Guiding Synthesis Proofs

Vincent Lombart*
Unité d'Informatique
Université Catholique de Louvain
Louvain-la-Neuve
Belgium

Geraint Wiggins†
Department of Artificial Intelligence
University of Edinburgh
Edinburgh
Scotland

Yves Deville‡
Unité d'Informatique
Université Catholique de Louvain
Louvain-la-Neuve
Belgium

Extended Abstract

1 Introduction

In this paper, we explore some possibilities of the *Whelk* logic program synthesis system. This exploration is intended to be a step towards an automation of the synthesis process using *Whelk* as a basic building block.

Whelk is based on the constructive synthesis approach, it synthesizes logic programs from proofs: a conjecture (called a *specification conjecture*) is interactively proven in a constructive way, and *Whelk* automatically generates the corresponding program.

In [Wiggins 92b], it is explained how *Whelk* synthesizes programs from proofs. Here, we focus on how to guide a proof to get a logic program with a given structure. This will be done through the synthesis of a sample problem: from the specification of the *subset* relation, we synthesize two programs with completely different structures. We then explain how the close structural relationship between a proof and the synthesized program has been used to guide the synthesis towards those program structures.

Whelk is designed to be interfaced with the CLaM proof planner [Bundy 88, Bundy *et al.* 90b]. The techniques and heuristics illustrated here can be integrated as CLaM proof plans or as proof "critics" [Ireland 92], and the operation of the system will be discussed alongside the proofs.

In the following sections, the aspects of *Whelk* important for this paper are first described. Two proofs of the *subset* specification conjecture are then developed, and the synthesized pro-

*Email: vl@info.ucl.ac.be
†Email: geraint@ed.ac.uk
‡Email: yde@info.ucl.ac.be

grams are built in parallel. Finally, both programs are analysed, and issues of time complexity, directionalities and proof difficulties are discussed.

2 The *Whelk* system

Whelk is a proof development system. Under certain conditions, *Whelk* is able to extract a logic program from a developed proof. This makes it a valuable tool in the logic program synthesis and transformation domain.

Whelk is based on a Gentzen Sequent Calculus, and uses a first order, typed, constructive logic with equality. To derive a logic program from a proof in *Whelk*, the *proofs-as-programs* paradigm, which usually synthesize functional programs, has been adapted to synthesize logic programs. This adaptation raised two important problems, due to the differences between functional and relational programs:

- There is more than one way to use a relational program (multidirectionality).
- For a given directionality, there are potentially many – or no – outputs.

A solution to those problems is to consider only the all-ground directionality [Bundy *et al.* 90a]. The predicate can then be seen as a boolean valued function. This is the key idea to transform *proofs-as-functional-programs* into *proofs-as-relational-programs*

The form of a specification conjecture to prove is largely a matter of taste. In *Whelk* a new operator, ∂, has been introduced with the meaning "It is decidable whether..." [Wiggins *et al.* 91, Wiggins 92b]. To synthesize a program with vector of arguments $\vec{a}$ of type $\vec{\tau}$ and specification $S(\vec{a})$, we must prove a theorem of the form

$$\vdash \forall \vec{a} : \vec{\tau}.\ \partial S(\vec{a}). \tag{1}$$

3 The `subset/2` predicate synthesis

We want to synthesize a predicate `subset/2` which holds iff its first argument is a subset of its second argument. Those arguments are lists of integers considered as sets of integers. The specification of the subset relation can be approximated as

$$subset(SubL, SupL) \leftrightarrow (\forall X.\ X \in SubL \rightarrow X \in SupL) \tag{2}$$

(Note: this definition is an approximation because it allows multisets, which will lead to problems in running the synthesised program later. However, the full specification, where the arguments are forced to be sets, would make the examples opaque.)

This problem has been chosen for two reasons:

- It is simple enough (in this form) to be easily tractable.
- It is complex enough to have the possibility to synthesize (at least) two completely different programs from its specification.

