Combining Proof Plans with Partial Order Planning for Imperative Program Synthesis

Andrew Ireland and Jamie Stark
School of Mathematical & Computer Sciences
Heriot-Watt University,
Edinburgh, Scotland, UK
a.ireland@hw.ac.uk

Abstract

The structured programming literature provides methods and a wealth of heuristic knowledge for guiding the construction of provably correct imperative programs. We investigate these methods and heuristics as a basis for mechanizing program synthesis. Our approach combines *proof planning* with conventional *partial order planning*. Proof planning is an automated theorem proving technique which uses high-level *proof plans* to guide the search for proofs. Proof plans are structured in terms of *proof methods*, which encapsulate heuristics for guiding proof search. We demonstrate that proof planning provides a local perspective on the synthesis task. In particular, we show that proof methods can be extended to represent heuristics for guiding program construction. Partial order planning complements proof planning by providing a global perspective on the synthesis task. This means that it allows us to reason about the order in which program fragments are composed. Our hybrid approach has been implemented in a semi-automatic system called BERTHA. BERTHA supports partial correctness and has been tested on a wide range of non-trivial programming examples.

1 Introduction

Within the context of constructing provably correct programs, Dijkstra [Dij72] identifies:

> *"... a number of patterns of abstraction that play a vital role in the whole process of composing programs."*

Dijkstra argues that programming typically involves the use of a small number of these *vital patterns of abstraction*. Dijkstra's vision is reflected in [Gri81], where Gries advocates methods which support the development of a program and its proof hand-in-hand. The application of such methods involves the use of heuristics. The structured programming literature [Bac86, Dij76, Gri81, Kal90] provides a wealth of heuristic knowledge for guiding program construction. While this literature focuses on programming as a manual activity, we are motivated by mechanization. That is, we are interested in these methods and heuristics as a basis for mechanizing program synthesis. Many alternative approaches to the synthesis task exist, *e.g.* transformation-based approaches. Our motivation, however, is to investigate how structured programming can be exploited directly in mechanizing imperative program synthesis.

Our starting point is *proof planning*, a theorem proving technique which uses high-level proof outlines, known as *proof plans*, to provide heuristic guidance in the search for proofs. There are strong similarities between Dijkstra's *"patterns of abstraction"* and proof plans, *i.e.* they both represent common patterns of reasoning. However, in order to apply proof planning to the program synthesis task we extend the proof plan representation to include heuristics for guiding program construction. This extension provides a local perspective on the synthesis task, *i.e.* the ability to reason about the synthesis of program fragments for individual goals. The synthesis task

$$\frac{P \rightarrow Q[E/V]}{\{P\}V := E\{Q\}}\ \text{assign}$$

$$\frac{\{P\}S_1\{R\},\ \{R\}S_2\{Q\}}{\{P\}S_1; S_2\{Q\}}\ \text{seq}$$

$$\frac{(I \wedge \neg B) \rightarrow Q,\ \ \{I \wedge B\}S\{I\}}{\{I\}\textbf{while}\ B\ \textbf{begin}\ \ invariant\{I\}\ S\ \textbf{end}\ \{Q\}}\ \text{while}$$

Figure 1: Example Proof Rules for Floyd/Hoare Style Program Reasoning.

also requires the ability to reason about the order in which such program fragments are composed. This is important where interaction between goals is possible. We use partial order planning to provide this global perspective. Our hybrid approach exploits, therefore, the complementary nature of proof planning and partial order planning.

We see three key contributions to our work. Firstly, we have extended the proof planning method schema to allow the representation of programming heuristics. Secondly, we have integrated proof planning within a partial order refinement planning framework. Thirdly, we have demonstrated how various structured programming and proof heuristics can be effectively represented within this hybrid approach. The approach has been implemented in a system called BERTHA, which supports partial correctness, *i.e.* BERTHA plans functional correctness proofs for the programs that it synthesizes. BERTHA is semi-automatic in that when it determines the need for iteration, the user is prompted for a loop invariant and guard.

In §2 we review the logical basis for reasoning about imperative programs as well as proof planning and partial order planning. The problems that arise during the synthesis of imperative programs are discussed in §3, while §4 shows how proof planning can be extended to tackle these problems. An overview of the synthesis methods is presented in §5, with particular emphasis on the heuristics associated with the methods for assignment and while-loops. Implementation details and a transcript of the BERTHA system are presented in §6. Experimental results are presented in §7 while in §8 related work is discussed. Potential future avenues of research and conclusions are given in §9 and §10 respectively.

2 Background

2.1 Reasoning about imperative programs

Based upon Floyd/Hoare logic [Flo67, Hoa69], we reason about a simple programming language, containing assignments, conditionals and while-loops. We adopt the conventional notation for a partial correctness specification:

$$\{P\}C\{Q\}$$

where P and Q are logical assertions while C denotes program code. P is known as the precondition while Q is known as the postcondition. Partial correctness is defined as follows: if P is true before C is executed and the execution of C terminates then Q is true after the execution terminates.

We relate program variables to initial values using constants, *e.g.* $x0, y0$, etc. These constants denote values that remain unchanged for a certain execution of the program, but may differ from execution to execution. They do not appear in the program code. To illustrate, consider the following specification:

$$\{x = x0\}C\{x = x0 + 1\}$$

The postcondition here states that the program variable x is equal to its initial value plus one. Hoare gives us a semantics for a simple programming language in the form of proof rules. Examples of these are given in Figure 1. It can be shown that this semantics is sound [Krz81]. Dijkstra's weakest liberal precondition predicate transformer $wlp(C, Q)$ [Dij76] can be used to calculate the weakest possible precondition of the code C given that it will achieve Q on termination.

2.2 Proof planning

Tactic-based reasoning [GMW79] involves the use of programs, known as *proof tactics*, which control the application of inference rules. Proof construction corresponds to proof tactic composition. Within proof planning [Bun88] the process of composing parameterized proof tactics is guided by high-level proof plans. A proof plan is structured in terms of *proof methods*. Heuristic preconditions associated with proof methods provide the guidance for composing proof tactics. Proof planning represents an extension to tactic-based reasoning, where proof tactics guarantee soundness while proof methods guide proof search.

The *Clam* proof planner [BvHHS90] has been used for a number of applications, but the main focus has been on proof by mathematical induction. The success of the proof plan for induction is due to the *rippling* heuristic [BSvH$^+$93]. Within rippling, syntactic differences between inductive conclusion and hypothesis are automatically identified. A syntactic class of rewrite rules, called *wave-rules*, are then used to manipulate the differences such that the hypothesis can be used to simplify the conclusion. The use of a hypothesis to simplify a conclusion is known as *fertilization*. Since rippling was developed for induction, it is not surprising that it is also applicable to the verification of while-loop programs. For a full account of rippling see [BW96, BSvH$^+$93].

A major strength of proof planning which we build upon is the notion of a *proof critic* [Ire92, IB96b]. Proof critics support the automatic analysis and patching of failed proof attempts. Previously, proof critics have been successfully applied to the automatic discovery of what are often referred to as *eureka steps* during the construction of a proof. For instance, proof critics are able to automatically generate auxiliary lemmas and conjecture generalizations [IB96a, IB96b, IB99] as well as loop invariants [IS97, SI98, IS01, EI03, EI04, IEI04]. Such proof patching steps are typically provided by the user of a theorem proving system. Proof patching can be viewed as an application of a general technique known as *middle-out reasoning* [BSH90]. Within middle-out reasoning meta-variables are used to delay choice during a proof. The motivation is that the middle of a proof is typically more constrained than the start. Middle-out reasoning has also been successfully applied to the synthesis of functional [ASG97, SG95] and logic programs [KBB93]. As will be shown in §5.1.2, both middle-out reasoning and rippling play a central role in our synthesis of while-loop programs. As a prelude to the presentation of the actual synthesis mechanism, we illustrate below in §2.3 how rippling can be used within the verification of a while-loop program.

2.3 Proof planning for while-loop verification

Consider the specification and program given in Figure 2, where the definition of $\sum$ is given as follows:

$$\begin{aligned} \sum_{m=0}^{0} m &= 0 \\ \sum_{m=0}^{X+1} m &= (X+1) + \left(\sum_{m=0}^{X} m \right) \end{aligned} \tag{1}$$

As mentioned above, rippling restricts rewriting to a syntactic class of rewrite rules known as wave-rules. Wave-rules are generated automatically from definitions and properties. For example,

$\{x = x0 \wedge x0 \geq 0\}$
$i := 0;$
$r := i;$
while not $(i = x)$ **do**
begin
$i := i + 1;$
$r := i + r$
end
$\{r = \sum_{m=0}^{x0} m\}$

Figure 2: Specification and Program: Summing First $x0$ Natural Numbers

(1) gives rise to a set of wave-rules which includes:

$$\sum_{m=0}^{\boxed{X}+1} m \Rightarrow \boxed{(X+1) + \left(\sum_{m=0}^{X} m\right)} \tag{2}$$

Note that we use shading to denote the wave-fronts, *i.e.* the syntactic differences between the left-hand side and right-hand side of the rule. The parts of the terms that are not shaded are known as *skeleton*. A key property of rippling is that the skeleton terms are preserved. As will be shown below, skeleton preservation is crucial to the proof strategy. The application of wave-rules requires that a conjecture is first annotated with wave-fronts. In order to illustrate, consider $r = \sum_{m=0}^{i} m$ as a candidate loop invariant for the code given in Figure 2. The verification condition generated from the while-loop takes the form:

$$(r = \sum_{m=0}^{i} m \wedge i \neq x) \rightarrow (i+1) + r = \sum_{m=0}^{i+1} m$$

Using wave-fronts to annotate the differences between hypothesis and conclusion gives:

$$(r = \sum_{m=0}^{i} m \wedge i \neq x) \rightarrow \boxed{(i+1) + \boxed{r}} = \sum_{m=0}^{\boxed{i}+1} m$$

The goal of rippling is to manipulate the wave-fronts so that the hypothesis can be used to simplify the conclusion. The associated ripple proof takes the form:

$(r = \sum_{m=0}^{i} m \wedge i \neq x) \rightarrow \boxed{(i+1) + \boxed{r}} = \sum_{m=0}^{\boxed{(i+1)}} m$ ripple using (2)

$(r = \sum_{m=0}^{i} m \wedge i \neq x) \rightarrow \boxed{(i+1) + \boxed{r}} = \boxed{(i+1) + (\boxed{\sum_{m=0}^{i} m})}$ fertilize

$(r = \sum_{m=0}^{i} m \wedge i \neq x) \rightarrow (i+1) + r = (i+1) + r$ elementary

Note that the fertilization step involves using the hypothesis to rewrite the conclusion. The ripple proof is relatively trivial, requiring a single wave-rule (2) application: one could argue that the meta-level overheads of rippling are not necessary in order to obtain a proof. Later we see that these overheads are essential in order to constrain the search for synthesis proofs.

2.4 Partial order planning

Within proof planning, the ordering of individual steps within a plan corresponds to the order in which goals are achieved. This approach to planning is known as *total order planning*. In contrast, *partial order planning* allows for greater flexibility in the ordering of steps during the search for a plan. That is, the ordering of individual steps within a plan is determined on a least commitment basis. There are a number of planners that support partial order planning [Tat77, LW96, TDL00]. Starting with a null plan, a partial order planner incrementally selects actions to achieve individual goals. Crucially, the position of each new action is not totally ordered with respect to the other actions within the plan. However, during the planning process auxiliary constraints, *i.e.* ordering relations, are maintained which ensure the consistency between the actions. Reasoning is then required to order the goal achievements. Similarly, our planning approach to imperative program synthesis requires the ability to reason about the order in which program fragments are composed. The ability to reason about partially ordered plan steps means that the exponential search of all possible orderings of plan steps can be avoided.

