IsaWhelk: Whelk Interpreted in Isabelle*

David A. Basin
Max-Planck-Institut für Informatik
Im Stadtwald
D-66123, Saarbrücken, Germany
Email: basin@mpi-sb.mpg.de
Phone: (49) (681) 302-5435
Fax: (49) (681) 302-5401

Abstract

The Whelk logic has been proposed as a foundation for logic program synthesis. Here, I interpret the rules of Whelk as rules of first-order logic and derive them in Isabelle. Theoretically, this provides a means to understand the meta-theory behind Whelk, and its correctness. The interpretation suggests simplifications, corrections, and extensions. Practically, it provides a way to construct logic programs from proofs of their correctness by applying the formalized proof rules using higher-order resolution.

1 Introduction

Background

I have used Isabelle, a logical framework supporting proof construction by higher-order resolution, as a tool to reconstruct, simplify, implement, and use a proposed logic. The logic is Whelk, developed by Geraint Wiggins in [20], which represents an adaptation of the proofs-as-programs idea to logic program synthesis; the logic has been implemented and is in use at Edinburgh and other Universities. The Isabelle theory, which I call *IsaWhelk*, has functionality similar to Whelk, but is formally verified and very simple to understand.

In [20], Whelk is presented as a new kind of logic where specifications are manipulated in a special kind of "tagged" formal system. A tagged formula A is of the form $[\![A]\!]_{P(\overline{x})\leftrightarrow\phi}$.[1] The tag subscript represents a *pure logic program* [4] where P is an atom and ϕ a formula in a restricted subset of first-order logic. The Whelk logic manipulates these tags so that the tagged (subscripted) program should satisfy two properties. First, the tagged program should be logically *equivalent* to formula it tags in the appropriate first-order theory. To achieve this the proof rules state how to build programs for a given goal from programs corresponding to the subgoals. Second, the tagged program should be *decidable*, which means as a program it terminates in all-ground mode. One other feature of Whelk is that a proof may begin with a subformula of the starting goal labeled by a special operator ∂. At the end of the proof the Whelk system *extracts* the tagged program labeling this goal; hence Whelk may be used to synthesize logic programs.

*This research was funded by the German Ministry for Research and Technology (BMFT) under grant ITS 9102. The responsibility for the contents lies with the author. I thanks Tobias Nipkow and Larry Paulson for helping me with my teething pains with Isabelle. I also thank Alan Bundy, Ina Kraan, Sean Matthews, Andreas Tönne, and members of the Edinburgh Mathematical Reasoning Group for their encouragement and criticism of this and related work.

[1] For $\overline{x}$ some collection of variables. Both formulas and sequents are tagged. See sections 3.2 and 3.3 for the full story.

In my interpretation I replace the tagging mechanism by formulating tagged rules directly as equivalences between specifications and program schemas; hence, the Whelk rules constitute a simple calculus for manipulating equivalences. I have used Isabelle to derive these rules in the appropriate first-order theory or, in several cases, to uncover flaws in proposed rules. Under this interpretation, the tagged formula in the starting specification is translated to an equivalence between the specification and a meta-variable which is later incrementally instantiated to a program during resolution with the proof rules. I have used Isabelle's higher-order resolution facility to use these rules to synthesize programs in the manner of Whelk.

Motivation and Contributions

My motivations are twofold. First, I am interested in applying frameworks like Isabelle to implement verification and synthesis calculi where higher-order unification can be used to construct programs during proof [1, 2]. Second, to understand appropriate frameworks and techniques to automate the synthesis of logic programs. In [10, 11], we show how second-order unification combined with rewriting can be used to automatically synthesize logic programs from correctness proofs. This case study combines these motives, recasting Whelk directly as a set of rules in first-order logic where programs may be synthesized using higher-order unification.

This kind of formalization should be of interest to researchers working in logic programming, mechanized reasoning, and formal methods. First, just as formally developing and reasoning about the correctness of programs on a computer is worthwhile, so is the formalization of programming logics themselves and the analysis of their correctness. This work demonstrates how we may use a logical framework like Isabelle to formally study and give a machine checked account of the proof rules of a proposed logical system. My interpretation recasts each Whelk rule into an Isabelle rule which I show to be derivable. The correctness of the proposed logic can thereby be established with respect to the equivalence criteria above.

My second contribution is pedagogical: the interpretation simplifies and clarifies. Rather than formalizing the notation of Whelk directly I explain it using concepts available in the Isabelle theory. Tagging is accomplished by replacing tags with explicit equivalences in the relevant formulas. Extraction is accomplished via meta-variable instantiation. And whereas Whelk manipulates environments of parameter variables when reasoning about quantifiers, my approach instead uses higher-order unification to manipulate and propagate these variables in a uniform way. As well as simplifying, the interpretation brings us back to the familiar mathematical setting of first-order logic where questions of derivability and correctness may be treated using conventional and well understood means. The interpretation quickly exposed problems: some of the Whelk rules were invalid. Of course, these defects were present in the original Whelk theory, but their presence there is perhaps harder to ascertain.

This second contribution leads into the third: the interpretation provides a clear illustration of the ideas behind schematic proofs developed in [1]. The idea there is that many logics designed for program synthesis and transformation can be recast in within a logical framework like Isabelle (or ELF [18]) whereby higher-order unification is used to construct programs during proofs. This idea of unifying verification and synthesis through metavariables is not new. It goes back to Green's use of resolution not only for checking answers to queries, but also for synthesizing programs [9]. But what is surprising is that, cast in a higher-order setting, many different appearing calculi and synthesis techniques can be explained and simplified using this idea. Whelk is one such calculus. Related work in logic program synthesis is [10, 11], which has looked at using higher-order (pattern) unification as a means of "middle-out" synthesizing logic programs; Whelk is seen as an alternative to this approach. This work demonstrates that the two development methodologies are essentially the same.

Finally, this reconstruction demonstrates the practical benefits of using systems like Isabelle to implement such a synthesis calculus. Not only could I use Isabelle's higher-order unification to synthesize programs, I could also use the normalization and simplification tactics that are distributed with the system. I used these routines both to verify the translated Whelk rules, usually fully automatically, and also to construct synthesis proofs. For example, the subset theorem required only 15 simple steps in IsaWhelk as opposed to 105 in Whelk where such routines are only now being developed.

Organization

In 2 I provide background on Whelk and Isabelle. In 3 I present the translation and in 4 argue its correctness. In 5 I give the IsaWhelk proof of the example used to illustrate Whelk in [20]: the subset predicate. Finally I draw conclusions. Three appendicies are also included: the published Whelk rules, the derived IsaWhelk rules, and the entire proof script for the subset proof.

2 Background to Interpretation

Due to space considerations I only give very brief overviews of Whelk and Isabelle. Details on Whelk are found in [20][2] (see also [4]) and on Isabelle in [14, 15, 16, 17].

2.1 Overview of Whelk

The following summarizes [20]. Whelk is a sequent calculus implemented in Prolog. Its logical basis is constructive sorted first-order logic with equality, extended with an operator ∂ and rules for reasoning about ∂. Whelk proofs begin with a specification conjecture of the form $\forall \overline{x}.\partial\, S(\overline{x})$ where S is some formula, not involving ∂, possibily containing the x_i in the list of variables $\overline{x}$. At the end of the proof, a program named P is extracted that is equivalent to the specification within the ∂ operator, i.e., $\forall \overline{x}.S(\overline{x}) \leftrightarrow P(\overline{x})$. The Whelk rules are formulated top-down ("refinement style") and formulas are tagged; the tags explain how to build the extraction. [20] does not list all the rules for Whelk, only those necessary for the given example. These rules are summarized in my Appendix A.

