Deriving and Applying Logic Program Transformers*

Penny Anderson and David Basin
Max-Planck-Institut für Informatik,
Im Stadtwald, D-66123 Saarbrücken, Germany
{anderson,basin}@mpi-sb.mpg.de
phone: +49 681 302-5435 (fax: 302-5401)

Abstract

We present a methodology for logic program development based on the use of verified transformation templates. We use the Isabelle Logical Framework to formalize transformation templates as inference rules. We derive these rules in higher-order logic and afterwards use higher-order unification to apply them to develop programs in a deductive synthesis style. Our work addresses the pragmatics of template formalization and application as well as which theories and semantics of programs and data we require to derive templates.

Key words: program transformation, logic programs, higher-order unification, logical frameworks.

1 Introduction

We investigate the transformation of logic programs based on the use of *transformation templates* which formalize equivalences between schemata representing logic programs. Our focus and contributions center on three problems:

1. How may we acquire and formalize useful templates?
2. What foundations are appropriate for formally demonstrating the correctness of templates?
3. How should we organize transformation proofs so that logic programs can be developed alongside their proofs of correctness in a 'deductive synthesis' development style.

Our motivation is to provide a foundation in which templates may be derived and then applied to program development. We seek a uniform approach in which the developments of both templates and programs themselves are formal, machine supported activities which take place in a given mathematical theory.

We explore these problems within the context of the Isabelle Logical Framework [19], a theorem proving system which provides a *metalogic* and support for specifying and using *object logics*. Within a particular object logic, higher-order logic (HOL), we specify

*The authors were funded by the German Ministry for Research and Technology (BMFT) under grant ITS 9102. Responsibility for the content lies with the authors. The authors thank Alan Smaill for pointing them to some of the literature on logic programs as inductive definitions.

$$\textit{Functional Definition for } F$$

$$\neg B(x) \Rightarrow F(x) = T(x) \tag{1}$$

$$B(x) \Rightarrow F(x) = R(F(H(x)), K(x)) \tag{2}$$

$$\textit{Functional Definition for } G$$

$$\neg B(x) \Rightarrow G(x, y) = R(T(x), y) \tag{3}$$

$$B(x) \Rightarrow G(x, y) = G(H(x), R(K(x), y)) \tag{4}$$

$$\textit{Applicability Conditions:}$$

$$R(x, N) = x \tag{5}$$

$$R(R(x, y), z) = R(x, R(y, z)) \tag{6}$$

$$\textit{Correctness:}$$

$$F(x) = G(x, N) \tag{7}$$

Figure 1: Template for Tail Recursion Introduction, Functional Form

and reason about templates, which are formalized as proof rules expressing the equivalence of two logic program schemata. These rules are not taken as axioms, but rather derived within a conservative extension of higher-order logic. After derivation, the rules are applied, using higher-order unification, to transform logic programs during proofs of their correctness. Below we discuss these points in more detail, providing an overview to the technical development in following sections.

The first problem concerns acquiring and formalizing templates. For acquisition, we show that 'technology transfer' is possible by reformulating templates developed for functional program transformation. Although logic programs have a different operational semantics than functional programs, many of the large catalog of transformations developed for functional programs (e.g., [15, 12, 17]), when suitably reformulated, are also useful for optimizing logic programs. Figure 1 presents an example that we will consider in detail in this paper: a template for transforming functions to a tail-recursive form taken from [12] (see also [4]). This template defines, as conditional equations, functions F (Equations 1–2) and G (Equations 3–4) and states an equality between them (Equation 7) whenever two conditions hold. These applicability conditions state that the function R has a right identity N and is associative. In the context of functional program transformation, such templates have been applied using second-order matching to transform particular instances of F to G [15]; the above template, for example, may be applied to transform a naive implementation of list reversal or factorial to a more efficient tail recursive version.

We transform functional templates by *flattening* the functional schemata they contain to obtain relational templates suitable for transforming logic programs; that is, we iteratively replace n-ary functions with $n+1$-ary predicates which represent their graphs. The flattened template corresponding to the above is given in Figure 2. In the logic program context, the transformation performs *last call optimization*, which is the logic programming equivalent of tail recursion removal (e.g., see [16]).

In our work, transformation templates are formalized in the metalogic as rules, and formally verified by deriving them in an appropriate formal metatheory. However, to derive

Relational Definition for F

$$\neg B(x) \Rightarrow (F(x,y) \Leftrightarrow T(x,y)) \tag{8}$$

$$B(x) \Rightarrow (F(x,y) \Leftrightarrow \exists x_1, x_2, x_3. H(x,x_1) \wedge K(x,x_2) \wedge F(x_1,x_3) \wedge R(x_3,x_2,y)) \tag{9}$$

Relational Definition for G

$$\neg B(x) \Rightarrow (G(x,y,z) \Leftrightarrow \exists x_1. T(x,x_1) \wedge R(x_1,y,z)) \tag{10}$$

$$B(x) \Rightarrow (G(x,y,z) \Leftrightarrow \exists x_1, x_2, x_3. H(x,x_1) \wedge K(x,x_2) \wedge R(x_2,y,x_3) \wedge G(x_1,x_3,z)) \tag{11}$$

Applicability Conditions:

$$R(x,N,y) \Leftrightarrow (x = y) \tag{12}$$

$$(\exists y_1. R(x,y,y_1) \wedge R(y_1,z,r)) \Leftrightarrow (\exists z_1. R(y,z,z_1) \wedge R(x,z_1,r)) \tag{13}$$

Correctness:

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

Figure 2: Template for Tail Recursion Introduction, Relational Form

relational templates there is a problem to overcome: the correctness of functional templates are typically established using proof techniques such as fixedpoint induction that are not available for logic programs, which have different semantics.

