Logic Frameworks for Logic Programs*

David A. Basin

Max-Planck-Institut für Informatik
Saarbrücken, Germany
Email:basin@mpi-sb.mpg.de

Abstract. We show how logical frameworks can provide a basis for logic program synthesis. With them, we may use first-order logic as a foundation to formalize and derive rules that constitute program development calculi. Derived rules may be in turn applied to synthesize logic programs using higher-order resolution during proof that programs meet their specifications. We illustrate this using Paulson's Isabelle system to derive and use a simple synthesis calculus based on equivalence preserving transformations.

1 Introduction

Background

In 1969 Dana Scott developed his Logic for Computable Functions and with it a model of functional program computation. Motivated by this model, Robin Milner developed the theorem prover LCF whose logic PPλ used Scott's theory to reason about program correctness. The LCF project [13] established a paradigm of formalizing a programming logic on a machine and using it to formalize different theories of functional programs (e.g., strict and lazy evaluation) and their correctness; although the programming logic was simple, within it complex theories could be developed and applied to functional program verification.

This paradigm can be characterized as *formal development from foundations*. Type theory and higher-order logic have been also used in this role. A recent example is the work of Paulson with ZF set theory. Although this theory appears primitive, Paulson used it to develop a theory of functions using progressively more powerful derived rules [24].

Most work in formalized program development starts at a higher level; foundations are part of an informal and usually unstated meta-theory. Consider, for example, transformation rules like Burstall and Darlington's well known fold-unfold rules [7]. Their rules are applied to manipulate formulas and derive new ones; afterwards some collection of the derived formulas defines the new program. The relationship of the new formulas to the old ones, and indeed which constitute the new program is part of their informal (not machine formalized) metatheory. So is the correctness of their rules (see [18, 8]). In logic programming the situation is similar; for example, [30, 29] and others have analyzed conditions required for fold-unfold style transformations to preserve equivalence of logic programs and indeed what "equivalence" means.

* 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 advice on Isabelle. I also thank Alan Bundy, Ina Kraan, Sean Matthews, and members of the Edinburgh Mathematical Reasoning Group for helpful discussions and collaboration on related work.

Development from Foundations in Logic Programming

We propose that, analogous to LCF, we may begin with a programming logic and derive within it a program development calculus. Derived rules can be applied to statements about program correctness formalized in the logic and thereby verify or synthesize logic programs. Logic programming is a particularly appropriate domain to formalize such development because under the declarative interpretation of logic programs as formulas, programs are formalizable within a fragment of first-order logic and are therefore amenable to manipulation in proof systems that contain this fragment. Indeed, there have been many investigations of using first-order logic to specify and derive correct logic programs [9, 10, 11, 17, 19]. But this work, like that of Burstall and Darlington, starts with the calculus rather than the foundations. For example in [17] formulas are manipulated using various simplification rules and at the end a collection of the resulting formulas constitutes the program. The validity of the rules and the relationship of the final set of formulas (which comprise the program) to the specification is again part of the informal meta-theory.

Our main contribution is to demonstrate that without significant extra work much of the informal metatheory can be formalized; we can build calculi from foundations and carry out proofs where our notion of correctness is more explicit. However, to do this, a problem must be overcome: first-order logic is too weak to directly formalize and derive proof rules. Consider for example, trying to state that in first-order logic we may replace any formula $\forall x.A$ by $\neg\exists x.\neg A$. We might wish to formulate this as $\forall x.A \rightarrow \neg\exists x.\neg A$. While this is provable for any instance A, such a generalized statement cannot be made in first-order logic itself; some kind of second-order quantification is required.[2] In particular, to formalize proof rules of a logic, one must express rules that (in the terminology of [15]) are *schematic* and *hypothetical*. The former means that rules may contain variables ranging over formula. The latter means that one may represent logical consequence; in the above example consequence has been essentially internalized by implication in the object language.

