J. Symbolic Computation (1999) **11**, 1–000

Program Development Schemata as Derived Rules

PENNY ANDERSON[†] AND DAVID BASIN[‡]

[†]*Department of Computer Science, The College of New Jersey,*
Ewing, NJ 08628-0718, USA

[‡]*Institut für Informatik, Universität Freiburg*
D-79110 Freiburg, Germany

(Received 18 May 1999)

We show how the formalization and application of schemata for program development can be reduced to the formalization and application of derived rules of inference. We formalize and derive schemata as rules in theories that axiomatize program data and programs themselves. We reduce schema-based program development to ordinary theorem proving, where higher-order unification is used to apply rules. Conceptually, our formalization is simple and unifies divergent views of schemata, program synthesis, and program transformation. Practically, our formalization yields a simple methodology for carrying out development using existing logical frameworks; we illustrate this in the domain of logic program synthesis and transformation using the Isabelle logical framework.

1. Introduction

Research on program schemata originated in the 1960s as computer scientists investigated programming concepts related to the control structure of programs, such as recursion and iteration. They developed techniques that could be applied to obtain proofs of program properties, in particular equivalence (Cooper, 1966; de Bakker and de Roever, 1973; Manna and Vuillemin, 1972). For the sake of generality, proofs were constructed not only for particular programs (like McCarthy's well-known "91 function") but also for classes of programs that were instances of general schemata. It was quickly seen that program schemata could be used not only for post-hoc verification of program equivalence, but for transformation as well, and as the basis for program transformation systems (Huet and Lang, 1978).

Schemata play an important role in modern-day systems for program development, including the synthesis of programs from non-executable specifications as well as the transformation of one executable specification into another. As an example of this, and to motivate the discussion ahead, consider Figure 1, which is a schema for transforming a functional program to a tail-recursive form. The schema is adapted from Hoffmann and Krieg-Brückner (1993). It uses conditional equations to define F and G and states that an equality between them holds, whenever two applicability premises also hold. These premises state that R has a right identity N, and is associative. The variables F and

0747–7171/90/000000 + 00 $03.00/0

Functional Definition for F

$$\neg B(x) \quad \rightarrow \quad F(x) = T(x) \tag{1.1}$$

$$B(x) \quad \rightarrow \quad F(x) = R(F(H(x)), K(x)) \tag{1.2}$$

Functional Definition for G

$$\neg B(x) \quad \rightarrow \quad G(x, y) = R(T(x), y) \tag{1.3}$$

$$B(x) \quad \rightarrow \quad G(x, y) = G(H(x), R(K(x), y)) \tag{1.4}$$

Applicability Premises:

$$R(x, N) = x \tag{1.5}$$

$$R(R(x, y), z) = R(x, R(y, z)) \tag{1.6}$$

Conclusion:

$$F(x) = G(x, N) \tag{1.7}$$

Figure 1. Development Schema for Tail Recursion Introduction

G do not name programs themselves, but rather *program schemata*, since their defining bodies can be instantiated to different programs depending on how the schema variables they contain are instantiated.† Hence this development schema expresses the equality of infinitely many pairs of programs, each pair corresponding to an instantiation that meets the applicability premises. When this schema is correct, we can use it for program verification: given concrete functions f and g and a term n, we can verify that for every x, $f(x) = g(x, n)$, when f and g are instances of the program schemata and the applicability premises hold. Alternatively, the schema can be used for program synthesis or program transformation, depending on whether f itself is already executable. Given a particular f, the schema can be applied to generate a corresponding g and n. In the case of this particular schema, we can use it to transform a naive implementation of a list reversal or factorial function to a more efficient tail recursive version.

1.1. Contributions

This paper makes several contributions towards a clarified view of schema-based program development. First, we propose that schemata can be understood, formalized, and used in a simple way: *program development schemata are derived rules.* We mean this in the standard sense of a derived rule of inference in logic. A schema like Figure 1 can be formulated as a rule stating that the conclusion follows from the premises defining F, G, and the applicability conditions. By deriving the rule in an axiomatic theory, we validate a semantic statement about it: the conclusion of the rule holds in every model where both the axioms of the theory and the premises of the rule are true. Hence, by selecting a language to work in we control which development schemata are formalizable, and by selecting a theory we determine which schemata are derivable.

This view of schemata is simple but powerful. It can be applied to formalize and derive a large variety of schemata that are used in practice (see §5.3 for some of the limitations) and we give examples that show applications associated with rather different semantics.

Second, we clarify and advocate the role of logical frameworks in schema development

† We use capital letters to denote schema variables, lower case letters near the end of the alphabet (e.g., x, y, and z) to denote ordinary (term) variables, lower case letters like f, g, and h to denote concrete (non-schematic) functions, and lower case letters like m, n, s, and t to denote terms.

and application. If development schemata are derived rules, formal reasoning about them requires the ability to reason about the rules of a theory, not merely the formulae of a theory. We propose the use of logical frameworks based on formal metalogics for this task. Specifically, a metalogic based on a minimal higher-order logic can provide a foundation for formalizing the syntax and rules of object theories. All activities centered around verifying and using schemata are reduced to theorem proving in a higher-order setting. Logical frameworks such as the implementations of the Edinburgh LF (Pfenning, 1991; Luo and Pollack, 1992) or Isabelle (Paulson, 1994b) are based on such metalogics and are well suited for the kind of formalization we describe in this paper.

Third, we show how program verification, schema-based program development, and even the development of schemata are closely related and can be viewed as points on a continuum of theorem-proving activities. A schema is an inference rule, and can be applied, like other rules, to program verification, for example to verify the equality between two functions $f(x) = g(x, n)$. If we use higher-order unification to apply rules, then schema application can be used for program synthesis or transformation; that is, we can begin with one of the functions as an unknown named by a schema variable, and synthesize an instance for it during proof. We show too that similarly we can synthesize not only programs, but also development schemata themselves.

It is common folklore that there is a close connection between verification and synthesis; for example, Prolog programmers know that if a predicate $p(X)$ is provable and resolution generates an instance t for X, then the same resolution steps (using instances of the resolved clauses) verify $p(t)$. Our work shows a number of ways to extend and apply this kind of paradigm. Instead of using an axiomatized set of clauses as in Prolog, we use general derived rules. Instead of using a fixed strategy for proof construction like SLD resolution, we use general interactive theorem proving. And instead of using first-order syntax to build terms, we use higher-order syntax to build recursively defined functions. Through examples, we illustrate that this generality is useful for different kinds of schema-based reasoning, including the development of schemata themselves.

Finally, we show how our approach can unify schema-based program development with other development methodologies. The use of schemata has sometimes been considered complementary to deductive approaches to program development based on the application of inference rules. These approaches are sometimes called *rules+strategies* (Pettorossi and Proietti, 1998) since development proceeds by applying inference rules possibly guided by high-level strategies, which control the development process. By viewing schemata as rules and schema application as logical inference, the distinction between schema-based development and development based on rules+strategies disappears.

This unification is not just conceptually interesting, but also practically useful. Our formalization yields a simple methodology for carrying out schema-based development in existing logical frameworks and experimenting with approaches that combine what are traditionally seen as distinct development methodologies. The practical usefulness of this is ultimately an empirical question. However, we give examples that suggest that it is very easy to use an "off-the-shelf" logical framework to formalize, derive, and apply rule schemata, to experiment with different semantics for schemata, and to carry out non-trivial developments. We believe this is of interest both to specialists seeking to develop and understand particular program development methodologies and to practitioners interested in implementing and using such methodologies.

1.2. Organization

The remainder of this paper is organized as follows. In §2 we formalize the connection between development schemata and inference rules, and explain how derived rules can be used for program development. In §3 we show how schemata may be interpreted and derived in theories with different associated semantics. For the sake of concreteness, we use logic program development as an example and present, in some detail, a simple application in §4. A more extensive example is treated at a higher level in §5, where we also consider the role schemata can play together with other kinds of development methods. In §6 we discuss related work and we draw conclusions in §7.

2. Development Schemata as Rules

Our focus in this section is on inference rules, their formalization in a logical metatheory, and the mechanisms underlying their use in program derivation.

2.1. Rules

We begin by specifying what we mean by an *inference rule*. Following Hindley and Seldin (1986), we fix a logical language with a set of formulae $\mathcal{F}$. For n a natural number, every partial function ϕ from $\mathcal{F}^n$ to $\mathcal{F}$ determines a rule $R(\phi)$. We call each n-tuple $\langle A_1, \ldots, A_n \rangle$ in the domain of ϕ the *premises*, and if $\phi(A_1, \ldots, A_n) = B$ then B is the corresponding *conclusion*. As formalized, rules have finitely many premises, which is enough for our purposes.

In practice, rules are formalized using *rule schemata*. For example, a rule schema for Modus Ponens is

$$\frac{A \to B \quad A}{B}. \tag{2.1}$$

A and B are not formulae but rather variables that range over formulae. We call these *schema variables*. A rule schema defines a rule: the partial function in which instances of the premises are mapped to corresponding instances of the conclusion. Since schemata are typically used to present rules, there is little harm in identifying the two concepts; we will do that here too when no confusion can arise.

Given a language, a *deductive system* for it is given by a set of formulae called the *axioms*, and a set of rules. A *derivation* of a formula B from a set of formulae $\mathcal{A}$ is a formula-labeled tree: the leaves are labeled with either axioms or members of $\mathcal{A}$, and the root is labeled with B. A node labeled by A with children $A_1, \ldots, A_n$ is an instance of a rule with premises A_i and conclusion A. The set $\mathcal{A}$ is called the set of *assumptions* and B is *derivable* from $\mathcal{A}$; when there are no assumptions, B is said to be *provable*. The deductive system determines a *theory*, namely the set of all provable sentences.

We call a rule *derivable* when, for each instance, there is a derivation of the conclusion from the premises.† Derivability has a semantic side. Assume we have a class of models for the language in question and a notion of what it means for a formula to be true in a model from the class. A derived rule defines a consequence relation between the premises

† In contrast, a rule is *admissible* if the conclusion is provable whenever the premises are. Note, therefor, that there are different possible notions of what it means to say that a rule is *correct*.

and the conclusion: a rule is derivable if for any model where all the premises are true, so is the conclusion. In §3 we show how different semantics for rules can be captured by derivability in different theories.

2.2. Formalizing Rule Schemata

Formalizing rule schemata means representing them in a formal language and reasoning about them in a theory with a well-defined semantics. We now give a brief overview of representation and derivability using languages and theories based on type theories.

We begin by considering syntax and the role of types. When schema variables range over formulae, as in (2.1), instantiation is standard substitution for a first-order language and instances may be found using first-order matching or unification. However, in program development schemata, variables usually range over different kinds of syntactic entities. For example, in the premise $R(x, N) = x$ from Figure 1, the schema variable N ranges over terms and R ranges over those functions that given two terms deliver a term. "Ranges over" corresponds to a typing requirement, but such a function type for R is not a type in the programming language, and cannot be treated by first-order substitution.

The simply typed λ-calculus provides a basis for formally representing and reasoning about typed syntax like the above. This language constitutes a *metalanguage* in which an *object language* is encoded. This idea goes back to Church's use of the λ-calculus to represent syntax in higher-order logic (Church, 1940). Since then, researchers have explored other logics based on the λ-calculus, and in particular their use as *metalogics*, which form the basis of *metatheories* (i.e., the metalogic is extended with *metaformulae* as axioms) that represent the deductive systems of other *object logics*. These metalogics typically extend the typed λ-calculus with (minimal) implication and universal quantification over terms at all types. Examples include the metalogics of the Automath project (de Bruijn, 1980), the Edinburgh LF (Harper *et al.*, 1993), Isabelle (Paulson, 1994b), and λ-Prolog (Felty and Miller, 1988). Deductive support for reasoning in these metalogics is often based on higher-order unification (Huet, 1975; Pfenning, 1991), which can be used to generate appropriate higher-order instances for schemata. We will assume some familiarity with such logics and their use as metalogics; the reader lacking this background should consult Harper *et al.* (1993).

Let $\mathcal{V}$ denote the set of (typed) variables of the λ-calculus. We divide the set of variables $\mathcal{V}$ into two kinds: *term variables* and *schema variables.* As a notational convention, we write term variables as lower-case variables like x, y, and z, and schema variables as upper-case variables like R and N. We call a well-typed term (possibly) containing schema variables a *schematic term.* When no schema variables are present, we call it a *ground term.* When working with schematic terms, we will usually consider only substitutions where the domain consists of schematic variables, i.e., term variables are mapped to themselves (effectively treated as constants). A *program schema* is a schematic term whose instances correspond to programs in some language. A *rule schema*, as we have seen, is given by an $n+1$-tuple of terms written

$$\frac{A_1, \ldots, A_n}{B} \tag{2.2}$$

where the A_i and B are schematic terms. Finally, we call a derivation a *schematic derivation* (or *schematic proof*, when there are no assumptions) when formulae labeling the derivation are themselves schematic.