We present two solutions to this problem: template derivations based on induction over inductively defined datatypes, and derivations based on induction over inductively defined relations. In the first approach, we specialize the datatypes of some of the arguments, specialize some predicate variables, and prove the specialized rule by induction over datatypes. For example, we take the template of Figure 2, specialize the type of x to lists, instantiate some of the schematic variables to concrete predicates over lists, and prove the equivalence by structural induction on x. This is possible because, in Figure 2, we have left types implicit and interpret them polymorphically. We may specialize the template by proving it for a specialized type instance where we interpret some of the schematic variables as functions over these types. Isabelle supports this view of type polymorphism through type variables and type instantiation.

The second approach we examine yields more general templates, but it requires a more specialized semantics for logic programs; namely we take the view of logic programs put forth by Hagiya and others [20, 2, 10] where programs are considered not as sets of formulae constituting a Horn-clause theory, but rather as rules constituting an inductive definition. Under this view we may derive the equivalence of two relational schemata F and G by induction over the definitions of F and G. This approach is analogous to proofs of equivalence for functional schemata based on fixedpoint induction.

With respect to the third problem, organizing transformation proofs, we show how it is possible to generate transformed programs in the course of proving the correctness of the transformation itself. Our approach is based on applying proof rules which formalize transformation templates, using second-order unification (not matching, as is traditional) in the course of higher-order resolution. In our example, $F(x,y)$ is transformed to $G(x,N,y)$ by proving a theorem $\forall x, y. F(x,y) \Leftrightarrow ?G(x,N,y)$. Here F is the name of the source program to be transformed; the question mark in $?G$ indicates that $?G$ is a second-order metavariable that is instantiated into the new transformed program during the correctness proof. Application of a transformation rule yields subgoals corresponding to the program definitions and applicability conditions which are further instantiated and simplified during later resolution steps. At the end of the proof, we are left with a definition for the transformed program G and a proof of its equivalence to F.

The remainder of the paper is organized around the problems discussed above and a running example of tail-recursion introduction is used to illustrate our proposed methodology. We begin with background, in particular in §2 we provide necessary details on Isabelle, which is the basis for our implementation of this work and will be used to illustrate our development. In §3 we show how transformation templates are formalized as inference rules. In §4 and §5 we present our approaches based on datatype and relational induction for demonstrating the correctness of these rules. In §6 we show how transformation rules may be applied and transformation proofs organized in a deductive synthesis style. Finally, in §7 we discuss experience, provide historical background, and compare our work to related approaches.

2 Background

All the work presented here has been implemented in Isabelle, and some Isabelle-specific background is necessary for the examples that follow. Note that Isabelle is not the only possible platform for this work. Any implementation of higher-order logic supporting inductively defined datatypes and relations would be sufficient; however the use of a logical framework has advantages in that we can use the metalogic to directly formalize and derive *rules* of higher-order logic as opposed to just *formulae* (this distinction will be illustrated shortly). The overview to Isabelle given here is necessarily limited; [19] provides a full account.

Isabelle is an interactive theorem prover which serves as a logical framework; this means that its logic is a metalogic in which object logics (e.g., first-order logic, set theory, etc.) are encoded. Isabelle's metalogic is a minimal higher-order logic supporting polymorphic typing. Object logics are encoded in the metalogic by declaring a signature and proof rules. Afterwards, proofs are interactively constructed by applying rules using higher-order resolution. Proof construction may be automated using *tactics* which are ML programs in the tradition of LCF that construct proofs.

Rules of object logics encoded in Isabelle are formulae in Isabelle's metalogic. For example, if we have declared a type *Form* of formulae, we might then formalize the natural deduction proof rule

$$\frac{A \land B}{A}$$

as

$$A \land B \Longrightarrow A$$

where $\Longrightarrow$ is implication in Isabelle's metalogic. Note that Isabelle supports type-reconstruction, and that free variables are implicitly universally quantified; hence the above is shorthand for

$$\forall A, B : Form.\, A \land B \Longrightarrow B\,,$$

where $\forall$ is universal quantification in Isabelle's metalogic. Quantification and implication in the metalogic should not be confused with declared syntax for the object logic (such as $\wedge$). In general, a rule has the form

$$[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi \tag{14}$$

where the notation $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$ is shorthand for the iterated implication $\phi_1 \Longrightarrow \ldots \Longrightarrow (\phi_n \Longrightarrow \phi)$.

A rule can also be viewed as a proof-state, for the purposes of top-down proof construction, where ϕ is the goal to be established and the ϕ_i represent the subgoals to be proven. Under this view an initial proof state has the form $\phi \Longrightarrow \phi$, i.e., it has one subgoal, namely ϕ. The final proof state *is itself* the desired theorem ϕ.