Rather than moving to a more powerful logic like higher-order logic, we show that one can formalize program development using weak logics embedded in logical frameworks such as Paulson's Isabelle system [25] or Pfenning's ELF [28]. In our work, a programming logic (also called the *object logic*) is encoded in the logic of the logical framework (the *meta-logic*). For example, the meta-logic of Isabelle, which we use, is fragment of higher-order logic containing implication (to formalize hypothetical rules) and universal quantification (to formalize schematic rules). Within this meta-logic we formalize a theory of relevant data-types like lists and use this to specify our notion of program correctness and derive rules for building correct programs. Moreover, Isabelle manipulates rules using higher-order unification and we use this to build programs during proof where meta-variables are incrementally instantiated with logic programs.

We illustrate this development paradigm by working through a particular example in detail. Within an Isabelle theory of first-order logic we formulate and derive a calculus for reasoning about equivalences between specifications and representations of logic programs in first-order logic. The derived calculus can be seen as a formal development of a logic for program development proposed in Wiggins (see Section 3.4). After derivation, we apply these rules using higher-order unification

[2] First-order *logic* is too weak, but it is possible to formalize powerful enough first-order *theories* to express such rules by axiomatizing syntax, e.g., [32, 3, 23]. However, such approaches require some kind of reflection facility to establish a link between the formalized meta-theory and the desired theory where such rules are used. See [4] for a further discussion of this. Moreover, under such an approach unification cannot be used to identify program verification and synthesis.

to verify that logic programs meet their specifications; the logic programs are given by meta-variables and each unification step during proof incrementally instantiates them.

Our experience indicates that this development is quite manageable. Isabelle comes with well-developed tactic support for rewriting and simplification. As a result, our derivation of rules was mostly trivial and involved no more than typing them in and invoking the appropriate first-order simplification tactics. Moreover, proof construction with these rules was partially automated by the use of Isabelle's standard normalization and simplification procedures. We illustrate this by developing a program for list subset.

2 Background to Isabelle

What follows is a brief overview of Isabelle [25, 26, 27] as is necessary for what follows. Isabelle is an interactive theorem prover developed by Larry Paulson. It is a logical framework: its logic serves 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 that construct proofs.

Isabelle provides a fine basis for our work. Since it is a logical framework, we may encode in it the appropriate object logic, first-order logic (although we indicate in Section 5 other possible choices). Isabelle's metalogic is based on the implicational fragment of higher-order logic where implication is represented by "`==>`" and universal quantification by "`!!`"; hence we can formalize and derive proof rules which are both hypothetical and schematic. Rules, primitive and derived, may be applied with higher-order unification during higher-order resolution; unification permits meta-variables to occur both in rules and proofs. We use this to build logic programs by theorem proving where the program is originally left unspecified as a higher-order meta-variable and is filled in incrementally during the resolution steps; the use of resolution is similar to the use of "middle out reasoning" to build logic programs as demonstrated in [20, 21, 6, 33].

Isabelle manipulates objects of the form[3] `[|F1; ...; Fn|] ==> F`. A proof proceeds by applying rules to such formulas which result in zero or more subgoals, possibly with different assumptions. When there are no subgoals, the proof is complete. Although Isabelle proof rules are formalized natural deduction style, the above implication can be read as an intuitionistic sequent where the `Fi` are the hypotheses. Isabelle has resolution tactics which apply rules in a way the maintains this illusion of working with sequents.

3 Encoding A Simple Equivalence Calculus

We give a simple calculus for reasoning about equivalence between logic programs and their specifications. Although simple, it illustrates the flavor of calculus and program development we propose.

3.1 Base Logic

We base our work on standard theories that come with the Isabelle distribution. We begin by selecting a theory of constructive first-order predicate calculus and

[3] We shall use `typewriter font` to display concrete Isabelle syntax.

augment this with a theory of lists to allow program development over this data-type (See *IFOL* and *List* in [27]). The list theory, for example, extends Isabelle's first-order logic with constants for the empty list "[]", cons ".", and standard axioms like structural induction over lists. In addition, we have extended this theory with two constants called `Wfp` (well-formed program) and `Def` with the property that `Wfp(P) = Def(P) = P` for all formulas `P`; their role will be clarified later.

