall s'.(\ \boxed{l_1 \lhd s' = l_2 \lhd s' \wedge} \\
\boxed{\lfloor r_1 \rfloor \lhd \boxed{s_l \diamond s'} = \lfloor r_2 \rfloor \lhd \boxed{s_l \diamond s'}} \\
\rightarrow s_r \diamond s' = s')
\end{array}\right) \\
\vee \\
\forall s.\ \boxed{l_1 \lhd s \neq l_2 \lhd s \vee} \\
\boxed{\lfloor r_1 \rfloor \lhd \boxed{s_l \diamond s} \neq \lfloor r_2 \rfloor \lhd \boxed{s_l \diamond s}}
\end{array}$$

Note that this step has enabled the application of wave-rule (8) to move the created front into a sink position. Repeated application of this wave-rule rewrites the right conjunct into:

$$\begin{array}{c}
\left(\begin{array}{c}
\lfloor r_1 \lhd s_l \rfloor \lhd s_r = \lfloor r_2 \lhd s_l \rfloor \lhd s_r \\
\wedge \\
\forall s'.(\ \boxed{l_1 \lhd s' = l_2 \lhd s' \wedge} \\
\boxed{\lfloor r_1 \lhd s_l \rfloor \lhd s' = \lfloor r_2 \lhd s_l \rfloor \lhd s'} \\
\rightarrow s_r \diamond s' = s')
\end{array}\right) \\
\vee \\
\forall s.\ \boxed{l_1 \lhd s \neq l_2 \lhd s \vee \boxed{\lfloor r_1 \lhd s_l \rfloor \lhd s \neq \lfloor r_2 \lhd s_l \rfloor \lhd s}}
\end{array}$$

Now all the sinks contain the same terms and the blocked wave-fronts can be removed by means of the following two domain-independent wave-rules:

$$\boxed{\boldsymbol{A} \wedge B} \overset{\rightarrow}{\Rightarrow} A \qquad \boxed{\boldsymbol{A} \vee B} \overset{\leftarrow}{\Rightarrow} A$$

The resulting goal is then readily solved by **fertilization**.

Comments

The sequence of rippling steps shown above (including the instantiations of the meta-variables) is discovered automatically by our planning engine.

The simultaneous structural induction over e_1 and e_2 selected at the beginning of the proof is revised in favour of a more sophisticated induction principle whose second induction hypothesis is of the form $\Phi(r_1 \lhd s_l, r_2 \lhd s_l)$ where s_l is the most general unifier for l_1 and l_2. To illustrate consider the program (fragment) corresponding to the proof (fragment) we presented above:

$$\begin{array}{l} \mathit{unify}(l_1 \cdot r_1, l_2 \cdot r_2) = \\ \qquad \textbf{let } s_l = \mathit{unify}(l_1, l_2) \textbf{ in } s_l \diamond \mathit{unify}(r_1 \lhd s_l, r_2 \lhd s_l) \end{array}$$

On the other hand, the initial induction would have led us (if successful) to:

$$\begin{array}{l} \mathit{unify}(l_1 \cdot r_1, l_2 \cdot r_2) = \\ \qquad \textbf{let } s_l = \mathit{unify}(l_1, l_2) \textbf{ in } s_l \diamond \mathit{unify}(r_1, r_2) \end{array}$$

It can be easily verified that this last program incorrectly returns a `unifier' if applied to the S-expressions $(x \cdot x)$ and $(a \cdot b)$ (where x is a variable and a, b are two distinguished constants). (Clearly this last program fragment can not possibly be extracted from any proof of our specification.)

The induction scheme must of course be shown to be valid by exhibiting some well-foundedness proof. Additionally, we have only shown the step case part of the proof: the base cases are largely straightforward, with the exception of the variable/term case, where the occurs check arises. The system currently cannot automatically deal with the well-foundedness proof nor with the difficult base case.