Whelk rules are designed to produce programs equivalent to their specifications. The program is built using tagging information which is not displayed during proof. The constructed programs are represented as formulas in a *programming logic* $\mathcal{L}_P$ which is distinct from the *specification logic* $\mathcal{L}_S$. The programming logic is not fully defined but basically it is a subset of the specification logic where sorts are omitted as well as logical operators like negation (which is considered undesirable in logic programs). At the end of a Whelk proof, a *pure logic program* in $\mathcal{L}_P$ is constructed. These are universally quantified equivalences between an atom (the program name) and a program body where the body is a formula in the language $\mathcal{L}_P$; they are similar to the *logic descriptions* of [7]. For example, the following is a pure logic program for list membership (where *cons* is ".").[3]

$$\forall x\, y.p(x, y) \leftrightarrow (x = [] \wedge False) \vee (\exists v_0\, v_1.x = v_0.v_1 \wedge (y = v_0 \vee p(v_1, y))) \qquad (1)$$

Such programs can be translated to Horn clauses or run directly in a programming language like Gödel.

2.2 Overview of Isabelle

Isabelle is an interactive theorem prover developed by Larry Paulson at Cambridge. It is a logical framework; this means its logic is used as a meta-logic in which object logics (e.g., first-order logic, set theory, etc.) are encoded. Proofs are interactively constructed by applying proof rules using higher-order resolution. Proof construction may be automated using *tactics* which are ML programs in the tradition of LCF [8] that construct proofs.

[2] Although Wiggins tells me the Whelk logic is still under development and some rules have been revised since [20]. I have kept to rules and definitions in [20], since this is the most recent published account of the logic.

[3] "." is overloaded and also is used in the syntax of quantifiers; e.g., $\forall x\, y.\phi$ which abbreviates $\forall x.\forall y.\phi$.

Isabelle provides a fine basis for this work. First, since it is a logical framework, I could select an object logic appropriate for my interpretation. Fortunately, Isabelle is distributed with a constructive first-order logic with equality, similar to Whelk (without ∂); this spared me encoding this logic myself. This theory also came with well-developed tactic support for rewriting and simplification. Moreover, as Isabelle's metalogic is based on higher-order logic, it can formalize proof rules which are both hypothetical (depending on other formulas) or schematic (generalized over formulas); i.e., one can express and derive proof rules, and I used this to derive equivalents of the Whelk rules. Rules in Isabelle, primitive and derived, may be applied with higher-order unification where higher-order meta-variables may occur not only in proof rules, but also in the proof as well. I used this to construct proofs that built pure logic programs (as in Whelk) where the program was originally left unspecified as a higher-order meta-variable and filled in incrementally during the resolution steps. Hence Whelk extraction corresponds to a substitution instance.

Isabelle manipulates objects of the form[4] `[|F1; ...; Fn|] ==> F` where formulas may be schematic (contain metavariables). The symbol `==>` is meta-level implication and the above represents the formula `F1 ==> (... ==> (Fn ==> F)...)` in Isabelle's metalogic, which is a fragment of higher-order logic. (The reader should not confuse Isabelle's implication, `==>`, with implication in the object logic, or its higher-order universal quantifier "`!!`" with quantifiers in the object logic.) A proof in Isabelle proceeds by applying rules to such formulas which result in zero or more subgoals, possibly with different assumptions. When there are no more subgoals, the proof is complete.

Note that although proof rules are formalized natural deduction style, the above meta-level implication can be read as an intuitionistic sequent where the `Fi` are the hypotheses. Isabelle has resolution tactics which apply the natural deduction rules in a way the maintains this illusion of working with sequents.

3 The Translation

3.1 Language and Logical Foundation

Whelk without ∂ is sorted intuitionistic first-order logic with equality and I used this as the basis of the IsaWhelk theory. The only differences are that Isabelle's calculus is natural deduction (as opposed to sequent) and is not sorted. The first difference is not theoretically important (since we don't intend to study the proof theory, e.g., cut-elimination properties, of Whelk) nor is it a practical concern since, as noted above, Isabelle maintains the illusion of working with a constructive sequent. As for sorts, Isabelle is simply typed at the meta-level, and these meta-level types enforce similar restrictions as Whelk's. Hence, in what follows, types are omitted. They are explicitly given in the Whelk system and automatically inferred in Isabelle.

The Whelk theory comes with "built-in" data-types/axioms for natural numbers and lists. There are standard Isabelle theories for these data-types that we employed. E.g., a theory of lists that contains the empty list [], cons ".", and standard axioms like induction. The Whelk axioms were not published but they are likely similar. To sum up, the language and theory of our IsaWhelk theory is constructive first-order logic with equality extended with theories for numbers and lists. The exact formalization of these may be found in the standard Isabelle distribution (the theories IFOL, Nat2 and List) and the documentation [16]. I have extended this theory with two constants called `Extract` and `Def` with the property that `Extract(P) = Def(P) = P` for all formulas `P`; their role will be clarified later.

3.2 Translating ∂

Rather than adding the operator ∂ and tags to our Isabelle theory and axiomatizing the corresponding Whelk rules, I instead translate this notation and derive corresponding rules. The translation shall capture

[4] We shall use `typewrite font` to display concrete Isabelle syntax.

that ∂A represents the equivalence between A and a formula E which represents a program. In the goal, we do not know what E will be — it is to be constructed — so initially it is simply a meta-variable. In the rules, it is a combination of meta-variables that explains how to construct E from proofs of the subgoals. Below I expand on this and give examples.

3.2.1 Translation in the Specification Conjecture

∂A in the initial goal is translated to `A <-> Extract(?E(x1, ..., xn))` where the `xi` are the variables bound in the outer scope of A. The `xi` are necessary as formulas substituted for `E` should be able to reference such bound variables; this requires that `E` is second-order and takes the bound variables as arguments (as substitution is capture avoiding). For example, consider the subset specification which will constitute our synthesis example later.

$$\forall l.\forall m.\partial\,(\forall z.z \in l \to z \in m) \tag{2}$$

The synthesized program will be a function of l and m but not z. In Isabelle, we represent this as follows.

```
ALL l m. (ALL z. In(z,l) --> In(z,m)) <-> Extract(?E(l,m)))
```

The "?" symbol indicates metavariables in Isabelle. Note that the restriction in Whelk that ∂ appears exactly once in a formula of the form $\forall\overline{x}.\partial\,(S(\overline{x}))$ is unimportant in our setting. We could reason about multiple equivalences or conditional specifications.

3.2.2 Translation in the Rules

Formulas involving ∂ have associated tags that say how to build an equivalent formula. Tagged hypotheses look like $[\![\partial\,A]\!]_{P(\overline{x})\leftrightarrow\phi}$ whereas tagged conclusions have the tag placed around the entire sequent. Given a formula $\partial\,A$ with associated tag $[\![\cdot]\!]_{P(\overline{x})\leftrightarrow\phi}$, we remove the tag and translate $\partial\,A$ to `A <-> Ext`, where `Ext` represents ϕ. Hence, we simply eliminate the tag and express the equivalence directly in our logic. Moreover, we do away with the predicate name P as it is not needed, except for recursive calls, and we will present our solution to this problem shortly.

As an example, consider the Whelk rule for $\wedge$-I in the presence of ∂.[5]

$$\partial\wedge\text{-}I\;\frac{[\![\Gamma\vdash\partial\,A]\!]_{P(\mathcal{E})\leftrightarrow\phi}\quad[\![\Gamma\vdash\partial\,B]\!]_{P(\mathcal{E})\leftrightarrow\psi}}{[\![\Gamma\vdash\partial\,(A\wedge B)]\!]_{P(\mathcal{E})\leftrightarrow\phi\wedge\psi}}$$

Under our translation this rule becomes

```
[| A <-> Extract(ExtA); B <-> Extract(ExtB) |] ==> A & B <-> Extract(ExtA & ExtB)
```

which is Isabelle's way of representing the following natural deduction rule.

$$\frac{A\leftrightarrow \mathit{Extract}(\mathit{ExtA})\quad B\leftrightarrow \mathit{Extract}(\mathit{ExtB})}{A\wedge B\leftrightarrow \mathit{Extract}(\mathit{ExtA}\wedge \mathit{ExtB})}$$

The above rule is derived (recall that `Extract(P) = P`) in one step with Isabelle's simplification tactic for intuitionistic logic, so it is a valid rule. In 4.1 I argue it faithfully implements the Whelk rule. Note that this rule is initially postulated with free variables like `A` and `ExtA` which are treated as constants during proof of the rule; this prevents their premature instantiation during proof, which would lead to a proof of something more specialized. When the proof is completed, these variables are replaced by metavariables.