For some problems, the choice of the induction parameter is crucial since different programs can then be constructed [Deville 90, Dayantis 87]. With an induction on the sublist, one gets:

$$\texttt{subset}(SubL,\ SupL) \leftarrow$$
$$\quad SubL = [\,]$$
$$\vee\ SubL = [V_0|V_1] \wedge \texttt{member}(V_0, SupL) \wedge \texttt{subset}(V_1, SupL)$$

where $SubL$, $SupL$ are universally quantified and V_0, V_1 are existentially quantified (as usual), and where $\texttt{member}(V, S)$ holds iff $V \in S$. Procedurally speaking, the outer loop is on the sublist, and the inner loop is on the superlist through the `member` procedure.

Using an induction on the superlist, one obtains:

$$\texttt{subset}(SubL,\ SupL) \leftarrow$$
$$\quad SupL = [\,] \wedge SubL = [\,]$$
$$\vee\ SupL = [V_0|V_1] \wedge \texttt{delete\_all}(V_0, SubL, AuxL) \wedge \texttt{subset}(AuxL, V_1)$$

where $\texttt{delete\_all}(V_0, SubL, AuxL)$ holds iff $AuxL$ is $SubL$ with all occurrences of V_0 deleted. Procedurally speaking, the outer loop is on the superlist, and the inner loop is on the sublist through the `delete_all` procedure.

There is a third way to construct the `subset` predicate, through the so-called "iteration through negation", using the negation-as-failure rule:

$$\texttt{subset}(SubL,\ SupL) \leftarrow$$
$$\quad \neg \texttt{p}(SubL, SupL)$$
$$\texttt{p}(SubL, SupL) \leftarrow$$
$$\quad \texttt{member}(X, SubL) \wedge \neg\texttt{member}(X, SupL)$$

This is the program obtained by the Lloyd-Topor transformation [Lloyd & Topor 84] of the `subset` specification into a logic program. This program suffers badly from floundering in all but the all-ground mode, and so will not be considered here.

Let us now show how it is possible to synthesize the first two programs from the same specification.

3.1 Preliminaries

3.1.1 Object Level Notation

The type "integer" will be noted *nat*, and the type "list of integers" *lnat*. We will need two axioms to define the "$\in$" relation used in the definition of *subset*:

$$\forall X : nat.\ \neg X \in [\,] \tag{3}$$
$$\forall X : nat.\ \forall Hd : nat.\ \forall Tl : lnat.\ X \in [Hd|Tl] \leftrightarrow X = Hd \vee X \in Tl \tag{4}$$

and we also need some axioms on integers and lists of integers:

$$\forall X : nat.\ \forall Y : nat.\ X = Y \vee \neg X = Y \tag{5}$$
$$\forall K : lnat.\ \forall L : lnat.\ K = L \vee \neg K = L \tag{6}$$
$$\forall L : lnat.\ L = [\,] \vee \exists Hd : nat.\ \exists Tl : lnat.\ L = [Hd|Tl] \tag{7}$$
$$\ldots$$

The proofs will be presented as they are built in *Whelk*, in refinement style. In other words, when a rule such as

$$\frac{B \quad C}{A}$$

is used, it is done backwards: If A has to be proven, when using the rule, we are then left with B and C to prove. How these rules are used by *Whelk* to extract a program is explained in [Wiggins 92b]. In this paper, the automatically generated predicate names will be renamed to improve readability.

A sequent can be written horizontally or vertically (or mixed).

$$H_1 \ldots H_n \vdash T \qquad \text{or} \qquad \begin{array}{l} H_1 \\ \vdots \\ H_n \\ \vdash T \end{array}$$

where $H_1 \ldots H_n$ are the hypotheses, and T is the conjecture. To help the reader, when new hypotheses are introduced, they are marked with a star on their left.