The choice of lists was arbitrary; to develop programs over numbers, trees, etc. we would employ axiomatizations of these other data-types. Moreover, the choice of a constructive logic was also arbitrary. Classical logic suffices too as the proof rules we derive are clearly valid after addition of the law of excluded middle. This point is worth emphasizing: *higher-order unification, not any feature of constructivity, is responsible for building programs from proofs in our setting.*

In this theory, we reason about the equivalence between programs and specifications. "Equivalence" needs clarification since even for logic programs without impure features there are rival notions of equivalence. The differences though (see [22, 5]) are not so relevant in illustrating our suggested methodology (they manifest themselves through different formalized theories). The notion of equivalence we use is equivalence between the specification and a logic program represented as a *pure logic program* in the above theory. *Pure logic programs* themselves are equivalences between a universally quantified atom and a formula in a restricted subset of first-order logic (see [6] for details); they are similar to the *logic descriptions* of [12].

For example, the following is a pure logic program for list membership, where "." is *cons*.[4]

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

Such programs can be translated to Horn clauses or run directly in a language like Gödel [16].

3.2 Problem Specification

As our notion of correctness is equivalence between programs and specifications, our proofs begin with formulas of the form $\forall \overline{x}.(spec(\overline{x}) \leftrightarrow E(\overline{x}))$. The variables in $\overline{x}$ represent parameters to both the specification *spec* and the logic program E; we do not distinguish input from output. *spec* is a first-order specification and E is either a concrete (particular) pure logic program or a schematic (meta) variable standing in for such a program. If E is a concrete formula then a proof of this equivalence constitutes a *verification* proof as we are showing that E is equivalent to its specification. If E is a second-order meta-variable then a proof of this equivalence that instantiates E serves as a *synthesis* proof as it builds a program that meets the *spec*. If *spec* is already executable we might consider such a proof to be a *transformation* proof.

An example we develop in this report is synthesizing a program that given two lists l and m is true when l is a subset of m. This example has been proposed by others, e.g., [17, 33]. Slipping into Isabelle syntax we specify it as

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

Note that `ALL`, `-->` and `<->` represent first-order universal quantification, implication, and equivalence, and are declared in the definition of first-order logic. The "`?`" symbol indicates metavariables in Isabelle. Note that `?E` is a function of the input lists `l` and `m` but `z` is only part of the specification. Higher-order unification, which we use to build an instance for `?E` will ensure that it is only a function of `l` and `m`.

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

3.3 Rules

We give natural deduction rules where the conclusion explains how to construct `?E` from proofs of the subgoals. These rules form a simple calculus for reasoning about equivalences and can be seen as a reconstruction of those of the Whelk system (see Section 3.4). Of course, since `A <-> A` is valid, the synthesis specification for subset can be immediately proven by instantiating `?E` with the specification on the left hand side of the equivalence. While this would lead to a valid proof, it is uninteresting as the specification does not suggest an algorithm for computing the subset relation. To make our calculus interesting, we propose rules that manipulate equivalences with restricted right-hand sides where the right hand side can be directly executed.

Specifically, we propose rules that admit as programs (on the right hand sides of equivalences) formulas like the body of the membership predicate given above, but exclude formula like `ALL z. In(z,l) --> In(z,m)`. To do this we define inductively the set of such admissible formulas. They are built from a collection of (computable) base-relations and operators for combining these that lead to computable algorithms provided their arguments are computable. In particular, our base relations are the relations `True`, `False`, equality and inequality. Our operators will be the propositional connectives and existential quantification restricted to a use like that in the membership example, i.e., of the form $\exists v_0 v_1 . x = v_0 . v_1 \wedge P$ where P is admissible. This limited use of existential quantification is necessary for constructing recursive programs in our setting; it can be trivially compiled out in the translation to Horn clauses.

Note that to be strictly true to our "foundations" paradigm, we would specify the syntax of such well-formed logic programs in our theory (which we could do by recursively defining a unary well-formedness predicate that captures the above restrictions). However, to simplify matters we capture it by only deriving rules for manipulating these equivalences where the right-hand sides meet these restrictions. To ensure that only these rules are used to prove equivalences we will resort to a simple trick. Namely, we wrap all the right hand sides of equivalences in our rule, and in the starting specification with the constructor `Wfp`. E.g., our starting goal for the subset proof would really be