Unfortunately, the translation is not completely uniform because Wiggins' did not use his notation in a completely consistent way. The ∂ operator is supposed to indicate what parts of the specification should contribute to the synthesized program (see Section 3.3 of [20]) yet the logic includes rules for manipulating tagged formulas without this operator. Consider the Whelk rule "axiom".

[5] To differentiate rules involving ∂ from those without, I preface their name with ∂, e.g., $\partial\wedge$-I. Note that "I" and "E" refer to introduction and elimination as in [20]; "Left" and "Right" are actually standard for a sequent calculus presentation.

$$axiom\ \frac{}{[\![\Gamma, [\![A]\!]_\phi, \Delta \vdash A]\!]_{P(nil)\leftrightarrow\phi}}$$

This rule is only needed when A is headed by ∂, otherwise we can use the normal assumption rule of first-order logic. Hence I translate it as if A were ∂A. The result is derivable in Isabelle as

```
[| A <-> Extract(Ext) |] ==> A <-> Extract(Ext).
```

(Incidentally, this rule is not needed in our IsaWelk theory as it is a special case of Isabelle's standard assumption rule (applied with "assume_tac") which unifies the goal against an assumption.) The Whelk elimination rules and substitution rules also required this kind of translation. This revealed a problem with the substitution rule, discussed in 3.3.

There are two other subtleties in the translation — parameter variables and recursive naming — which I clarify below.

Predicate Parameters and Variable Environments

Whelk rules manipulate "environments" of variables (arguments to the defined predicate P, i.e., $\mathcal{E}$ in the above) which represent parameters of the synthesized predicate. I simplify this by listing parameters up front as bound variables to which the meta-variables in the starting goal are applied. For example, l and m in the subset specification. Such parameters need not be mentioned in the rules themselves (with one exception noted shortly) as Isabelle's higher-order unification properly propagates these parameters to subgoals. I explain this below.

Isabelle automatically *lifts* rules during higher-order resolution (see [14, 15]); this is a sound way of dynamically matching types of meta-variables during unification by applying them to new universally quantified parameters when necessary. This idea is best explained by an example. Consider applying the derived $\partial\wedge$-I rule to the following (made-up) goal.

```
ALL l m. ((ALL z. In(z,l)) & (Exists z. ~In(z,m))) <-> Extract(?E(l,m)))
```

In IsaWhelk, we begin proving goals by "setting up a context" where initial universally quantified variables become eigenvariables.[6] Applying $\forall$-I twice yields the following.

```
!! l m. ((ALL z. In(z,l)) & (Exists z. ~In(z,m))) <-> Extract(?E(l,m)))
```

Now if we try to apply the above derived rule for conjunction, Isabelle will automatically lift this rule to

```
!! l m. [| ?A(l,m) <-> Extract(?ExtA(l,m)); ?B(l,m) <-> Extract(?ExtB(l,m)) |] ==>
    ?A(l,m) & ?B(l,m) <-> Extract(?ExtA(l,m) & ?ExtB(l,m)),
```

which now unifies with `?A(l,m) = ALL z. In(z,l)`, `?B(l,m) = Exists z. ~In(z,m)`, and the program is instantiated with `?E(l,m) = ?ExtA(l,m) & ?ExtB(l,m)`. As the proof proceeds `?ExtA` and `?ExtB` are further instantiated.

Recursive Definitions

As explained, we translate formulas $[\![\partial A]\!]_{P(\overline{x})\leftrightarrow\phi}$ to an equivalence between A and a predicate `Ext` representing ϕ, without naming $P(\overline{x})$. This is a problem if `Ext` must recursively call itself (e.g., Equation 1) and arises during proof by induction where in the recursive step, use of the induction hypothesis yields a recursive call of the program we are building.

In Whelk, an induction rule sets up an auxiliary predicate definition that may be called in the induction hypothesis. Our analog is to prove theorems within a context that may be extended by new predicate

[6] By eigenvariables, we mean variables universally quantified outermost in the context. Recall universal quantification is the operator "`!!`" in Isabelle's meta-logic. See [15] for more details.

definitions. That is, the context contains not only axioms for defined function symbols (e.g., like `In` in the subset example) but it also contains a meta-variable that is instantiated during induction with new predicate definitions.

Back to the subset example; our starting goal actually includes a context which defines the axioms for `In` and includes a variable `?H` ("wrapped" by `Def`) which expands to a definition or series of definitions. These will be called from the program that instantiates `?E`.

```
[| ALL x. ~In(x,[]);  ALL x h t. In(x,h.t) <-> x = h | In(x,t) |]
   ==> Def(?H) --> (ALL l m. (ALL z. In(z,l) --> In(z,m)) <-> Extract(?E(l,m)))
```

The wrappers `Extract` and `Def` (recall these are the identity) restrict which IsaWhelk rules will unify against them. Without them, `?H` and `?E` would unify against everything and could be trivially instantiated by standard Isabelle tactics. By wrapping them unification is restricted to assumptions and rules that also contain explicitly wrapped variables (e.g., $\partial\wedge$-I above). This is further discussed in 4.1.

Definitions are set up during induction. Consider the Whelk-rule for list induction.

$$\textit{induction}\ \frac{\begin{array}{c}[\![\Gamma \vdash A\langle[\,]/x\rangle]\!]_{P'(\mathcal{E})\leftrightarrow\phi} \\ [\![\Gamma, [\![A\langle v_1/x\rangle]\!]_{P'(\mathcal{E}')} \vdash A\langle[v_0|v_1]/x\rangle]\!]_{P'(\mathcal{E})\leftrightarrow\psi}\end{array}}{[\![\Gamma \vdash A]\!]_{P(\mathcal{E})\leftrightarrow P'(\mathcal{E})}}$$

In this rule, $\mathcal{E}'$ is the binding environment at the time of application of the rule, and P' is defined by

$$P'(\mathcal{E}') \leftrightarrow x = [\,] \wedge \phi \vee \exists v_0.\exists v_1.x = [v_0|v_1] \wedge \psi.$$

This corresponds to the following derived rule which builds the program `P(x,y)`, which represents $P'(\mathcal{E})$; the first assumption sets up a new schematic definition for this program and the second and third correspond to the induction cases of the Whelk rule.

```
[| Def(ALL x y. P(x,y) <-> (x = [] & EA(y)) | (EX v0 v1. x = v0.v1 & EB(v0,v1,y)));
   ALL y. A([],y) <-> Extract(EA(y));
   !!m. ALL y. A(m,y) <-> Extract(P(m,y)) ==> ALL h y. A(h.m,y)  <-> Extract(EB(h,m,y)) |]
 ==> A(x,y) <-> Extract(P(x,y))
```

As in [1] we have written a tactic that applies such induction rules. Its resolution with the goal creates three subgoals (corresponding to the three assumptions above) but the first is immediately discharged by unifying against a `Def(?H)` in the context which sets up a recursive definition. This is precisely the role that `Def(?H)` serves. Actually, to allow for multiple recursive definitions, the induction tactic first duplicates the `Def(?H)`[7] before resolving with the induction rule. Also, it thins out (weakens) the instantiated definition in the two remaining subgoals.

There is an additional subtlety in the induction rule which concerns parameter arguments. Other rules did not take additional parameters but this is the exception; here `P` takes two arguments even though the induction is on only one of them. This is necessary as the rule must establish (in the first assumption) a definition for a predicate with a fixed number of universally quantified parameters and the number of these cannot be predicted at the time of the induction. My solution to this problem is ad hoc; I derive in Isabelle a separate induction rule for each number of arguments needed in practice (two are needed for the predicates synthesized in the subset example). Less ad hoc solutions exist; e.g., a tactic could uniformly generate such rules and their proofs in Isabelle, but this is not an interesting problem.

[7] This follows as `Def(?H)` equals `?H` and if we have an hypothesis `?H` then we can instantiate it with `?H1 & ?H2`. Instantiation is performed by unification with `&`-elimination and results in the two new assumptions `?H1` and `?H2` which are rewrapped with `Def`. This "engineering with logic" is accomplished by resolving with a derived rule that performs these manipulations.