2.5 A unified planning framework

Instead of building our own partial order planner, we have instead exploited a unified planning framework [KKY95]. This framework was developed to allow theoretical and empirical comparisons to be made between the various existing planning algorithms. At the core of the framework is a generic *refinement search* algorithm. Refinement search is a process by which a set of partial solutions (plans) are incrementally transformed until a complete solution is obtained. There are two parts to the transformation process. Firstly, detail is added to the partial solutions, and secondly, partial solutions that are found to be inconsistent with the overall constraints of the planning task are pruned. Nearly all classical planning systems, including partial order planners, use refinement search. As a consequence, the unified planning framework provided an ideal basis upon which to combine proof and partial order planning.

We now describe the essential details of refinement search with the unified planning framework. Within the unified planning framework, a planning problem is represented in terms of an initial world state $\mathcal{I}$ and a desired world state $\mathcal{G}$. Search nodes within the framework are represented as a 5-tuple:

$$\langle \mathcal{T}, \mathcal{O}, \mathcal{B}, \mathcal{ST}, \mathcal{L} \rangle$$

where:

- $\mathcal{T}$ denotes the set of actions within the plan.
- $\mathcal{O}$ denotes a partial ordering relation over the actions in $\mathcal{T}$.
- $\mathcal{B}$ denotes a set of binding constraints on the variables that appears within the precondition and postconditions of actions.
- $\mathcal{ST}$ denotes a mapping between action names and operators.
- $\mathcal{L}$ denotes a set of auxiliary constraints on the actions that appear within the plan.

An initial search node takes the following form:

$$\langle \{t_I, t_G\}, \{t_I \prec t_G\}, \{\}, \{t_I \mapsto \texttt{start}, t_G \mapsto \texttt{finish}\}, \{\} \rangle$$

where `start` and `finish` denote dummy actions. Refinement search is specified by a set of *refinement strategies* **R** and a solution constructor function `sol`, where `sol`$(\mathcal{P}, \mathcal{G})$ determines whether or not plan $\mathcal{P}$ achieves the desired world state $\mathcal{G}$. From the initial node, the search for a plan progresses by the application of a refinements, chosen from **R**. This process generates child nodes, *i.e.* refined partial plans. Partial plans that are shown to be inconsistent are pruned while the set of consistent partial plans is further refined. The generic refinement planning algorithm is given in Figure 3.

Algorithm: Find-Plan($\mathcal{I}$, $\mathcal{G}$)
Parameters: `sol`: solution constructor function.

1. Construct the null plan $\mathcal{P}_0$.
2. Initialize search queue with $\mathcal{P}_0$.
3. **Begin loop**
   (a) Nondeterministically pick a node $\mathcal{N}$, with partial plan $\mathcal{P}$ from the search queue.
   (b) If `sol`$(\mathcal{P}, \mathcal{G})$ returns a solution then success (terminate).
   (c) If `sol`$(\mathcal{P}, \mathcal{G})$ returns *fail* then skip to step 3a.
   (d) Call `Refine-Plan`$(\mathcal{P})$ to generates refinements to $\mathcal{P}$.
   (e) Add all refinements to the search queue.

   End loop

Figure 3: Kambhampati's Generic Refinement Planning Algorithm.

The process of applying refinements is controlled by a routine called `Refine-Plan`, the details of which are given in Figure 4. `Refine-Plan` has essentially three roles. Firstly it selects goals within the plan to work on. Secondly it attempts to achieve the selected goals via plan refinements. Lastly, it records the consequences of the plan refinements via constraints and checks for plan consistency with respect to the constraints. The unified planning framework supports two primitive types of constraints. These are *interval preservation constraints* (IPCs) and *point truth constraints* (PTCs):

- An IPC takes the form $\langle t_i, c, t_j \rangle$ where c denotes a condition that is to be preserved at all points between steps t_i and t_j within the plan.
- A PTC takes the form $\langle c@t \rangle$ where c denotes a condition that is to be true immediately before step t within the plan.

Kambhampati et al [KKY95] claim that these constraint types, correspond to the use of constraints used within the majority of planners. Note that in §4 we consider the instantiation of the unified planning framework for our synthesis problem.

3 The synthesis problem

It is recognized that the activity of programming not only concerns the satisfaction of goals but also requires the management of interactions between the achievement of several goals:

> *"Often the specification of a program will require the simultaneous satisfaction of more than one goal. ..., the special interest of this problem lies in the interrelatedness*

Algorithm: Refine-Plan($\mathcal{P}$)
Parameters: `pick-open`: a routine for picking open goals.
`pre-order`: a routine which adds orderings to the plan to make conflict resolution tractable.
`conflict-resolve:` a routine which resolves conflicts with auxiliary constraints.

1. **Goal selection:** Using `pick-open`, choose an open-goal from the agenda associated with plan $\mathcal{P}$, *i.e.* $\langle C@t \rangle$ where C is the precondition associated with step t of plan $\mathcal{P}$. *Not a backtrack point.*

2. **Goal establishment:** Non-deterministically select a new or existing plan step t' to achieve $\langle C@t \rangle$. Introduce constraints that record i) $t' \prec t$, ii) t' has the effect C, and iii) C is preserved by all steps between t' and t. *Backtrack point.*

3. **Book keeping:** Add auxiliary constraints to ensure that goal establishment steps are protected by later refinements.

4. **Tractability refinements:** (Optional)

   (a) **Pre-ordering:** impose additional orderings between every pair of steps of the partial plan (`pre-order`).

   (b) **Conflict resolution:** add orderings and bindings to resolve conflicts between plan steps and auxiliary constraints (`conflict-resolve`).

5. **Consistency check:** If the refinement of plan $\mathcal{P}$ is consistent then return the refined plan else prune the refined plan.

Note that the order in which goals are selected does not determine the order in which goals are achieved, i.e. *does not affect the ordering of actions within the plan. Consequently, goal selection is not a backtrack point within the algorithm. Kambhampati et al [KKY95] identify two types of pre-ordering strategies,* total ordering *and* unambiguous *ordering. The total ordering strategy orders every pair of steps in the plan, while unambiguous ordering strategy only orders pairs when two steps interact. During conflict resolution the plan is refined until all conflicts are resolved.*

Figure 4: Kambhampati's Generic Refinement Algorithm.

of the goals."

Manna and Waldinger [MW77, Chapter 3]

As a consequence, the synthesis problem becomes two problems: Firstly, the goal satisfaction problem, which is concerned with how to achieve a goal; secondly, the simultaneous goal problem, which is concerned with how to interleave the achievement of one goal with the achievement of other goals in such a way as to preserve the original achievement. Below we analyze each problem separately.

3.1 The goal satisfaction problem

Satisfying a goal in deductive program synthesis involves finding a sequence of operators which achieve the desired result. Gries [Gri81, Principle 14.4] observes that *Programming is a goal-oriented activity*. By this he means the best place to start, when trying to satisfy a goal, is the goal itself. Since we are concerned with finding not only a program, but also a proof of its correctness, we use the Floyd/Hoare proof rules in a backward manner to transform the goal. The key problem, however, is in deciding which proof rule to use and how it should be applied. For instance, we may decide to choose an assignment over a conditional statement and then choose what expression to assign to which variable. Such search is not feasible without the use of declarative heuristics to control the application of the proof rules and procedural heuristics to order which declarative heuristics to try before others.

In terms of declarative heuristics, we exploit common problem-solving heuristics, such as: *Did you use all the data?* [Pol45]. Within the context of satisfying a postcondition, this corresponds to bridging the gap between pre- and postconditions. This can be achieved by exploring the knowledge encoded within the preconditions about the initial values of variables. We can also use other postconditions to give the values of variables and then arrange it so that the planning mechanism protects values at the right point. Other declarative heuristics may involve reasoning directly about object-level terms. For instance, can we express a term in the target programming language? Examples of procedural heuristics include attempting the least expensive program operators first, and arranging that certain problems are only attempted using certain declarative heuristics.

3.2 The simultaneous goal problem

In [Wal77], the simultaneous goal problem is expressed succinctly as follows:

> "*It is often easier to achieve either of two goals than it is to achieve both at the same time. In the course of achieving the second goal we may undo the effects of achieving the first.*"

Ernst and Newell's GPS system [EN69] was unable to find a solution for swapping the contents of two registers without help. Its failure was due to its search heuristic (means end analysis), which reduced differences one by one, *i.e.* it tried to achieve goals independently. Sussman's anomaly [Sus75] is another example of a problem which requires a solution to interleave goal achievement, *i.e.* constructing an ordered stack of blocks can lead to problems if the individual stacking operations are planned in isolation. The problem of swapping the value of two registers is easily represented in a Floyd/Hoare style. It corresponds to the problem of swapping the values of two program variables, *e.g.*

$$\{x = x0 \wedge y = y0\}C\{x = y0 \wedge y = x0\}$$

Where C is a meta-variable, a place-holder for the program we wish to synthesize. As a consequence, we must take account of the potential for interaction between goals, *i.e.* conjuncts in the program's postcondition. To deal with this problem, we have to use a search procedure which opens up the space of solutions to include a solution to the problem, *i.e.* one that does not achieve goals independently.

4 Overall approach

To integrate program synthesis and program proof our approach combines the existing proof planning mechanism (see §2.2) with a partial order planning capability. This is achieved by instantiating the unified planning framework presented in §2.5. The details of the instantiation are described below.

4.1 Proof tactics

Gordon [Gor89] shows how it is possible to embed the semantics of a simple programming language into a higher order logic using Floyd/Hoare axiomatic semantics. Gordon uses HOL [Gor88a], a tactic-based theorem prover, to do this. By embedding the proof rules and axioms into a theorem prover he has shown that not only are the semantics sound, but that any proof developed using these rules is guaranteed to be correct. Gordon shows how it is possible to construct interactive verification proofs of programs by mechanizing these proof rules as tactics. It is these tactics which our meta-level partially describes.

4.2 Planning methods

Within our combined approach, a planning action corresponds to an extension of the proof planning method given in §2.2. The method schema is extended to include program statements and specifications. The details of the extended method schema are presented in Figure 5. Note that within proof planning a precondition describes the heuristic constraints on the applicability of tactics. To avoid confusion we call these *heuristic preconditions* and use *state preconditions* to refer to constraints on program specifications. A method is selected if both the associated state and heuristic preconditions hold for a given goal condition. Once selected, the method's effects are executed to generate the add and delete list elements which are used in the construct of the state postconditions. An example method is given in Figure 6.

4.3 Planning critics

In tackling the synthesis task we use both proof critics and planning critics. While proof critics are used to patch failed proof attempts, planning critics are applied in order to resolve plan inconsistencies. We delay further discussion until §5.2, after the details on the overall planning mechanism have been described.

4.4 Planning process

While the structured programming literature [Dij76, Gri81, Kal90] is full of heuristics for achieving isolated goals, it provides little guidance as to how to treat simultaneous goals. This is where we exploit partial order planning techniques. As mentioned above, we use a generic refinement search algorithm. In particular, we build upon a partial order planner generated from this generic search algorithm and described in [KKY95]. This provides a foundation for combining proof planning with partial order planning.