The simply typed λ-calculus is an adequate representation language for program schemata. A metalogic based on implication ($\Longrightarrow$) and universal quantification ($\forall$), like those previously mentioned, can be used as the basis of a metatheory that is adequate to represent rule schemata.[†]

Consider a very simple example: representing an ordinary schematic inference rule like modus ponens, given in (2.1). Let the language of the object logic (e.g. first-order logic) be encoded in the typed λ-calculus. For instance, let o be a type whose elements encode formulae of the object logic; we can then represent object logic implication using the logical constant imp of type $o \Rightarrow o \Rightarrow o$. To represent inference rules we formulate assertions about them as axioms. Let the unary predicate $tr(A)$ represent the assertion that the formula represented by A is true. Then modus ponens is encoded as

$$\forall A^o\ B^o.\ tr(imp\ A\ B) \Longrightarrow tr(A) \Longrightarrow tr(B)\,, \tag{2.3}$$

i.e., from the truth of $A \to B$ and A, the truth of B follows. Note that implication in the metalogic (which we associate to the right) and typed universal quantification should not be confused with implication and quantification in the object logic.

To aid reading metaformulae, we can (as done in the Isabelle system) write iterated implication $A_1 \Longrightarrow \ldots \Longrightarrow A_n \Longrightarrow A$ as $[\![A_1; \ldots; A_n]\!] \Longrightarrow A$, use the infix notation $\to$ instead of imp, and omit the predicate tr, which "lifts" object logic formulae to formulae in the metatheory. Under these conventions, the above inference rule can be written as

$$\forall A^o\ B^o.\ [\![A \to B; A]\!] \Longrightarrow B\,. \tag{2.4}$$

Observe that the schema variables in a rule correspond to variables in the metalogic that are universally quantified at the front of the formula. This is sensible: a schema makes a statement about all its instances in the same way a universally quantified formula does. This motivates the identification of schema variables in rules of the object logic with variables in formulae of the metalogic that are outermost universally quantified. As notational shorthand, consistent with this identification, we will often leave these universal quantifiers implicit. Under all these conventions, the resulting formula is similar syntactically to (it is a one-dimensional representation of) the rule schema it that formalizes:

$$[\![A \to B; A]\!] \Longrightarrow B\,. \tag{2.5}$$

In general, a rule schema like (2.2) determines a rule ϕ where $\phi(\sigma(A_1), \ldots, \sigma(A_n)) = \sigma(B)$ for every (type correct, higher-order) substitution σ. This rule is derivable whenever we can derive $\sigma(B)$ from the assumptions $\sigma(A_1), \ldots, \sigma(A_n)$. Given a formalization of $\Longrightarrow$ as minimal implication in the metalogic, derivability of the rule in the object logic is equivalent to the provability of the metaformula

$$[\![\sigma(A_1); \ldots; \sigma(A_n)]\!] \Longrightarrow \sigma(B)$$

in the metatheory. This in turn follows (e.g., by $\forall$-elimination in the metalogic) from the provability of (the universal closure) of

$$[\![A_1; \ldots; A_n]\!] \Longrightarrow B\,.$$

[†] Adequacy is a technical notion (Harper *et al.*, 1993). Very roughly, adequacy for a language means that every well-typed term represents a term of the appropriate syntactic category and vice versa. Adequacy for rule schemata corresponds to capturing derivability in the metatheory.

Hence, rule schemata for an object logic can be identified with formulae of the metatheory. A schema like that of Figure 1 corresponds to a metaformula stating that the premises imply the conclusion, i.e.,

$$[\![(1.1);(1.2);(1.3);(1.4);(1.5);(1.6)]\!] \Longrightarrow (1.7)\,.$$

The formalization of the object logic in the metatheory determines a class of models, and the derivability of this formula expresses the semantic property that in any model in which (1.1) – (1.6) are true, (1.7) is true.

2.3. Construction of Derivations and Deductive Synthesis

As stated above, a derivation is a formula-labeled tree whose leaves are either axioms or assumptions. When considering the process of interactively building derivations, we take the view implemented in the Isabelle system (Paulson, 1994b): it is useful to allow the assumption set to vary while constructing a derivation. At the start of the derivation, the initial goal is also the initial assumption, and at any point in the derivation the goal is derivable under the assumptions labeling the leaves. Under this view, the derivation is a proof precisely when all leaves are labeled by axioms.

A derivation is constructed by applying a rule to a leaf. When a rule is represented by a metaformula

$$[\![A_1;\ldots;A_n]\!] \Longrightarrow B\,,$$

it can be applied to a formula F labeling a leaf whenever there is a higher-order substitution σ such that $\sigma(B) = F$, where $=$ is equality in the λ-calculus. In this case, the derivation is extended so that F has the children $\sigma(A_1),\ldots,\sigma(A_n)$. Observe though that if we allow schematic derivations, then F itself may contain metavariables and higher-order unification can be used to find a σ where $\sigma(B) = \sigma(F)$. The substitution must furthermore be applied to all the formulae labeling the derivation tree, since they may share schematic variables with F. Applying inference rules this way is called *higher-order resolution* (Paulson, 1985; Paulson, 1989). In general σ is not unique; each rule may have a set of applications at a leaf, each computing a different substitution and resulting in a different derivation.

The application of substitutions to schematic derivations can be used as the mechanism for deductive program development (e.g., program synthesis or transformation). The process of constructing a derivation iteratively applies substitutions, and this in turn incrementally instantiates schema variables with terms that represent refined specifications or, when executable, programs.

More precisely, we can impose a total order on a derivation that respects the partial order induced by the tree structure. Let an *ordered derivation* be a derivation tree of m nodes that is further labeled by the natural numbers $1\ldots m$, where the root is labeled 1, and on any path from the root to a leaf, all the labels are strictly increasing. Let the substitution computed by an application of the rule applied to node i be σ_i. With every ordered derivation $\mathcal{D}$ we may associate a substitution σ_D, defined as the composition

$$\sigma_m \circ \sigma_{m-1} \circ \ldots \circ \sigma_1\,.$$

Now if G is the initial goal formula of an ordered derivation $\mathcal{D}$, the root of $\mathcal{D}$ is labeled with $\sigma_D(G)$. As a result, if G contains schema variables in the domain of σ_D, the formula derived is not G but the substitution instance of it determined by σ_D. Moreover, any leaf formula introduced as a premise A_i of an inference rule containing schema variables is also the substitution instance $\sigma_D(A_i)$. If $\{A_1, \ldots, A_n\}$ is the set of assumptions in the derivation, then the derivation demonstrates the derivability of the rule represented by

$$[\![\sigma_D(A_1), \ldots, \sigma_D(A_n)]\!] \Longrightarrow \sigma_D(G) .$$

As we will see in §3, some of the assumptions $\sigma_D(A_i)$ may define programs or program schemata (like F and G in Figure 1) and $\sigma_D(G)$ may be a statement about these programs or schemata. Thus deductive program development may be used to derive programs or new program development schemata. In §3.2 we give an example showing how deduction can be used in the derivation and specialization of schematic rules.

2.4. Realization in a Logical Framework

Our account of schemata as rules and development as deduction by higher-order resolution has practical applications. There are logical frameworks (proof assistants based on formal metalogics) that support the formalization, derivation, and application of rules as described above. We have used the Isabelle Logical Framework for our case-studies in schematic development. There are other possibilities, e.g., Elf (Pfenning, 1991); however Isabelle fits well with our requirements. It provides a formal metalogic with a sophisticated type system, proof construction by higher-order resolution, support for hierarchical construction of theories, and support for automation. The overview of Isabelle given here is limited; Paulson (1994b) provides a full account.

Isabelle's metalogic is a minimal higher-order logic, based on implication ($\Longrightarrow$) and universal quantification ($\bigwedge$), supporting polymorphic typing augmented with type classes. Polymorphic typing introduces another kind of schematic variable, namely schematic variables that range over types. This has important applications in our work: in §3 we derive rules containing schematic type variables and later specialize these rules both with respect to term and type instances.

Object logics are encoded in Isabelle's metalogic by declaring types and term constructors to represent the language syntax, followed by metaformulae that represent inference rules. Free variables in formulae are implicitly universally quantified; hence a formula like (2.4) is written as (2.5). The Isabelle distribution comes with a number of encoded object logics and theories. For our case studies we used an implementation of higher-order logic (Isabelle/HOL) extended with theories of typed sets, inductive definitions, well-founded orders, and simple program data types.

Derivations in Isabelle are constructed by applying rules, encoded as metaformulae, using higher-order resolution. For the purposes of top-down proof construction, a metaformula $[\![A_1; \ldots; A_n]\!] \Longrightarrow B$ can be viewed as a proof state, where B is the goal to be established and the A_i represent the subgoals to be proven. Under this view an initial proof state has the form $G \Longrightarrow G$, i.e., it has one subgoal, namely G. The final proof state *is itself* the desired theorem G. Rules are applied interactively, but their application can be automated using programs, called *tactics*, that construct proofs. Isabelle comes with a well-developed collection of tactics for common theorem proving tasks, such as rewriting. This was important for our development: we could use standard theories and tactics provided by Isabelle to carry out case studies quickly.

3. Deriving Development Schemata

We now turn to examples to illustrate our thesis that development schemata are derived rules of inference and that schematic proof techniques can be used to develop schemata themselves. Our examples are three schemata for logic program development, and they demonstrate the flexibility of our approach in developing related schemata for different programming language semantics. Our derivations provide evidence that it is possible to use a generic logical framework for schematic development, building on standard theories with little additional effort.

Schemata for functional programs have been thoroughly studied, and there is a large body of published program transformations (Huet and Lang, 1978; Partsch, 1990; Hoffmann and Krieg-Brückner, 1993). In previous work (Basin *et al.*, 1993), we have explored the formalization of such transformations in a framework similar to that proposed here. These ideas extend to logic programs, as demonstrated in (Basin, 1994; Anderson and Basin, 1995). Further experiments have shown that functional and logic program notations can be combined, allowing the development of programs in mixed notations like Gödel (Hill and Lloyd, 1991), and allowing the development of logic programs from partially functional specifications. Our examples will be drawn from these experiments, and in this section we consider development schemata for logic programs that are straightforward translations of functional ones.

We can translate the functional development schema of Figure 1 to a relational form suitable for logic programs by *flattening* it, which entails replacing n-ary functions with $n+1$-ary relations and replacing nested functions with conjoined relations that share existentially quantified variables. For example, the conditional equation (1.2) of Figure 1

$$B(x) \to F(x) = R(F(H(x)), K(x))$$

is flattened by forming the following conditional equivalence

$$B(x) \to (F(x, y) \leftrightarrow \exists x_1\ x_2\ x_3.\ H(x, x_1) \wedge K(x, x_2) \wedge F(x_1, x_3) \wedge R(x_3, x_2, y))$$

i.e., (3.2) of Figure 2. Flattening all the equations of Figure 1 in a similar way yields the translated schema in Figure 2.†

When translated into Isabelle/HOL, this schema is a sentence of the metalogic but it is not derivable (there are counter-models). The following sections show how variants of it can be formally derived. In the functional setting we can reason about the equivalence of programs by reasoning about their meaning relative to a given semantics. For example, if recursive functions are interpreted as fixedpoints of functionals, then we can reason about a function by reasoning about the limit of its approximating functions. The equivalence of F and G in Figure 1 is provable by such reasoning. This approach is not available for reasoning about relational schemata.

In the remainder of the section, we present three variants of this schema that are derivable. The first two, presented in §3.1 and §3.2, are based on induction principles for the data of the programs. The third, in §3.3, instead formalizes the programs as inductively defined relations, i.e. it formalizes a particular semantics of logic programs.

† All the examples in this paper have been implemented in Isabelle/HOL. However, we will often use conventional logical notation, e.g., when writing schemata, and reserve more computer-oriented syntax for when we want to emphasize the "look and feel" of formal, computer-supported, development.