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

As `Wfp` was defined to be the identity (i.e., `Wfp(P)` equals `P`) it does not effect the validity of any of the rules. It does, however, affect their applicability. That is, after rule derivation we remove the definition of `Wfp` from our theory so the only way we can prove the above is by using rules that manipulate equivalences whose right hand side is also labeled by `Wfp`. In particular, we won't be able to prove

```
ALL l m. (ALL z. In(z,l) --> In(z,m)) <-> Wfp(ALL z. In(z,l) --> In(z,m)).
```

Basic Rules Figure 1 contains a collection of typical derived rules about equivalences. Many of the rules serve simply to copy structure from the specification to the program. These are trivially derivable, for example

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

Translating this into Isabelle we have

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

This rule is derivable (recall that `Wfp(P) = P`) in one step with Isabelle's simplification tactic for intuitionistic logic, so it is a valid rule. The rule allows us essentially

```
RAllI: [| !!x. A(x) <-> Wfp(Ext) |] ==> (ALL x.A(x)) <-> Wfp(Ext)

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

ROrI: [| A <-> Wfp(ExtA); B <-> Wfp(ExtB) |] ==> A | B <-> Wfp(ExtA | ExtB)

RImpI: [| A <-> Wfp(ExtA); B <-> Wfp(ExtB) |] ==> (A --> B) <-> (Wfp(ExtA --> ExtB))

RTrue: [| A |] ==> A <-> Wfp(True)

RFalse: [| ~A |] ==> A <-> Wfp(False)

RAllE: [| ALL x.A(x) <-> Wfp(Ext(x)) |] ==> A(x) <-> Wfp(Ext(x))

ROrE: [| A ==> (C <-> Wfp(ExtA)); B ==> (C <-> Wfp(ExtB)); A | B |] ==>
         C <-> Wfp(ExtA | ExtB)

EqInstance:  A = B <-> Wfp(A = B)
```

Fig. 1. Examples of Basic Rules

to decompose synthesizing logic programs for a conjunction into synthesizing programs for the individual parts. 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, which would lead to a proof of something more specialized. When the proof is completed, these variables are replaced by metavariables, so the rule may be later applied using unification.

There are two subtleties in the calculus we propose: parameter variables and induction rules. These are explained below.

Predicate Parameters Recall that problems are equivalences between specifications and higher-order meta-variables applied to parameters, e.g., `l` and `m` in the subset specification. We would like our derived rules to be applicable independent of the number of parameters involved. Fortunately, these do not need to be mentioned in the rules themselves (with one exception noted shortly) as Isabelle's higher-order unification properly propagates these parameters to subgoals. This is explained below.

Isabelle automatically *lifts* rules during higher-order resolution (see [25, 26]); 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 above conjunction rule to the following (made-up) goal.

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

In our theory, we begin proving goals by "setting up a context" where initial universally quantified variables become eigenvariables.[5] Applying ∀-I (∀-intro of first-order logic) twice yields the following.

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

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

[5] By eigenvariables, we mean variables universally quantified outermost in the context. Recall universal quantification is the operator "`!!`" in Isabelle's meta-logic. See [26] for more details.

```
!! 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 resolves 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 Our calculus so far is trivial; it copies structure from specifications into programs. One nontrivial way of transforming specifications is to admit proofs about equivalence by induction over the recursively defined data-types. But this introduces a problem of how predicates recursively call themselves.

We solve this by proving theorems in a context and proof by induction can extend this context with new predicate definitions.[6] In particular, the context will contain not only axioms for defined function symbols (e.g., like `In` in the subset example) but it also contains a meta-variable ("wrapped" by `Def`) 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` 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)) <-> Wfp(?E(l,m)))
```

The wrapper `Def` (recall this, like `Wfp` is the identity) also serves to restrict unification; in particular, only the induction rule which creates a schematic pure logic program can instantiate `Def(?H)`.

Definitions are set up during induction. Consider the following rule corresponding to structural induction over lists. This rule builds a schematic program `P(x,y)` contained in the first assumption. The second and third assumption correspond to the base case and step case of a proof by induction showing that this definition is equivalent to the specification formula `A(x,y)`. This rule is derived in our theory by list induction.