Isabelle supports proof construction through higher-order resolution, which is roughly analogous to resolution in Prolog. That is, given a proof state with subgoal ψ and a rule like (14), then we higher-order unify ϕ with ψ. If this succeeds, then the unification yields a substitution σ and the proof state is updated by applying σ to it, replacing ψ with the subgoals $\sigma(\phi_1), \ldots, \sigma(\phi_n)$. This resolution step can be justified by a sequence of proof steps in the metalogic. Note that resolution provides a way to combine proof-rules, that is, we can derive new proof rules formally in the logic. For example, given our rule above for conjunction elimination, we could derive a rule like

$$(A \wedge B) \wedge C \Longrightarrow A\,.$$

Note too, that since unification is used to apply rules, the proof state itself may contain metavariables. We will show that this supports program transformation and synthesis during development. Finally, although rules are formalized in a natural deduction style, (14) can be read as an intuitionistic sequent where the ϕ_i are the hypotheses. Isabelle's resolution tactics apply rules in a way that maintains this illusion of working with sequents.

3 Transformation Templates as Inference Rules

We proceed in two parts. First we discuss how transformation templates for functional programs can be recast as templates for logic programs. Then we show how templates are formalized as proof rules.

3.1 From Functional to Logic Templates

A transformation template is a four-tuple consisting of:

1. a program schema (i.e., a program given in a typed higher-order language) representing the source program,
2. a program schema representing the transformed program,
3. applicability conditions, which specify requirements on relations or functions in the schemata, and
4. a correctness relation, which is the relation that holds between the original and transformed schemata.

Examples of functional and relational templates have been given in Figure 1 and Figure 2. Note that this formalization is identical to that of Huet and Lang in [15], except that they omit the correctness relation since they only consider functional programs transformation where the relation is always equality.

Functional transformation templates have been well studied and there is a large body of published examples (e.g., see [15, 17, 12]). Our interest is in transforming logic programs, so we employ relational reformulations of such functional templates. We translate functional templates by *flattening* the functional schemata to relational schemata.

Consider a functional schema of the form

$$L(x_1, \ldots x_n) = R(x_1, \ldots x_n) \tag{15}$$

where L and R represent terms built from function symbols and the x_i. We flatten such an equation to a logical equivalence by flattening each side with respect to some new variable y that represents the computed output. The term on each side of the equality corresponds to a formula involving relation symbols (including the special equality symbol =) on each side of the equivalence. If L or R is a variable, say x_i, then the flattened result is the equation $x_i = y$. Otherwise the term consists of m functions applied to variables and these are flattened to a conjunction of m relations: each n-ary function becomes an $n+1$-ary relation, representing the function's graph. Values passed between nested functions are now passed between relations by using existentially quantified variables to represent the intermediate values. This is best illustrated with an example. Consider (2) given in Figure 1. The conditional equation

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

is flattened by forming the following conditional equivalence (we retain the condition which is already a relation)

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

which is given in (9). Flattening all the equations of Figure 1 yields the translated template in Figure 2.

3.2 From Templates to Rules

A template can be recast as a rule of inference stating that the correctness condition holds under the hypotheses given by the definitions and applicability conditions. That is, if our template contains definitions $D_1, \ldots, D_m$, and applicability conditions $C_1, \ldots, C_n$, and the correctness relation G, we can express it as the rule

$$[\![D_1; \ldots; D_m; C_1 \ldots C_n]\!] \Longrightarrow G$$

In the case of both functions and relations, these templates can be formalized as rules of first-order or higher-order logic. We have chosen to work with HOL because, as we will see shortly, we will want to view logic programs as inductive definitions.

Figure 3 gives the formal rule of inference corresponding to the relational template of Figure 2, expressed in Isabelle's concrete syntax. Some explanation follows. First, the object logic we use is an encoding of HOL based on Church's that comes distributed with

```
[| ALL x y. ~B(x) --> (F(x,y) = T(x,y));
   All x y. B(x) -->  (F(x,y) =  EX x1 x2 x3.  H(x,x1) & K(x,x2) & F(x1,x3) &  R(x3,x2,y));
   ALL x y z. ~B(x) --> (G(x,y,z) = Ex x1. T(x,x1) &  R(x1,y,z));
   ALL x y z. B(x) --> (G(x,y,z) =  Ex x1 x2 x3. H(x,x1) &  K(x,x2) & R(x2,y,x3) & G(x1,x3,z))
   All x y. R(x,N,y) = (x = y);
   All x y z r. (Ex y1. R(x,y,y1) & R(y1,z,r)) = (Ex z1. R(y,z,z1) &  R(x,z1,r))
|] ==>  All x y. F(x,y) = G(x,N,y)
```

Figure 3: Rule for Tail Recursion Introduction, Relational Form

the Isabelle system and is documented in [19]. Second, quantification for first-order variables in each equivalence is made explicit; recall that all remaining free variables are outermost universally quantified in the metalogic. Finally, note that in HOL logical equivalence is written as equality, which is defined as a relation at all types; hence we use the same symbol both for logical equivalence of propositions and equality.