3.3 Errors Uncovered by the Translation

I translated and derived most of Whelks rules as described above. The Whelk rules are listed in Appendix A (ordered as in [20]) and their translations in Appendix B. The translated rules are given with their names, which are bound to ML identifiers for use during proof construction.

Most rules were easily translated and derived, analogous to $\partial\wedge$-I rule above. Below are the rules which require some explanation as they involve non-obvious parameter manipulation or the translation/derivation revealed problems with the given Whelk rule. The proofs for the derivable rules were usually constructed automatically using Isabelle's standard proof procedure for intuitionistic logic.

$\partial\forall$-I

[20] gives two forms of $\forall$ introduction involving ∂, distinguishing between $\forall$ outside and inside the scope of ∂ (see Appendix A). In IsaWhelk the first is not necessary given our translation of parameters; $\forall$-I achieves the same effect.

$\partial\forall$-E

Whelk has two rules depending if $\forall$ is inside or outside of ∂. [20] explains that in both, "$\mathcal{E}''$ is $\mathcal{E}'$ prepended to $(t).nil$". This means that t is an argument to the appropriate program.

$$\partial\forall\text{-}E\;\frac{[\![\Gamma, [\![\partial\forall x.A]\!]_{P(\mathcal{E}')}, \Delta, [\![\partial A\langle t/x\rangle]\!]_{P(\mathcal{E}'')} \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{[\![\Gamma, [\![\partial\forall x.A]\!]_{P(\mathcal{E}')}, \Delta \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}$$

$$\partial\forall\text{-}E\;\frac{[\![\Gamma, [\![\forall x.\partial A]\!]_{P(\mathcal{E}')}, \Delta, [\![\partial A\langle t/x\rangle]\!]_{P(\mathcal{E}'')} \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{[\![\Gamma, [\![\forall x.\partial A]\!]_{P(\mathcal{E}')}, \Delta \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}$$

The translations of these are subtle since they involve parameter variables. Let us start with the second rule. If Whelk were natural deduction it would be of the form

$$\frac{[\![\forall x.\partial A]\!]_{P(\mathcal{E}')}}{[\![\partial A\langle t/x\rangle]\!]_{P(\mathcal{E}'')}}.$$

As ∂A occurs within the scope of $\forall x$ the premise is translated as `ALL x.A(x) <-> Extract(Ext(x))`. In the conclusion, the variable x replaced by an arbitrary t; this may be modeled by letting x be a meta-variable which can later be unified to some t. In Isabelle this is accomplished by stating the rule with a constant `x` and after derivation `x` is generalized to a meta-variable. The extension of the variable environment $\mathcal{E}''$ with t means that in the synthesized fragment P the variable x receives the same instantiation. Hence the conclusion translates to `A(x) <-> Extract(Ext(x))` and the IsaWhelk rule is

```
(ALL x.A(x) <-> Extract(Ext(x))) ==>  A(x) <-> Extract(Ext(x)).
```

which is indeed derivable. If the reader finds this translation confusing, he may find it helpful to reflect on the axiom for standard $\forall$-E in Isabelle which is `(ALL x.F(x)) ==> F(x)`

What about the first rule? As the universally quantified variable is within the scope of ∂ it cannot be an argument to the synthesized program fragment. I.e., the premise of the inference rule must be `(All x. A(x)) <-> Extract(Ext)` with `Ext` *not* dependent on `x`. The conclusion, as in the second rule, extends the environment of the synthesized predicate P with t. I.e., the synthesized fragment is to be instantiated. This translates to `A(x) <-> Extract(Ext(x))`. But we cannot combine these into

```
((All x. A(x)) <-> Extract(Ext)) ==> A(x) <-> Extract(Ext(x)).
```

The problem is that Isabelle's type checker assigns different types to the `Ext` in the premise and the conclusion (one is a function, the other isn't). This is not merely a problem caused by restrictions in Isabelle; a real problem has been revealed, which could also be observed by close inspection of the Whelk rule. That is, the predicate P being defined cannot be instantiated by t since there is no corresponding variable to be instantiated. That is, the universally quantified variable x is within, not without ∂ and the predicate P cannot be defined in terms of this variable.

Tagged Formulas without ∂

In 3.2 I noted that the Whelk notation is not used consistently as there are rules manipulating tagged formulas without the ∂ operator (i.e., *axiom*, $\partial \vee -E$, *induction*, *sub*, *rewrite*, and *lemma* in Appendix A). I translated these as if the goal formulas were headed by ∂ and this led to derivable rules and was consistent with intention of Whelk to synthesize programs for such formulas. The only problem that arose is that the substitution rule was invalid under this interpretation.

The Whelk substitution rule is the following.[8]

$$sub\ \frac{\Gamma \vdash [\![B \leftrightarrow C]\!]_{P(\mathcal{E})\leftrightarrow\psi} \qquad \Gamma \vdash [\![A\langle C/B\rangle]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{\Gamma \vdash [\![A]\!]_{P(\mathcal{E})\leftrightarrow\psi\wedge\phi}}$$

How do we interpret a formula like $[\![B \leftrightarrow C]\!]_{P(\mathcal{E})\leftrightarrow\psi}$ which does not have a leading ∂ operator but is tagged none-the-less? If we translate all of these tagged formulas into equivalences then the resulting rule is not derivable.[9] It is hard to imagine what was intended by this rule. I have omitted it; it is not necessary for proof construction.

4 Correctness of Translation and Extensions

4.1 Adequacy and Faithfulness of Encoding

I have used Isabelle to formally establish the correctness of the translated rules when this was the case. But do these rules really model the Whelk rules? That is, is there a bijection between Whelk proofs and IsaWhelk proofs that respects our interpretation of ∂ and constructs the same program.

No such bijection exists if we allow unrestricted use of Isabelle's rules in constructing IsaWhelk proofs. That is, Isabelle has more rules available than Whelk (we needn't consider tactics here as these expand to sequences of primitive rules), in particular, rules for manipulating Isabelle's metalogic. For example, we may directly instantiate an Isabelle meta-variable rather than instantiate it using unification during resolution with an IsaWhelk rule; this has no counterpart in Whelk as there are no meta-variables present and extraction is manipulated out of reach of the user.

If we restrict ourself to "core" IsaWhelk, consisting of the application of the rules of first-order logic (by the resolution tactics or simplifiers) and resolution using the IsaWhelk rules, then the correspondence holds.[10] This must be established by induction on proofs analyzing each proof rule case-by-case. Such a proof is straightforward but tedious. We sketch part of one case here for concreteness.

Consider the $\partial\wedge$-I rule from 3.2.2. Let $\partial(A \wedge B)$ be a formula in a Whelk proof to which $\partial\wedge$-I is applied. That is, we prove $\partial(A \wedge B)$ building the program $P(\mathcal{E}) \leftrightarrow \phi \wedge \psi$ from proofs of ∂A and ∂B which have built programs $P(\mathcal{E}) \leftrightarrow \phi$ and $P(\mathcal{E}) \leftrightarrow \psi$ respectively. Assume for concreteness that $\mathcal{E}$ is

[8] The rule in [20] contained a typo. Wiggins tells me that this rule has now been deleted from Whelk.

[9] Here is a countermodel. Let $A[B]$ denote a formula A with a distinguished subformula B (where B does not contain any free variables). The rule states that we can derive $A[B] \leftrightarrow \psi \wedge \phi$ from the assumptions $A[C] \leftrightarrow \phi$ and $(B \leftrightarrow C) \leftrightarrow \psi$. Let $B = True$, $C = False$, $A[C] = C$ (so $A[B] = B$), and $\psi = \phi = False$; the assumptions are true but the conclusion is false.