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

As in [2] we have written a tactic that applies induction rules. Resolution with this rule yields three subgoals (corresponding to the three assumptions above) but the first is 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; `P` takes two arguments even though the induction is on only one of them. This is

---

[6] The ability to postulate new predicate definitions can, of course, lead to inconsistency. We lack space here for details, but it is not hard to prove under our approach that defined predicates are defined by well-founded recursion and may be consistently added as assumptions.

[7] `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 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.

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. Our solution to this problem is ad hoc; we 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, but more complex, solutions are also possible.

3.4 Relationship to Other Calculi

The derived calculus, although very simple, is motivated by and is similar to the Whelk Calculus developed by Wiggins in [33]. There 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 $[\![A]\!]_{P(\overline{x}) \leftrightarrow \phi}$. Both formulas and sequents are tagged and the tag subscript represents part of a pure logic program. 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.

The rules I give can be seen as a reinterpretation of the rules of Whelk where tagged formulas are formulated directly as equivalences between specifications and program schemas (for full details see [1]); hence, seen in this light, the Whelk rules constitute a simple calculus for manipulating equivalences. For example, the Whelk rule for reasoning about conjunctions is

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

and can be straightforwardly translated into the rule RAndI given in Section 3.3 (ϕ and ψ play the role of $ExtA$ and $ExtB$ and P and its parameters $\mathcal{E}$ are omitted.) Our derivation of many of these rules provides a formal verification that they are correctness preserving with respect to the above mentioned equivalence criteria. In [1] we describe the translation and derivation of the Whelk rules in considerable detail. Interestingly, not all of the Whelk rules given could be derived; the reinterpretation led to rules which were not derivable (counter models could be given) and hence helped uncover mistakes in the original Whelk calculus. This confirms that just as it is useful to have machine checked proofs of program correctness, it is also important to certify calculi formally.

4 Program Development

We now illustrate how the derived rules can be applied to program synthesis. Our example is synthesizing the subset predicate (over lists). We choose this as it is a standard example from the literature. In particular, our proof is almost identical to one given in [33].

Our proof requires 15 steps and is given in Figure 2 with comments. Here we replay Isabelle's response to these proof steps, i.e., the instantiated top-level goal and subgoals generated after each step. The output is taken directly from an Isabelle session except, to save space, we have combined a couple of steps, "pretty printed" formulas, and abbreviated variable names.

```
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)) <-> Wfp(?E(l,m)))";

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

(* Base Case *)
br RAllI 1;
br RTrue 1;
by (cut_fast_tac [inbase] 1);

(* Step Case *)
by(SIMP_TAC (list_ss addrews [instep,AndImp,AllAnd,AllEqImp]) 1);
br RAndI 1;

(* Prove 2nd case 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 RFalse 1;   (* Induction Base Case *)
by(SIMP_TAC (list_ss addrews [inbase]) 1);
by(SIMP_TAC (list_ss addrews [instep]) 1);  (* Induction Step Case *)
br ROrI 1;
br EqInstance 1;
by (etac allE 1 THEN assume_tac 1);  (* Apply induction hypothesis *)
```

Fig. 2. Isabelle Proof Script for Subset Proof

The proof begins by giving Isabelle the subset specification. Isabelle prints back the goal to be proved (at the top) and the subgoals necessary to establish it. As the proof proceeds, the theorem we are proving becomes specialized as `?H` is incrementally instantiated with a 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)) <-> Wfp(?E(l, m)))
 1. Def(?H) --> (ALL l m. (ALL z. In(z, l) --> In(z, m)) <-> Wfp(?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 the previous section, promotes universally quantified variables to eigenvariables so our rules may be used via lifting.

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

Next, we invoke our induction tactic that applies the derived list induction rule, specifying induction on `l`. The execution of the tactic 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)) <-> Wfp(?P(l, m)))
 1. !!l m y. Def(?Q) ==> (ALL z. In(z, []) --> In(z, y)) <-> Wfp(?EA10(y))
 2. !!l m ma h y.
       [| Def(?Q); ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Wfp(?P(ma, y)) |] ==>
       (ALL z. In(z, h . ma) --> In(z, y)) <-> Wfp(?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). First we apply `RAllI` which promotes the ∀-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)) <-> Wfp(?EA10(y))
```

Next we apply `RTrue` which 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 `Wfp`) 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)) <-> Wfp(?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 base case of the definition for `In`. Isabelle leaves us 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)) <-> Wfp(?P(ma, y)) |] ==>
       (ALL z. In(z, h . ma) --> In(z, y)) <-> Wfp(?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 recursive case of the definition for `In` 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 (automatically!) proven 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)) <-> Wfp(?P(ma, y)) |] ==>
       In(h, y) & (ALL v. In(v, ma) --> In(v, y)) <-> Wfp(?EB11(h, ma, y))
```