We now have a proof rule that represents the relational transformation template given in Figure 2. It is a sentence of the metalogic but unfortunately it is not derivable; in the following two sections we show how variants of it can be formally derived.

4 Datatype Induction

In the functional setting we can reason about the equivalence of programs or program schemata by reasoning about their meaning relative to a given semantics. For example, if recursive functions are implemented as fixedpoints of functionals, then we can reason about a function by reasoning about the limit of its approximating functions in the semantics. This semantically justifies proof rules like Scott's fixedpoint induction. We could use such an induction rule to prove the equivalence of F and G in Figure 1.

This approach is not available for reasoning about rules corresponding to our relational templates. The rule just given is not derivable in first or higher-order logic: there are counter-models. Hence we must either specialize the schemata, the semantics, or both.

4.1 Template Specialization

We consider first datatype specialization, as it is the simpler of the two kinds of specialization that we shall address. The idea is that a relational template can be specialized in two steps: first we specialize the types of schematic variables to relations over given datatypes. Second, we partially instantiate schematic variables and simplify the resulting expression. The result of this translation is a new template which can be formulated as a rule and derived by structural induction over datatypes.

Let us work through an example, specializing the template given in Figure 2 to the one in Figure 4. To begin with, we add type declarations. In this case, we consider F and G as relations (in HOL, objects of type *bool*) over polymorphic lists. We augment our template with these type declarations, leaving types of other relations implicit (when this is translated to a rule in Isabelle, Isabelle can reconstruct the types of remaining predicates and constants).

We now instantiate some of the schematic variables and simplify the resulting schemata. In particular, we specialize the schema so that F and G compute by structural recursion

$$
\begin{aligned}
&\textit{Type Declarations} && \\
F &: \alpha\ \mathit{list} \times \beta\ \mathit{list} \rightarrow \mathit{bool} && (16) \\
G &: \alpha\ \mathit{list} \times \beta\ \mathit{list} \times \beta\ \mathit{list} \rightarrow \mathit{bool} && (17) \\
&\textit{Defining Relation for } F && \\
F([], y) &\Leftrightarrow T(y) && (18) \\
F(h\#t, y) &\Leftrightarrow \exists x_1, x_2. K(h, t, x_1) \wedge F(t, x_2) \wedge R(x_2, x_1, y) && (19) \\
&\textit{Defining Equations for } G && \\
G([], y, z) &\Leftrightarrow \exists x_1. T(x_1) \wedge R(x_1, y, z) && (20) \\
G(h\#t, y, z) &\Leftrightarrow \exists x_1, x_2. K(h, t, x_1) \wedge R(x_1, y, x_2) \wedge G(t, x_2, z) && (21) \\
&\textit{Applicability Conditions:} && \\
&\quad R(x, N, y) \Leftrightarrow (x = y) && (22) \\
&\quad (\exists y_1. R(x, y, y_1) \wedge R(y_1, z, r)) \Leftrightarrow (\exists z_1. R(y, z, z_1) \wedge R(x, z_1, r)) && (23) \\
&\textit{Correctness:} && \\
&\quad F(x, y) \Leftrightarrow G(x, N, y) &&
\end{aligned}
$$

Figure 4: Transformation Rule for Tail Recursion Introduction, Specialized to Lists

over lists; hence, we let $B(x)$ be the predicate $x \neq nil$. Now, in a theory of lists where lists are either the empty list, [], or built from (infix) # denoting *cons*, we can eliminate the conditions $B(x)$ by substituting in the two possible cases. For example, the first defining equation

$$\neg B(x) \Rightarrow (F(x, y) \Leftrightarrow T(x, y))$$

becomes

$$\neg(x \neq []) \Rightarrow (F(x, y) \Leftrightarrow T(x, y))$$

which in turn is equivalent to

$$F([], y) \Leftrightarrow T([], y) \,.$$

Furthermore, schematic variables like T need only be functions of variables (since any instance may contain constants like []); hence we can further simplify to

$$F([], y) \Leftrightarrow T(y) \,,$$

which is the first defining equivalence in Figure 4.

Continuing, the second defining equivalence for F in Figure 2 is

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

Substituting a cons $h\#t$ for x and simplifying away B gives us

$$F(h\#t, y) \Leftrightarrow \exists x_1, x_2, x_3. H(h\#t, x_1) \wedge K(h\#t, x_2) \wedge F(x_1, x_3) \wedge R(x_3, x_2, y) \,.$$

Now, if F is to compute by recursion on its first argument, then $H(h\#t, x_1)$ should be the tail relation, i.e., $x_1 = t$. Making this specialization we simplify the above again to

$$F(h\#t, y) \Leftrightarrow \exists x_2, x_3. K(h\#t, x_2) \wedge F(t, x_3) \wedge R(x_3, x_2, y) \,.$$

Finally, as above where we eliminated [] from T, we can eliminate the cons constructor in K by replacing it with a ternary relation over h, t, and x_2. Making this replacement, and renaming bound variables yields

$$F(h\#t, y) \Leftrightarrow \exists x_1, x_2.K(h, t, x_1) \wedge F(t, x_2) \wedge R(x_2, x_1, y)\ ,$$

which is the second defining relation in Figure 4. Continuing the specialization in this fashion results in the new template given there.