Relational Definition for F

$$\neg B(x) \;\rightarrow\; (F(x,y) \leftrightarrow T(x,y)) \tag{3.1}$$

$$B(x) \;\rightarrow\; (F(x,y) \leftrightarrow \exists x_1\, x_2\, x_3.\, H(x,x_1) \wedge K(x,x_2) \wedge F(x_1,x_3) \wedge R(x_3,x_2,y)) \tag{3.2}$$

Relational Definition for G

$$\neg B(x) \;\rightarrow\; (G(x,y,z) \leftrightarrow \exists x_1.\, T(x,x_1) \wedge R(x_1,y,z)) \tag{3.3}$$

$$B(x) \;\rightarrow\; (G(x,y,z) \leftrightarrow \exists x_1\, x_2\, x_3.\, H(x,x_1) \wedge K(x,x_2) \wedge R(x_2,y,x_3) \wedge G(x_1,x_3,z)) \tag{3.4}$$

Applicability Conditions:

$$R(x,N,y) \leftrightarrow (x = y) \tag{3.5}$$

$$(\exists y_1.\, R(x,y,y_1) \wedge R(y_1,z,r)) \leftrightarrow (\exists z_1.\, R(y,z,z_1) \wedge R(x,z_1,r)) \tag{3.6}$$

Conclusion:

$$F(x,y) \leftrightarrow G(x,N,y)$$

Figure 2. Schema for Tail Recursion Introduction, Relational Form

3.1. Induction on Well-Founded Data-Types

When a logic program (or program schema) is represented as a set of relations defined by recursive equivalences, as in Figure 2, inductive reasoning principles for it can be obtained when the terms it manipulates have an associated well-founded ordering and recursive calls to predicates manipulate terms that are smaller in the ordering. Here we explore schema derivation where we hypothesize the existence of such an ordering (the user must instantiate and prove this premise later with a particular ordering when applying the schema) and verify the schema by induction over the ordering.

The ordering is defined by a well-founded relation, which we call a *descent relation* by analogy to *descent functions* in functional programming. A descent relation must be used in such a way that recursive calls are smaller under this relation. Rather than defining conditions sufficient in general for this, we consider a special case sufficient for the examples at hand. If a predicate p has a defining recursive equivalence of the form

$$p(x_1,\ldots,x_n) \leftrightarrow \exists y_1 \ldots y_m.\, d(x_1,y_i) \wedge p(y_i,t_1,\ldots,t_{n-1}) \wedge q(x_1,\ldots,x_n,y_1,\ldots,y_m)$$

where $1 \leq i \leq m$ and the $t_1,\ldots,t_{n-1}$ are terms possibly containing the $x_1,\ldots,x_n$ and $y_1,\ldots,y_m$ free, then we say that d is a descent relation for p. This can be extended in the obvious way to schemata where p, d, and q are schema variables. A principle for reasoning about p by induction on its first argument can be obtained, without restricting instantiations of p to terms of a particular type, by explicitly stating an applicability premise that ensures the well-foundedness of the descent relation d.

For example, in the schema of Figure 2, F and G share the descent relation H. If we add a premise to the schema stating that H is well-founded, we obtain the schema of Figure 3, which is provable by induction on x.

This schema can be encoded as a metaformula to be proved in Isabelle/HOL. We exhibit the user input (with slight cosmetic adjustments and the addition of reference numbers). The numbered formulae are the premises of the meta-implication, while the final line is the conclusion. Although the conclusion appears in subsequent proof state displays, Isabelle hides the premises to reduce the amount of information presented to the user.

Relational Definition for F

$$\neg B(x) \quad \rightarrow \quad (F(x,y) \leftrightarrow T(x,y)) \tag{3.7}$$

$$B(x) \quad \rightarrow \quad (F(x,y) \leftrightarrow \exists x_1\ x_2\ x_3.\ H(x,x_1) \wedge K(x,x_2) \wedge F(x_1,x_3) \wedge R(x_3,x_2,y)) \tag{3.8}$$

Relational Definition for G

$$\neg B(x) \quad \rightarrow \quad (G(x,y,z) \leftrightarrow \exists x_1.\ T(x,x_1) \wedge R(x_1,y,z)) \tag{3.9}$$

$$B(x) \quad \rightarrow \quad (G(x,y,z) \leftrightarrow \exists x_1\ x_2\ x_3.\ H(x,x_1) \wedge K(x,x_2) \wedge R(x_2,y,x_3) \wedge G(x_1,x_3,z)) \tag{3.10}$$

Applicability Conditions:

$$\mathrm{wf}(H) \tag{3.11}$$

$$R(x,N,y) \leftrightarrow (x=y) \tag{3.12}$$

$$(\exists y_1.\ R(x,y,y_1) \wedge R(y_1,z,r)) \leftrightarrow (\exists z_1.\ R(y,z,z_1) \wedge R(x,z_1,r)) \tag{3.13}$$

Conclusion:

$$F(x,y) \leftrightarrow G(x,N,y)$$

Figure 3. Tail recursion schema based on well-founded relations

Proof goal 3.1

$[\![\ \forall\, x\ y.\ \neg\ B\ x\ \rightarrow\ F\ x\ y = T\ x\ y;$ [1]
$\quad \forall x\ y.\ B\ x\ \rightarrow\ F\ x\ y = (\exists x1\ x2\ x3.\ H\ x\ x1\ \ \&\ \ K\ x\ x2\ \ \&\ \ F\ x1\ x3\ \ \&\ \ R\ x3\ x2\ y);$ [2]
$\quad \forall x\ y\ z.\ \neg\ B\ x\ \rightarrow\ G\ x\ y\ z = (\exists x1.\ T\ x\ x1\ \ \&\ \ R\ x1\ y\ z);$ [3]
$\quad \forall x\ y\ z.\ B\ x\ \rightarrow$
$\qquad G\ x\ y\ z = (\exists x1\ x2\ x3.\ H\ x\ x1\ \ \&\ \ K\ x\ x2\ \ \&\ \ R\ x2\ y\ x3\ \ \&\ \ G\ x1\ x3\ z);$ [4]
$\quad \mathsf{wf}\{(x,y) \mid x\ y.\ H\ y\ x\};$ [5]
$\quad \forall x\ y.\ R\ x\ N\ y = (x = y);$ [6]
$\quad \forall x\ y\ z\ r.\ (\exists y1.\ R\ x\ y\ y1\ \ \&\ \ R\ y1\ z\ r) = (\exists z1.\ R\ y\ z\ z1\ \ \&\ \ R\ x\ z1\ r)$ [7]
$]\!]\Longrightarrow \forall x\ y.\ F\ x\ y = G\ x\ N\ y$

In Isabelle/HOL syntax, function application is denoted by juxtaposition and functions are curried, so that $F(x,y)$ is encoded as $F\ x\ y$. The equality sign $=$ denotes a polymorphically typed equality between propositions as well as terms, so that in, e.g., premise (6) the first $=$ is logical equivalence, while the second $=$ is equality between terms. The logical connectives and quantifiers $\rightarrow$, $\&$, $\forall$, and $\exists$ are object-level syntax, while the brackets $[\![\ ]\!]$, the quantifier $\bigwedge$, and the connectives ";" and $\Longrightarrow$ are meta-level syntax (described in §2.4). Parentheses are used for grouping, but also for tupling, so in premise (5), (x,y) is a pair. Set comprehension notation is nearly standard, but — as seen in premise (5) — the bound variables are explicitly listed to the right of the vertical bar. Free variables in the goal are implicitly outermost universally quantified at the metalevel.

Isabelle/HOL comes with a theory of well-founded orderings. This supports the direct formalization of the added premise 3.11 and provides associated induction principles. (The formalization of 3.11 as premise (5) swaps the order of arguments to the descent relation H because Isabelle/HOL defines well-foundedness to apply to a set of pairs where the first element of the pair is "smaller" in the well-founded ordering.)

The machine-assisted proof of Goal 3.1 formalizes a well-known inductive argument. We outline it by giving some of the intermediate proof states, in order to show how schema verification reduces to proof in this setting.

The first step of the proof introduces a generalization to be proved by induction, yielding two subgoals:

Proof state 3.2
$\forall x\ y.\ F\ x\ y = G\ x\ N\ y$
1. $\forall x\ y.$
   $\forall y\ z.\ (\exists v.\ F\ x\ v\ \ \&\ \ R\ v\ y\ z) = G\ x\ y\ z$
   $\Longrightarrow F\ x\ y = G\ x\ N\ y$
2. $\forall x\ y.$
   $\forall y\ z.\ (\exists v.\ F\ x\ v\ \ \&\ \ R\ v\ y\ z) = G\ x\ y\ z$

To solve subgoal 1, we show that the generalization (subgoal 2) implies the conclusion by instantiating y in the hypothesis with N and simplifying using premise 6 of Goal 3.1. We attack the remaining subgoal with well-founded induction over the first argument of F. In two-dimensional notation, an induction rule for a well-founded set r is:

$$\frac{\mathrm{wf}(r) \qquad \begin{array}{c}[\forall y.\ (y,x) \in r \to \phi(y)] \\ \vdots \\ \phi(x)\end{array}}{\phi(a)}$$

Applying the Isabelle encoding of this rule leads to the following proof state (set membership is denoted by infix " : ").

Proof state 3.3
$\forall x\ y.\ F\ x\ y = G\ x\ N\ y$
1. $\forall x\ x1.$
   $\forall y.\ (y, x1)\ :\ \{(x,y)|x\ y.\ H\ y\ x\}\ \to\ (\forall y1\ z.\ (\exists v.\ F\ y\ v\ \ \&\ \ R\ v\ y1\ z) = G\ y\ y1\ z)$
   $\Longrightarrow \forall y\ z.\ (\exists v.\ F\ x1\ v\ \ \&\ \ R\ v\ y\ z) = G\ x1\ y\ z$

We next do a case split on $B(x)$. When $\neg B(x)$ holds, Isabelle/HOL's simplifier solves the goal using the definitions of F and G (premises 1 – 4 of Goal 3.1). At the same time, the simplifier expands these definitions in the remaining subgoal:

Proof state 3.4
$\forall x\ y.\ F\ x\ y = G\ x\ N\ y$
1. $\forall x\ y\ z.$
   $[\![\forall y.\ (y,x) : \{(x,y)|x\ y.\ H\ y\ x\}\ \to\ (\forall y1\ z.\ (\exists v.\ F\ y\ v\ \ \&\ \ R\ v\ y1\ z) = G\ y\ y1\ z);$
   $B\ x]\!]$
   $\Longrightarrow (\exists v.\ (\exists x1.\ H\ x\ x1\ \ \&\ \ (\exists x2.\ K\ x\ x2\ \ \&\ \ (\exists x3.\ F\ x1\ x3\ \ \&\ \ R\ x3\ x2\ v)))$
   $\qquad \&\ \ R\ v\ y\ z)$
   $\quad = (\exists x1.\ H\ x\ x1\ \ \&\ \ (\exists x2.\ K\ x\ x2\ \ \&\ \ (\exists x3.\ R\ x2\ y\ x3\ \ \&\ \ G\ x1\ x3\ z)))$

After some rearranging of terms, this gives us the syntactic form needed to apply the associativity of R (premise 7 of Goal 3.1), yielding:

Proof state 3.5
$\forall x\ y.\ F\ x\ y = G\ x\ N\ y$
1. $\forall x\ y\ z.$
   $[\![\forall y.\ (y,x) : \{(x,y)|x\ y.\ H\ y\ x\}\ \to\ (\forall y1\ z.\ (\exists v.\ F\ y\ v\ \ \&\ \ R\ v\ y1\ z) = G\ y\ y1\ z);$
   $Bx]\!]$
   $\Longrightarrow (\exists x1.\ H\ x\ x1\ \ \&\ \ (\exists x2.\ K\ x\ x2\ \ \&\ \ (\exists x3.\ (\exists z1.\ R\ x2\ y\ z1\ \ \&\ \ R\ x3\ z1\ z)\ \ \&\ \ F\ x1\ x3)))$
   $\quad = (\exists x1.\ H\ x\ x1\ \ \&\ \ (\exists x2.\ K\ x\ x2\ \ \&\ \ (\exists x3.\ R\ x2\ y\ x3\ \ \&\ \ G\ x1\ x3\ z)))$

The conclusion now follows trivially from the induction hypothesis; the proof assistant solves the goal by simplification, followed by an automatic procedure based on depth-first search. Altogether the proof required five commands to the proof assistant.

$$
\begin{array}{l}
[\![\ \forall\, x\ y.\ \neg\ ?B\,x\ \rightarrow\ ?F\,x\ y = ?T\,x\ y;\\
\ \ \forall x\ y.\ ?B\,x\ \rightarrow\ ?F\,x\ y = (\exists x1\ x2\ x3.\ ?H\,x\ x1\ \ \&\ \ ?K\,x\ x2\ \ \&\ \ ?F\,x1\ x3\ \ \&\ \ ?R\,x3\ x2\ y);\\
\ \ \forall x\ y\ z.\ \neg\ ?B\,x\ \rightarrow\ ?G\,x\ y\ z = (\exists x1.\ ?T\,x\ x1\ \ \&\ \ ?R\,x1\ y\ z);\\
\ \ \forall x\ y\ z.\ ?B\,x\ \rightarrow\ ?G\,x\ y\ z = (\exists x1\ x2\ x3.\ ?H\,x\ x1\ \ \&\ \ ?K\,x\ x2\ \ \&\ \ ?R\,x2\ y\ x3\ \ \&\ \ ?G\,x1\ x3\ z);\\
\ \ \mathsf{wf}\{(x,y)\mid x\ y.\ ?H\,y\ x\};\\
\ \ \forall x\ y.\ ?R\,x\ ?N\,y = (x = y);\\
\ \ \forall x\ y\ z\ r.\ (\exists y1.\ ?R\,x\ y\ y1\ \ \&\ \ ?R\,y1\ z\ r) = (\exists z1.\ ?R\,y\ z\ z1\ \ \&\ \ ?R\,x\ z1\ r)\\
]\!]\Longrightarrow \forall x\ y.\ ?F\,x\ y = ?G\,x\ ?N\,y
\end{array}
$$

Figure 4. Isabelle/HOL encoding of Figure 3

Upon completion of the proof, Isabelle converts the rule to a standard form, replacing the free variables (F, G, and the other upper-case letters) by schema variables (prefixed by "*?*"), as shown in Figure 4. Since schema variables are subject to instantiation through higher-order unification when the rule is subsequently applied in program development, Figure 4 encodes the schema of Figure 3.