We decompose the conjunction with `RAndI`, which yields two subgoals.

```
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)) <-> Wfp(?P(l, m)))
 1. !!l m ma h y.
       [| Def(?Q); ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Wfp(?P(ma, y)) |] ==>
       In(h, y) <-> Wfp(?EA21(ma, h, y))
 2. !!l m ma h y.
       [| Def(?Q); ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Wfp(?P(ma, y)) |] ==>
       (ALL v. In(v, ma) --> In(v, y)) <-> Wfp(?EB22(ma, h, y))
```

We immediately solve the second subgoal by resolving with the induction hypothesis. I.e., after ∀-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)) <-> Wfp(?P(l, m)))
```

Returning to the first goal (to build a program for `?EA21`), we perform another induction; the base case is proved as in the first induction except rather than introducing `True` with `RTrue` we introduce `False` with `RFalse` and solve the remaining 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)) <-> Wfp(?P(l, m)))
 1. !!l m ma h y mb ha ya.
       [| ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Wfp(?P(ma, y));
          Def(?Q27); ALL y. In(y, mb) <-> Wfp(?Pa(y, mb)) |] ==>
       In(ya, ha . mb) <-> Wfp(?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)) <-> Wfp(?P(ma, y));
          Def(?Q27); ALL y. In(y, mb) <-> Wfp(?Pa(y, mb)) |] ==>
       ya = ha | In(ya, mb) <-> Wfp(?EB32(ha, mb, ya))
```

Applying `ROrI` 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)) <-> Wfp(?P(ma, y));
          Def(?Q27); ALL y. In(y, mb) <-> Wfp(?Pa(y, mb)) |] ==>
       ya = ha <-> Wfp(?EA40(mb, ha, ya))
 2. !!l m ma h y mb ha ya.
       [| ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Wfp(?P(ma, y));
          Def(?Q27); ALL y. In(y, mb) <-> Wfp(?Pa(y, mb)) |] ==>
       In(ya, mb) <-> Wfp(?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)) <-> Wfp(?P(l, m)))
 1. !!l m ma h y mb ha ya.
       [| ALL y. (ALL z. In(z, ma) --> In(z, y)) <-> Wfp(?P(ma, y));
          Def(?Q27); ALL y. In(y, mb) <-> Wfp(?Pa(y, mb)) |] ==>
       In(ya, mb) <-> Wfp(?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)) <-> Wfp(?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 [33]. Alternatively it can be directly translated into the following Prolog program.

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

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

5 Conclusion, Comparisons, and Future Work

The ideas presented here have applicability, of course, outside logic programming and Isabelle can be used to derive other calculi for verification and synthesis. [2, 4] describes other applications of this methodology. But logic programming seems an especially apt domain for such development due to the close relationship between the specification and programming language.

Other authors have argued that first-order logic is the proper foundation for reasoning about and transforming logic programs (e.g., [11, 9]). But there are benefits to using even richer logics to manipulate first-order, and possibly higher-order, specifications. For example, in this paper we used a recursion schema corresponding to structural induction over lists. But synthesizing logic programs with more complicated kinds of recursion (e.g., quick sort) requires general well-founded induction. But providing a theory where the user can provide his own well-founded relations necessitates formalizing well-foundedness which in turn requires quantifying over sets or predicates and, outside of set-theory, this is generally second-order. We are currently exploring synthesis based on well-founded induction in higher-order logic.

Another research direction is exploring other notions of equivalence. Our calculus has employed a very simple notion based on provability in a theory with induction principles over recursive data-types. There are other notions of equivalence and ways of proving equivalence that could be formalized of course. Of particular interest is exploring schematic calculi like that proposed by Waldau [31]. Waldau presents a calculus for proving the correctness of transformation schemata using intuitionistic first-order logic. In particular he showed how one can prove the correctness of fold-unfold transformations and schemata like those which replace recursion by tail recursion. The spirit of this work is similar to our own: transformation schemata should be proven correct using formal proofs. It would be interesting to carry out the kinds of derivations he suggests in Isabelle and use Isabelle's unification to apply his transformation schemata.