We now consider the instantiation of the unified planning framework for the program synthesis task. Given a partial correctness specification of the form:

$$\{P\}C\{Q\}$$

then the initial planning state $\mathcal{I}$ corresponds to $\{P\}C\{Q\}$ while $\mathcal{G}$, the desired final state, corresponds to $\{P\}\{P\}$. Note that C denotes a meta-variable, a place-holder for the program that is to be synthesized. Each step within a plan corresponds to a method instance. To initialise the plan we introduce method instances corresponding to the dummy operators $start$ and $finish$. These

```
method(   Name(Args),
          State Preconditions,
          Heuristic Preconditions,
          Effects,
          State Postconditions)
```

Where:

- `Name:` Method identifier.
- `Args:` A pair, where the first component is a program statement and the second component is a proof tactic. Note that if a method corresponds to a purely logical deduction then the program component will be empty, where empty is denoted by `noop`.
- `State Preconditions:` Preconditions relating to the partial correctness specification of the program being synthesized.
- `Heuristic Preconditions:` Auxiliary preconditions, often referred to as filter conditions in the planning literature.
- `Effects:` Code which is used to calculate the effects of applying the method within the current planning context.
- `State Postconditions:` An description of how the application of the method changes the partial correctness specification of the program being synthesized. The description is represented as an add-delete list, *i.e.* conditions that are being adding are prefixed by the $\oplus$ operator while conditions that are being deleted are prefixed by the $\ominus$ operator.

  Note that the program component of `Args`, `State Preconditions` *and the* `State Postconditions` *represent extensions to the original proof method schema.*

Figure 5: Extended Proof Method Schema

$$
\begin{array}{ll}
method(& assign(V := E, ASS\_TAC(V, E), \\
& [post(V = E)], \\
& [prog\_var(V), \\
& \ expressible(E)], \\
& [find\_wlp\_ass(V = E, V, E, WLP)], \\
& [\oplus(post(WLP)), \ominus(post(V = E))] \\
). &
\end{array}
$$

Where:

- $V := E$: Program assignment statement, *i.e.* the variable V is assigned the value of the expression E.
- $ASS\_TAC(V, E)$: Tactic corresponding to the assignment axiom.
- $post(C)$: C is a condition that occurs within the postcondition of a partial correctness specification.
- $prog\_var(V)$: is true if V denotes a program variable.
- $expressible(E)$: is true if E denotes an expression that can be expressed directly within the programming language (see §5.1).
- $find\_wlp\_ass(Q, V, Exp, WLP)$: Weakest liberal precondition calculation, *i.e.* replace every V in Q with Exp to give WLP.

Figure 6: An `assign` Method.

are used to introduce the initial and final states into the plan, *i.e.* the $start$ method introduces the initial state through its associated state postcondition slot:

$$\{\oplus(pre(P)), \oplus(post(Q))\}$$

The $finish$ method introduces the final state through its state precondition slot:

$$\{\oplus(pre(P)), \oplus(post(P))\}$$

For each plan, an agenda of open-goals is maintained, *i.e.* a set of terms of the form $post(G)@t$ where G denotes a goal condition and t is the step within the plan at which the condition appears. The initial planning node corresponds to the 5-tuple described in §2.5, with the exception that initial values are assigned to $\mathcal{B}$, the binding constraints, and $\mathcal{L}$, the auxiliary constraints. In the case of $\mathcal{B}$, the initial value is the meta-variable C, a place-holder for the program that is to be synthesized. The auxiliary constraints $\mathcal{L}$ are initialized with PTCs corresponding to the state preconditions associated with the $finish$ method.

In terms of the generic algorithms, the instantiation of the `Refine-Plan` routine (see Figure 4) requires the most work. Below we highlight how `Refine-Plan` is instantiated for the program synthesis task:

1. **Goal selection:** a goal is selected from the plan agenda, *i.e.* $\langle post(g)@t \rangle$.
2. **Goal establishment:** Select a method m such that:
   (a) $post(g)$ is described in m's state preconditions P.
   (b) P (partially) describes the state after t.

Check the heuristic preconditions associated with m (backtrack point). If the heuristic preconditions fail then apply proof critics associated with m (backtrack point). If the heuristic preconditions are true then execute the corresponding effects to construct the add and delete list elements for the state postconditions (not a backtrack point). Add a new step t_{new} to the plan (corresponding to the instance of m) such that $t \prec t_{new}$.

3. **Book keeping:** For every state precondition P of the new method m add the IPC $\langle t, P, t_{new} \rangle$ and the PTC $\langle P@t_{new} \rangle$. For every state postcondition Q of method m, such that there exists an auxiliary PTC $\langle Q@t_{after} \rangle$ and $t_{new} \prec t_{after}$, then add the IPC $\langle t_{new}, Q, t_{after} \rangle$.

4. **Tractability refinements:**

   (a) **Pre-ordering:** impose a total ordering on plan steps that is consistent with the ordering and auxiliary constraints.

   (b) **Conflict resolution:** if no consistent total ordering exists, but there exists an applicable planning critic, then use the critic weaken method m. Undo and rerun the book keeping and tractability refinements.

5. **Consistency check:** If a consistent total ordering exists then return the refined partial plan else prune the refined partial plan.

Method selection is based upon a *waterfall* approach, similar to that used within the Boyer-Moore theorem prover [BM79]. The methods are ordered so that simpler, more immediate, methods proceed the more complex methods. Where only partial success of a method is achieved then proof critics are used to guide proof patching. The details of method ordering and proof critics are presented in §5.

To illustrate the instantiation of the planning algorithm, consider the following specification:

$$\{true\}C\{x = 0 \wedge y = 0\}$$

This corresponds to the initial planning state and is represented by the *start* method's state postconditions as follows:

$$\{\ \oplus(post(x = 0)), \oplus(post(y = 0))\ \}$$

The final state corresponds to:

$$\{true\}\{true\}$$

and is represented by $\{\}$ through the state preconditions associated with the $finish$ method. As a consequence, the set of auxiliary constraints is also empty. The plan agenda records programming goals and is initialized with the specification postconditions, *i.e.*

$$\mathcal{A} : \{\langle goal(x = 0)@t_I \rangle, \langle goal(y = 0)@t_I \rangle\}$$

Now suppose the planner selects the goal on the agenda, *i.e.* $x = 0$ at step t_I. The planner will then search for a method M such that i) $post(x = 0)$ is contained within the state preconditions of M, and ii) the state preconditions of M (partially) describe the state postcondition corresponding to t_I. The `assign` method given in Figure 6 meets these conditions and results in the introduction of the assignment $x := 0$. The planner then adds ordering constraints:

$$t_I \prec t_1 \prec t_G$$

and a binding constraint:

$$C = (C_1; x := 0; C_2)$$

where C_1 and C_2 denote new meta-variables. In addition, auxiliary constraints are introduced, *i.e.* $\langle post(x = 0)@t_1 \rangle$ and $\langle t_I, post(x = 0), t_1 \rangle$. These ensure that the goal condition $x = 0$ is

preserved until its achievement, *i.e.* no step in between t_I and t_1 can affect the postcondition $x = 0$. Then new plan agenda becomes:

$$\mathcal{A} : \{\langle goal(y = 0)@t_I\rangle, \langle goal(0 = 0)@t_1\rangle\}$$

Note that the first goal on the agenda is achieved by the assignment $y := 0$ while the remaining goals are trivial and are achieved without the introduction of additional program constructs. The complete plan gives rise to the following program:

$$\{true\}\ y := 0; x := 0\ \{x = 0 \wedge y = 0\}$$

5 Overview of the proof methods and critics

Below we give a brief description of the methods developed for imperative program synthesis together with how they are organized:

`elementary:` attempts to prove that a postcondition can be derived from the preconditions and other postconditions using basic properties and axioms, *e.g.* $odd(x) \equiv \neg even(x)$, and simple equality/inequality reasoning.

`trans:` controls the rewriting of a postcondition by ensuring a decrease in the number of expressions in the postcondition which cannot be expressed within the programming language.

`assign:` introduces an assignment, where the variable and expression are chosen using the values of variables in the pre- and postconditions.

`eval:` assigns values to variables using definitions of properties. To do this we have to be sure that the variable is not constrained to take a conflicting value at that point in the plan.

`if:` introduces a conditional statement by exploiting a disjunction in the postcondition if a guard can be chosen and both auxiliary branches of the proof rule can be proof-planned.

`while:` a loop construct is introduced after asking the user to supply an invariant and guard, and after three heuristic preconditions are satisfied. These heuristic preconditions are discussed in §5.1.2.

The following three methods are used as sub-methods to the `while` method:

`wave:` provides the basis for rippling within postconditions.

`fertilize:` guides the application of hypotheses.

`casesplit:` guides case splitting.

The ordering of methods presented above reflects the ordering in which methods are selected. This means that simpler program constructs are explored before the more complex ones. For instance, we only attempt to construct a loop if we cannot find a sequence of assignments or conditional statements which satisfies a goal condition. Note that both the `if` and `while` methods have auxiliary calls to the planner, enabling the system to synthesize nested constructs.

5.1 Synthesis heuristics

Below we give examples of the kinds of heuristic preconditions which are embodied within our methods:

Expressibility: An important meta-logical concept within our work is the *expressibility* of a term. We say that a term is *expressible* if we can use it in the programming language. It is *inexpressible* if we cannot. For example, $x+1$ and $r*y$ are expressible since x, r, y are program variables and addition and multiplication are allowed in our programming language. However $\sum_{m=0}^{x0} m$ is inexpressible since neither $\sum$ nor $x0$ are allowed in our programming language. The notion of expressibility allows the heuristics a degree of generality and object language independence.

Positive effects: As programmers we are interested in both efficiency and conciseness. A program can be made both more efficient and more concise by making use of variables to hold previous calculations. This corresponds to using already established goals. Unfortunately, the planning algorithm only reasons about the harmful effects of simultaneous goal achievement. To make use of already established goals we must reason about the potential positive effects that an operation may have on a goal. By defining the *positive effects*, which one goal may have on other goals, we use the meta-level to reason about notions of conciseness and efficiency. Such a definition can only ever describe heuristically the kinds of programs which are preferred. The intuition behind our definition is as follows: Achieving a goal g with other simultaneous goals can sometimes be made easier if one of these simultaneous goals was achieved before g. Achieving this goal first may achieve a part of g, so we only have to achieve the remaining unachieved parts of g.

At the meta-level we cannot reason directly about the order of goal achievement. This is the job of the planner. We use a rewriting strategy, changing the goal we wish to achieve by replacing the already achieved parts. For example, consider the goals $i = i0 + 1$ and $r = (i0+1)+r0$. The planner allows the achievement of these two goals separately, forcing the assignment of r to occur before the assignment to i, yielding the possible solution:

$$\begin{array}{l}\{i = i0 \wedge r = r0\}\\ r := (i + 1) + r;\\ i := i + 1\\ \{i = i0 + 1 \wedge r = (i0 + 1) + r0\}\end{array}$$

However, there is a subtle interaction between the two goals. Both goals share the value $i0 + 1$ and furthermore i is equal to it. We say that the goal $i = i0 + 1$ has a positive effect on the goal $r = (i0 + 1) + r0$. If we firstly choose to achieve $i = i0 + 1$ then we make use of it when achieving $r = (i0 + 1) + r0$, using the value of i. This means that instead of achieving $r = (i0+1)+r0$, we choose to achieve $r = i+r0$ and indicate to the planner the preference that $i = i0 + 1$ should be achieved first. In doing so we have used our structured programming knowledge of how a goal can be affected by an assignment. In this example, by observing the positive effect, we gain the solution:

$$\begin{array}{l}\{i = i0 \wedge r = r0\}\\ i := i + 1;\\ r := i + r\\ \{i = i0 + 1 \wedge r = (i0 + 1) + r0\}\end{array}$$

Such a solution is clearly preferred.