3.2. Specialization to an Inductive Data Type

The rule of Figure 4 can be used to transform a large class of programs that compute over data types such as numbers, lists or strings. This wide applicability is a result of the generality of the schema: not only have we avoided committing to particular functions by using schema variables, the types of the schematic functions are polymorphic (they have schematic types). For example the type of F is $\alpha \Rightarrow \beta \Rightarrow \mathit{bool}$ and the type of G is $\alpha \Rightarrow \alpha \Rightarrow \beta \Rightarrow \mathit{bool}$, for any types α and β.

In practice, we may want to repeatedly apply the schema to improve programs operating over a particular data type, like lists. In this case, repeated application would require repeatedly proving the well-foundedness of the same descent relation. Although the well-foundedness property can be captured as a lemma, this solution is not ideal. Moreover, the program syntax of Figure 4 is awkward, compared to the pattern-matching usual in recursive logic programs. Specializing the schema to a particular data type can solve these problems.

As an example we show how to specialize the rule of Figure 4 to polymorphic lists, built from the constructors [] (the empty list) and # (the infix function constant "cons"). A simple way to do this is to *synthesize* the specialized rule schema from the more general one. The derivation is schematic in the sense of §2.2. Since the output program schema and the applicability premises are synthesized in the course of the derivation, the initial goal specifies only the types of the schematic variables, the input program schema and the equivalence between the input and output programs.

We give the goal here, but omit the type annotations. Note that F, T, etc. in the goal are free variables, implicitly universally quantified at the metalevel.

Proof goal 3.6

$$
\begin{array}{l}
[\![\forall y.\ F\ []\ y = T\ y;\\
\ \forall h\ t\ y.\ F\ (h\#t)\ y) = (\exists x1\ x2.\ K\ h\ t\ x1\ \ \&\ \ F\ t\ x2\ \ \&\ \ R\ x2\ x1\ y)\\
]\!]\Longrightarrow \forall x\ y.\ F\ x\ y = G\ x\ N\ y
\end{array}
$$

The two premises represent the input program schema, which has the usual form for logic programs defined by recursion over lists, instead of the conditional form of Figure 4.

The occurrences of the constructors "[]" and "#" guide the synthesis of the specialized rule.

The first step in the synthesis is to apply the schema of Figure 4, at the same time specifying instantiations for the guard B and the descent relation H:

$$B \leftarrow\!\!\dashv \lambda x.\, x \neq [\,],\quad H \leftarrow\!\!\dashv \lambda x\, y.\, y = tl\ \ x \wedge x \neq [\,]$$

In addition to these explicit instantiations, the unification of the goal with the conclusion of Figure 4 produces the obvious substitution:

$$?F \leftarrow\!\!\dashv F,\quad ?N \leftarrow\!\!\dashv N,\quad ?G \leftarrow\!\!\dashv G$$

This introduces subgoals that are partial instantiations of the premises of Figure 4:

Proof state 3.7
$\forall x\ y.\ F\ x\ y = G\ x\ N\ y$
1. $\forall x\ y.\ \neg x \neq [\,]\ \rightarrow\ F\ x\ y =\ ?T\ x\ y$
2. $\forall x\ y.\ x \neq [\,]\ \rightarrow$
   $F\ x\ y = (\exists x1\ x2\ x3.\ (x1 = tl\ \ x\ \ \&\ \ x \neq [\,])\ \ \&\ \ ?K\ x\ x2\ \ \&\ \ F\ x1\ x3\ \ \&\ \ ?R\ x3\ x2\ y)$
3. $\forall x\ y\ z.\ \neg x \neq [\,]\ \rightarrow\ G\ x\ y\ z = (\exists x1.\ ?T\ x\ x1\ \ \&\ \ ?R\ x1\ y\ z)$
4. $\forall x\ y\ z.\ x \neq [\,]\ \rightarrow$
   $G\ x\ y\ z = (\exists x1\ x2\ x3.\ (x1 = tl\ \ x\ \ \&\ \ x \neq [\,])\ \ \&\ \ ?K\ x\ x2\ \ \&\ \ ?R\ x2\ y\ x3\ \ \&\ \ G\ x1\ x3\ z)$
5. $\mathsf{wf}\{(x, y) | x\ y.\ x = tl\ \ y\ \ \&\ \ y \neq [\,]\}$
6. $\forall x\ y.\ ?R\ x\ N\ y = (x = y)$
7. $\forall x\ y\ z\ r.\ (\exists y1.\ ?R\ x\ y\ y1\ \ \&\ \ ?R\ y1\ z\ r) = (\exists z1.\ ?R\ y\ z\ z1\ \ \&\ \ ?R\ x\ z1\ r)$

In subsequent steps, subgoals 1, 2 and 5 are proved, producing further substitutions for the schema variables, while the others are left unproved but are further instantiated and simplified, producing the output program schema and applicability premises for the new development schema.

Because of the substitutions for B and H, subgoals 1 and 2 are partial instantiations of the defining equivalences for F of Figure 4. These two subgoals follow easily from the definition of F given in the goal. During the proof, the substitution is extended to

$$?K \leftarrow\!\!\dashv \lambda u\, v.\ K\ (hd\ \ u)\ (tl\ \ u)\ v,\quad ?T \leftarrow\!\!\dashv T,\quad ?R \leftarrow\!\!\dashv R,$$

yielding:

Proof state 3.8
$\forall x\ y.\ F\ x\ y = G\ x\ N\ y$
1. $\forall x\ y\ z.\ \neg x \neq [\,]\ \rightarrow\ G\ x\ y\ z = (\exists x1.\ T\ x1\ \ \&\ \ R\ x1\ y\ z)$
2. $\forall x\ y\ z.\ x \neq [\,]\ \rightarrow$
   $G\ x\ y\ z = (\exists x1\ x2\ x3.\ (x1 = tl\ \ x\ \ \&\ \ x \neq [\,])\ \ \&\ \ K\ (hd\ \ x)\ (tl\ \ x)\ x2\ \ \&\ \ R\ x2\ y\ x3\ \ \&\ \ G\ x1\ x3\ z)$
3. $\mathsf{wf}\{(x, y) | x\ y.\ x = tl\ \ y\ \ \&\ \ y \neq [\,]\}$
4. $\forall x\ y.\ R\ x\ N\ y = (x = y)$
5. $\forall x\ y\ z\ r.\ (\exists y1.\ R\ x\ y\ y1\ \ \&\ \ R\ y1\ z\ r) = (\exists z1.\ R\ y\ z\ z1\ \ \&\ \ R\ x\ z1 r)$

The rest of the derivation consists primarily of proving that the descent relation $\lambda x\, y.\ y = tl\ \ x \wedge x \neq [\,]$, given in subgoal 3, is well-founded. This is provable by structural induction over lists, a principle provided as part of Isabelle's data type facility (Paulson, 1994a).

$$
\begin{array}{l}
[\![\forall y.\ ?F\ [\,]\ y = ?T\ y; \\
\ \forall h\ t\ y.\ ?F\ (h\#t)\ y = (\exists x1\ x2.\ ?K\ h\ t\ x1\ \ \&\ \ ?F\ t\ x2\ \ \&\ \ ?R\ x2\ x1\ y); \\
\ \forall y\ z.\ ?G\ [\,]\ y\ z = (\exists x1.\ ?T\ x1\ \ \&\ \ ?R\ x1\ y\ z); \\
\ \forall x\ xa\ y\ z.\ ?G\ (x\#xa)\ y\ z = (\exists x1.\ ?K\ x\ xa\ x1\ \ \&\ \ (\exists x2.\ ?R\ x1\ y\ x2\ \ \&\ \ ?G\ xa\ x2\ z)); \\
\ \forall x\ y.\ ?R\ x\ ?N\ y = (x = y); \\
\ \forall x\ y\ z\ r.\ (\exists y1.\ ?R\ x\ y\ y1\ \ \&\ \ ?R\ y1\ z\ r) = (\exists z1.\ ?R\ y\ z\ z1\ \ \&\ \ ?R\ x\ z1\ r) \\
]\!]\Longrightarrow \forall x\ y.\ ?F\ x\ y = ?G\ x\ ?N\ y
\end{array}
$$

Figure 5. Isabelle encoding of tail recursion introduction schema specialized to lists

The result is:

Proof state 3.9

$\forall x\ y.\ F\ x\ y = G\ x\ N\ y$

1. $\forall x\ y\ z.\ \neg x \neq [\,]\ \rightarrow\ G\ x\ y\ z = (\exists x1.\ T\ x1\ \ \&\ \ R\ x1\ y\ z)$
2. $\forall x.\ x \neq [\,]\ \rightarrow$
   $\forall y\ z.\ G\ x\ y\ z = (\exists x2.\ K\ (hd\ \ x)\ (tl\ \ x)\ x2\ \ \&\ \ (\exists x3.\ R\ x2\ y\ x3\ \ \&\ \ G\ (tl\ \ x)\ x3\ z))$
3. $\forall x\ y.\ R\ x\ N\ y = (x = y)$
4. $\forall x\ y\ z\ r.\ (\exists y1.\ R\ x\ y\ y1\ \ \&\ \ R\ y1\ z\ r) = (\exists z1.\ R\ y\ z\ z1\ \ \&\ \ R\ x\ z1\ r)$

The remaining subgoals are not proved, but rather discharged at the meta-level to become premises of the new derived rule. But first it is desirable to simplify the defining equations for G to remove the conditionals guarding them. After the simplification we discharge the remaining four assumptions to obtain the rule of Figure 5.

This example is a first look at the deductive synthesis process outlined in §2.3. It uses a development schema, not to transform a concrete program, but to synthesize a new, more specialized, development schema. Beginning with a schematic definition of F, the deductive process synthesizes a schematic definition of G, as well as the applicability premises, by resolution with the more general schematic rule of Figure 3, followed by further theorem proving. Because we use a general-purpose logical framework rather than a special-purpose program development system, there is little conceptual or practical difference between the development of a new schema and the development of a program.

3.3. Relational Induction

The validity of the previous two schemata follows from properties of the data manipulated by logic programs represented as recursive equivalences. An alternative is to formalize a more restrictive semantics for logic programs. We explore this alternative here and formalize a semantics where logic programs inductively define sets.

This approach can be seen as specializing our theory to reflect one of the standard semantics of logic programs — that given by the least Herbrand model (van Emden and Kowalski, 1976). This model corresponds to viewing a program as constituting an inductive definition of a set of predicates: a Herbrand model is a fixedpoint of this inductive definition. First-order logic cannot characterize this model, but it can be directly expressed in higher-order logic by an inductive definition. The view of logic programs as inductive definitions has been explored (primarily in the context of negation-as-failure) by a number of authors including Hagiya and Sakurai (1984), Aronsson *et al.* (1991), and Paulson and Smith (1991). Here we adopt this view and use it to establish the correctness of another way of formalizing development schemata.

3.3.1. Inductive Definitions

First we briefly review the kind of inductive definitions we use; the general theory can be found, for example, in Aczel (1977). An inductive definition is given by a set of definite clauses that define a relation, i.e., a set. A trivial example is the program

$$\begin{array}{rcl} \mathit{Num}(0) & & \\ \mathit{Num}(s(x)) & \leftarrow & \mathit{Num}(x) \end{array}$$

which defines a set Num containing all natural numbers built from 0 and the successor function s. The first clause constitutes the base case, and the second explains how, given a term in the set, we can add a new term to the set. Iterating the second clause (perhaps transfinitely) we will reach a fixedpoint that corresponds to the minimal Herbrand model.