We conclude with a brief comparison of related approaches to program synthesis based on unification. This idea can be traced back to [14] who proposed the use of resolution not only for checking answers to queries, but also for synthesizing programs and the use of second-order matching by Huet and Lang to apply schematic transformations. Work in unification based program synthesis that is closest in spirit to what we described here is the work of [20, 21], which used higher-order (pattern) unification to synthesize logic programs in a "middle-out" fashion. Indeed, synthesis with higher-order resolution in Isabelle is very similar as in our work, the meta-variable standing in for a program is a second-order pattern and it is only unified against second-order patterns during proof. [20, 21] emphasizes, however, the automation of such proofs via rippling while we concentrate more on the use of logical frameworks to give formal guarantees to the programming logic itself. Of course, these approaches are compatible and can be combined.

References

1. David Basin. Isawhelk: Whelk interpreted in Isabelle. Abstract accepted at the 11th International Conference on Logic Programming (ICLP94). Full version available via anonymous ftp to mpi-sb.mpg.de in pub/papers/conferences/Basin-ICLP94.dvi.Z.
2. 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.
3. David Basin and Robert Constable. Metalogical frameworks. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 1–29. Cambridge University Press, 1993.
4. 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.
5. A. Bundy. Tutorial notes; reasoning about logic programs. In G. Comyn, N.E. Fuchs, and M.J. Ratcliffe, editors, *Logic programming in action*, pages 252–277. Springer Verlag, 1992.
6. 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.
7. R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, 1977.
8. Wei Ngan Chin. *Automatic Methods for Program Transformation.* Ph. D. thesis, Imperial College Department of Computer Science, March 1990.
9. 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.
10. K.L. Clark. Predicate logic as a computational formalism. Technical Report TOC 79/59, Imperial College, 1979.
11. 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.
12. 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.
13. 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.
14. Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the IJCAI-69*, pages 219–239, 1969.
15. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
16. P. Hill and J. Lloyd. The Gödel Report. Technical Report TR-91-02, Department of Computer Science, University of Bristol, March 1991. Revised in September 1991.
17. C.J. Hogger. Derivation of logic programs. *JACM*, 28(2):372–392, April 1981.
18. L. Kott. About a transformation system: A theoretical study. In *Proceedings of the 3rd International Symposium on Programming*, pages 232–247, Paris, 1978.
19. Robert A. Kowalski. Predicate logic as a programming language. In *IFIP-74.* North-Holland, 1974.

20. Ina Kraan, David Basin, and Alan Bundy. Logic program synthesis via proof planning. In *Proceedings of LoPSTr-92*. Springer Verlag, 1992.
21. 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.
22. M.J. Maher. Equivalences of logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann, 1987.
23. Sean Matthews, Alan Smaill, and David Basin. Experience with FS_0 as a framework theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 61–82. Cambridge University Press, 1993.
24. Lawrence C. Paulson. Set theory for verification: I. From foundations to functions. *Journal of Automated Reasoning*. In press; draft available as Report 271, University of Cambridge Computer Laboratory.
25. Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.
26. Lawrence C. Paulson. Introduction to Isabelle. Technical Report 280, Cambridge University Computer Laboratory, Cambridge, January 1993.
27. Lawrence C. Paulson. Isabelle's object-logics. Technical Report 286, Cambridge University Computer Laboratory, Cambridge, February 1993.
28. Frank Pfenning. Logic programming in the LF logical framework. In *Logical Frameworks*, pages 149 – 181. Cambridge University Press, 1991.
29. Taisuke Sato. Equivalence-preserving first-order unfold/fold transformation systems. *Theoretical Computer Science*, 105:57–84, 1992.
30. H. Tamaki and T. Sato. Unfold/fold transformations of logic programs. In *Proceedings of 2nd ICLP*, 1984.
31. 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.
32. Richard W. Weyhrauch. Prolegomena to a theory of formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.
33. 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.