3.1.2 Meta-Level Notation

Meta-level annotations, which control the use of rewrite rules in CLaM, are given in the following way. A *wave front*, which surrounds the part of an conjecture which must be moved away to allow matching with a hypothesis, is surrounded by a box. Any subexpression of that part which is required for the match, the *wave hole*, is underlined. The wave front is superscripted with a direction, $\uparrow$ when the front is to move out, and $\downarrow$ when it is to move in. Finally, universally quantified variables in the hypothesis are marked in the conclusion as *sinks* – these are targets towards which inward wave fronts can be *rippled*.

An example of a step case sequent with an out-going wave-front – the list constructor and V_0 – a corresponding wave hole – V_1 – and a sink – $SupL$– is

$$\begin{array}{ll} \star & V_0 : nat, V_1 : lnat, \\ \star & \forall SupL : lnat.\ \partial(\forall X : nat.\ X \in V_1 \rightarrow X \in SupL) \\ & \vdash \forall SupL : lnat.\ \partial(\forall X : nat.\ X \in \boxed{[V_0|\underline{V_1}]}^{\uparrow} \rightarrow X \in \lfloor SupL \rfloor) \end{array}$$

This conclusion might be rewritten – rippled – by the application of the following *wave rule*, which is derivable automatically from the definition of $\in$.

$$X \in \boxed{[Y_0|\underline{Y_1}]}^{\uparrow} \quad \Rightarrow \quad \boxed{X = Y_0 \vee \underline{X \in Y_1}}^{\uparrow}$$

Application of this rule will yield the conclusion

$$\vdash \forall SupL : lnat.\ \partial(\forall X : nat.\ X \in \boxed{X = V_0 \vee \underline{X \in V_1}}^{\uparrow} \rightarrow X \in \lfloor SupL \rfloor)$$

Matching both the object and meta parts of the wave rule with those of the conjecture allows us to select a rewrite rule which moves exactly the symbols we want moved, in order to match with our induction hypothesis.

Rules which move the unwanted parts of expressions into sinks are useful because then the universal quantification in the hypothesis can be used to allow the substitution necessary for a successful match.

Examples of all of the above will be given in the following sections.

3.2 The Synthesis Conjecture

We start off with the specification conjecture

$$\vdash \forall SubL : lnat.\ \forall SupL : lnat.\ \partial(\forall X : nat.\ X \in SubL \rightarrow X \in SupL) \tag{8}$$

where $SubL$ is the sublist and $SupL$ is the superlist.

Whelk is orientated towards proofs by induction, and that technique will therefore be used in the proofs. But we have here two reasonable possibilities of induction: either on the sublist, or on the superlist.

3.3 Induction on the sublist

When proving the specification conjecture, applying induction on $SubL$ gives us two subconjectures:

Base Case

$$\vdash \forall SupL : lnat.\ \partial(\forall X : nat.\ X \in [\,] \rightarrow X \in SupL) \tag{9}$$

Step Case

$$\begin{aligned} &\star\ \ V_0 : nat, V_1 : lnat, \\ &\star\ \ \forall SupL : lnat.\ \partial(\forall X : nat.\ X \in V_1 \rightarrow X \in SupL) && (10) \\ &\quad \vdash \forall SupL : lnat.\ \partial(\forall X : nat.\ X \in \boxed{[V_0|\underline{V_1}]}^{\uparrow} \rightarrow X \in \lfloor SupL \rfloor) && (11) \end{aligned}$$

At this point, the program fragment generated looks like

$$\begin{aligned} &\texttt{subset}(SubL, SupL) \leftarrow \\ &\quad SubL = [\,] \wedge \ldots \\ &\vee\ SubL = [V_0|V_1] \wedge \ldots \end{aligned}$$

The base case (9) is always true, by definition (3), so *Whelk* replaces the first "..." in the program by *true*. For the step case, we unfold $X \in [V_0|V_1]$ according to the definition (4), which is a wave rule (see section 3.1.2), and with some rewriting we get

$$\vdash \partial(\boxed{(\forall X : nat.\ X = V_0 \rightarrow X \in \underline{\lfloor SupL \rfloor}) \wedge (\forall X : nat.\ X \in V_1 \rightarrow X \in \underline{\lfloor SupL \rfloor})}^{\uparrow}) \tag{12}$$

The outward wave front is now rippled as far as it will go.