We now consider in more detail the heuristics associated with the `assignment` and `while` methods.

5.1.1 The assignment method

Given a postcondition of the form $V = Exp$ the `assign` method has two cases. The first case is applicable when Exp is expressible in the programming language. If Exp is expressible and V is a program variable then the assignment of Exp to V is introduced. This corresponds to Figure 6

and was illustrated in §4.4. The second case is applicable when Exp cannot be expressed in the programming language. The assignment method attempts to find subexpressions of Exp which are equal to variables in the pre- and postconditions. Then the method introduces a new variable to any (proper) subexpression, which is still inexpressible, adding an extra subgoal that equates this new variable to the inexpressible subexpression. For example, consider the specification:

$$\{x = x0\}C\{r = x0 + \textstyle\sum_{m=0}^{x0} m\}$$

Note that $x0$ and $\sum_{m=0}^{x0} m$ are both inexpressible in the programming language. Since $x = x0$ appears in the precondition we can replace the first $x0$ with x. Because $\sum_{m=0}^{x0} m$ is still inexpressible we replace it with a new variable v. This gives the following sub-specification:

$$\begin{array}{l}\{x = x0\}\\ C';\\ \{x + v = x0 + \sum_{m=0}^{x0} m \wedge v = \sum_{m=0}^{x0} m\}\\ r := x + v\\ \{r = x0 + \sum_{m=0}^{x0} m\}\end{array}$$

The aim of this heuristic is to reduce the complexity of such equational conditions, a kind of divide and conquer strategy.

5.1.2 The while-loop method

The most complex method is the `while` method. Recall that the proof rule for a while-loop is:

$$\frac{(I \wedge \neg B) \rightarrow Q \quad \{I \wedge B\}S\{I\}}{\{I\}\textbf{while } B \textbf{ do } invariant(I)\ S \textbf{ end}\{Q\}}$$

Gries provides an abstraction pattern for developing a loop [Gri81, Strategy 15.1.4][1]:

> **Strategy for developing a loop:** Given an invariant (I) and a guard (B) such that the invariant and the negated guard imply the postcondition (Q), *i.e.* $(I \wedge \neg B) \rightarrow Q$; then develop the loop body (S) so that it reestablishes the loop invariant.

We represent Gries' strategy as heuristic preconditions to the `while` method. Note that middle-out reasoning plays a crucial role in the evaluation of these preconditions as described below:
Heuristic preconditions to the `while` method:

1. Given an invariant I and negated guard $\neg B$ we can establish $(I \wedge \neg B) \rightarrow Q$.
2. There exists a weakest liberal precondition WLP of the loop body such that verification condition $(I \wedge B) \rightarrow WLP$ holds.
3. There exists a loop body S which satisfies $\{WLP\}S\{I\}$.

The first heuristic precondition uses the `elementary` and `trans` methods to provide a proof that the invariant and negated guard imply the postcondition. The second heuristic precondition builds upon rippling and middle-out reasoning. A middle-out version of the proof illustrated in §2.3 is attempted. If the attempt is successful then we are left in the paradoxical situation of knowing the weakest liberal precondition of a loop body, and an invariant, without knowing the structure of the loop body. However, using the weakest liberal precondition, an auxiliary specification of the loop body can be generated. The third precondition succeeds if the auxiliary specification can be achieved. We now explore the details of the second and third heuristic preconditions.

In the domain of imperative program synthesis, we are unable to prove the verification conditions without the weakest liberal precondition of the invariant and the loop body. We would like, however, to apply Gries' strategy where the proof leads the way. As mentioned above, to achieve

[1] Here we focus on a partial-correctness version.

this we use middle-out reasoning to determine the weakest liberal precondition of the loop body. This involves creating a schematic goal known as the *synthesis condition* which represents the counterpart of a verification condition. They share the same hypothesis, *i.e.* the loop invariant and guard, but the conclusion of a synthesis condition comprises the invariant with every program variable replaced by a first order meta-variable. Such a conclusion speculates the structure of the loop's weakest liberal precondition. For example, consider the invariant $r = \sum_{m=0}^{i} m$ and the guard $i \neq x$. The corresponding synthesis condition is:

$$(r = \sum_{m=0}^{i} m \wedge i \neq x) \rightarrow R = \sum_{m=0}^{I} m$$

where I and R are first-order meta-variables. We use an extended version of the verification condition proof plan to guide the search for instantiations for I and R. The proof plan consists of the waterfall of methods: `elementary`, `casesplit`, `fertilize` and `wave`. The `casesplit` method deals with disjunctive postconditions. Disjunctions suggest that a conditional construct may be needed in the body of the loop. We treat each disjunct separately by standardizing the first-order meta-variables apart. This means that we give new names to the variables so that the disjuncts are treated separately. Conditional rewrite rules suggest the need for a disjunction and we use a proof critic mechanism [IB96b] to transform the plan in a similar manner to Smaill and Green [SG95]. The ripple proof of the above synthesis condition is as follows:

$(r = \sum_{m=0}^{i} m \wedge i \neq x) \rightarrow \quad R = \sum_{m=0}^{I} m$

ripple using (2)
where $\{I \mapsto I' + 1\}$

$(r = \sum_{m=0}^{i} m \wedge i \neq x) \rightarrow \quad R = (I' + 1) + (\sum_{m=0}^{I'} m)$

fertilize
where $\{I' \mapsto i\}$

$(r = \sum_{m=0}^{i} m \wedge i \neq x) \rightarrow \quad R = (i + 1) + r$

elementary
where $\{R \mapsto (i + 1) + r\}$

$(r = \sum_{m=0}^{i} m \wedge i \neq x) \rightarrow \quad (i + 1) + r = (i + 1) + r$

Note that I and R are instantiated to be $i + 1$ and $(i + 1) + r$ respectively. Note also that this proof is similar to the verification condition proof in §2.3, *i.e.* we have proved the verification condition resulting from the loop body. From this we extract the differences added to each variable in order to engineer an auxiliary specification. The program synthesized using this auxiliary specification is used for the loop body. We have made certain assumptions which may lose us correctness at the meta-level. Tactic execution, however, which involves no search, provides an object-level correctness check. This property of proof planning is very appealing as it allows us to experiment with heuristics while still having a guarantee of correctness.

The auxiliary specification is constructed by equating all the program variables within the invariant to initial values in the precondition. The postcondition equates the program variables to terms derived from differences extracted from the weakest liberal precondition calculation. These terms are derived by replacing every program variable appearing in the difference with its

corresponding initial value. For example, the auxiliary specification for the example above is:

$$\{i = i0 \wedge r = r0\}C\{i = i0 + 1 \wedge r = (i0 + 1) + r0\}$$

This auxiliary specification is passed to the planning mechanism using the top-level-waterfall. The planner then constructs a program for the loop body. For example, the auxiliary specification above can be achieved by the two assignments:

$$\begin{array}{l}\{i = i0 \wedge r = r0\}\\ i := i + 1;\\ r := i + r\\ \{i = i0 + 1 \wedge r = (i0 + 1) + r0\}\end{array}$$

The program synthesized so far is:

$$\begin{array}{l}\{x = x0\}\\ C';\\ \{r = \sum_{m=0}^{i} m\}\\ \textbf{while not } (i = x)\\ \quad \textbf{begin}\\ \qquad i := i + 1;\\ \qquad r := i + r\\ \quad \textbf{end}\\ \{r = \sum_{m=0}^{x0} m\}\end{array}$$

We finish the synthesis by initializing the loop invariant using the `eval`, `assign` and `elementary` methods. The final program is:

$$\begin{array}{l}\{x = x0\}\\ i := 0;\\ r := i;\\ \{r = \sum_{m=0}^{i} m\}\\ \textbf{while not } (i = x)\\ \quad \textbf{begin}\\ \qquad i := i + 1;\\ \qquad r := i + r\\ \quad \textbf{end}\\ \{r = \sum_{m=0}^{x0} m\}\end{array}$$

Note that the `while` method makes use of heuristics developed for loop invariant verification. It uses the space of possible verification condition proofs to constrain the synthesis of the code. The above synthesis example is semi-automatic. The user has to supply an invariant and guard when prompted to do so by the system. However, the system decides when a loop is needed. When asked to supply an invariant, the user is always given the postcondition which is being weakened as a guide, *i.e.* the user does not have to specify an invariant with the original specification. Note that alternative invariants may cause different code to be synthesized. Although the search problem is much more difficult than its verification counterpart, proof planning and the constraints of rippling make synthesis feasible.

5.2 Proof and planning critics

As mentioned earlier, proof critics are used to patch failed proof attempts while planning critics resolve conflicts with respect to the plan's constraint set. Our synthesis approach uses two proof critics and one planning critic as described below:

`trans:` a proof critic motivated by Gries' strategy for developing an alternative command [Gri81, (14.7)]. Failure to prove a condition attached to a rewrite rule is used to transform the associated postcondition into a disjunction. Note that a disjunctive postcondition allows for a casesplit using the `if` method.

`wave:` a proof critic which introduces casesplits based upon failure within the context of a ripple proof. While triggered by the failure of the `wave` method, it has strong similarities with the `trans` casesplit critic.

`assign:` a planning critic which triggers when a step threatens a property introduced by an `assign` method relying on the value of a program precondition. The `assign` method is weakened by assigning the same variable to a temporary variable, rather than the expression to which it was previously assigned.

In order to illustrate the `assign` planning critic, we return to the swap example discussed in §3.2. Recall that the specification takes the form:

$$\{x = x0 \wedge y = y0\}C\{x = y0 \wedge y = x0\}$$

The agenda corresponding to the initial plan node takes the form:

$$\mathcal{A} : \{\langle goal(x = y_0)@t_I\rangle, \langle goal(y = x_0)@t_I\rangle\}$$

Assuming the first goal condition is selected, *i.e.* $x = y_0$, then the `assign` method would introduce the assignment $x := y$ (see §5.1.1). The achievement of this goal condition would introduce auxiliary constraints that include the following IPCs:

$$\{\langle t_I, pre(x = x0), t_1\rangle, \langle t_I, pre(y = y0), t_1\rangle, \langle t_I, post(x = y0), t_1\rangle, \langle t_1, post(y = y0), t_G\rangle\}$$

where t_1 denotes the node corresponding to the application of the `assign` method. In terms of ordering constraints, the following is recorded:

$$t_I \prec t_1 \prec t_G$$

and the binding constraint becomes:

$$C = (C_1; x := y; C_2)$$

The updated plan agenda is now:

$$\mathcal{A} : \{\langle goal(y = x_0)@t_I\rangle, \langle goal(y = y_0)@t_1\rangle\}$$

Again assuming the first goal condition is selected then the `assign` method would introduce the assignment $y := x$, and give rise to the following auxiliary constraints:

$$\{\langle t_I, pre(x = x0), t_2\rangle, \langle t_I, pre(y = y0), t_2\rangle, \langle t_I, post(y = x0), t_2\rangle, \langle t_2, post(x = x0), t_G\rangle\}$$

where t_2 denotes the node corresponding to the second application of the `assign` method. The corresponding ordering and binding constraints take the form:

$$t_I \prec t_2 \prec t_G$$

and

$$C = (C_3; y := x; C_4)$$

respectively. Now the planner attempts to impose a total order on the plan steps. There are two possible orderings to consider. Each ordering needs to be checked for consistency:

- $t_I \prec t_1 \prec t_2 \prec t_G$: To check whether this ordering is consistent the planner checks that no IPCs are violated. Since t_2 updates y the IPC $\langle t_1, post(y = y0), t_G\rangle$ would be violated by this ordering.
- $t_I \prec t_2 \prec t_1 \prec t_G$: To check whether this ordering is consistent the planner checks that no IPCs are violated. Since t_1 updates x the IPC $\langle t_2, post(x = x0), t_G\rangle$ would be violated by this ordering.