4.2 Formalization and Derivation in Isabelle

The formalization of the above template as a rule in HOL is straightforward. We extend HOL with an inductively defined list datatype.[1] Then we form a rule from the template, as described in §3.2.

```
1) [| ALL y. F([],y) = T(y);
2)    ALL h t y. F(h#t,y) = (EX x1 x2. K(h,t,x1) & F(t,x2) & R(x2,x1,y));
3)    ALL y z. G([],y::('a list),z) = (EX x1. T(x1) & R(x1,y,z));
4)    ALL h t y z. G(h#t,y,z) = (EX x1 x2. K(h,t,x1) & R(x1,y,x2) & G(t,x2,z));
5)    ALL x y. R(x,N::('a list),y) = (x = y);
6)    ALL x y z r. (EX y1. R(x,y,y1) & R(y1,z,r)) = (EX z1. R(y,z,z1) & R(x,z1,r))
   |] ==> ALL x y. F(x,y) = G(x,N,y)
```

Note in the above that the second argument to both G and R in hypotheses numbered[2] 3 and 5 are annotated with their type; the notation '$X :: ty$' annotates X with the type ty. Remaining types are inferred automatically, and these two in-line type declarations, plus the fact that list operators have fixed types, suffice for Isabelle to reconstruct the types given in Figure 4.

This rule may now be derived in Isabelle/HOL by a simple list induction which we briefly sketch here. We prove the conclusion $F(x, y) = G(x, N, y)$ by proving instead the generalization

$$\forall y, z.(\exists v.F(x, v) \wedge R(v, y, z)) \Leftrightarrow G(x, y, z) \tag{24}$$

This generalization implies the conclusion by instantiation of y with N and simplification using hypothesis 5.

We proceed to prove the generalization by induction over x using structural induction for the list datatype.

$$\frac{x \in \tau\ list \qquad \phi([]) \qquad \begin{array}{c}[a \in \tau, \phi(x)]\\ \vdots \\ \phi(a\#x)\end{array}}{\phi(x)}$$

In the base case we must show

$$\forall y, z.(\exists v.F([], v) \wedge R(v, y, z)) \Leftrightarrow G([], y, z)$$

[1] We omit the definition of this type here; it is defined in a standard way using Isabelle's inductive datatype package, documented in [18].

[2] We have numbered the assumptions for further reference in the text; these do not appear in the actual Isabelle declaration. Other than the omission of line continuation characters, the rest of the text is taken verbatim from an Isabelle session.

Both sides simplify to $(\exists v.T(v) \wedge R(v,y,z))$ using hypothesis 1 and 3 on the left and right hand sides respectively. In the step case we must show

$$\forall y,z.(\exists v.F(a\#x,v) \wedge R(v,y,z)) \Leftrightarrow G(a\#x,y,z).$$

Expanding the definitions of F and G and rearranging terms gives

$$\begin{array}{rl} \forall y,z. & (\exists v,x_1,x_2.R(x_2,x_1,v) \wedge R(v,y,z) \wedge K(a,x,x_1) \wedge F(x,x_2)) \\ \Leftrightarrow & (\exists x_1,x_2.K(a,x,x_1) \wedge R(x_1,y,x_2) \wedge G(x,x_2,z). \end{array}$$

Using associativity (hypothesis 6) and again rearranging terms gives

$$\exists x_1,z'.(\exists x_2.F(x,x_2) \wedge R(x_2,z',z)) \wedge R(x_1,y,z') \wedge K(a,x,x_1))$$

on the left. We then use the induction hypothesis to replace $F(x,x_2) \wedge R(x_2,z',z)$ by $G(x,z',z)$. Elimination of unused quantifiers and variable renaming then yields syntactically identical right and left-hand sides.

The Isabelle proof of this theorem, constructed interactively, required eight steps and directly follows the above outline.[3]

5 Relational Induction

We now present our second approach to proving templates correct, which can be seen as specializing the theory to reflect the intended semantics. Logic programs are definite clauses (clauses with exactly one positive literal) and have a standard semantics given by the least Herbrand model [21]. This model corresponds to viewing a program as constituting an inductive definition of a set of predicates: the Herbrand model is the 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 [10, 2, 20]. Here we adopt this view and use it to establish the correctness of the kind of templates we are considering.

5.1 Inductive Definitions

To make the paper self-contained we will give a very brief review to the kind of inductive definitions we use; the general theory can be found, for example, in [1]. We view a program as consisting of rules that define a relation (i.e., a set). A trivial example is the program

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

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

[3] We omit the actual proof due to length limitations. Proof scripts for all theorems appearing in this paper are available, on request, from the authors.

An inductively defined set justifies an elimination (or induction) principle corresponding to the introduction (formation) rules. In particular, an induction principle for the above would be the following familiar structural induction principle.