Any inductively defined set justifies an induction principle corresponding to its defining clauses. For example, the set Num has the following familiar induction principle.

$$\frac{\mathit{Num}(x) \quad \phi(0) \quad \begin{array}{c}[\phi(y)]\\ \vdots \\ \phi(s(y))\end{array}}{\phi(x)}$$

We may view schemata like Figure 2 as specifying an inductive definition corresponding to each program schema. We can orient each equivalence in a schematic definition from right-to-left, bringing the condition into the body of the clause. For example, the first two equivalences correspond to the clauses†

$$F(x,y) \leftarrow \neg B(x) \wedge T(x,y) \tag{3.14}$$

and

$$F(x,y) \leftarrow B(x) \wedge H(x,x_1) \wedge K(x,x_2) \wedge R(x_3,x_2,y) \wedge F(x_1,x_3)\,. \tag{3.15}$$

Now, we can take F to be the set that is inductively defined by these clauses. The corresponding induction principle is:

$$\frac{F(x,y) \quad \begin{array}{c}[\neg B(x), T(x,y)]\\ \vdots \\ \phi(x,y)\end{array} \quad \begin{array}{c}[B(x), H(x,x_1), K(x,x_2), R(x_3,x_2,y), \phi(x_1,x_3)]\\ \vdots \\ \phi(x,y)\end{array}}{\phi(x,y)}$$

Similarly, the induction principle corresponding to $G(x,y,z)$ is:

$$\frac{G(x,y,z) \quad \begin{array}{c}[\neg B(x), T(x,x_1), R(x_1,y,z)]\\ \vdots \\ \phi(x,y,z)\end{array} \quad \begin{array}{c}[B(x), H(x,x_1), K(x,x_2), R(x_2,y,x_3), \phi(x_1,x_3,z)]\\ \vdots \\ \phi(x,y,z)\end{array}}{\phi(x,y,z)}$$

As we shall see next, these two induction rules suffice to justify the schema of Figure 2.

† These are not, strictly speaking, definite clauses since some atoms such as B occur negatively. However, they constitute parameterized inductive definitions where the set being defined, e.g., F, occurs only positively in the defining body.

inductive "$f\ B\ T\ H\ K\ R\ N$"
intrs
fBase "$[\![\neg(B\ x);\ T\ x\ y]\!] \Longrightarrow (x,\ y)\ :\ f\ B\ T\ H\ K\ R\ N$"
fStep "$[\![B\ x;\ H\ x\ x1;\ K\ x\ x2;\ R\ x3\ x2\ y;\ (x1,\ x3)\ :\ f\ B\ T\ H\ K\ R\ N$
$]\!] \Longrightarrow (x,\ y)\ :\ f\ B\ T\ H\ K\ R\ N$"

inductive "$g\ B\ T\ H\ K\ R\ N$"
intrs
gBase "$[\![\neg(B\ x);\ T\ x\ x1;\ R\ x1\ y\ z]\!] \Longrightarrow (x,\ y,\ z)\ :\ g\ B\ T\ H\ K\ R\ N$"
gStep "$[\![B\ x;\ H\ x\ x1;\ K\ x\ x2;\ R\ x2\ y\ x3;$
$(x1,\ x3,\ z)\ :\ g\ B\ T\ H\ K\ R\ N]\!] \Longrightarrow (x,\ y,\ z)\ :\ g\ B\ T\ H\ K\ R\ N$"

Figure 6. Inductive definitions of program schemata

3.3.2. Representation and Proof in Isabelle

Formalizing the meaning of a logic program as an inductive definition requires a logic or theory capable of formalizing and reasoning about inductive definitions. Both higher-order logic and set theory are candidates for this because inductive definitions can be expressed in both of them based on an internally formalized account of the Knaster-Tarski fixedpoint theorem (see Paulson (1994a)). Theorem provers like Isabelle/HOL and the HOL system come with packages that support the formalization of inductive definitions. The user inputs a sets of clauses and from these the system constructs an inductive definition (as the least fixedpoint of an appropriate function) and supplies and proves the associated induction principles, which can be used in subsequent proofs about the defined sets.

In Isabelle/HOL, the inductive definitions of F and G are given by clauses (encoded as metaformulae) that formalize the two inductively defined sets. However, inductive definitions in Isabelle define *constants*, and we want to prove the rule for any F and G. Thus the problem is to represent a development schema using inductively-defined constants. The solution is to define each inductive set in terms of parameters and to supply metavariables for these parameters when formulating the schema.

In our tail-recursion example, the first step is to provide parameterized inductive definitions of F and G; Figure 6 presents the Isabelle source.† These definitions directly formalize the clausal definitions of F and G as metaformulae. For example, the clause *fBase* corresponds to (3.14); $F(x,y)$ is rendered as $(x,\ y)\ :\ f\ B\ T\ H\ K\ R\ N$ because f is a parameterized inductively defined set, not a predicate (and recall that Isabelle/HOL functions are curried). As before, " : " represents set membership and "(,)" is pairing. (Although sets and predicates are equal in HOL, they are not identical in the metalogic.)

From this definition, Isabelle automatically constructs and derives a number of inference rules, including induction principles for f and g corresponding to those given previously. These allow us to prove Goal 3.10, which encodes the applicability conditions and the conclusion of Figure 2. Together with the inductive definitions of f and g, this is a complete representation in Isabelle of the development schema.

Proof goal 3.10
$[\![\forall\ x\ y.\ (R\ x\ N\ y) = (x = y)$
$\forall\ x\ y\ z\ r.\ (\exists y1.\ (R\ y1\ z\ r)\ \&\ (R\ x\ y\ y1)) = (\exists z1.\ (R\ x\ z1\ r)\ \&\ (R\ y\ z\ z1));$
$]\!] \Longrightarrow \forall\ x\ y.\ (x,y)\ :\ f\ B\ T\ H\ K\ R\ N = (x,N,y)\ :\ g\ B\ T\ H\ K\ R\ N;$

† We have omitted type declarations and annotations for the sake of readability.

Note that, in this goal, f and g are constants, while the capitalized variables are free variables, which become schema variables in the derived rule.

We have used the induction principles to justify this rule in a theory that extends HOL with the inductive definitions of f and g given above. The machine-assisted proof was somewhat harder than the one for the corresponding rule based on well-founded sets; here we just give the main ideas. We use the same generalization as in the proof by well-founded induction. Then we split the equivalence into separate proofs for each direction. To show that the right-hand side implies the left-hand side, we use the assumption $g(x, y, z)$; using the induction rule for g we prove the induction formula $\exists v.\ f(x, v) \wedge R(v, y, z)$. This follows by the expansion of definitions and simplification with the applicability conditions. In the converse direction we have $f(x, v)$, and using the induction rule for f we prove the induction formula $\forall y\ z.\ R(v, y, z) \to g(x, y, z)$.

3.3.3. Discussion

The representation of logic programs as inductively defined sets has the virtues of semantic accuracy and wide applicability. It avoids confusion between logic programs, represented as inductively defined sets, and purely logical applicability conditions, represented as logical formulae. Because the encoding faithfully reflects the semantics of logic programs, inductive reasoning is possible for any encoded program and it is not necessary to find induction principles based on the program's data.

These advantages come at some cost in readability, intuitiveness, and ease of proof. Instead of the predicate notation familiar to logic programmers, set membership notation is mixed with predicate notation in program encodings. Theorem proving is complicated by the need to explicitly convert sets to predicates.

Although this representation of logic programs is quite different from the use of recursive equivalences, the encoding and proof of development schemata differs only in the details. The schema development methodology is independent of the choice of the underlying theory.

4. Application of Transformation Schemata

Isabelle's proof facilities can be used in a uniform way both to derive schemata and to apply them in program derivation. The derivation in §3.2 demonstrates the application of one schema in the derivation of another. This section further illustrates schema application by showing its use in program development: we describe how to apply the specialized rule for tail recursion developed in §3.2 to the familiar list reverse problem. For the sake of concreteness we include snapshots of the Isabelle proof states.

A naive Prolog program to reverse a list is:

```
reverse([],[]).
reverse([H|T],Y) :- reverse(T,I1), append(I1,[H],Y).
```

Its running time is quadratic in the size of the input because it uses **append** to construct the result Y. Transforming such programs to obtain linear time complexity is a standard exercise. We show how to do this using the schema of Figure 5.

The derivation begins with the statement of the goal[†]

[†] We again use pretty-printed Isabelle syntax, omitting type annotations and using # to denote *cons*.

Proof goal 4.1
$[\![\forall y.\ \mathbf{reverse}([], y) = (y = []);$
$\forall h\ t\ y.\ \mathbf{reverse}(h\#t, y) = (\exists x1\ x2.\ x1 = [h]\ \ \&\ \ \mathbf{reverse}(t, x2)\ \ \&\ \ \mathbf{append}(x2, x1, y))$
$]\!] \Longrightarrow \mathbf{reverse}(x, y) = ?Q\ x\ y$

The Prolog program **reverse** is represented by the premises and the goal is to show its equivalence to the (not yet given) program named by $?Q$. We define programs to operate on tuples in order to obtain a syntax closer to ordinary logic programs. Thus, although schematic functions are written in curried notation, the notation for concrete programs is uncurried. Note that using $\mathbf{reverse}([], [])$ in place of the first premise would not be accurate in this representation of programs; by using an equivalence we represent "negative information" (see §6.2 for more on this point) and rule out the possibility that the second argument is non-empty, e.g., $\mathbf{reverse}([], [1, 2, 3])$.

As the derivation proceeds, a definition for $?Q$ is synthesized. The first step is to resolve with the development schema of Figure 5, at the same time specifying the instantiation $\lambda x\, y\, z.\ \mathbf{qrev}(x, y, z)$ for $?G$. Isabelle responds with six subgoals corresponding to the premises of the schema.

Proof state 4.2
$\forall x\ y.\ \mathbf{reverse}(x, y) = \mathbf{qrev}(x, ?N1, y)$
1. $\forall y.\ \mathbf{reverse}([], y) = ?T1\ y$
2. $\forall h\ t\ y.\ \mathbf{reverse}(h\#t, y) = (\exists x1\ x2.\ ?K1\ h\ t\ x1\ \ \&\ \ \mathbf{reverse}(t, x2)\ \ \&\ \ ?R1\ x2\ x1\ y)$
3. $\forall y\ z.\ \mathbf{qrev}([], y, z) = (\exists x1.\ ?T1\ x1\ \ \&\ \ ?R1\ x1\ y\ z)$
4. $\forall h\ t\ y\ z.\ \mathbf{qrev}(h\#t, y, z) = (\exists x1\ x2.\ ?K1\ h\ t\ x1\ \ \&\ \ ?R1\ x1\ y\ x2\ \ \&\ \ \mathbf{qrev}(t, x2, z))$
5. $\forall x\ y.\ ?R1\ x\ ?N1\ y = (x = y)$
6. $\forall x\ y\ z\ r.\ (\exists y1.\ ?R1\ x\ y\ y1\ \ \&\ \ ?R1\ y1\ z\ r) = (\exists z1.\ ?R1\ y\ z\ z1\ \ \&\ \ ?R1\ x\ z1\ r)$

Resolution has instantiated $?F$ in the schema to $\lambda x\, y.\ \mathbf{reverse}(x, y)$. The derivation proceeds by solving subgoals 1, 2, 5, and 6, causing Isabelle to instantiate the remaining metavariables. When the derivation is complete, the synthesized program will be represented by the fully instantiated subgoals 3 and 4.

In the next step, resolution with the definition of **reverse** trivially solves the first two subgoals, and instantiates the schema variables $?T1$, $?K1$, and $?R1$, giving a fully instantiated definition for the new function **qrev** (now subgoals 1 and 2).