Note that the consistency checking relies upon the weakest liberal precondition calculation [Dij76], and the method state postconditions, *i.e.* the add-delete lists.

Although a consistent ordering is not possible, the planner forces an ordering and then attempts to resolve the conflict. Let us suppose that its chooses ordering $t_I \prec t_2 \prec t_1 \prec t_G$. Since t_1 violates the property $x = x0$ which is required for t_2, a planning critic weakens the form of assignment so that y is not assigned directly to x. That is, the critic introduces an auxiliary variable into the goal condition with the expectation that we can drag this new goal condition back through the conflicting assignment, and so make use of the precondition as required. After the critic has executed the binding constraint becomes:

$$C = (C_1; y := x; y := temp)$$

and $\langle goal(temp = x0)@t_2 \rangle$ is added to the plan agenda. The achievement of this goal condition results in the following binding constraint:

$$C = (temp := x; y := x; y := temp)$$

This completes the synthesis for the swap example.

6 Implementation

The ideas described above are implemented within a Prolog-based system called BERTHA. The BERTHA system consists of a planning mechanism, pretty printing utilities, definitions of meta-logical terms and a library containing a set of methods, critics, definitions, lemmas and example specifications. BERTHA is built directly on the shell of the *Clam* proof planning system [BvHHS90]. However, BERTHA is a significant extension to the *Clam* proof planner. Only the core of *Clam* is used to provide a representation for terms and other meta-level constructs. Method and critic data structures have been extended to deal with partial descriptions of state. Our methods are general enough to allow arbitrary combinations and nesting of constructs. BERTHA shows the success of the meta-level heuristics and planning mechanism. It is easy enough to replace the heuristics by writing new methods. Therefore, given new methods we would be able to compare the benefits of different heuristics.

To illustrate BERTHA, a session is given below. The output presented is taken directly from BERTHA, with only certain details abstracted for a clearer presentation. Note that in BERTHA a wave-front $\boxed{x}+1$ is expressed as `''{x}+1''<out>` while conjunction and disjunction are denoted by `#` and `/` respectively. In addition, we use `sum(X)` to denote $\sum_{m=0}^{X} m$. In this example, BERTHA is required to synthesize a program which sums the first $x0$ natural numbers. Initially BERTHA is given the specification:

$$\{x = x0\}C\{r = \textstyle\sum_{m=0}^{x0} m\}$$

We assume that any definitions and lemmas, which are required to complete the proof, are included in a background theory, which has already been loaded into BERTHA. We start the synthesis attempt:

```
|-? poplan(sum).
```

BERTHA then shows the specification of the problem.

```
Specification:
{x=x0}
{r=sum(x0)}
```

BERTHA searches for methods which are applicable. As no other methods are applicable BERTHA attempts the `while` method. Next BERTHA shows the current goal $r = \sum_{m=0}^{x0} m$ and asks the user to supply an invariant and guard to use when synthesizing a loop.

```
Postcondition: r=sum(x0)
Choose Invariant:
 |: r=sum(i).
Choose Guard:
 |: not(i=x).
```

In this case the user supplies the invariant $r = \sum_{m=0}^{i} m$ and the guard $i \neq x$. If the user does not want to apply the `while` method they can cause this precondition to fail by typing `fail` when asked to supply an invariant. The heuristic preconditions associated with the `while` method are then evaluated. The first precondition attempts to show that the given invariant and guard provide a valid weakening, *i.e.*

$$(r = \sum_{m=0}^{i} m \wedge i = x \wedge x = x0) \rightarrow r = \sum_{m=0}^{x0} m$$

This is achieved by BERTHA using the elementary method:

```
Planning Weakening
 Last method fired: elementary(...)
PROGRAM:
{r=sum(i)#not(not(i=x))#x=x0}
{r=sum(x0)}
--------Solution Found--------
```

Note that the elementary method makes use of the precondition $x = x0$. Since the precondition is used it is passed to the planning mechanism to be used as a constraint; *i.e.* x must take the value $x0$ at all times between the precondition and the end of the loop being synthesized. The planning mechanism is responsible for the reasoning required to keep this constraint.

The second heuristic precondition of the `while` method involves synthesizing the verification condition proof involving the loop invariant. This is done by setting up an appropriate synthesis condition, which is achieved using the ripple method:

```
Planning Synthesis Condition
 Last method fired: wave(sum1)
PROGRAM:
{ r=sum(i)# not(i=x)}
{R=sum(‘‘{I}+1’’<out>)}
 Last method fired: fertilize(...)
PROGRAM:
{r=sum(i)# not(i=x)}
{R=sum(‘‘{i}+1’’<out>)}
 Last method fired: elementary(...)
PROGRAM:
{r=sum(i)# not(i=x)}
{( i+1)+r=sum(‘‘{i}+1’’<out>)}
--------Solution Found--------
```

The above transcript from BERTHA shows how the original postcondition is incrementally instantiated as a side-effect of the ripple guided proof. The associated ripple proof is given in §5.1.2. Note that `sum1` corresponds to wave-rule (2).

Next an auxiliary specification to the loop body is generated using information from the proof of the synthesis condition. The third heuristic precondition succeeds if the auxiliary specification can be achieved. Here the auxiliary specification is achieved using two assignments:

```
Planning Body
 Last method fired: ass1(i,i+1)
PROGRAM:
{x=x0#i=iv1temp0#r=rv2temp0}
     i:=i+1
{i=iv1temp0+1#r=(iv1temp0+1)+rv2temp0}
 Last method fired: elementary(...)
PROGRAM:
{x=x0#i=iv1temp0#r=rv2temp0}
     i:=i+1
{i=iv1temp0+1#r=(iv1temp0+1)+rv2temp0)}
 Last method fired: ass1(r,i+r)
PROGRAM:
{x=x0#i=iv1temp0#r=rv2temp0}
     i:=i+1;
     r:=i+r
{i=iv1temp0+1#r=(iv1temp0+1)+rv2temp0}
 Last method fired: elementary(...)
PROGRAM:
{x=x0#i=iv1temp0#r=rv2temp0}
     i:=i+1;
     r:=i+r
{i=iv1temp0+1#r=(iv1temp0+1)+rv2temp0}
--------Solution Found--------
```

Note that the goal:

$$i = iv1temp0 + 1$$

has a positive effect on the goal:

$$r = (iv1temp0 + 1) + rv2temp0$$

The assignment method reasons about the appropriate assignment needed and leaves the planner to reason about the order in which they should be attempted to achieve the conjunctive goal. Having satisfied these three preconditions, BERTHA is able to fire the `while` method, synthesizing a loop:

```
Planning Termination
 Last method fired: while(...)
PROGRAM:
{x=x0}
     while(not i=x) do
      begin
       {r=sum(i)}
       i:=i+1;
       r:=i+r
      end
{r=sum(x0)}
```

After the loop there remains a subgoal to prove:

$$\{x = x0\}C'\{r = sum(i)\}$$

i.e. we must initialize the loop invariant. Doing so will synthesize the code which will initialize the loop. BERTHA chooses to evaluate this goal, by choosing values for i and r. It first uses the definition of $\sum$ to do this.

```
 Last method fired: eval(sum0)
PROGRAM:
{x=x0}
     while(not i=x) do
      begin
       {r=sum(i)}
       i:=i+1;
       r:=i+r
      end
{r=sum(x0)}
```

This results in assigning values of 0 and i to the program variables i and r.

```
 Last method fired: ass1(r,i)
PROGRAM:
{x=x0}
     r:=i;
     while(not i=x) do
      begin
       {r=sum(i)}
       i:=i+1;
       r:=i+r
      end
{r=sum(x0)}
 Last method fired: elementary(...)
...
 Last method fired: ass1(i,0)
PROGRAM:
{x=x0}
     i:=0;
     r:=i;
     while(not i=x) do
      begin
       {r=sum(i)}
       i:=i+1;
       r:=i+r
      end
{r=sum(x0)}
 Last method fired: elementary(...)
...
--------Solution Found--------
```

Having completed the synthesis, BERTHA reports success by displaying the completed program:

```
PROGRAM:
{x=x0}
     i:=0;
     r:=i;
     while(not i=x) do
      begin
       {r=sum(i)}
       i:=i+1;
       r:=i+r
      end
{r=sum(x0)}
```

and presents the proof found in the form of a compound tactic:

```
TACTIC:
SEQ_TAC THENL [
  WHILE_TAC(not(i=x),r=sum(i)) THENL
    WEAKEN_TAC THENL [
      ELEMENTARY_TAC(...),
      SKIP_TAC],
    SEQ_TAC THENL [
      ASS_TAC(r,i+r),
      SEQ_TAC THENL [
        ASS_TAC(i,i+1),
        WEAKEN_TAC THENL [
          REWRITE_TAC(sum1,[2]),
          WEAKEN_TAC THENL [
            FERTILIZE_TAC,
            WEAKEN_TAC THENL [
              ELEMENTARY_TAC(...),
              SKIP_TAC]]]]],
    PARTIAL_CORRECTNESS_TAC]
  WEAKEN_TAC THENL [
    EVAL_TAC(sum0),
    SEQ_TAC THENL [
      ASS_TAC(r,i),
      SEQ_TAC THENL [
        ASS_TAC(i,0),
        WEAKEN_TAC THENL [
          ELEMENTARY_TAC(...),
          WEAKEN_TAC THENL [
            ELEMENTARY_TAC(...),
            SKIP_TAC]]]]]
```

This example highlights three important properties of the BERTHA system and the synthesis proof plan:

1. Apart from asking the user to supply an invariant and guard, the synthesis attempt shown above is fully automatic. BERTHA chooses which methods to apply, how the achievement of goals is ordered, which wave-rules to apply, etc.

2. The synthesis proof plan is designed to be very prescriptive. There was no search in the example above and little search required in general. The search strategy is a simple depth-first backtracking algorithm.

3. Rippling successfully constrains middle-out reasoning in the presence of first-order meta-variables. The only change required was the generalization of method preconditions to

allow first-order meta-variables as wave-terms. Eager fertilization provides control over the potentially non-terminating rewriting.

7 Results

Experimental results with BERTHA are presented in the appendices. In appendix A, specifications and the associated synthesized programs are presented. Appendix B provides all the information from which BERTHA synthesized each program, *e.g.*, specification, definitions, lemmata and invariants. Below the results of the experiments are divided into four classes: each class is discussed in detail.