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

We may view templates like that of Figure 2 as defining a set of rules, one corresponding to each relational definition. In particular, we can orient each equivalence from right-to-left, bring the condition into the body of the definition. For example, the first two equivalences correspond to the clauses[4]

$$F(x, y) \Leftarrow \neg B(x) \wedge T(x, y) \tag{25}$$

and

$$F(x, y) \Leftarrow B(x) \wedge H(x, x_1) \wedge K(x, x_2) \wedge R(x_3, x_2, y) \wedge F(x_1, x_3)\ . \tag{26}$$

Now, we can view F as being defined by the inductive definition constituted by these clauses. In this case, the corresponding induction principle is as follows.

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

Similarly, the inductive schema corresponding to the definition of $G(x, y, z)$ is the following.

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

5.2 Proving Templates by Relational Induction

To formalize the template given in Figure 2 we begin by representing F and G as inductively defined relations rather than sets of formulae. Given a relatively straightforward encoding of these inductive definition, Isabelle's support for inductive/co-inductive definitions supplies the induction principles needed for the proof (see [18] for full details). However, inductive definitions in Isabelle define *constants*, and we want to prove the rule for any F and G. Thus the problem is to somehow represent a transformation rule using such inductively-defined constants instead of metavariables. The solution is to define each inductive set in terms of parameters and to supply metavariables for these parameters when formulating the transformation rule.

In our tail-recursion example, we proceed by giving Isabelle the following parameterized inductive definitions of F and G:[5]

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

[5]We have omitted some type annotations from the Isabelle source for the sake of readability.

```
consts
 F ::  "['a  => bool, ['a, 'b] => bool, ['a, 'a] => bool, ['a, 'b] => bool,
         ['b, 'b, 'b] => bool, 'b]
          => ('a * 'b)set"

 G :: "['a  => bool, ['a, 'b] => bool, ['a, 'a] => bool, ['a, 'b] => bool,
        ['b, 'b, 'b] => bool, 'b]
           => ('a * 'b * 'b)set"

inductive "F(B,T,H,K,R,N)"
 intrs
  fBase  "[| ~ B(x); T(x,y) |] ==> <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)
          |] ==> <x,y>:F(B,T,H,K,R,N)"

inductive "G(B,T,H,K,R,N)"
 intrs
  gBase  "[| ~ B(x); T(x,x1); R(x1,y,z) |]  ==> <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)
         |] ==> <x,y,z>:G(B,T,H,K,R,N)"
end
```

The definition consists of type declarations followed by the introduction rules for the relations. Except for notational differences, the introduction rules directly formalize the clausal definitions of F and G as formulas in Isabelle's metalogic. For example, the clause `fBase` corresponds to (25); $F(x,y)$ is rendered as `<x,y>:F(B,T,H,K,R,N)` because `F` is a parameterized inductively defined set, not a predicate; `:` 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 proof rules, including induction principles for `F` and `G` corresponding to those of the previous section. These allow us to prove the following goal, which, together with the inductive definitions of `F` and `G`, represents the transformation rule of Figure 2.

```
[| ALL x. R(x,N,y) = (x = y);
   ALL x y z r . (EX y1. R(y1,z,r) & R(x,y,y1)) =  (EX z1. R(x,z1,r) & R(y,z,z1))
 |] ==> ALL x y. (<x,y>:F(B,T,H,K,R,N)) = (<x,N,y>:G(B,T,H,K,R,N))
```

Note that this representation has the advantage of avoiding confusion between logic programs, represented as inductively defined relations, and purely logical applicability conditions, represented as logical formulae.

We have used the above induction principles to justify this rule in a theory that extends HOL with the inductive definitions of F and G given above. The proof of this rule was harder than the previous one and required 20 steps in Isabelle. Rather than bore the reader with details we just give the main proof ideas. First, we require the same generalization as in the previous section in (24). Then we split the equivalence into separate proofs for each direction. To show the RHS implies the LHS; we have the assumption $G(x,y,z)$, so using the induction rule for G we prove the induction formula ϕ given by

$$\phi(x,y,z) \equiv \exists v.F(x,v) \wedge R(v,y,z)\,.$$

This follows by suitable definition expansion 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

$$\phi(x,v) \equiv \forall y,z.R(v,y,z) \rightarrow G(x,y,z)\,.$$

6 Application of Transformation Schemata

We use Isabelle both to derive templates as rules and to apply rules to transform programs: both activities become theorem proving. Template application exploits metavariables in schemata and goals to synthesize the transformed program by means of higher-order unification in the course of theorem-proving.

In this section we describe template application by continuing our running example: we show how the rule for tail-recursion developed in §4.2 can be applied for program optimization. After verification of the tail-recursion rule, Isabelle returns the following proof rule in which all undeclared constants become metavariables (denoted by names beginning with '?'):

```
[| ALL y. ?F([], y) = ?T(y);
   ALL h t y. ?F(h # t, y) = (EX x1 x2. ?K(h, t, x1) & ?F(t, x2) & ?R(x2, x1, y));
   ALL y z. ?G([], y, z) = (EX x1. ?T(x1) & ?R(x1, y, z));
   ALL h t y z. ?G(h # t, y, z) = (EX x1 x2. ?K(h, t, x1) & ?R(x1, y, x2) & ?G(t, x2, z));
   ALL x y. ?R(x, ?N, y) = (x = y);
   ALL x y z r.
       (EX y1. ?R(x, y, y1) & ?R(y1, z, r)) = (EX z1. ?R(y, z, z1) & ?R(x, z1, r)) |] ==>
 ALL x y. ?F(x, y) = ?G(x, ?N, y)
```

We will use this rule to optimize the naive reversal program *rev* given by the Prolog program:

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

This program is 'naive' because its use of *append* to place the constant H at the end of I_1 results in quadratic running time.