$\wedge$ Introduction splits the conjecture (12) in two:

$$\vdash \partial(\forall X : nat.\ X = V_0 \rightarrow X \in SupL) \tag{13}$$

$$\vdash \partial(\forall X : nat.\ X \in V_1 \rightarrow X \in SupL) \tag{14}$$

Using a substitution rule, the first subconjecture (13) is reduced to

$$\vdash \partial(V_0 \in SupL) \tag{15}$$

This can be proven by a further application of induction, again guided by rippling. Its proof will synthesize the `member/2` predicate.

The second subconjecture (14) is trivially proven using the induction hypothesis (10)

$$\forall SupL : lnat.\ \partial(\forall X : nat.\ X \in V_1 \rightarrow X \in SupL)$$

Whelk has by now replaced the second "..." in the program by

$$\texttt{member}(V_0, SupL) \wedge \texttt{subset}(V_1, SupL)$$

Hence the synthesized program extracted from the proof is the following:

$$\begin{array}{l}\texttt{subset}(SubL,\ SupL) \leftarrow \\ \quad SubL = [\,] \\ \vee\ SubL = [V_0|V_1] \wedge \texttt{member}(V_0, SupL) \wedge \texttt{subset}(V_1, SupL)\end{array}$$

3.4 Induction on the superlist

When proving the specification conjecture (8)

$$\vdash \forall SubL : lnat.\ \forall SupL : lnat.\ \partial(\forall X : nat.\ X \in SubL \rightarrow X \in SupL)$$

applying induction on $SupL$ gives us two subconjectures:

Base Case

$$\vdash \forall SubL : lnat.\ \partial(\forall X : nat.\ X \in SubL \rightarrow X \in [\,]) \tag{16}$$

Step Case

$$\begin{array}{ll}\star & V_0 : nat, V_1 : lnat, \\ \star & \forall SubL : lnat.\ \partial(\forall X : nat.\ X \in SubL \rightarrow X \in V_1) \qquad (17) \\ & \vdash \forall SubL : lnat.\ \partial(\forall X : nat.\ X \in \lfloor SubL \rfloor \rightarrow X \in \boxed{[V_0|\underline{V_1}]}^{\uparrow}) \qquad (18)\end{array}$$

Using the definition of "$\in$" (3, 4) and an axiom on lists (7), the base case is true if $SubL = [\,]$. We now have the program fragment

$$\begin{array}{l}\texttt{subset}(SubL, SupL) \leftarrow \\ \quad SupL = [\,] \wedge SubL = [\,] \\ \vee\ SupL = [V_0|V_1] \wedge \ldots\end{array}$$

For the step case, guided by rippling as for the previous proof, we unfold $X \in [V_0|V_1]$ according to definition (4), to give

$$\vdash \partial(\forall X : nat.\ X \in \lfloor SubL \rfloor \rightarrow \boxed{X = V_0 \vee \underline{X \in V_1}}^{\uparrow}) \tag{19}$$

and then according to the propositional wave rule

$$A \rightarrow \boxed{B \vee \underline{C}}^{\uparrow} \Rightarrow \boxed{\underline{A} \wedge \neg B}^{\downarrow} \rightarrow C \tag{20}$$

where the direction of the wave motion is altered because the rule is a *transverse wave rule* – see [Bundy *et al.* 91] for more information. This leaves us with

$$\vdash \partial(\forall X : nat.\ \boxed{\underline{X \in \lfloor SubL \rfloor} \wedge \neg X = V_0}^{\downarrow} \rightarrow X \in V_1) \tag{21}$$