Simple programs: Examples requiring the synthesis of programs involving either no program constructs or only one assignment are shown in Table 2. Examples `easy1` and `easy2` require no program constructs to be synthesized. Their proofs rely on the elementary method. Examples `one, incx, xoplusxo` and `xplusy` all involve simple reasoning about the values of variables in the pre- and postcondition. The programs synthesized are simple assignments. Examples `dble, square` and `cubed` involve the transformation of the postcondition into an equivalent form using the `trans method` and so rely on a property in the postcondition and various lemmata to find a weaker, but more amenable, specification. The notions of more "expressible" and "embeddedness" enable the postcondition to be transformed until the assignment method can be applied.

Sequences of assignments: Examples requiring the synthesis of programs involving sequences of assignments are presented in Table 3. The first three examples require no temporary variables, since the conjunctive goals can be dealt with separately. The remaining six examples require the use of a temporary variable. BERTHA uses the assignment planning critic when it discovers that it cannot order the achievement of the conjunctive goals. This critic weakens an existing assignment by introducing a temporary variable, so as to delay the conflicting assignment.

Conditionals: Examples requiring the synthesis of programs involving conditionals are presented in Table 4. Note again the use of temporary variables. This highlights that BERTHA can reason about conflicting goals with program constructs other than assignments. The examples `max` and `ifdblecubed` require the transformation of goals. For `max` this transformation process fails since the conditions on the rewrite rules cannot be met. This failure causes the trans critic to fire, which introduces a disjunctive goal, which in turn suggests the application of the `if` method.

While-loops: Examples requiring the synthesis of programs involving loops are presented in Table 5. Note that the examples `exp1` and `exp2` have exactly the same specification. The use of different invariants and the use of the background theory in a different way gives rise to different algorithms, one with an increasing iterator and the other with a decreasing one. The example `sumexp` shows that BERTHA can generate loops within conditional branches, and the `sumodd` example shows that BERTHA can generate conditionals within loops. This example involves the use of the wave casesplit critic to introduce a disjunctive goal. Examples `sumfexp1` and `sumfsum1` introduce nested loops. The user is asked to supply invariants for the loops on two separate occasions. The inner loop is generated when BERTHA attempts to synthesize the auxiliary specification of the outer loop. Example `sumfsum2` has the same specification as `sumfsum1`, but the user supplies a stronger invariant using the principle advocated by Gries in [Gri76]. This results in the synthesis of a more efficient algorithm, *i.e.* by exploiting the relationship between loop variables i and j, the nested loop structure of `sumfsum1` can be replaced by a single loop, as shown in `sumfsum2`.

8 Related work

Manna and Waldinger [MW80] introduce a deductive approach to recursive program synthesis. Proof planning enables us to use a similar approach, but allows us to identify the various types of knowledge needed for such a task. Proof planning has been used for logic [KBB93] and functional program [ASG97, SG95] synthesis. This work extended an established proof plan for induction. For imperative program synthesis we could no longer rely on induction directly and needed to develop a new proof plan and search procedure.

Dershowitz [Der85] gives a set of synthesis rules for the top-down transformation of specification into code. These rules mix planning, heuristic and proof knowledge together, making them hard to understand. There is no deductive component and search issues are ignored. Cheng [Che94] extends Dershowitz's ideas and couples them with a theorem prover. Small pieces of code are synthesized then verified to ensure correctness. Christensen's system [Chr93] first attempts to break a specification down using program constructs, and then proves any verification conditions. Dependencies between variables are reasoned about using an equational reasoning tool. If there are still assignments to be chosen, these are then *guessed* by the system.

Our approach to imperative program synthesis differs from those highlighted above in that we are able to couple the deductive and heuristic components within the proof planning framework. Middle-out reasoning enables us to prove the program correct while it is being synthesized. This provides a powerful, yet understandable approach, which gives provably correct programs.

A distinctive feature of our approach is that we are tackling synthesis from first principles. This raises the issue of scalability. Scalability might be achieved by using our approach to combine software components, *e.g.* iterating over pre-specified library components. This is essentially how AMPHION [LPPU94] achieves scalability, *i.e.* building upon powerful subroutine libraries. AMPHION, however, only supports straight-line code.

An alternative and more semantic approach to program synthesis is achieved through program schemas [FZH98, Smi90, Smi96]. Each program schema provides a template that is filled out during synthesis. Correctness is partially dealt with off-line, once for each program schema. In contrast our approach selects program constructs based on heuristic choice and builds up a correctness argument as the program is synthesized. In [RF03] a unification of proof planning and schema-based synthesis is outlined. This hybrid approach potentially provides greater flexibility compared with conventional schema-based synthesis approaches. In particular, they extend the use proof methods to represent heuristics for guiding the application of program schemas. This is similar to our use of proof methods in representing heuristics for guiding program construction. It would be interesting to investigate how schema-based techniques could be exploited within our approach, in particular to support reuse.

Schema-based approaches typically support the development of functional programs. In terms of the synthesis of imperative programs, the *Evolving Specifications* (ESPECS) framework [PS01] supports the composition of behavioural specifications and their refinement to imperative code. Specifications within ESPECS are represented as state machines. Building upon the Specware-Designware framework [Smi96], composition and refinement of specifications are achieved via colimit and diagram morphism respectively. BERTHA is not refinement based in the formal sense of Dromey [Dro89] and Morgan [Mor94]. However, the partial order planner upon which BERTHA is development supports refinement in the AI planning sense. Like BERTHA, ESPECS requires auxiliary specification, such as loop invariants, to be provided as input. A potential advantage of the BERTHA approach is that the system prompts the user when such input is required. A key difference between the ESPECS and BERTHA approaches is that while BERTHA guides the synthesis process ESPECS relies upon external assistance, *e.g.* in [PS01] reference is made to a design tactic as providing the algorithmic details.

9 Limitations and future work

Like all heuristic techniques, our approach is incomplete. Potential avenues for future investigation include:

- By relying on the rippling heuristic, we are restricting the loop body to be, in some sense, structure preserving. Exploring ways in which rippling can be relaxed, *e.g.* to allow renaming of variables, would increase the range of programs which could be synthesized.
- Building upon our invariant discovery work [IS97, SI98, IS01, EI03, EI04, IEI04] we believe that progress can be made in terms of automating synthesis without user-supplied loop invariants.
- Building upon the meta-level notion of expressibility, we believe automating the reuse of previously synthesized code is possible.
- The link to the object-level is unimplemented. Currently we check the tactics by hand. However, we do not perceive this to be a problem since the tactics are already implemented in HOL [Gor89] and there exists a link between HOL and *Clam* [BSBG98].
- The proof plan currently deals with partial correctness; termination has not been addressed. To show total correctness, the `while` method needs to be extended to find a decreasing and bounded property of the loop.
- Our synthesis ideas are demonstrated using a simple programming language, similar to the language used in [Gor88b]. Recently we have been investigating the use of proof planning [EI03, EI04, IEI04] in terms of the SPARK approach to high integrity software development [Bar03]. SPARK is an Ada subset which supports Floyd/Hoare style program development. The simplicity of the language definition was a key factor in the design of SPARK, *e.g.* functions are not permitted to have side-effects, and recursion and dynamic storage allocation are prohibited. This simplicity would also make SPARK an ideal language for us to further test and extend our synthesis work. One such extension would be to include arrays and records within BERTHA.
- As mentioned in §8, the scalability of our approach may be achieved through component based synthesis, *i.e.* synthesis based upon pre-specified subprograms rather than primitive program constructs. We plan to investigate component based synthesis within the context of SPARK.

10 Conclusion

We have presented an approach to imperative program synthesis which combines proof planning with partial order planning. The approach has been implemented within the BERTHA system and has been used to synthesize a wide range of programs. Using a hybrid approach has enabled us to combine structured programming and proof heuristics within a single framework. We believe that our approach makes progress towards mechanizing Gries' vision of developing *"a program and its proof hand-in-hand, with the proof ideas leading the way!"* [Gri81].

Acknowledgments

The research reported was supported by EPSRC studentship award 96307451 and EPSRC grant GR/R24081. An earlier version of this work was published in the proceedings of the *Automated Software Engineering Conference* [SI99]. The authors would like to thank the three anonymous ASE-99 reviewers and the ASE Journal reviewers for their constructive feedback. We greatly appreciated the many fruitful discussions which we have had with Julian Richardson and other members of Alan Bundy's DR^EAM group. Thanks also goes to Maria McCann for her assistance in the preparation of this paper.

A Program Synthesis Examples

`easy1 :` $\vdash \quad \{x = 0\}$ $\quad\quad \{x = 0\}$	`easy2 :` $\vdash \quad \{x = x0 \wedge y = y0\}$ $\quad\quad \{x + y = x0 + y0\}$
`one :` $\vdash \quad \{true\}$ $\quad\quad x := 1$ $\quad\quad \{x = 1\}$	`incx :` $\vdash \quad \{x = x0\}$ $\quad\quad x := x + 1$ $\quad\quad \{x = x0 + 1\}$
`xoplusxo :` $\vdash \quad \{x = x0\}$ $\quad\quad x := x + x$ $\quad\quad \{x = x0 + x0\}$	`xplusy :` $\vdash \quad \{x = x0 \wedge y = y0\}$ $\quad\quad x := x + y$ $\quad\quad \{x = x0 + y0\}$
`dble :` $\vdash \quad \{x = x0\}$ $\quad\quad x := x + x$ $\quad\quad \{x = dble(x0)\}$	`square :` $\vdash \quad \{x = x0\}$ $\quad\quad x := x * x$ $\quad\quad \{x = exp(x0, 2)\}$
`cubed :` $\vdash \quad \{x = x0\}$ $\quad\quad x := x * (x * x)$ $\quad\quad \{x = exp(x0, 3)\}$	`incxy :` $\vdash \quad \{x = x0 \wedge y = y0\}$ $\quad\quad y := y + 1;$ $\quad\quad x := x + 1$ $\quad\quad \{x = x0 + 1 \wedge y = y0 + 1\}$
`rename :` $\vdash \quad \{x = x0 \wedge y = y0\}$ $\quad\quad p := x;$ $\quad\quad q := y$ $\quad\quad \{q = y0 \wedge p = x0\}$	`xplusyincy :` $\vdash \quad \{x = x0 \wedge y = y0\}$ $\quad\quad x := x + y;$ $\quad\quad y := x + 1$ $\quad\quad \{x = x0 + y0 \wedge y = x + 1\}$
`swap :` $\vdash \quad \{x = x0 \wedge y = y0\}$ $\quad\quad temp := x;$ $\quad\quad x := y;$ $\quad\quad y := temp$ $\quad\quad \{x = y0 \wedge y = x0\}$	`swap3 :` $\vdash \quad \{x = x0 \wedge y = y0 \wedge z = z0\}$ $\quad\quad temp := x;$ $\quad\quad x := y;$ $\quad\quad y := z;$ $\quad\quad z := temp$ $\quad\quad \{x = y0 \wedge y = z0 \wedge z = x0\}$
`dble_swap :` $\vdash \quad \left\{ \bigwedge \begin{array}{l} x = x0 \wedge y = y0 \\ z = z0 \wedge w = w0 \end{array} \right\}$ $\quad\quad temp := x;$ $\quad\quad x := y;$ $\quad\quad y := temp;$ $\quad\quad temp := z;$ $\quad\quad z := w;$ $\quad\quad w := temp$ $\quad\quad \left\{ \bigwedge \begin{array}{l} x = y0 \wedge y = x0 \\ z = w0 \wedge w = z0 \end{array} \right\}$	`dble_swap2 :` $\vdash \quad \left\{ \bigwedge \begin{array}{l} x = x0 \wedge y = y0 \\ z = z0 \wedge w = w0 \end{array} \right\}$ $\quad\quad temp := x;$ $\quad\quad x := y;$ $\quad\quad y := temp;$ $\quad\quad temp := z;$ $\quad\quad z := x + (y + w);$ $\quad\quad w := temp$ $\quad\quad \left\{ \bigwedge \begin{array}{l} x = y0 \wedge y = x0 \\ z = x0 + (y0 + w0) \\ w = z0 \end{array} \right\}$
`xplusy1 :` $\vdash \quad \{x = x0 \wedge y = y0\}$ $\quad\quad temp := x;$ $\quad\quad x := x + y;$ $\quad\quad y := temp$ $\quad\quad \{x = x0 + y0 \wedge y = x0\}$	`xplusyincy1 :` $\vdash \quad \{x = x0 \wedge y = y0\}$ $\quad\quad temp := x + 1;$ $\quad\quad x := x + y;$ $\quad\quad y := temp$ $\quad\quad \{x = x0 + y0 \wedge y = x0 + 1\}$