We begin by stating a theorem to be proved: the equivalence of the above logic program and an unknown one *?Q*. We formalize the above by translating it to list theory (which includes a definition of the append relation) and turning Horn clauses into equivalences.[6]

```
> val assums = goal thy
"[| ALL y. rev([],y) = (y = []);
    ALL h t y. rev(h#t,y) = (EX i2 i1.i2 = [h] & rev(t,i1) & append(i1,i2,y)) |]
  ==> ALL x y. rev(x,y) = ?Q(x,y)";
ALL x y. rev(x, y) = ?Q(x, y)
 1. ALL x y. rev(x, y) = ?Q(x, y)
```

Our proof steps are taken, except for slight pretty-printing, directly from an Isabelle session. The first 4 lines are typed by the user. The first states that we are beginning a new theorem and that the assumptions will be bound to the variable *assums*. The next two lines contain the assumptions, which axiomatize the above naive reverse program. This is followed by the conclusion to be proved; the metavariable *?Q* will be instantiated during proof with the transformed program. That is, metavariables in goals are interpreted existentially and the goal is satisfied if proven for some instance of *?Q*. The final two lines of transcript are Isabelle's response: the current proof state, consisting of the ultimate goal to be proved and a list of current subgoals (in this case, one subgoal identical to the ultimate goal).

[6] The use of an equivalence is necessary if we are to prove that the transformed program is equivalent to the original program (*sound* and *complete* in the terminology of logic programming), instead of merely implied by it. Moreover, we have developed our templates with this application in mind.

Our first step is to resolve with the transformation rule, which we have named *tail_opt*, at the same time specifying the name *qrev* for the new function represented by *?G* in the rule. This is the first line and Isabelle responds with six subgoals.

```
> by (res_inst_tac  [("G","qrev")] tail_opt 1);
ALL x y. rev(x, y) = qrev(x, ?N1, y)
 1. ALL y. rev([], y) = ?T1(y)
 2. ALL h t y. rev(h # t, y) = (EX x1 x2. ?K1(h, t, x1) & rev(t, x2) & ?R1(x2, x1, y))
 3. ALL y z. qrev([], y, z) = (EX x1. ?T1(x1) & ?R1(x1, y, z))
 4. ALL h t y z. qrev(h # t, y, z) = (EX x1 x2. ?K1(h, t, x1) & ?R1(x1, y, x2) & qrev(t, x2, z))
 5. ALL x y. ?R1(x, ?N1, y) = (x = y)
 6. ALL x y z r. (EX y1. ?R1(x, y, y1) & ?R1(y1, z, r)) = (EX z1. ?R1(y, z, z1) & ?R1(x, z1, r))
```

The subgoals correspond to the six premises of *tail_opt* where resolution has instantiated *?F* in the rule to *rev*. Our plan is to solve subgoals 1, 2, 5, and 6, causing Isabelle to instantiate remaining metavariables, and to discharge subgoals 3 and 4 as assumptions, which will constitute the definition of the new function *qrev*.

The first two subgoals state the definition of the program to be transformed; we can trivially solve them by resolving with the the assumptions (bound to *assums*).

```
> by (REPEAT (resolve_tac assums 1));
ALL x y. rev(x, y) = qrev(x, ?N1, y)
 1. ALL y z. qrev([], y, z) = (EX x1. x1 = [] & append(x1, y, z))
 2. ALL h t y z. qrev(h # t, y, z) = (EX x1 x2. x1 = [h] & append(x1, y, x2) & qrev(t, x2, z))
 3. ALL x y. append(x, ?N1, y) = (x = y)
 4. ALL x y z r. (EX y1. append(x, y, y1) & append(y1, z, r)) =
       (EX z1. append(y, z, z1) & append(x, z1, r))
```

This step instantiates the metavariables *?T1*, *?K1*, and *?R1*, giving a definition for the new function *qrev* (now subgoals 1 and 2).

Goals 3 and 4 follow from simple properties of the *append* relation which we have proved and named *Append_neutral* and *Append_assoc_3*. We apply them in two resolution steps combined as follows:

```
> by (REPEAT (resolve_tac [Append_neutral, Append_assoc_3] 3));
ALL x y. rev(x, y) = qrev(x, [], y)
 1. ALL y z. qrev([], y, z) = (EX x1. x1 = [] & append(x1, y, z))
 2. ALL h t y z.
       qrev(h # t, y, z) = (EX x1 x2. x1 = [h] & append(x1, y, x2) & qrev(t, x2, z))
```