[10] Overlooking a trivial difference: in Whelk the constructed program $P(\overline{x}) \leftrightarrow \phi$ always must be named (i.e., P) whereas names are only created in IsaWhelk by applying the induction rule. However, this can be viewed as a feature of IsaWhelk rather than a bug. In Whelk, extra dummy predicates end up being defined, e.g., when a proof begins with an induction.

the vector of variables $x_1 \ldots x_k$. This Whelk step translates directly to an IsaWhelk step. The goal proved translates to `A & B <-> Extract(?E(x1, ..., xk)`. We resolve this with the corresponding IsaWhelk rule (`WAndI`) and Isabelle lifts this rule over the parameters `x1, ..., xk`. The lifted rule produces the two subgoals `A <-> Extract(?EA(x1, ..., xk))` and `B <-> Extract(?EB(x1, ..., xk))` and unifies `?E(x1, ..., xk)` to `?EA(x1, ..., xk) & ?EB(x1, ..., xk)`. But these are just the translations of the corresponding Whelk subgoals and we have produced the same program fragment in both cases. Arguing the bijection in the other direction (the IsaWhelk rule to the Whelk rule) is similar.

The above argument should not be surprising. The IsaWhelk rules are simply the natural deduction implementation of the Whelk rules where equivalence and the `Extract` label are used to indicate Whelk's tags. The only essential difference is the way we handle bound variables and lifting accounts for the correspondence here. Despite this simple correspondence, I find that casting the Whelk rules as first-order rules makes them easier to understand and reason about.

Note that `Extract` and `Def` wrappers play an important role in the above. Without them, the meta-variables they surround could become accidentally instantiated in an IsaWhelk proof. For example, we could prove any theorem by assumption since we have a meta-variable in our assumption list which could match any goal. The `Extract` label ensures that the only way to build an extraction is with the specific IsaWhelk rules that manipulate these tagged formulas. Similarly, the Induction rules are the only ones that set up new definitions and can instantiate `Def` wrapped variables.

4.2 Decidability

Given the above, any property holding for Whelk programs also holds for IsaWhelk programs. One such property, called "decidability" is presented in [20]. Before examining this though, we consider more generally what properties we might desire from programs synthesized through IsaWhelk or Whelk proofs.

In IsaWhelk it is explicit that proofs may extend a context with new definitions and hence are proofs under assumption. If the assumptions synthesized are inconsistent, the predicate may not define the relation expected. This is also the case in Whelk; in both, we wish that the predicates we define really define mathematical relations. This is analogous to proofs in provers like Boyer-Moore NQTHM prover [3] where, when users define functions, the system checks to see if these really define total functions. In the case of NQTHM this is demonstrated by showing termination. That is, recursive function calls are on arguments smaller in some well-founded order. I prove something analogous here.

Theorem 4.1 (IsaWhelk Rules Define Predicates) *If P is defined by a (completed) IsaWhelk proof, then recursive calls in P are on arguments smaller under some well-founded ordering.*

Proof 4.1 This follows by analysis of the induction rules which set up recursive definitions. Consider WListInduct2 given in 3.2.2. This sets up the definition

$$\forall x\, y. P(x, y) \leftrightarrow (x = [] \wedge EA(y)) \vee (\exists v_0 v_1. x = v_0.v_1 \wedge EB(v_0, v_1, y)))$$

with two subgoals that serve to instantiate EA and EB. In the first goal, the base-case, we must prove $\forall y. A([], y) \leftrightarrow \mathit{Extract}(EA(y))$ without additional assumptions. So $EA(y)$ cannot reference the named predicate P as it is not a part of the context of this goal. In the second subgoal we must prove $\forall h\, y. A(h.m, y) \leftrightarrow \mathit{Extract}(EB(h, m, y))$ under the assumption $\forall y. A(m, y) \leftrightarrow \mathit{Extract}(P(m, y))$ with m an eigenvariable. Here now is the essence of the proof: if we wish to instantiate $EB(h, m, y)$ with a term containing the recursive call, that recursive call must be of the form $P(m, t)$ for some term t. Critically though, m is fixed (an eigenvariable) and hence there is a well-founded order that the recursive call is smaller on; namely the first argument (the argument we did induction on) is smaller in the order given by the induction schema. □

Note that this theorem is weaker than saying that the defined predicate, when executed as a logic program, will terminate on all-ground queries. But this holds as a corollary when the rules meet certain simple syntactic conditions. That is when all base relations (e.g., $True$ and $False$ and also equality) are associated with recursive procedures and the non-inductive proof rules combine predicates with operators whose truth we can decide via (taking boolean combinations of) the truth of the subcomponents. This is the case for the IsaWhelk and Whelk rules. As for the induction rule, the theorem above indicates calls are on smaller arguments so evaluation of these will terminate. Note that only the induction rule introduces quantifiers into the program body but these are existential quantifiers which appear in formulas of the form $\exists v_0 v_1 . x = v_0 . v_1 \ldots$ which can trivially be decided (or compiled out in the translation to Horn clauses).

The above properties are different though from the "decidability criteria" that [20] claims for Whelk. Whelk rules are said to preserve decidability, in the sense that if ∂A is proven then we know $A \vee \neg A$. The proof of this property (labeled "Correctness of Synthesis Logic" in Section 3.6) is the following:

> *Theorem:* The rules of Whelk including ∂ are correct with respect to the interpretation of ∂.
> *Proof:* By definition, $\vdash \partial A$ iff $\vdash A \vee \neg A$. Rewrite any Whelk rule under this equivalence. The resulting rule is necessarily derivable from the rules of $LJ_{\tau,=}$.

It seems, however, what is really desired is that we have an effective way of executing the bodies of our pure logic programs. [20] explains:

> In order to ensure that we can derive a program from the proof, and to ensure that the program is then executable, we must place restrictions on the proof system we use. The most straightforward way to achieve the necessary restriction is to require that the logic we use for the proof be *constructive*.

I believe termination is best argued as I have above. Furthermore, my account does not rely on the fact that the proof system is constructive. We can use excluded middle to reason about the equivalence between a program and its specification; what is important is the syntactic form of the program, as discussed above.

4.3 Extensions

The above discussion highlights what we require from our rules, and hence suggests how we could enhance and extend IsaWhelk. The equivalence criteria stipulates that the IsaWhelk rules must form a calculus of equivalence reasoning. Additional criteria can be formalized for example to ensure that programs terminate in all ground mode. Or perhaps additionally, that they don't contain negation or other operators that might effect their behavior in other modes. When these criteria are reducible to syntactic properties of the programs synthesized, they may be easily checked for any IsaWhelk rule, or extension we might make.

As one example, we may choose to add new base predicates. Since equality is a part of most programming logic languages, we have added the rules `EqInstance` and `UnEqInstance` to IsaWhelk. These are `A = B <-> Extract(A = B)` and `~(A = B) <-> Extract(~(A = B))` respectively. These rules are trivially derived (definition expansion). As another example, we may form new rules in Isabelle by using resolution to combine given rules. For example, if we have the following rule `WImpiI`

```
[| ?A <-> Extract(?ExtA); ?B <-> Extract(?ExtB) |] ==> (?A --> ?B) <-> Extract(?ExtA --> ?ExtB)
```

then we may resolve it against itself (typing `WImpI RS WImpI`) resulting in the new rule

```
[| ?A1 <-> Extract(?ExtA1); ?B1 <-> Extract(?ExtB1); ?B <-> Extract(?ExtB) |] ==>
  ((?A1 --> ?B1) --> ?B) <-> Extract((?ExtA1 --> ?ExtB1) --> ?ExtB)
```

If we know that the original rules are correct, then their composition is also correct, both logically, and under syntactically based citeria (e.g., avoiding negation, etc.) we have established.

5 Program Development

I illustrate program synthesis in IsaWhelk using the subset relation example given in [20]. The proof I give closely follows the proof outline (Wiggins gave only selected steps) there. The example illustrates the translation and how IsaWhelk proofs reflect Whelk proofs. It also illustrates the schematic proof idea, and the benefits of building on top of a system like Isabelle that has a well developed tactic collection.

The entire proof required 15 steps and is given in Appendix C along with comments. Here I replay Isabelle's response to these proof steps, i.e., the instantiated top-level goal and subgoals that it generates after each step. The output is taken directly from an Isabelle session except, to save space, I have combined a couple steps, "pretty printed" formulas, and abbreviated variable names (e.g., contracting `ExtB11` to `EB11`).