`ifdblecubed :`

$\vdash \quad \{x = x0 \wedge y = y0\}$
if $x = y + y$ **then**
$x := y * (y * y)$
$\left\{ \bigvee \begin{pmatrix} \wedge & x0 = dble(y0) \\ & x = exp(y0, 3) \end{pmatrix} \atop (x0 \neq dble(y0)) \right\}$

`ifswap :`

$\vdash \quad \{x = x0 \wedge y = y0\}$
if $x \geq y$ **then**
$temp := x;$
$x := y;$
$y := temp$
$\left\{ \bigvee \begin{matrix} (x0 \geq y0 \wedge y = x0 \wedge x = y0) \\ (x0 < y0 \wedge x = x0 \wedge y = y0) \end{matrix} \right\}$

`max :`

$\vdash \quad \{x = x0 \wedge y = y0\}$
if $x \geq y$ **then**
$r := x$
else
$r := y$
$\{r = max(x0, y0)\}$

`not :`

$\vdash \quad \{x = x0\}$
if $x = 0$ **then**
$x := 1$
else
$x := 0$
$\left\{ \bigvee \begin{matrix} (x0 = 0 \wedge x = 1) \\ (x0 \neq 0 \wedge x = 0) \end{matrix} \right\}$

`notandy :`

$\vdash \quad \{x = x0\}$
$y := x;$
if $x = 0$ **then**
$x := 1$
else
$x := 0$
$\left\{ \wedge \begin{pmatrix} \vee & (x0 = 0 \wedge x = 1) \\ & (x0 \neq 0 \wedge x = 0) \end{pmatrix} \atop (y = x0) \right\}$

`swapandnot :`

$\vdash \quad \{x = x0 \wedge y = y0\}$
$temp := x;$
if $y = 0$ **then**
$x := 1$
else
$x := 0; y := temp;$
$\left\{ \wedge \begin{pmatrix} \vee & (y0 = 0 \wedge x = 1) \\ & (y0 \neq 0 \wedge x = 0) \end{pmatrix} \atop (y = x0) \right\}$

`exp1 :`

$\vdash \quad \{x = x0 \wedge y = y0\}$
$r := 1;$
while not $(y = 0)$ **do**
begin
$r := r * x;$
$y := y - 1$
end
$\{r = exp(x0, y0)\}$

`exp2 :`

$\vdash \quad \{x = x0 \wedge y = y0\}$
$r := 1;$
$i := 0;$
while not $(i = y)$ **do**
begin
$r := r * x;$
$i := i + 1$
end
$\{r = exp(x0, y0)\}$

`sum2 :`

$\vdash \quad \{x = x0\}$
$i := 0;$
$r := 0;$
while not $(i = x)$ **do**
begin
$i := i + 1;$
$r := i + r$
end
$\{r = \sum_{m=0}^{x0} m\}$

`fac2 :`

$\vdash \quad \{x = x0\}$
$i := 0;$
$r := 1;$
while not $(i = x)$ **do**
begin
$i := i + 1;$
$r := r * i$
end
$\{r = fac(x0)\}$

`sumodd :`

$\vdash \quad \{x = x0\}$
$i := 0;$
$r := i;$
while not $(i = x)$ **do**
begin
$i := i + 1$
if not $even(i)$ **then**
$r := i + r$
end
$\{r = \sum_{m=0}^{x0} (m | odd(m))\}$

```
sumfsum1 :
⊢  {x = x0}
   i := 0;
   r := 0;
   while not (i = x) do
      begin
         i := i + 1;
         j := 0;
         v := 0;
         while not (j = i) do
            begin
               j := j + 1;
               v := j + v
            end
         r := v + r
      end
   {r = Σ_{m=0}^{x0} Σ_{n=0}^{m} n}
```

```
sumfsum2 :
⊢  {x = x0}
   i := 0;
   r := 0;
   z := i;
   while not (i = x) do
      begin
         i := i + 1;
         z := i + z;
         r := z + r
      end
   {r = Σ_{m=0}^{x0} Σ_{n=0}^{m} n}
```

```
sumexp :
⊢  {x = x0 ∧ y = y0}
   if x < y then
         i := 0;
         r := i;
         while not (i = x) do
            begin
               i := i + 1;
               r := i + r
            end
    else
         i := 0;
         r := i + 1;
         while not (i = y) do
            begin
               i := i + 1;
               r := x * r
            end
   { ⋁ (x0 < y0 ∧ r = Σ_{m=0}^{x0} m)
     (x0 ≥ y0 ∧ r = exp(x0, y0)) }
```

```
sumfexp1 :
⊢  {x = x0 ∧ y = y0}
   i := 0;
   r := i;
   while not (i = x) do
      begin
         i := i + 1;
         k := 0;
         v := k + 1;
         while not (k = y) do
            begin
               k := k + 1;
               v := i * v
            end
         r := v + r
      end
   {r = Σ_{m=0}^{x0} exp(m, y0)}
```

B Synthesis Results

B.1 Definitions & Lemmas

No	Definitions & Lemmas
L1	$X - X = 0$
L2	$X - (Y - Z) = (X - Y) + Z$
L3	$X * 1 = X$
L4	$exp(X, 0) = 1$
L5	$exp(X, Y + 1) = X * exp(X, Y)$
L6	$X * exp(X, Y - 1) = exp(X, Y)$
L7	$fac(0) = 1$
L8	$fac(X + 1) = (X + 1) * fac(X)$
L9	$\sum_{m=0}^{0} m = 0$
L10	$\sum_{m=0}^{X+1} m = (X + 1) + \sum_{m=0}^{X} m$
L11	$X + \sum_{m=0}^{X-1} m = \sum_{m=0}^{X} m$
L12	$\sum 0 = 0$
L13	$\sum_{m=0}^{X+1} f(m) = f(X + 1) + \sum_{m=0}^{X} f(m)$
L14	$\sum_{m=0}^{X+1} f(m, Y) = f(X + 1, Y) + \sum_{m=0}^{X} f(m, Y)$
L15	$\sum_{m=0}^{0}(m \mid odd(m)) = 0$
L16	$even(X + 1) \rightarrow \sum_{m=0}^{X+1}(m \mid odd(m)) = \sum_{m=0}^{X}(m \mid odd(m))$
L17	$odd(X + 1) \rightarrow \sum_{m=0}^{X+1}(m \mid odd(m)) = (X + 1) + \sum_{m=0}^{X}(m \mid odd(m))$
L18	$X \geq Y \rightarrow max(X, Y) = X$
L19	$X < Y \rightarrow max(X, Y) = Y$
L20	$dble(X) = X + X$

Table 1: Definitions & Lemmas Used For Program Synthesis.

B.2 Program Specifications

Algorithm	Specification	Lemmas
`easy1`	$\{x = 0\}C\{x = 0\}$	–
`easy2`	$\{x = x0 \wedge y = y0\}C\{x + y = x0 + y0\}$	–
`one`	$\{true\}C\{x = 1\}$	–
`incx`	$\{x = x0\}C\{x = x0 + 1\}$	–
`xoplusxo`	$\{x = x0\}C\{x = x0 + x0\}$	–
`xplusy`	$\{x = x0 \wedge y = y0\}C\{x = x0 + y0\}$	–
`dble`	$\{x = x0\}C\{x = dble(x0)\}$	L20
`square`	$\{x = x0\}C\{x = exp(x0, 2)\}$	L3,L4,L5
`cubed`	$\{x = x0\}C\{x = exp(x0, 3)\}$	L3,L4,L5

Table 2: Simple Programs Successfully Synthesized.

Algorithm	Specification
incxy	$\{x = x0 \wedge y = y0\} C \{x = x0 + 1 \wedge y = y0 + 1\}$
rename	$\{x = x0 \wedge y = y0\} C \{q = y0 \wedge p = x0\}$
xplusyincy	$\{x = x0 \wedge y = y0\} C \{x = x0 + y0 \wedge y = x + 1\}$
swap	$\{x = x0 \wedge y = y0\} C \{x = y0 \wedge y = x0\}$
swap3	$\{x = x0 \wedge y = y0 \wedge z = z0\} C \{x = y0 \wedge y = z0 \wedge z = x0\}$
dble_swap	$\{x = x0 \wedge y = y0 \wedge z = z0 \wedge w = w0\} C \{x = y0 \wedge y = x0 \wedge w = z0 \wedge z = w0\}$
dble_swap2	$\{x = x0 \wedge y = y0 \wedge z = z0 \wedge w = w0\} C \{x = y0 \wedge y = x0 \wedge w = z0 \wedge z = x0 + (y0 + w0)\}$
xplusy1	$\{x = x0 \wedge y = y0\} C \{x = x0 + y0 \wedge y = x0\}$
xplusyincy1	$\{x = x0 \wedge y = y0\} C \{x = x0 + y0 \wedge y = x0 + 1\}$

Table 3: Programs Involving Assignment Sequences Successfully Synthesized.

Algorithm	Specification	Lemmas
ifdblecubed	$\{x = x0 \wedge y = y0\} C \{\bigvee \begin{array}{l}(x0 = dble(y0) \wedge x = exp(y0, 3)) \\ (x0 \neq dble(y0))\end{array}\}$	L20,L4, L5,L3
ifswap	$\{x = x0 \wedge y = y0\} C \{\bigvee \begin{array}{l}(x0 \geq y0 \wedge y = x0 \wedge x = y0) \\ (x0 < y0 \wedge x = x0 \wedge y = y0)\end{array}\}$	–
max	$\{x = x0 \wedge y = y0\} C \{r = max(x0, y0)\}$	L18,L19
not	$\{x = x0\} C \{\bigvee \begin{array}{l}(x0 = 0 \wedge x = 1) \\ (x0 \neq 0 \wedge x = 0)\end{array}\}$	–
notandy	$\{x = x0\} C \{\bigwedge \begin{array}{l}\left(\bigvee \begin{array}{l}(x0 = 0 \wedge x = 1) \\ (x0 \neq 0 \wedge x = 0)\end{array}\right) \\ (y = x0)\end{array}\}$	–
swapandnot	$\{x = x0 \wedge y = y0\} C \{\bigwedge \begin{array}{l}\left(\bigvee \begin{array}{l}(y0 = 0 \wedge x = 1) \\ (y0 \neq 0 \wedge x = 0)\end{array}\right) \\ (y = x0)\end{array}\}$	–

Table 4: Programs Involving Conditionals Successfully Synthesized.