The search problem is clearly worse in the case of synthesis, compared to verification — for example, the termination of rippling for ground formulas, as demonstrated in [2], no longer holds. However, the constraints imposed by the proof planning framework are such that significant examples such as above become feasible.

3 Related work

We have already mentioned various systems that make use of the ideas of proofs-as-programs; as they stand these systems have only a small amount of automation. Such systems support to varying degrees Gries's desire to see programming and proof intertwined. The use of meta-variables in place of existential witnesses is a standard technique (dating back to early work in resolution-based synthesis [11, 26]). Automatic program synthesis and theorem proving diverged somewhat: clausal resolution was felt unsuited to recursive program synthesis because of the difficulty of incorporating induction. Additionally, inductive provers (notably Nqthm) did not support existential quantifiers and so were not well suited for synthesis. Synthesis systems went the way of dedicated program construction calculi.

Manna and Waldinger reunited the two in [19], providing an interactive, non-clausal, deductive tableaux approach. Snark [25] automates proof-search in parts of the framework of [19]. It has been used within the Amphion system to synthesise programs via the *composition of existing subroutines*, rather than from non-executable specifications. The Amphion system is designed to operate at this high level; it has little support for inductive reasoning and so iterative and recursive programs are largely outside its capabilities. Induction can be done, but only interactively. Snark's clausal representation suffers the drawbacks mentioned above. The system only shows the partial correctness of the resulting programs.

Biundo skolemizes $\forall\exists\forall$ specifications and uses heuristics to construct a partial proof tree [3]. A subset of leaves of the partial proof form the definition of the Skolem function: the remainder of the proof is completed (verified) with respect to this definition. Synthesis is effected by *clause-set translations* (e.g., rewriting and case-splitting) which induce an AND/OR search space. Induction, which is not one of the translations, is carried out before formulae are put into clausal form, and thus is given a distinguished status in the synthesis strategy. Multiple inductions are not possible. This limitation could probably be overcome in principle (as it could in Snark), but heuristic control has not been forthcoming. The *specialization* translation allows instances of induction hypotheses to be made to enable fertilization. This is quite similar to our approach; however, there is no heuristic which guides the application of other translations into making this instance obvious, or even necessary. Our approach is much more controlled because of the need for skeleton preservation.

Kraan has used proof planning in the synthesis of logic programs [16]. Her approach uses middle-out reasoning to construct predicate definitions from specifications in classical logic. Proving the equivalence of specification and implementation does not guarantee total correctness. In contrast, the type theory ensures we only extract terminating functions and so yields total correctness. A related proposal is due to Wiggins [27].

Basin [1] advocates the use of middle-out reasoning using higher-order meta-variables in interactive synthesis proofs. His approach, though interactive, is similar to ours but, like in Snark and in Biundo's approach, this only demonstrates partial correctness. The instantiation of meta-variables to programs is limited to an executable fragment of the logic, which is specified manually, so total correctness must be proven by meta-logical reasoning by the user. Again, we lean on the type theory to deal with questions of constructivity.

We believe that type-theory provides a firmer mathematical foundation than is found in these systems, and that the type system provides a better base for structuring larger software development. In the longer term, we see work on composing software components as in [25] as complementary to our work on more direct synthesis of smaller, but recursive, programs, as reported here.

3.1 Applications to the unification problem

The unification problem has been used frequently as a test-bed for various formal methods techniques, though none of the case studies have had the degree of automation provided by proof planning.

For example, an interactive verification (rather than a synthesis) in the system LCF of the correctness of a unification algorithm is found in [24]; this provides a good discussion of the problem of finding a good formalisation for machine supported reasoning.

[21] provides an interactive synthesis using the ideas of [19], showing the effectiveness of the calculus in interactive use, but again not attempting to automate the synthesis process. On the Logic Programming side, a calculus for the development logic programs was given in [13], and used in [9] to synthesise a unification algorithm. The development there was done by hand, the proof involving thousands of proof steps.