Proof state 4.3
$\forall x\ y.\ \mathbf{reverse}(x, y) = \mathbf{qrev}(x, ?N1, y)$
1. $\forall y\ z.\ \mathbf{qrev}([], y, z) = (\exists x1.\ x1 = []\ \ \&\ \ \mathbf{append}(x1, y, z))$
2. $\forall h\ t\ y\ z.\ \mathbf{qrev}((h\#t), y, z)$
   $= (\exists x1\ x2.\ x1 = [h]\ \ \&\ \ \mathbf{append}(x1, y, x2)\ \ \&\ \ \mathbf{qrev}(t, x2, z))$
3. $\forall x\ y.\ \mathbf{append}(x, ?N1, y) = (x = y)$
4. $\forall x\ y\ z\ r.\ (\exists y1.\ \mathbf{append}(x, y, y1)\ \ \&\ \ \mathbf{append}(y1, z, r))$
   $= (\exists z1.\ \mathbf{append}(y, z, z1)\ \ \&\ \ \mathbf{append}(x, z1, r))$

It remains to prove the subgoals 3 and 4, which we do by resolution with two lemmas stating simple properties of **append**. This step instantiates $?N$ to $[]$. No schema variables remain, and we have a complete implementation of **reverse** in terms of **qrev**, represented by the remaining subgoals. Discharging these, we obtain the metatheorem:

$[\![\forall y.\ \mathbf{reverse}([], y) = (y = []);$
$\forall h\ t\ y.\ \mathbf{reverse}(h\#t, y) = (\exists x1\ x2.\ x1 = [h]\ \ \&\ \ \mathbf{reverse}(t, x2)\ \ \&\ \ \mathbf{append}(x2, x1, y));$
$\forall y\ z.\ \mathbf{qrev}([], y, z) = (\exists x1.\ x1 = []\ \ \&\ \ \mathbf{append}(x1, y, z));$
$\forall h\ t\ y\ z.\ \mathbf{qrev}(h\#t, y, z) = (\exists x1.\ x1 = [h]\ \ \&\ \ (\exists x2.\ \mathbf{append}(x1, y, x2)\ \ \&\ \ \mathbf{qrev}(t, x2, z)))$
$]\!] \Longrightarrow \forall x\ y.\ \mathbf{reverse}(x, y) = \mathbf{qrev}(x, [], y)$

The definition of **qrev** represents a reverse algorithm that can be translated directly to the following Prolog program:

```
qrev([],Y,Z) :- (X1 = []), append(X1,Y,Z).
qrev([H|T],Y,Z) :- (X1 = [H]), append(X1,Y,X2), qrev(T,X2,Z).
rev(X,Y) :- qrev(X,[],Y).
```

This program is more efficient than the original. Each call to append executes in constant time since the first argument is bound to either an empty list or a singleton list; hence, the entire program executes in linear time, and in both directions (e.g., reversing input X into output Y or vice versa). Although in practice one would obtain further "constant factor" improvements by additional theorem proving (amounting to partial evaluation), we defer discussion of this kind of development to the next section.

This simple example demonstrates the benefits of representing development schemata as derived rules in a logical framework with higher-order unification. We have synthesized a program while proving its correctness and the proof is extremely simple. Nearly all the difficult theorem-proving is encapsulated in the derivation of the development schema, which can be reused for similar problems. In this sense, as well as summarizing "programming knowledge", schemata also compile many of the difficult development steps. This compilation of knowledge and development itself is one of the topics of our next section.

5. The Role of Schemata in Development

The list reverse example provides a proof of concept. However, the development is very easy and the programs involved are deterministic – essentially translations of functional programs. In this section, we discuss a more substantial and representative example of the role of schemata in development: a verified development of an efficient solution to the n-queens problem. Our goal is to give the reader some insight into the power of derived schematic rules, and show how they can be combined with different development methods, including lower-level verification steps and the interactive realization of higher-level development strategies. We also discuss the need for these other methods to compensate for the limitations of schemata.

In order to avoid excessive detail, the examples in this section are presented (translated from an Isabelle session) using the informal logical and programming language notation introduced for development schemata in previous sections.

5.1. The n-Queens Example

Our example is a verified version of the n-queens program development described by Sterling and Shapiro (1994). This development begins with a generate-and-test program, which constitutes an executable specification, and derives an improved version that interleaves the generation and testing of solutions. Sterling and Shapiro assert the equivalence of the two programs without proof; they describe the development process as "pushing the tester ... into the generator". This is a good description, but glosses over details that cost significant effort in a formalized development. Here we sketch some of them.

Informally, a solution to the n-queens problem is a placement of n queens on an

$$
\begin{array}{rrcl}
 & \mathbf{queens}(n, qs) & \leftrightarrow & (\exists ns.\ \mathbf{range}(0, n-1, ns) \wedge \mathbf{perm}(ns, qs) \wedge \mathbf{safe}(qs)) \\
n \not< m \ \rightarrow & \mathbf{range}(n, m, l) & \leftrightarrow & (l = [n]) \\
n < m \ \rightarrow & \mathbf{range}(n, m, l) & \leftrightarrow & (\exists ns.\ \mathbf{range}(n+1, m, ns) \wedge l = [n|ns]) \\
 & \mathbf{perm}(xs, []) & \leftrightarrow & (xs = []) \\
 & \mathbf{perm}(xs, [h|t]) & \leftrightarrow & (\exists ys.\ \mathbf{select}(xs, h, ys) \wedge \mathbf{perm}(ys, t)) \\
 & \mathbf{safe}([]) & & \\
 & \mathbf{safe}([h|t]) & \leftrightarrow & (\mathbf{safe}(t) \wedge \mathbf{not\_attack}(t, h)) \\
 & \mathbf{not\_attack}(xs, x) & \leftrightarrow & \mathbf{not\_attack\_3}(xs, x, 1) \\
 & \mathbf{not\_attack\_3}([], x, n) & & \\
 & \mathbf{not\_attack\_3}([h|t], x, n) & \leftrightarrow & (x \neq h + n \wedge x \neq h - n \wedge \mathbf{not\_attack\_3}(t, x, n+1)) \\
xs = [y|ys] \ \rightarrow & \mathbf{select}(xs, y, ys) & & \\
xs \neq [y|ys] \ \rightarrow & \mathbf{select}(xs, y, ys) & \leftrightarrow & (\exists h\ t\ zs.\ xs = [h|t] \wedge \mathbf{select}(t, y, zs) \wedge ys = [h|zs])
\end{array}
$$

Figure 7. Specification of the n-queens problem

n-by-n chessboard so that no queen attacks any other. The development of a formal specification from this informal statement is an interesting topic, but beyond the scope of this paper. (See Wirth (1971) and Smith (1990) for further discussions of this problem in the imperative and functional settings.) Here we merely give a brief description of the specification (Figure 7), which represents an executable though inefficient logic program.

The specification says that $\mathbf{queens}(n, qs)$ holds whenever qs is a permutation of the natural numbers from 0 to $n-1$ representing a solution to the n-queens problem. The predicate $\mathbf{range}(0, n-1, ns)$ computes the list $ns = [0, \ldots, n-1]$; **perm** computes some permutation of its input list, and **safe** checks that no queen attacks any other. Viewed as a logic program, the specification makes an essential use of nondeterminism: it generates a candidate placement of n queens, tests to see if it is a solution, and backtracks if not; all solutions can be obtained by backtracking.

We represent a placement of k queens on the first k rows of an n-by-n chessboard ($k \leq n$) by a list of k numbers between 0 and $n-1$. The ith element of the list is the column number where we have placed a queen in row i. This representation leads to the definitions of the predicates **safe**, **not_attack**, and **not_attack_3**. A list represents a safe placement if its tail is safe, and its head does not attack any member of its tail. The representation chosen guarantees that no row contains more than one queen, and working only with permutations of $[0, \ldots, n-1]$ guarantees that no column contains more than one queen. Thus **not_attack** need only check for attacks on diagonals. It does this by using an auxiliary parameter n, representing the distance between rows; a queen in column x attacks one in column h on the diagonal when $x = h \pm n$.

We derive a new definition of **queens** in terms of a new predicate **safe_perm**.

$$
\begin{array}{rrcl}
 & \mathbf{queens}(n, qs) & \leftrightarrow & (\exists ns.\ \mathbf{range}(0, n-1, ns) \wedge \mathbf{safe\_perm}(ns, [], qs)) \\
 & \mathbf{safe\_perm}([], ms, qs) & \leftrightarrow & (qs = ms) \\
ns \neq [] \ \rightarrow & \mathbf{safe\_perm}(ns, ms, qs) & \leftrightarrow & (\exists d_1\ d_2.\ \mathbf{select}(ns, d_2, d_1) \wedge \\
 & & & \mathbf{not\_attack}(ms, d_2) \wedge \mathbf{safe\_perm}(d_1, [d_2|ms], qs))
\end{array}
$$

This represents the logic program

```
queens(N, Qs) :- range(1, N, Ns), safe_perm(Ns, [], Qs)).

safe_perm([], Qs, Qs).
safe_perm(Ns, Ms, Qs) :-
        select(Ns, D2, D1),
        not_attack(Ms, D2),
        safe_perm(D1, [D2|Ms], Qs).
```

This is precisely Sterling and Shapiro's solution, except that it does not use negation.

Our synthesis is organized around three schema application steps, part of which we will see in the next section. The first uses a "loop jamming" schema to merge the definitions of **safe** and **perm** into a new predicate **perm_safe**. The second does the real work: it exploits the syntactic context established by loop jamming to synthesize a new predicate **safe_perm** that tests partial solutions and discards them as soon as a conflict is found. Finally we obtain a linear speedup by eliminating list reversals and by using list *cons* instead of **append** to build up the candidate solutions. The schema applications are supplemented by verification steps, used to develop the theory of the specification as well as to "glue together" the schematic proofs.

5.2. Schemata as Compilations of General Programming Knowledge

Derived rules compile proof knowledge in a precise sense: there is a schematic derivation that justifies the rule. The schema variables express generality, and in the case of program development schemata, they express generality of programming expertise. The question arises of what kind of development knowledge can be formalized by schemata. We illustrate some partial answers through two of the steps of the n-queens development: one applying a purely syntactic schema, the other applying a schema with semantic applicability conditions. Both are instances of well-known programming concepts: loop jamming in one case and incremental computation in the other.

A purely syntactic schema generalizes knowledge expressible through syntactic properties of programs, enforcing these properties through higher-order unification. Since function-typed schematic variables unify with functions from syntax to syntax, schemata can express generalizations about function composition, scope restrictions, and (using segment variables) the structure of sequences of associative operators (see Huet and Lang (1978) for a discussion). Sometimes these syntactic restrictions are strong enough that the development schema can be proved without any applicability premises. "Loop jamming" for a recursive data type like lists is an example (see Figure 8).

The application of a syntactic schema is particularly simple: The developer merely resolves the schema with a suitable goal and solves the resulting proof obligations by resolution with the appropriate program definitions. In the n-queens problem, the schema allows us to merge the predicates **safe** and **perm** (Figure 9).

In contrast to syntactic schemata are the more powerful schemata that include schematic applicability premises. These premises enforce program properties expressible in the development theory, supporting general reasoning about unknown functions and relations based on their semantic as well as syntactic properties.

The potential for greater generality of such schemata is illustrated by the first two tail-recursion schemata (Figures 4 and 5) developed in §3. The more general rule applies

Recursive definitions to be jammed

$$
\begin{array}{rcl}
P([], x) & \leftrightarrow & P_1(x) \\
P([h|t], x) & \leftrightarrow & \exists i.\ P_2(x, h, i) \wedge P(t, i) \\
Q([]) & & \\
Q([h|t]) & \leftrightarrow & Q(t) \wedge Q_1(h, t)
\end{array}
$$

New definition

$$
\begin{array}{rcl}
R([], x) & \leftrightarrow & P_1(x) \\
R([h|t], x) & \leftrightarrow & \exists i.\ P_2(x, h, i) \wedge R(t, i) \wedge Q_1(h, t)
\end{array}
$$

Conclusion:

$$
P(x, y) \wedge Q(x) \quad \leftrightarrow \quad R(x, y)
$$

Figure 8. A loop jamming schema

Input definitions:

$$
\begin{array}{rcl}
\mathbf{perm}(xs, []) & \leftrightarrow & (xs = []) \\
\mathbf{perm}(xs, [h|t]) & \leftrightarrow & (\exists ys.\ \mathbf{select}(xs, h, ys) \wedge \mathbf{perm}(ys, t)) \\
\mathbf{safe}([]) & & \\
\mathbf{safe}([h|t]) & \leftrightarrow & (\mathbf{safe}(t) \wedge \mathbf{not\_attack}(t, h))
\end{array}
$$

Output definition:

$$
\begin{array}{rcl}
\mathbf{perm\_safe}(xs, []) & \leftrightarrow & (xs = []) \\
\mathbf{perm\_safe}(xs, [h|t]) & \leftrightarrow & (\exists ys.\ \mathbf{select}(xs, h, ys) \wedge \mathbf{perm\_safe}(ys, t) \wedge \mathbf{not\_attack}(t, h))
\end{array}
$$

Figure 9. Loop jamming in the n-queens development

to programs over any data type for which an appropriate well-founded set can be found, thanks to the applicability premise (3.11 in Figure 3). The cost is the extra proof obligation imposed on the user. In the specialized version, the presence of list constructors, i.e., the syntactic form of the arguments, is enough to justify the derivation step, leading to a rule narrower in scope but simpler to apply.