The proof begins by giving Isabelle the subset specification (see 3.2). Isabelle prints back the goal to be proved (at the top) and the subgoals necessary to establish it. As the proof proceeds, the theorem becomes specialized as `?H` is incrementally instantiated with our program. We have also given the names `inbase` and `instep` to the context assumptions that define the membership predicate `In`.

```
Def(?H) --> (ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Extract(?E(l, m)))
 1. Def(?H) --> (ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Extract(?E(l, m)))
val inbase = "ALL x. ~ In(x, [])"
val instep = "ALL x h t. In(x, h . t) <-> x = h | In(x, t)"
```

The first proof step, invoked by the tactic `SetUpContext`, moves the definition variable `?H` into the assumption context and, as discussed in 3.2.2, promotes universally quantified variables to eigenvariables so our IsaWhelk rules may be used via lifting.

```
Def(?H) --> (ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Extract(?E(l, m)))
 1. !!l m. Def(?H) ==> (ALL z. In(z, l) --> In(z, m)) <-> Extract(?E(l, m))
```

Next, we invoke our induction tactic that applies the IsaWhelk rule `WListInduct2` with the appropriate instantiations to specify induction on `l`. This instantiates our schematic definition `?H` with the first schematic definition `?P` and a placeholder `?Q` for further instantiation. Note too that `?E` has been instantiated to this schematic program `?P`.

```
Def((ALL x y. ?P(x, y) <-> x = [] & ?EA10(y) | (EX v0 v1. x = v0 . v1 & ?EB11(v0, v1, y))) &
    ?Q) -->
(ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Extract(?P(l, m)))
 1. !!l m y. Def(?Q) ==> (ALL z. In(z, []) --> In(z, y)) <-> Extract(?EA10(y))
 2. !!l m ma h y.
       [| Def(?Q); ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Extract(?P(ma, y)) |] ==>
       (ALL z. In(z, h . ma) --> In(z, y)) <-> Extract(?EB11(h, ma, y))
```

We now prove the first case, which is the base-case (and omit printing the step case in the next two steps — Isabelle maintains a goal stack). Wiggins applies Whelk's $\partial\forall$-I rule followed by introduction of `True`; we do the same. First we apply IsaWhelk's `WAllI` which promotes the $\forall$-quantified variable `z` to an eigenvariable. The new subgoal becomes (as this step does not instantiate the theorem we are proving, we omit redisplaying it) the following.

```
 1. !!l m y z. Def(?Q) ==> (In(z, []) --> In(z, y)) <-> Extract(?EA10(y))
```

Next we apply `WTrue` (see Appendix C) which models the rule in Whelk that states if `?EA10(y)` is `True`, the above is provable provided `In(z, []) --> In(z, y)` is provable. This reduces the goal to one of ordinary logic (without `Extract`) as it instantiates the base case with the proposition `True`.

```
Def((ALL x y. ?P(x, y) <-> x = [] & True | (EX v0 v1. x = v0 . v1 & ?EB11(v0, v1, y))) &
    ?Q) -->
(ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Extract(?P(l, m)))
 1. !!l m y z. Def(?Q) ==> In(z, []) --> In(z, y)
```

Finally we complete this step by applying Isabelle's predicate-calculus simplification routines augmented with the formula `inbase`. We are left with the following step case (which is now the top goal on the stack and hence numbered 1).

```
 1. !!l m ma h y.
       [| Def(?Q); ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Extract(?P(ma, y)) |] ==>
       (ALL z. In(z, h . ma) --> In(z, y)) <-> Extract(?EB11(h, ma, y))
```

We now normalize this goal by applying the tactic

```
SIMP_TAC (list_ss addrews [instep,AndImp,AllAnd,AllEqImp]) 1
```

This calls Isabelle's simplification tactic which applies basic simplifications for the theory of lists, `list_ss`, augmented with the equation `instep` and the following lemmas `AndImp`, `AllAnd` and `AllEqImp`.

```
(A | B --> C) <-> (A --> C) & (B --> C)
ALL v. A(v) & B(v)) <-> (ALL v.A(v)) & (ALL v.B(v))
(ALL v. v = w --> A(v)) <-> A(w)
```

Each of these had been previously proven (in one step) with Isabelle's predicate calculus simplifier. This normalization step simplifies our subgoal to the following.

```
 1. !!l m ma h y.
       [| Def(?Q); ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Extract(?P(ma, y)) |] ==>
       In(h, y) & (ALL v. In(v, ma) --> In(v, y)) <-> Extract(?EB11(h, ma, y))
```

Analogous to the Whelk proof, we apply `WAndI`, which yields two subgoals which are (simplifications of) the equations found in (4) and (5) in [20]. The predicate `?P` we are building parallels that which Wiggins builds in his proof.

```
Def((ALL x y. ?P(x, y) <-> x = [] & True | (EX v0 v1. x = v0 . v1 & ?EA21(v1, v0, y) & ?EB22(v1, v0, y))) &
    ?Q) -->
(ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Extract(?P(l, m)))
 1. !!l m ma h y.
       [| Def(?Q); ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Extract(?P(ma, y)) |] ==>
       In(h, y) <-> Extract(?EA21(ma, h, y))
 2. !!l m ma h y.
       [| Def(?Q); ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Extract(?P(ma, y)) |] ==>
       (ALL v. In(v, ma) --> In(v, y)) <-> Extract(?EB22(ma, h, y))
```

We immediately solve the second subgoal by resolving with the induction hypothesis. I.e., after $\forall$-E we unify the conclusion with the induction hypothesis using Isabelle's assumption tactic. This instantiates the program we are building by replacing `?EB22` with a recursive call to `?P` as follows.

```
Def((ALL x y. ?P(x, y) <-> x = [] & True | (EX v0 v1. x = v0 . v1 & ?EA21(v1, v0, y) & ?P(v1, y))) & ?Q) -->
(ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Extract(?P(l, m)))
```

Returning to the first goal (to build a program for `?EA21`), we induce again and the base case is proved identically to the first induction except rather than introducing `True` with `WTrue` we introduce `False` with `WFalse` and solve the remaining propositional goal by simplification. This leaves us with the step case.

```
Def((ALL x y. ?P(x, y) <-> x = [] & True | (EX v0 v1. x = v0 . v1 & ?Pa(v0, y) & ?P(v1, y))) &
    (ALL x y. ?Pa(y, x) <-> x = [] & False | (EX v0 v1. x = v0 . v1 & ?EB32(v0, v1, y))) &
    ?Q27) -->
(ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Extract(?P(l, m)))
 1. !!l m ma h y mb ha ya.
       [| ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Extract(?P(ma, y));
          Def(?Q27); ALL y. In(y, mb) <-> Extract(?Pa(y, mb)) |] ==>
       In(ya, ha . mb) <-> Extract(?EB32(ha, mb, ya))
```

As before, we normalize this subgoal with Isabelle's standard simplifier.

```
 1. !!l m ma h y mb ha ya.
       [| ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Extract(?P(ma, y));
          Def(?Q27); ALL y. In(y, mb) <-> Extract(?Pa(y, mb)) |] ==>
       ya = ha | In(ya, mb) <-> Extract(?EB32(ha, mb, ya))
```

Applying `WOrI` unifies `?EB32(v0, v1, y)` with `?EA40(v1, v0, y) | ?EB41(v1,v0, y)` and yields a subgoal for each disjunct.

```
 1. !!l m ma h y mb ha ya.
       [| ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Extract(?P(ma, y));
          Def(?Q27); ALL y. In(y, mb) <-> Extract(?Pa(y, mb)) |] ==>
       ya = ha <-> Extract(?EA40(mb, ha, ya))
 2. !!l m ma h y mb ha ya.
       [| ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Extract(?P(ma, y));
          Def(?Q27); ALL y. In(y, mb) <-> Extract(?Pa(y, mb)) |] ==>
       In(ya, mb) <-> Extract(?EB41(mb, ha, ya))
```