On the type-theoretic side, the framework closest to what we propose can be found in [8], though here again the synthesis (in this case of a matching algorithm) was carried out interactively — in particular the existential witnesses in the various cases have to be supplied manually, though there is automation of the more straightforward aspects of the proof.

4 Conclusions, Future Work

We have presented a demonstration of the significant automation of the synthesis of a unification algorithm in a constructive type theory. The formal framework ensures that the resulting programs are totally correct with respect to their specification. The key to the idea is to use proof-planning to control the required inductive proofs, and middle-out reasoning to allow existential witnesses to be developed hand-in-hand with the proof. The proof-plan does not yet deal with the validation of the induction rule, for which a difficult well-founded measure is required, nor does it deal with the base case where the `occurs check' arises.

In building the declarative methods that are needed, we also pave the way for higher-level interaction with a synthesis proof system, along the lines of [17]. Increased levels of automation are needed in systems designed as integrated programming and proving environments.

The next stages in this investigation are to look at more in the same style; to look at different search strategies for the planner; and to reduce the current redundancy in the planner so that search becomes more efficient, without losing current solutions.

Acknowledgments Alan Bundy originated the idea of proof-planning, rippling and middle-out reasoning in the context of proof-planning. 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*.

References

[1] D. Basin. Logic frameworks for logic programs. In *4th International Workshop on Logic Program Synthesis and Transformation, (LOPSTR'94)*, number 883 in LNCS. Springer-Verlag, 1994.

[2] D. Basin and T. Walsh. Annotated rewriting in inductive theorem proving. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.

[3] 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.

[4] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[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. Also available from Edinburgh as DAI Research Paper No. 567.

[6] 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.

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

[8] 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.

[9] L.-H. Eriksson. Synthesis of a unification algorithm in a logic programming calculus. *Journal of Logic Programming*, 1:3–18, 1984.

[10] J. K. Gallagher. *The Use of Proof Plans in Tactic Synthesis*. PhD thesis, University of Edinburgh, 1993.

[11] C. Green. Application of theorem proving to problem solving. In IJCAI-69 [15], pages 219–239.

[12] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[13] Å. Hanson and Tärnlund, S-Å. A natural programming calculus. In *Proc. 6th IJCAI*, pages 348–355, Tokyo, 1979. International Joint Conference on Artificial Intelligence.

[14] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[15] *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, Washington, D. C., U. S. A., 1969.

[16] I. Kraan, D. Basin, and A. Bundy. Logic program synthesis via proof planning. In K. K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, pages 1–14. Springer-Verlag, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-92-244 and Edinburgh DAI Research Report 603.

[17] H. Lowe. Proof planning: a methodology for developing AI systems involving design. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 8(4), 1994. Special Issue on Research Methodology.

[18] Z. Manna and R. Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1:5–48, 1981.

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

[20] D. Miller. Abstractions in logic programs. In P. Oddifreddi, editor, *Logic and Computer Science*, pages 329–360. Academic Press, London, 1990.

[21] D. Nardi. Formal synthesis of a unification algorithm by the deductive-tableau method. *Journal of Logic Programming*, 7:1–43, 1989.

[22] K. Nordström, B. Petersson and J. Smith. *Programming in Martin-Löf Type Theory*. Oxford University Press, 1990.

[23] C. Paulin-Mohring. Extracting F_ω's programs from proofs in the calculus of constructions. *ACM Proc. POPL*, 1989.

[24] L. Paulson. Verifying the unification algorithm in lcf. Report 50, Computer Laboratory, University of Cambridge, 1984.

[25] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In A. Bundy, editor, *12th Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, Vol. 814, pages 341–355, Nancy, France, 1994. Springer-Verlag.

[26] R. J. Waldinger and R. C. T. Lee. PROW: A step toward automatic program writing. In IJCAI-69 [15], pages 241–252.

[27] G. A. Wiggins. Synthesis and transformation of logic programs in the Whelk proof development system. In K. R. Apt, editor, *Proceedings of JICSLP-92*, pages 351–368. M. I.T. Press, Cambridge, MA, 1992.