Generality can lead to surprising opportunities for reuse. A small modification (discussed in §5.3) of the schema for logic programs over well-founded relations of Figure 3 in §3.1 plays a major role in the n-queens development. This is surprising because the initial program makes use of backtracking, and as a result the schema might seem to offer no improvement: the program stack cannot be eliminated because of the need to record backtracking points. However, the transformation is used in this case not to eliminate the stack, but to incrementally build candidate solutions. This allows the search to discard partial candidates as soon as an attacking queen is generated. Of course, the relation between tail recursion transformations and other transformations that increase incrementality is well-known (Bird, 1984), but this development schema formalizes a wide range of incrementality improvements in one rule.

The result of the application of the schema to the n-queens problem is shown in Figure 10; the input definition is obtained from the output definition of Figure 9 by methods to be discussed later. The new predicate **l_not_attack** is a generalization to lists of **not_attack**, introduced so that the program types are compatible with the schema.

Input definition:

$$
\begin{array}{llll}
xs = [] & \rightarrow & (\mathbf{perm\_safe}(xs, ys) & \leftrightarrow \; ys = []) \\
xs \neq [] & \rightarrow & (\mathbf{perm\_safe}(xs, ys) & \leftrightarrow \; (\exists i\; q\; q_1\; q_2.\; \mathbf{select}(xs, q, i) \wedge \mathbf{perm\_safe}(i, q_2) \\
 & & & \quad \wedge\; q_1 = [q] \wedge \mathbf{l\_not\_attack}(q_1, q_2) \wedge \mathbf{append}(q_1, q_2, ys)))
\end{array}
$$

Output definition:

$$
\begin{array}{llll}
xs = [] & \rightarrow & (\mathbf{safe\_perm}(xs, a, ys) & \leftrightarrow \; ys = a) \\
xs \neq [] & \rightarrow & (\mathbf{safe\_perm}(xs, a, ys) & \leftrightarrow \; (\exists i\; q\; q_1\; q_2.\; \mathbf{select}(xs, q, i) \wedge q_1 = [q] \wedge \mathbf{l\_not\_attack}(q_1, a) \\
 & & & \quad \wedge\; \mathbf{append}(q_1, a, q_2) \wedge \mathbf{safe\_perm}(i, q_2, ys)))
\end{array}
$$

Figure 10. Application of a tail-recursion schema in the n-queens development

In the output definition of Figure 10, the accumulator a is tested for conflicts with the candidate placement q, and the recursive call is not executed if a conflict is found.

The two schema applications illustrated here demonstrate how some kinds of programming knowledge can be "compiled" into derived rules of inference. At the same time, they illustrate some of the limitations of schemata.

5.3. Limitations of Schemata

Schemata can capture some generalizations about programming, but their power is limited by the over-specificity of both program syntax and logical syntax. Here we consider some types of programming knowledge that are either poorly captured by our schemata or not expressible at all.

To begin with, schemata force a commitment to a syntactic representation that is often too specialized. For example, the "loop jamming" schema of Figure 8 is not fully general, even for programs defined by recursion over lists. The arity of predicates as well as the order of conjuncts in the bodies of the definitions have been tailored to the problem at hand. In the design of schemata it is often necessary to trade generality for ease of application and readability. For example, packaging all parameters of each predicate or function into a tuple can sometimes enable one schema to apply to programs with different arities, at the expense of tedious detail in the proofs as well as obscuring the syntax of the programs. A better solution is probably to automatically generate and prove members of a "schema family" from user specifications of arities.

Another example is the use of the "tail-recursion" schema to introduce incremental computation into the n-queens program. There, the schema of Figure 3 had to be modified, because of different numbers of existential variables that represent sharing between the conjuncts of a definition. A fully general version of this schema would be harder to verify and apply. For the n-queens problem, the original schema was altered to allow the input to be split by the descent relation nondeterministically into two parts, represented by two existential variables. A more general version might allow splitting into an arbitrary number of parts, represented by a tuple-typed existential variable. Such a strategy has a cost in readability and ease of application.

Some generalizations about programming are not pattern matching even in the informal sense and hence cannot be captured by schemata. Design patterns (Gamma *et al.*, 1995), which broadly classify software architectures and solutions to software design problems, are an attempt at deeper generalization than can be expressed by schemata. Alternatively, development often involves insights, for instance which development steps to try and in what order, that are clearly beyond the scope of our schemata.

5.4. Mixed Strategies in a Logical Framework

The programmability, flexibility, and transparency of a logical framework like Isabelle offer possibilities for easily integrating the use of development schemata with other development methods, both more and less general than schemata.

At a lower level of abstraction, verification steps may be an appropriate part of developments based primarily on synthesis. In our n-queens synthesis, we verified concrete programs for two main reasons. First, we needed to prove properties of the specification. One such property is $\mathbf{safe}(qs) \leftrightarrow \mathbf{safe}(\mathbf{reverse}(qs))$, used to remove the use of the list reverse function introduced with the partial solution accumulator. Second, the result of one development schema may fail to match the premises of the next one to be used, because of trivial syntactic differences; this is illustrated by the mismatch between the output program of Figure 9 and the input program of Figure 10. The necessary simplification, partial evaluation, or even somewhat deeper transformations are often easier to state concretely and verify than to capture as abstract schemata. Mixing verification with schema application makes few demands on the user in a framework like Isabelle: since schema application is like the application of any other logical rule, schema-based development and verification proof require the same set of skills, allowing the user to select an appropriate method freely.

At a higher level of abstraction, tactics and general metaprograms can be used to generate families of schemata, or implement algorithmic development knowledge and heuristically guided search procedures, without compromising correctness. In principle, tactics could also be used to implement the knowledge formalized by a development schema. But schemata are better than tactics at explaining the knowledge they capture because they are transparent: they are declarative, and they are applied by the same mechanism used for any other inference rule. A schema is easier to understand than a tactic in the same way that a specification is easier to understand than a program. Schemata are also more robust than tactics since they are less sensitive to changes in the operation of the proof assistant. Thus a fruitful line of development is to formalize as much as possible schematically and supplement schemata with rewriting, decision procedures, heuristic search, and other tactics.

6. Related work

We consider here related approaches to program development in general as well as work in logic programming related to our examples.

6.1. Formal Program Development

The formal program development methodologies most closely related to our work are resolution based approaches, deductive synthesis, and constructive synthesis.

6.1.1. Resolution Based Development

In our work, different kinds of program development (verification, transformation, and synthesis) are all based on the same mechanism: formal proof where rules are applied using higher-order resolution. The idea of using resolution as a basis for development has a long history in theorem proving in first-order theories using first-order resolution.

An early application, due to Green (1969), was to show how resolution could be used to construct terms that represent plans or, more generally, functions.

In particular, to see if $\forall x.\ \exists y.\ R(x,y)$ follows from a set of formulae, Green negates it and turns it into the clause $\{\neg R(a,y)\}$, where a is a new constant. He then uses resolution to derive an inconsistency in some appropriate theory, yielding a unifier for y. For example, in a theory of arithmetic we might get the unifier $a+2$ for y. Hence resolution can be used to solve for y as a "function" of a, and only first-order variables and unification have been used in the proof process. First-order variables and unification suffice because skolemizing removes quantifiers, which are the only binding operators in first-order logic, and at the same time (implicitly) extends the signature (e.g., with a) in which the theorem is proved. Resolution does not really synthesize a function, but rather a first-order term; however, since a is arbitrary, it can be generalized afterwards, i.e., y is the function $\lambda a.\, a+2$.

The above kind of development is first-order in two respects: first-order theories formalize data types and programs and first-order resolution builds programs during proofs. This is in contrast to our work, which is higher-order both with respect to the logics involved and with respect to resolution. As explained in §2.4, we use a higher-order metalogic to formalize rules. This supports a more direct representation of rules than a clausal form would, and higher-order variables play an essential role in formalizing program schemata, i.e., specifying the form and function of program instances.

Another important distinction is the way we introduce possibly recursive program definitions: a program schema defines a class of functions where the defined function may be used in the definition. In our developments, schema application leads to concrete recursive functions, which arise as subgoals during proofs. For example, in §4, we apply our tail recursion schema to a goal stating the equality of **reverse** and a program schema; the definition of **qrev** manifests itself in the proof in the form of subgoals (the third and fourth), which are discharged at the end of the derivation by making them assumptions. Hence, program development (like schema development) constitutes the derivation of a rule in the metalogic: the equivalence of the specification and the program follows from premises that constitute the new program definition. This approach exploits the duality between proof states and rules (see §2.3) and is not possible in the setting of a conventional first (or higher-order) resolution prover.

6.1.2. Deductive Synthesis

There are a number of approaches loosely classified under the heading of deductive synthesis that also have close ties to resolution based development, even if they are formalized otherwise. A standard example of deductive synthesis is the *Deductive Tableau Methodology* developed by Manna and Waldinger (1992). Under this methodology, one proves a $\forall/\exists$ goal like $\forall x.\ \exists y.\ R(x,y)$ by manipulating $R(x,y)$ in a tableau where x is turned into an eigenvariable and y is placed in a special "output column". Essentially, y plays the role of a metavariable that is incrementally instantiated during subsequent proof steps. At the end of the proof, if variables have been instantiated to ground terms meeting certain side conditions then the tableau yields a completed program.

As with the work of Green, it is possible to recast Deductive Tableau proofs as schematic proofs constructed using resolution. The essence is showing that one can derive rules in an appropriate theory that can simulate Deductive Tableau proof steps and whose application by higher-order resolution builds programs (see Ayari and Basin (1996) for details). The idea of implementing or reinterpreting different approaches to deductive synthesis using schematic or resolution based proofs has also been investigated by others (Coen, 1992; Heisel *et al.*, 1990; Kraan *et al.*, 1993).

The idea of formalizing transformation rules within higher-order logic has been examined by other researchers in different contexts. For example, Shankar (1996) formalizes program transformations for functional programs as theorems in higher-order logic in PVS. These can be manually applied (by providing substitutions to "backchain" through lemmata) to carry out transformations. Another example is the work of Kolyang *et al.* (1996), who formalize a "global search" transformation schema in Isabelle/HOL, using the ideas presented here; they also show how such transformation can be supported with a graphical interface.

6.1.3. Constructive Synthesis

A distinction is sometimes made in the literature between deductive synthesis methods and constructive synthesis methods, e.g, (Deville and Lau, 1993). The latter are typically based on some kind of constructive logic, or type theory, equipped with a notion of realizability, and one develops programs by proving that a term t (the program) belongs to a type T (the specification), i.e., $t \in T$.

Metavariables can be employed in type theories in ways similar to what we have described here. For example, Paulson (1994b) documents an implementation in Isabelle of a constructive type theory (one of Martin-Löf's). Rules are applied with higher-order resolution and, in proving $t \in T$, t may be a ground term of the type theory, a metavariable, or even a combination of the two. In the first case we have verification, in the second, synthesis, and in the third, a hybrid, where some parts of the program structure are known and others are left unspecified (see Hesketh *et al.* (1992) for uses of such hybrid verification/synthesis). Note too that T may be a metavariable, in which case theorem proving corresponds to type reconstruction. Similarly, Anderson (1994) uses higher-order unification for program derivation in constructive logic, representing transformations of *proof objects* (rather than programs) by derived schematic rules in the metalogic LF.

Thus, the idea of development based on schematic proof suffices to explain program synthesis, at least in some constructive logics. A common explanation of program synthesis is that it is possible because the logic is constructive: a proof yields a program that exhibits the implicit construction. This is correct but the emphasis can mislead. Unification during schematic developments is enough to account for synthesis. Constructivity does not play a role in the *synthesis* of programs, but rather in their *execution* and *meaning*. That is, because the logic is constructive, the terms extracted can be executed and their evaluation behavior agrees with the semantics of the type theory.

6.2. Logic Program Development

Given the close relationship between logic programs and first-order theories, it is not surprising that formal program development has been an active research topic in the logic programming community (Deville and Lau, 1993; Pettorossi and Proietti, 1997). There has been a considerable cross-fertilization of ideas also from research on development in general: many approaches for synthesizing or transforming functional and imperative programs have their logic program counterparts, e.g., unfold-fold style derivations

(Tamaki and Sato, 1984) or other approaches to deductive synthesis based on substituting formulae with equivalent formulae (Clark and Tärnlund, 1977; Hogger, 1981).