We now have a problem: it is not straightforward to use the induction hypothesis

$$\forall SubL : lnat.\ \partial(\forall X : nat.\ X \in SubL \rightarrow X \in V_1) \tag{22}$$

to complete the proof. The rippling tactic used to rewrite the conjecture to match the induction hypothesis is not immediately usable here, because there is no suitable rewrite rule. We have two differences between the conjecture and the hypothesis:

- $X \in SubL$ (in the hypothesis) is replaced by $X \in SubL \wedge \neg X = V_0$
- the presence of the quantifier $\forall SubL : lnat$ in the hypothesis

This last point is expressed by the appearance of the sink annotation around $SubL$, and will be used to continue the proof. The quantification can be seen as a degree of freedom for the use of the hypothesis.

We need a wave rule which will ripple the remaining wave front as close as possible to the sink. The obvious one is

$$\boxed{\underline{X \in \lfloor SubL \rfloor} \wedge \neg X = V_0}^{\downarrow} \Rightarrow X \in \boxed{\underline{SubL} \ominus V_0}^{\downarrow} \tag{23}$$

where $\ominus$ is the delete operation on lists. However, the first order nature of our proof would favour a more relational style of rule, which is currently beyond the definition of wave rules. In order to generate the rule we need, we must conjecture a lemma, defining an intermediate structure $AuxL$ such that

$$X \in AuxL \leftrightarrow X \in SubL \wedge \neg X = V_0 \tag{24}$$

With such an hypothesis, the conjecture can be transformed into

$$\vdash \partial(\forall X : nat.\ X \in AuxL \rightarrow X \in V_1) \tag{25}$$

It is then straightforward to use the induction hypothesis (22). The extracted program fragment now has the form:

$$\begin{aligned} &\texttt{subset}(SubL, SupL) \leftarrow \\ &\quad SupL = [\,] \wedge SubL = [\,] \\ &\vee\ SupL = [V_0|V_1] \wedge \ldots \wedge \texttt{subset}(AuxL, V_1) \end{aligned}$$

But how can this intermediate structure be introduced in the proof ? We have to use the cut rule:

$$\frac{\Gamma \vdash A \quad \Gamma, A \vdash T}{\Gamma \vdash T}$$

This rule is often considered as a "eureka" step in a proof, as it introduces a completely new hypothesis, A. But here, this hypothesis can easily be deduced from the differences

between the induction hypothesis and the conjecture to prove. In the CLaM proof planner, classes of situations such as this can be characterised in terms of *blocked rippling*, and a *proof critic* [Ireland 92] can be triggered to suggest the necessary solution, as we did by inspection, above.

The exact hypothesis we cut in is

$$\exists AuxL : lnat.\ \forall X : nat.\ X \in AuxL \leftrightarrow X \in SubL \wedge \neg X = V_0 \tag{26}$$

and its proof will consist of the building of a witness $AuxL$. This will synthesize the `delete_all/3` predicate, and finally give the extracted program

$$\begin{aligned} &\texttt{subset}(SubL,\ SupL) \leftarrow \\ &\quad SupL = [\,] \wedge SubL = [\,] \\ &\vee\ SupL = [V_0|V_1] \wedge \texttt{delete\_all}(V_0, SubL, AuxL) \wedge \texttt{subset}(AuxL, V_1) \end{aligned}$$

3.5 The proof structure

We can abstract both proofs as follows

- Choose an induction parameter, and apply induction
- Deal with the base case, using the non-recursive part of the definition of "$\in$". This involves some rewriting, and is easy to manage.
- Unfold the step case, according to the recursive part of the definition of "$\in$", then try to match a part of the conjecture to the induction hypothesis. This can be guided by the rippling paradigm. If necessary, introduce auxiliary structures, which may be suggested by the temporary breakdown of rippling.