Algorithm	Specification	Invariants supplied	Lemmas
`exp1`	$\{x = x0 \wedge y = y0\}C\{r = exp(x0, y0)\}$	$r = exp(x0, y0 - i)$	L1, L2, L4,L5
`exp2`	$\{x = x0 \wedge y = y0\}C\{r = exp(x0, y0)\}$	$r = exp(x0, i)$	L4,L5
`fac2`	$\{x = x0\}C\{r = fac(x0)\}$	$r = fac(i)$	L7,L8
`sum2`	$\{x = x0\}C\{r = \sum_{m=0}^{x0} m\}$	$r = \sum_{m=0}^{i} m$	L10,L11
`sumexp`	$\{x = x0 \wedge y = y0\}C\{\bigvee \begin{pmatrix} \bigwedge^{(x0 < y0)}_{(r = \sum_{m=0}^{x0} m)} \end{pmatrix} \begin{pmatrix} \bigwedge^{(x0 \geq y0)}_{(r = exp(x0, y0))} \end{pmatrix}\}$	**then** loop: $r = \sum_{m=0}^{i} m$ **else** loop: $r = exp(x0, i)$	L4, L5 L9,L10
`sumodd`	$\{x = x0\}C\{r = \sum_{m=0}^{x0}(m\|odd(m))\}$	$r = \sum_{m=0}^{i}(m\|odd(m))$	L15, L16 L17
`sumfexp1`	$\{x = x0 \wedge y = y0\}C\{r = \sum_{m=0}^{x0} exp(m, y0)\}$	Outer loop: $r = \sum_{m=0}^{i} exp(m, y0)$ Inner loop: $r = exp(i, k)$	L4,L5, L12, L14
`sumfsum1`	$\{x = x0\}C\{r = \sum_{m=0}^{x0}\sum_{n=0}^{m} n\}$	Outer loop: $r = \sum_{m=0}^{i}\sum_{n=0}^{m} n$ Inner loop: $\sum_{n=0}^{j} n$	L9,L10, L12, L13
`sumfsum2`	$\{x = x0\}C\{r = \sum_{m=0}^{x0}\sum_{n=0}^{m} n\}$	$\bigwedge^{(r = \sum_{m=0}^{i}\sum_{n=0}^{m} n)}_{(z = \sum_{m=0}^{i} n)}$	L9,L10, L12,L13

Table 5: Programs Involving Loops Successfully Synthesized.

References

[ASG97] A. Armando, A. Smaill, and I. Green. Automatic synthesis of recursive programs: The proof-planning paradigm. In M. Lowry and Y. Ledru, editors, *Proceedings of ASE-97: The 12th IEEE Conference on Automated Software Engineering*, pages 2–9. IEEE Computer Society Press, 1997.

[Bac86] R.C. Backhouse. *Program Construction and Verification*. Prentice Hall, 1986.

[Bar03] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.

[BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. Academic Press, 1979. ACM monograph series.

[BSBG98] R. Boulton, K. Slind, A. Bundy, and M. Gordon. An interface between CLAM and HOL. In J. Grundy and M. Newey, editors, *Proceedings of the 11th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'98)*, volume 1479 of *Lecture Notes in Computer Science*, pages 87–104, Canberra, Australia, September/October 1998. Springer.

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

[BSvH+93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[Bun88] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[BvHHS90] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[BW96] D. Basin and T. Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180, 1996.

[Che94] B Cheng. Applying formal methods in automated software development. *Journal of Computer and Software Engineering*, 2(2):137–164, 1994.

[Chr93] H. Christensen. Synthesis of programs from logic specifications using programming methodology. *Structured Programming*, 14:173–186, 1993.

[Der85] N. Dershowitz. Synthetic programming. *Artificial Intelligence*, 25:323–373, 1985.

[Dij72] E. Dijkstra. The Humble Programmer. *CACM*, 15(10):859–866, October 1972.

[Dij76] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Dro89] G. Dromey. *The Development of Programs From Specifications*. Addison-Wesley, 1989.

[EI03] B.J. Ellis and A. Ireland. Automation for exception freedom proofs. In *Proceedings of the* 18^{th} *IEEE International Conference on Automated Software Engineering*, pages 343–346. IEEE Computer Society, 2003. Available from School of Mathematical and Computer Sciences, Heriot-Watt University, Technical Report HW-MACS-TR-0010.

[EI04] B.J. Ellis and A. Ireland. An integration of program analysis and automated theorem proving. In E.A. Boiten, J. Derrick, and G. Smith, editors, *Proceedings of 4th International Conference on Integrated Formal Methods (IFM-04)*, volume 2999 of *Lecture Notes in Computer Science*, pages 67–86. Springer Verlag, 2004. Available from School of Mathematical and Computer Sciences, Heriot-Watt University, Technical Report HW-MACS-TR-0014.

[EN69] G. Ernst and A. Newell. *GPS: A Case Study in Generality and Problem Solving*. Academic Press, 1969.

[Flo67] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32. American Mathematical Society, 1967.

[FZH98] P. Flener, H. Zidoum, and B. Hnich. Schema-guided synthesis of constraint logic programs. In D.F. Redmiles and B. Nuseibeh, editors, *Proceedings of ASE'98*, pages 168–176. IEEE Computer Society Press, 1998.

[GMW79] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Gor88a] M. J. Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer, 1988.

[Gor88b] M. J. Gordon. *Programming Language Theory and its Implementation*. International Series in Computer Science. Prentice-Hall, 1988.

[Gor89] M. J. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.

[Gri76] D. Gries. An illustration of current ideas on the derivation of correctness proofs and correct programs. *IEEE Transactions on Software Engineering*, 2:238–244, December 1976.

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–583, 1969.

[IB96a] A. Ireland and A. Bundy. Extensions to a Generalization Critic for Inductive Proof. In M. A. McRobbie and J. K. Slaney, editors, *13th International Conference on Automated Deduction*, pages 47–61. Springer-Verlag, 1996. Springer Lecture Notes in Artificial Intelligence No. 1104. Also available from Edinburgh as DAI Research Paper 786.

[IB96b] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996. Also available as DAI Research Paper No 716, Dept. of Artificial Intelligence, Edinburgh.

[IB99] A. Ireland and A. Bundy. Automatic Verification of Functions with Accumulating Parameters. *Journal of Functional Programming: Special Issue on Theorem Proving & Functional Programming*, 9(2):225–245, March 1999. A longer version is available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/11.

[IEI04] A. Ireland, B.J. Ellis, and T. Ingulfsen. Invariant patterns for program reasoning. In R. Monroy, G. Arroyo-Figueroa, L.E. Sucar, and H. Sossa, editors, *Proceedings of 3rd Mexican International Conference on Artificial Intelligence (MICAI-04)*, volume 2972 of *Lecture Notes in Artificial Intelligence*, pages 190–201. Springer Verlag, 2004. Available from School of Mathematical and Computer Sciences, Heriot-Watt University, Technical Report HW-MACS-TR-0011.

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

[IS97] A. Ireland and J. Stark. On the Automatic Discovery of Loop Invariants. In *Proceedings of the Fourth NASA Langley Formal Methods Workshop – NASA Conference Publication 3356*, 1997. Also available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/1.

[IS01] A. Ireland and J. Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):65–97, February 2001. An earlier version is available as Research Memo RM/00/3, Dept. of Computing and Electrical Engineering, Heriot-Watt University.

[Kal90] A. Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice Hall, 1990.

[KBB93] I. Kraan, D. Basin, and A. Bundy. Logic program synthesis via proof planning. In K. K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, pages 1–14. Springer-Verlag, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-92-244 and Edinburgh DAI Research Report 603.

[KKY95] S. Kambhampati, C. Knoblock, and Q. Yang. Planning as refinement search: A unified framework for evaluating design tradeoffs in partial-order planning. *Artificial Intelligence*, 76:167–238, July 1995.

[Krz81] R. A. Krzysztof. Ten years of hoare's logic: A survey - part i. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[LPPU94] M. Lowry, A. Philpot, T. Pressburger, and I. Underwood. Amphion: Automatic programming for scientific subroutine libraries. In *Proc. 8th Intl. Symp. on Methodologies for Intelligent Systems*, Charlotte, North Carolina, October 1994.

[LW96] J. Lee and D.E. Wilkins. Using sipe-2 to integrate planning for military air campaigns. *IEEE Expert*, 11(6):11–12, December 1996.

[Mor94] C. Morgan. *Programming from Specifications*. Prentice Hall, 2 edition, 1994.

[MW77] Z. Manna and R. Waldinger. *Studies in Automatic Programming Logic*. Elsevier, 1977.

[MW80] Z. Manna and R.J. Waldinger. A deductive approach to program synthesis. *Journal of Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[Pol45] G. Polya. *How to Solve It*. Princeton University Press, 1945.

[PS01] D. Pavlovic and D.R. Smith. Composition and refinement of behavioral specifications. In *Proceedings of the* 16^{th} *IEEE International Conference on Automated Software Engineering*, pages 157–165. IEEE Computer Society, 2001.

[RF03] J. Richardson and P. Flener. Program schemas as proof methods. Uppsala University Department of Information Technology Technical Report 2003-008, 2003.

[SG95] A. Smaill and I. Green. Automating the synthesis of functional programs. Research paper 777, Dept. of Artificial Intelligence, University of Edinburgh, 1995.

[SI98] J. Stark and A. Ireland. Invariant discovery via failed proof attempts. In P. Flener, editor, *Logic-Based Program Synthesis and Transformation*, LNCS 1559, pages 271–288. Springer-Verlag, 1998.

[SI99] J. Stark and A. Ireland. Towards automatic imperative program synthesis through proof planning. In *The* 14^{th} *IEEE International Conference on Automated Software Engineering*, pages 44–51. IEEE Computer Society, 1999.

[Smi90] D. R. Smith. KIDS: A semi-automatic program development system. *Transactions on Software Engineering – Special Issue on Formal Methods*, 16(9):1024–1043, September 1990.

[Smi96] D. R. Smith. Toward a classification approach to design. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology, AMAST'96*, LNCS 1101. Springer Verlag, 1996.

[Sus75] G. J. Sussman. *A Computer Model of Skill Acquisition*. Artificial Intelligence Series. North Holland, 1975. Also MIT AI Lab Memon no. AI-TR-297.

[Tat77] A. Tate. Generating project networks. In R. Reddy, editor, *Proceedings of IJCAI-77*, pages 888–893, Boston, Ma, 1977. International Joint Conference on Artificial Intelligence.

[TDL00] A. Tate, J. Dalton, and J. Levine. O-plan: A web-based AI planning agent. In *AAAI/IAAI*, pages 1131–1132, 2000.

[Wal77] R. Waldinger. *Achieving Several Goals Simultaneously*, volume 8 of *Machine Intelligence*, chapter 6, pages 94–138. Halstead and Wiley, New York, 1977.