The fact that logic programs are both formulae and programs may give the impression that logic program development is simply reasoning about the equivalence of formulae. Unfortunately, neither the notion of which formulae constitute programs nor what equivalence means is straightforward. For instance, Horn clauses constitute an obvious class of programs, but are typically inadequate for reasoning about program properties as they formalize only "positive" information. For this reason, many researchers prefer to adopt a closed-world assumption and reason about the so-called (Clark) completion of a logic program. Furthermore, given some representation of programs as formulae (e.g., Horn clauses or their completions) there are many rival notions of equivalence. For instance, Maher shows that there is a surprisingly large number of distinct "reasonable" notions of equivalence, which are based on the logical, operational, or functional reading of programs (Maher, 1987). The considerable body of literature in logic program development is partially a manifestation of this complexity as different authors consider different classes of programs and notions of equivalence.

In our work, we have formalized logic programs in two different ways. First, as first-order formulae in a restricted language (see §3.1 and §3.2), where each program is associated with a set of formulae that define it for different possible inputs. This formalism is that used by Bundy, Smaill and Wiggins (1990) and is similar to the *logic descriptions* of Flener and Deville (1991). Second, as Horn clauses (see §3.3), which constitute an inductive definition. We have also explored different notions of equivalence. With respect to the first formalization we have viewed equivalence as provable equivalence in either a theory of well-founded relations or a theory that axiomatizes recursive data types with their own induction principles. With respect to the formalization based on inductive definitions, equivalence means provable equivalence in higher-order logic in a theory where the program schemata are formalized as inductively defined relations. This approach, of viewing pure logic programs as defining not a first-order theory, but rather a set of inductive definitions, has been also taken by Paulson and Smith (1991), Aronsson *et al.* (1991) and Hagiya and Sakurai (1984).

Of course, there are many other approaches possible; ours simply serve as examples chosen to illustrate the ideas of this paper. In the end, which approach one takes depends on what properties should hold for the synthesized or transformed program. In our examples, the second approach yields more general schemata than the first, but it requires a more specialized semantics for logic programs. That is, pure logic programs (or their completions) no longer constitute a first-order theory, with a corresponding notion of semantic entailment, but rather an inductive definition, which corresponds to the 'T_P'-semantics of van Emden and Kowalski (1976).

7. Conclusion and Future Work

Our focus has been on the unification of distinct concepts in a common framework and the practical gains that result from this. The use of a formal metalogic and proof by higher-order resolution are the two key ingredients. The formal metalogic supports the uniform formalization of development schemata and developments, and higher-order resolution allows us to construct different kinds of developments in a similar way, including synthesis, transformation, verification, and the derivation of development schemata themselves. The practical gains are based on the ability to use what has now become standard theorem proving technology in exploring and carrying out schema-based developments and combining this with other development styles.

One advantage of a common framework is that it aids comparison. We have made various comparisons throughout this paper, e.g., development methodologies (§6.1) and development styles (§5). Moreover, our framework provides, in a very general way, a formal basis for "comparative schematology": we can compare schemata themselves based on various formalizable relationships between them and their supporting theories. For example, within a fixed theory we can compare development schemata under different notions of subsumption. From the schema for general well-founded data types given in §3.1, we derived a tail recursion schema for lists in §3.2. The second schema is more specific than the first in two ways: its types are more specialized, and the well-founded order is fixed. These are independent specializations; the first constitutes a type instantiation and the second an instantiation of a term (relation) schema variable. There are other possible notions of subsumption too, e.g., one schema may logically entail another. The use of a logical framework allows us to study these relationships and their tradeoffs in practice. For example, specialized schemata often entail less deduction after application because the deduction saved is, essentially, compiled into the derivation of the schema.

Our framework also eases the comparison and development of schemata for different theories and even languages. At the heart of such a comparison is a quest for generality: much of program development is centered around algorithmic and data type refinement and these are often, when viewed from a high enough level of abstraction, language independent. The schemata we used for logic program development are variants of the functional program development schema given in Figure 1, which is derivable in a rather different formal system (e.g., using fixedpoint induction in some formalization of domain theory, such as LCF) . Similar ideas are presented by Basin and Friedrich (1996), where schemata for logic program development are used to develop circuit descriptions.

Finding useful development schemata is a difficult and creative process and our example suggests the possibility of "technology transfer" where, for example, development schemata for functional program transformation can be suitably reformulated for logic programs. Our exploration of different notions of equivalence can, in this light, be seen as an exploration of which semantics and corresponding deductive systems are sufficient to derive rules that are developed for, and derived in, rather different settings. By viewing development schemata as derived rules, we have the possibility of showing that certain translation techniques work uniformly over classes of schemata. Such an investigation could be based on proof theory (e.g., establishing sufficient conditions for relational schemata to be provable by relational induction when the corresponding functional schemata are provable by fixedpoint induction) or semantically (e.g., based on a common semantics for the rules). Investigating these possibilities, and, more generally, metatheory appropriate for establishing transfer results for development schemata, is an interesting area for future work.

References

Aczel, P. (1977). An introduction to inductive definitions. In Barwise, J., editor, *Handbook of Mathematical Logic*, pages 739–82. North-Holland, Amsterdam.

Anderson, P. (1994). Representing proof transformations for program optimization. In *Proceedings of the 12th International Conference on Automated Deduction*. Springer-Verlag. Available as INRIA Research Report 2229.

Anderson, P., Basin, D. (1995). Deriving and applying logic program transformers. In *Algorithms,* *Concurrency and Knowledge (1995 Asian Computing Science Conference)*, volume 1023 of *LNCS*, pages 301–318, Pathumthani, Thailand. Springer-Verlag.

Aronsson, M., Eriksson, L.-H., Hallnäs, L., Kreuger, P. (1991). A survey of GLCA: A definitional approach to logic programming. In *Extensions of Logic Programming*, volume 475 of *LNCS*, pages 49–100. Springer-Verlag.

Ayari, A., Basin, D. (1996). Generic system support for deductive program development. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS-96)*, volume 1055 of *LNCS*, pages 313–328, Passau, Germany. Springer-Verlag.

Basin, D., Bundy, A., Kraan, I., Matthews, S. (1993). A framework for program development based on schematic proof. In *7th International Workshop on Software Specification and Design*, pages 162–171, Los Angeles. IEEE Computer Society Press.

Basin, D., Friedrich, S. (1996). Modeling a hardware synthesis methodology in Isabelle. In *Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *LNCS*, pages 33–50, Turku, Finland. Springer-Verlag.

Basin, D. A. (1994). Logic frameworks for logic programs. In *4th International Workshop on Logic Program Synthesis and Transformation, (LOPSTR'94)*, volume 883 of *LNCS*, pages 1–16, Pisa Italy. Springer-Verlag.

Bird, R. S. (1984). The promotion and accumulation strategies in transformational programming. *ACM Transactions on Programming Languages and Systems*, **6**(4):487–504.

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

Church, A. (1940). A formulation of the simple theory of types. *Symbolic Logic*, **5**(1):56–68.

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

Coen, M. D. (1992). Interactive program derivation. Technical Report 272, Cambridge University Computer Laboratory, Cambridge.

Cooper, D. C. (1966). The equivalence of certain computations. *Computer Journal*, **9**:45–52.

de Bakker, J., de Roever, W. (1973). A calculus for recursive program schemes. In Nivat, M., editor, *Automata, Languages and Programming*, Amsterdam. North-Holland.

de Bruijn, N. (1980.). A survey of the project Automath. In *Essays in Combinatory Logic, Lambda Calculus, and Formalism.*, pages 589–606. Academic Press.

Deville, Y., Lau, K.-K. (1993). Logic program synthesis. *Journal of Logic Programming*, **12**.

Felty, A., Miller, D. (1988). Specifying theorem provers in a higher-order logic programming language. In *9th International Conference On Automated Deduction*, Argonne, Illinois.

Flener, P., Deville, Y. (1991). Towards stepwise, schema-guided synthesis of logic programs. In Clement, T., Lau, K.-K., editors, *Logic Program Synthesis and Transformation*, pages 46–64. Springer-Verlag.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., Booch, G. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Green, C. (1969). Application of theorem proving to problem solving. In *Proceedings of the IJCAI-69*, pages 219–239.

Hagiya, M., Sakurai, T. (1984). Foundation of logic programming based on inductive definition. *New Generation Computing*, **2**:59–77.

Harper, R., Honsell, F., Plotkin, G. (1993). A framework for defining logics. *Journal of the Association for Computing Machinery*, **40**(1):143–184.

Heisel, M., Reif, W., Stephan, W. (1990). Tactical theorem proving in program verification. In *10th International Conference On Automated Deduction*, pages 117–131, Kaiserslautern, Germany.

Hesketh, J., Bundy, A., Smaill, A. (1992). Using middle-out reasoning to transform naive programs into tail recursive ones. In *11th Conference on Automated Deduction*, number 607 in Lecture Notes in Artificial Intelligence, pages 310–324. Springer-Verlag.

Hill, P., Lloyd, J. (1991). The Gödel Report. Technical Report TR-91-02, Department of Computer Science, University of Bristol. Revised in September 1991.

Hindley, J. R., Seldin, J. P. (1986). *Introduction to Combinators and λ-Calculus.* Cambridge University Press, Cambridge.

Hoffmann, B., Krieg-Brückner, B., editors (1993). *Program Development by Specification and Transformation*, volume 680 of *LNCS*. Springer-Verlag.

Hogger, C. (1981). Derivation of logic programs. *Journal of the Association for Computing Machinery*, **28**(2):372–392.

Huet, G. (1975). A unification algorithm for typed lambda-calculus. *Theoretical Computer Science*, **1**(1):27–57.

Huet, G., Lang, B. (1978). Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, **11**:31–55.

Kolyang, Santen, T., Wolff, B. (1996). Correct and user-friendly implementation of transformation systems. In Gaudel, M.-C., Woodcock, J., editors, *FME '96 — Industrial Benefits and Advances in Formal Methods*, LNCS 1051, pages 629–648. Springer Verlag.

Kraan, I., Basin, D., Bundy, A. (1993). Middle-out reasoning for logic program synthesis. In *10th International Conference on Logic Programming (ICLP93)*, pages 441–455, Budapest Hungary. MIT Press.

Luo, Z., Pollack, R. (1992). Lego proof development system: User's manual. Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh.

Maher, M. (1987). Equivalences of logic programs. In Minker, J., editor, *Foundations of Deductive Databases and Logic Programming*. Morgan Kaufmann.

Manna, Z., Vuillemin, J. (1972). Fixpoint approach to the theory of computation. *Communication of the Association for Computing Machinery*, **15**(7):528–536.

Manna, Z., Waldinger, R. (1992). Fundamentals of the deductive program synthesis. *IEEE Transactions on Software Engineering*, **18**(8):674–704.

Partsch, H. A. (1990). *Specification and Transformation of Programs*. Springer-Verlag.

Paulson, L. C. (1985). Natural deduction as higher-order resolution. *Journal of Logic Programming*, **3**:237–258.

Paulson, L. C. (1989). The foundation of a generic theorem prover. *Journal of Automated Reasoning*, **5**:363–397.

Paulson, L. C. (1994a). A fixedpoint approach to implementing (co)inductive definitions. In *12th Conference on Automated Deduction*, number 814 in Lecture Notes in Artificial Intelligence, Nancy, France. Springer-Verlag.

Paulson, L. C. (1994b). *Isabelle : a generic theorem prover; with contributions by Tobias Nipkow*, volume 828 of *LNCS*. Springer, Berlin.

Paulson, L. C., Smith, A. W. (1991). Logic programming, functional programming, and inductive definitions. In *Extensions of Logic Programming*, volume 475 of *LNCS*, pages 283–310. Springer-Verlag.

Pettorossi, A., Proietti, M. (1997). Synthesis and transformation of logic programs using unfold/fold proofs. Technical Report R.457, Istituto di Analisi dei Sistemi ed Informatica (IASI-CNR), Roma, Italy. (To appear on the Journal of Logic Programming, 1999.).

Pettorossi, A., Proietti, M. (1998). Transformation of logic programs. In Gabbay, D. M., Hogger, C. J., Robinson, J. A., editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 697–787. Oxford University Press.

Pfenning, F. (1991). Logic programming in the LF logical framework. In *Logical Frameworks*, pages 149 – 181. Cambridge University Press.

Shankar, N. (1996). Steps towards mechanizing program transformations using PVS. *Science of Computer Programming*, **26**(1–3):33–57.

Smith, D. R. (1990). KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, **16**(9):1024–1043.

Sterling, L. S., Shapiro, E. (1994). *The Art of Prolog: Advanced Programming Techniques*. MIT Press.

Tamaki, H., Sato, T. (1984). Unfold/fold transformations of logic programs. In *Proceedings of 2nd ICLP*.

van Emden, M., Kowalski, R. (1976). The semantics of predicate logic as a programming language. *Journal of the Association for Computing Machinery*, **23**:733–42.

Wirth, N. (1971). Program development by stepwise refinement. *Communications of the ACM*, **14**(4):221–227.