4 Analysis of the synthesis

From those proofs, we can see that the choice of the induction parameter has induced the main structure of the synthesized programs: recursivity on the variable chosen as induction parameter. This is in fact the expression of the recursion-induction parallelism in *Whelk*. Such close relationships between the proofs and the synthesized programs are a great help in mastering the synthesis of a program: if we want to generate a program with a main loop on one of its arguments, we have to choose that argument as induction parameter. Such relationships are not limited to induction/recursion: it is possible also to decide when to introduce auxiliary structures, and when to split in cases; and these choices can be motivated automatically.

But how do those programs compare with respect to some important properties in the logic programming field ?

time complexity (in the all-ground directionality). We can calculate that the worst-case time complexity has the same asymptotic behaviour in both cases – that is: $O(\text{length}(SubL) \times \text{length}(SupL))$. On average,

- in the case of success, the number of iterations is divided by two with respect to the worst case. What will make the difference between both programs is the need to completely build an auxiliary structure in the second case (not simply taking the tail of a list), which will give a higher constant.
- in the case of failure, the first program will not have to loop through the whole sublist to detect the failure, but the second will loop through both whole lists. This will add to the previous difference to deduce that the first program will better behave on average.

directionalities Apart from the all-ground mode, the only common usage of these programs would be to generate sublists of a given list. Neither of them, however, will behave as we might wish in that case: with a superlist such as $[a, b, c]$, they generate sublists $[], [a], [a, a], [a, a, a], \ldots$. While there is a technique to remove infinite looping of programs called in the non-ground modes [Wiggins 92a], the problem is not in the program, but in the laxity of the "multiset" subset specification. It can only be solved by proving the "real" subset specification.

difficulty of the proofs The first proof is easier to manage, involving no eureka step. But with the technique we used to introduce an auxiliary structure in the second program, none of these proofs can be considered as really difficult. We have outlined how this technique can be automated in terms of the rippling paradigm.

5 Conclusion

In this paper, we have explained how the *Whelk* logic program synthesis system has been guided to derive significantly different programs from the same specification. This has been illustrated by the *subset* example. The relationship between the structures of the proofs and the corresponding programs has been used to guide the proofs. Such an analysis can be the basis for automation of a synthesis process using *Whelk* as the synthesis engine.

We have outlined how the technique of rippling may be used to guide search for a proof in both cases of induction variable, so that once the choice is made, the proofs may be automated.

References

[Bundy 88] Alan Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[Bundy *et al.* 90a] A. Bundy, A. Smaill, and G. A. Wiggins. The synthesis of logic programs from inductive proofs. In J. Lloyd, editor, *Computational Logic*, pages 135–149. Springer-Verlag, 1990. Esprit Basic Research Series. Also available from Edinburgh as DAI Research Paper 501.

[Bundy *et al.* 90b] 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.

[Bundy *et al.* 91] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. Research Paper 567, Dept. of Artificial Intelligence, University of Edinburgh, 1991. In the Journal of Artificial Intelligence.

[Dayantis 87] G. Dayantis. Logic program derivation for a class of first order logic relations. In *Proceedings of IJCAI-87*, pages 9–14, 1987.

[Deville 90] Y. Deville. *Logic programming: systematic program development.* Addison-Wesley Pub. Co., 1990.

[Ireland 92] 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.

[Lloyd & Topor 84] J. W. Lloyd and R. W. Topor. Making Prolog more expressive. *Journal of Logic Programming*, 1(3):225–240, 1984.

[Wiggins 92a] G. A. Wiggins. Negation and control in automatically generated logic programs. In A. Pettorossi, editor, *Proceedings of META-92.* Springer Verlag, Heidelberg, 1992. LNCS Vol. 649.

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

[Wiggins *et al.* 91] G. A. Wiggins, Alan Bundy, I. Kraan, and J. Hesketh. Synthesis and transformation of logic programs through constructive, inductive proof. In K-K. Lau and T. Clement, editors, *Proceedings of LoPSTr-91*, pages 27–45. Springer Verlag, 1991. Workshops in Computing Series.