In the first we apply `EqInstance` which instantiates `?EA40(v1, v0, y)` with `y = v0`. This completes the first goal leaving only the following.

```
Def((ALL x y.
        ?P(x, y) <-> x = [] & True | (EX v0 v1. x = v0 . v1 & ?Pa(v0, y) & ?P(v1, y))) &
    (ALL x y. ?Pa(y, x) <-> x = [] & False | (EX v0 v1. x = v0 . v1 & (y = v0 | ?EB41(v1, v0, y)))) &
    ?Q27) -->
(ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Extract(?P(l, m)))
 1. !!l m ma h y mb ha ya.
       [| ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Extract(?P(ma, y));
          Def(?Q27); ALL y. In(y, mb) <-> Extract(?Pa(y, mb)) |] ==>
       In(ya, mb) <-> Extract(?EB41(mb, ha, ya))
```

We complete the proof by resolving with the induction hypothesis. Isabelle prints back the following proven formula with no remaining subgoals.

```
  [| ALL x. ~ ?In(x, []);
     ALL x h t. ?In(x, h . t) <-> x = h | ?In(x, t) |] ==>
  Def((ALL x y. ?P(x, y) <-> x = [] & True | (EX v0 v1. x = v0 . v1 & ?Pa(y, v0) & ?P(v1, y))) &
      (ALL x y. ?Pa(x, y) <-> x = [] & False | (EX v0 v1. x = v0 . v1 & (y = v0 | ?Pa(v1, y)))) &
      ?Q) -->
  (ALL l m. (ALL z. ?In(z, l) --> ?In(z, m)) <-> Extract(?P(l, m)))
```

Note that the context remains open (`?Q`) as we might have needed to derive additional predicates. Also observe that Isabelle never forced us to give the predicates built (`?P` and `?Pa`) concrete names; these were picked automatically during resolution when variables were renamed apart by the system.

The constructed program can be simplified and translated into a Gödel program similar to the one in [20]. Alternatively it can be directly translated into the following Prolog program.

```
p([],Y).                                    pa([],Y)      :- false.
p([V0|V1],Y)  :- pa(Y,V0), p(V1,Y).         pa([V0|V1],V0).
                                            pa([V0|V1],Y) :- pa(V1,Y).
```

6 Conclusion and Future Work

This report has focused on a particular calculus, Whelk, but I believe the lessons to be drawn are more general. In particular, logical frameworks based on higher-order unification (Isabelle, and Pfenning's ELF come to mind) provide excellent foundations for program synthesis. As they formalize rules, we can give formal accounts of the correctness of calculi (e.g., see [19] for an account of a logic for reasoning about first-order program schemata). As they feature higher-order unification, we may build programs incrementally during proof, where meta-variables and binding operators properly interact. As they feature tactic languages, or other kinds of automation support, we may quickly apply the logics we embed to problems of reasonable complexity.

These lessons are general, but the formalization of logic programming calculi is particularly simple since the programming language is a sublanguage of the specification language. In general, we do not require that specifications are executable or even first-order. Moreover, the equivalence between specifications and programs need not be proven constructively. For example, IsaWhelk programs could have been developed in classical first-order logic (all the proof construction rules we used are derivable there too); all that matters is that the rules build programs from an appropriate (e.g., Horn) subset of the specification language.

Other authors have argued that first-order logic is the proper foundation for reasoning about and transforming logic programs (e.g.,[6, 5]). Our work corroborates this, although it demonstrates too that there is a role for logical frameworks as first-order logic is to weak to formalize first-order proof rules (as such rules contain variables ranging over formulas). But there are benefits also to using even richer logics to manipulate first-order, and possibly higher-order, specifications. For example, in this paper I used a recursion schema corresponding to structural induction over lists. But for synthesizing logic programs with more complicated kinds of recursion (e.g., quick sort) we require general well-founded induction where the user may suggest an appropriate well-founded ordering. But formalizing well-foundedness requires quantifying over sets or predicates and, outside of set-theory, this is generally second-order. Exploring synthesis based on well-founded induction in a higher-order setting is a direction for future work.

References

[1] David Basin, Alan Bundy, Ina Kraan, and Sean Matthews. A framework for program development based on schematic proof. In *7th International Workshop on Software Specification and Design*, Los Angeles, December 1993. IEEE Computer Society Press.

[2] David Basin and Sean Matthews. A conservative extension of first order logic and its applications to theorem proving. In *13th Conference of the Foundations of Software Technology and Theoretical Computer Science*, pages 151–160, December 1993.

[3] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, 1979.

[4] 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 Re search Paper 501.

[5] K. L. Clark and S-Å. Tärnlund. A first order theory of data and programs. In B. Gilchrist, editor, *Information Processing*, pages 939–944. IFIP, 1977.

[6] K.L. Clark and S. Sickel. Predicate Logic: a calculus for deriving programs. In R. Reddy, editor, *Proceedings of IJCAI-77*, pages 419–420. IJCAI, 1977.

[7] Pierre Flener and Yves Deville. Towards stepwise, schema-guided synthesis of logic programs. In T. Clement and K.-K. Lau, editors, *Logic Program Synthesis and Transformation*, pages 46–64. Springer-Verlag, 1991.

[8] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[9] Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the IJCAI-69*, pages 219–239, 1969.

[10] Ina Kraan, David Basin, and Alan Bundy. Logic program synthesis via proof planning. In *Proceedings of LoPSTr-92*. Springer Verlag, 1992.

[11] Ina Kraan, David Basin, and Alan Bundy. Middle-out reasoning for logic program synthesis. In *10th International Conference on Logic Programming (ICLP93)*, pages 441–455, Budapest Hungary, 1993.

[12] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. Technical Report ECS-LFCS-01-159, University of Edinburgh, LFCS, May 1991.

[13] Tobias Nipkow. Term rewriting and beyond — theorem proving in Isabelle. *Formal Aspects of Computing*, 1:320–338, 1989.

[14] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[15] Lawrence C. Paulson. Introduction to Isabelle. Technical Report 280, Cambridge University Computer Laboratory, Cambridge, January 1993.

[16] Lawrence C. Paulson. Isabelle's object-logics. Technical Report 286, Cambridge University Computer Laboratory, Cambridge, February 1993.

[17] L.C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 77–90. Academic Press, 1990.

[18] Frank Pfenning. Logic programming in the LF logical framework. In *Logical Frameworks*, pages 149 – 181. Cambridge University Press, 1991.

[19] Mattias Waldau. Formal validation of transformation schemata. In T. Clement and K.-K. Lau, editors, *Logic Program Synthesis and Transformation*, pages 97–110. Springer-Verlag, 1991.

[20] Geraint A. Wiggins. Synthesis and transformation of logic programs in the Whelk proof development system. In K. R. Apt, editor, *Proceedings of JICSLP-92*, 1992.

Appendix A. Whelk Rules

$$axiom\ \frac{}{[\![\Gamma, [\![A]\!]_{\phi}, \Delta \vdash A]\!]_{P(nil)\leftrightarrow\phi}}$$

$$\partial\forall\text{-}I\ 1\ \frac{[\![\Gamma \vdash \partial\, A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{[\![\Gamma \vdash \forall x.\partial\, A]\!]_{P((x).\mathcal{E})\leftrightarrow\phi}} \qquad \partial\forall\text{-}I\ 2\ \frac{[\![\Gamma \vdash \partial\, A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{[\![\Gamma \vdash \partial\,(\forall x.A)]\!]_{P(\mathcal{E})\leftrightarrow\phi}}$$

$$\partial\wedge\text{-}I\ \frac{[\![\Gamma \vdash \partial\, A]\!]_{P(\mathcal{E})\leftrightarrow\phi} \quad [\![\Gamma \vdash \partial\, B]\!]_{P(\mathcal{E})\leftrightarrow\psi}}{[\![\Gamma \vdash \partial\,(A \wedge B)]\!]_{P(\mathcal{E})\leftrightarrow\phi\wedge\psi}}$$

$$\partial\vee\text{-}I\ \frac{[\![\Gamma \vdash \partial\, A]\!]_{P(\mathcal{E})\leftrightarrow\phi} \quad [\![\Gamma \vdash \partial\, B]\!]_{P(\mathcal{E})\leftrightarrow\psi}}{[\![\Gamma \vdash \partial\,(A \vee B)]\!]_{P(\mathcal{E})\leftrightarrow\phi\vee\psi}}$$

$$\partial \rightarrow\text{-}I\ \frac{[\![\Gamma \vdash \partial\, A]\!]_{P(\mathcal{E})\leftrightarrow\phi} \quad [\![\Gamma, A \vdash \partial\, B]\!]_{P(\mathcal{E})\leftrightarrow\psi}}{[\![\Gamma \vdash \partial\,(A \rightarrow B)]\!]_{P(\mathcal{E})\leftrightarrow\phi\rightarrow\psi}}$$

$$\partial_{\,true}\text{-}I\ \frac{[\![\Gamma \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{[\![\Gamma \vdash \partial\, A]\!]_{P(nil)\leftrightarrow true}} \qquad \partial_{\,false}\text{-}I\ \frac{[\![\Gamma \vdash \neg A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{[\![\Gamma \vdash \partial\, A]\!]_{P(nil)\leftrightarrow false}}$$

$$\partial\forall\text{-}E\ 1\ \frac{[\![\Gamma, [\![\partial\forall x.A]\!]_{P(\mathcal{E}')}, \Delta, [\![\partial\, A\langle t/x\rangle]\!]_{P(\mathcal{E}'')} \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{[\![\Gamma, [\![\partial\forall x.A]\!]_{P(\mathcal{E}')}, \Delta \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}$$

$$\partial\forall\text{-}E\ 2\ \frac{[\![\Gamma, [\![\forall x.\partial\, A]\!]_{P(\mathcal{E}')}, \Delta, [\![\partial\, A\langle t/x\rangle]\!]_{P(\mathcal{E}'')} \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{[\![\Gamma, [\![\forall x.\partial\, A]\!]_{P(\mathcal{E}')}, \Delta \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}$$

$$\partial\vee\text{-}E\ \frac{\begin{array}{cc}[\![\Gamma, A, \Delta \vdash C]\!]_{P(\mathcal{E})\leftrightarrow\phi} & [\![\Gamma, B, \Delta \vdash C]\!]_{P(\mathcal{E})\leftrightarrow\psi}\\ [\![\Gamma, \Delta \vdash \partial\, A]\!]_{P(\mathcal{E})\leftrightarrow\alpha} & [\![\Gamma, \Delta \vdash \partial\, B]\!]_{P(\mathcal{E})\leftrightarrow\beta}\end{array}}{[\![\Gamma, A \vee B, \Delta \vdash C]\!]_{P(\mathcal{E})\leftrightarrow(\alpha\wedge\phi)\vee(\beta\wedge\psi)}}$$

$$induction\ \frac{\begin{array}{l}[\![\Gamma \vdash A\langle[\,]/x\rangle]\!]_{P'(\mathcal{E})\leftrightarrow\phi}\\ [\![\Gamma[\![A\langle v_1/x\rangle]\!]_{P'(\mathcal{E}')} \vdash A\langle[v_0|v_1]/x\rangle]\!]_{P'(\mathcal{E})\leftrightarrow\psi}\end{array}}{[\![\Gamma \vdash A]\!]_{P(\mathcal{E})\leftrightarrow P'(\mathcal{E})}}$$

$$sub\ \frac{\Gamma \vdash [\![B \leftrightarrow C]\!]_{P(\mathcal{E})\leftrightarrow\psi} \quad \Gamma \vdash [\![A\langle C/B\rangle]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{\Gamma \vdash [\![A]\!]_{P(\mathcal{E})\leftrightarrow\psi\wedge\phi}} \qquad rewrite\ \frac{\Gamma \vdash [\![E]\!]_{P(\mathcal{E})\leftrightarrow\phi} \quad \Gamma \vdash D \leftrightarrow E}{\Gamma \vdash [\![D]\!]_{P(\mathcal{E})\leftrightarrow\phi}}$$

$$lemma\ \frac{[\![\Gamma, [\![B]\!]_{P'(nil)} \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}{[\![\Gamma \vdash A]\!]_{P(\mathcal{E})\leftrightarrow\phi}}$$

In the $\partial\forall$-E rules $\mathcal{E}''$ is $\mathcal{E}'$ prepended to $(t).nil$. In the rewrite rule D may be simplified to an equivalent E by rewriting. In the induction rule $\mathcal{E}'$ is the binding environment at the time of application of the rule, and P' is defined by

$$P'(\mathcal{E}') \leftrightarrow x = [\,] \wedge \phi \vee \exists v_0.\exists v_1.x = [v_0|v_1] \wedge \psi$$

Appendix B. IsaWhelk Rules

```
WAxiom: [| A <-> Extract(Ext) |] ==> A <-> Extract(Ext)

WAllI: [| !!x. A(x) <-> Extract(Ext) |] ==> (ALL x.A(x)) <-> Extract(Ext)

WAndI: [| A <-> Extract(ExtA); B <-> Extract(ExtB) |] ==> A & B <-> Extract(ExtA & ExtB)

WOrI: [| A <-> Extract(ExtA); B <-> Extract(ExtB) |] ==> A | B <-> Extract(ExtA | ExtB)

WImpI: [| A <-> Extract(ExtA); B <-> Extract(ExtB) |] ==> (A --> B) <-> (Extract(ExtA --> ExtB))

WTrue: [| A |] ==> A <-> Extract(True)

WFalse: [| ~A |] ==> A <-> Extract(False)

WAllEIn: [| ALL x.A(x) <-> Extract(Ext(x)) |] ==> A(x) <-> Extract(Ext(x))

WOrE:
[| A ==> (C <-> Extract(ExtA)); B ==> (C <-> Extract(ExtB));
   A <-> Extract(Alpha);  B <-> Extract(Beta);
   A | B |] ==> C <-> Extract((Alpha & ExtA) | (Beta & ExtB))

Rewrite: [| (E <-> D); E <-> Extract(Ext) |] ==> D <-> Extract(Ext)

WListInduct2:
[| Def(ALL x y. P(x,y) <-> (x = [] & ExtA(y)) | (EX v0 v1. x = v0.v1 & ExtB(v0,v1,y)));
   ALL y. A([],y) <-> Extract(ExtA(y));
   !!m. ALL y. A(m,y) <-> Extract(P(m,y))
           ==> ALL h y. A(h.m,y)  <-> Extract(ExtB(h,m,y)) |]
   ==> A(x,y) <-> Extract(P(x,y))

EqInstance:  A = B <-> Extract(A = B)

UnEqInstance: ~(A = B) <-> Extract(~(A = B))
```

Appendix C. IsaWhelk Proof Input

```
val [inbase,instep] = goal thy
"  [| ALL x. ~In(x,[]); \
\     ALL x h t. In(x,h.t) <-> x = h | In(x,t) |] \
\ ==> Def(?H) --> (ALL l m. (ALL z. In(z,l) --> In(z,m)) <-> Extract(?E(l,m)))";

(* After performing forall introductions, perform induction *)
by SetUpContext;
by (IndTac WListInduct2  [("x","l"),("y","m")] 1);

(* Base Case *)
br WAllI 1;
br WTrue 1;
by (cut_fast_tac [inbase] 1);

(* Step Case *)
by(SIMP_TAC (list_ss addrews [instep,AndImp,AllAnd,AllEqImp]) 1);
br WAndI 1;  (* Yield simplifications of Wiggins' equations 4 & 5 *)

(* Prove 2nd case immediately with induction hypothesis! *)
by (etac allE 2 THEN assume_tac 2);

(* First Case --- Do an induction on y to synthesize member(h,y) *)
by (IndTac WListInduct2  [("x","y"),("y","h")] 1);
br WFalse 1;   (* Induction Base Case *)
by(SIMP_TAC (list_ss addrews [inbase]) 1);
by(SIMP_TAC (list_ss addrews [instep]) 1);  (* Induction Step Case *)
br WOrI 1;
br EqInstance 1;
by (etac allE 1 THEN assume_tac 1);  (* Apply induction hypothesis *)
val Subset  = result();
```