Now all the metavariables are instantiated: resolution with *Append_neutral*, which states that $append(x, [], y) = (x = y)$, provides the substitution `[]` for *?N1*, and we have a complete implementation of *rev* in terms of *qrev*, represented by subgoals 1 and 2. Discharging these remaining subgoals as assumptions via Isabelle's *uresult* function, we obtain the theorem:

```
[| ALL y. ?rev([], y) = (y = []);
   ALL h t y. ?rev(h # t, y) = (EX i2 i1. i2 = [h] & ?rev(t, i1) & append(i1, i2, y));
   ALL y z. ?qrev([], y, z) = (EX x1. x1 = [] & append(x1, y, z));
   ALL h t y z.
       ?qrev(h # t, y, z) = (EX x1 x2. x1 = [h] & append(x1, y, x2) & ?qrev(t, x2, z)) |] ==>
 ALL x y. ?rev(x, y) = ?qrev(x, [], y)
```

We are done! This theorem states the equivalence of *rev* and *qrev* where *qrev* is defined by the third and fourth assumption. These definitions represent a reverse algorithm that can be run directly in programming languages such as Gödel [11], or translated 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 indeed an improvement on the original. Each call to append will execute in constant time since the first argument is either bound to 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 the transformed program now has optimal asymptotic running time, we can perform further 'constant factor' improvements by additional theorem proving, which essentially amounts to partial evaluation. In particular, the equalities in the definition of *qrev* are unnecessary, and can be removed using simplification/tautology checking tactics that come with the Isabelle system. Continuing from the previous proof state (prior to *uresult*), we can perform perform further simplifications and Isabelle returns the following simplified proof state.

```
ALL x y. rev(x, y) = qrev(x, [], y)
 1. ALL y z. qrev([], y, z) = (z = y)
 2. ALL h t y z. qrev(h # t, y, z) = (EX v. append([h], y, v) & qrev(t, v, z))
```

We might like to further simplify, for example, by reducing the base case of *qrev* simply to `ALL y. qrev([],y,y)`, however, this is not logically equivalent to (1) above. Further simplification is possible, however, if we commit to the Horn clause of *qrev* by orienting each equivalence from right to left. In that case, we can further simplify our definition to the following familiar Prolog program.

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

The above example shows how applying rules corresponding to templates can transform programs during their correctness proofs. We have also performed the same optimization using the rule derived by relational induction in §5.2. The proof is similar to the one just given and the only differences are in representation (the use of sets instead of relations) and that there were technical difficulties in using resolution to generate instances for parameters in the inductive definitions of F and G.[7]

7 Experience and Related Work

Experience and Future Work

The methodology we described here appears to be viable for formalizing and using transformation templates and adapting functional templates to logic programs. We have translated and formally derived a number of templates taken from sources such as [15, 12]; these include templates for tail-recursion introduction, finite differencing, and tupling. These templates

[7] F and G are constants defined by parameterized inductive definitions and are not given as hypotheses in the transformation rule; therefore instances of the parameters cannot be generated by resolution as in the first step of the proof above. There are ways to trick Isabelle into generating these instances by unification, but such Isabelle specific details lie outside the scope of this paper.

can be easily recast as rules in higher-order logic; indeed, the recasting is sometimes simpler then the original presentation since the use of a typed metalogic and so-called higher-order abstract syntax often allow us to naturally capture side conditions on types and variable occurrences that must otherwise be explicitly stored with the transformation template (e.g., the *where* constraints of [12]). Derivation of relational rules is more difficult than in the functional case but not overwhelmingly so. The additional difficulties seem to be twofold. First, two inductions (on F and on G) must be done instead of one over an equivalence. The second difference, perhaps more serious, is a pragmatic one: there is better support in Isabelle for using rewriting to simplify functions than relations. However, in deriving both relational and functional templates, the main difficulty is the same and concerns the creative step of finding the proper generalizations.

In many respects, our work has just begun and there are a number of directions to pursue next. It is a bit unsatisfying that, through datatype specialization, we lose data polymorphism. A more general solution is to leave the schemata polymorphic, but to use Isabelle's type classes to restrict them to types with associated well-founded relations. We could then add an additional applicability condition to the template that states that the 'descent relation', which yields a value for the recursive call (e.g., in §4 this was H, which we specialized to the tail relation), is well-founded. Such rules are provable and have more general applicability; however their drawback is that application results in new (well-foundedness) proof obligations.

Another direction to pursue is formalizing and using wider classes of templates. Our work here was limited to templates drawn from functional programming; we would like to explore other kinds of templates, in particular those better suited for multi-mode logic programming, such as given by O'Keefe in [16].

Finally, our work has raised an interesting question concerning the existence of sufficient conditions for relational templates to be provable by relational induction when the corresponding functional templates are provable by fixedpoint induction. We are currently investigating this question.

Related Work

Work on template based program transformation began in the 1970s, propelled by two independent lines of research. The first was denotational semantics as carried out by Scott and others, which laid a foundation for understanding and reasoning about programs, including the equivalence of program schemata. The second was the development of higher-order unification by Huet [14], originally developed for resolution proofs in higher-order logic, but shortly thereafter applied to program transformation. For example, Huet and Lang in [15] combine these two strands: they use Scott-style domain theory (e.g., LCF) to reason about the equivalence of program schemata formulated in transformation templates and then use second-order matching to apply the templates to transform functional programs. Since then, this idea has been further developed in other projects and systems, e.g., [3, 12, 5].

In the logic programming community, transformation has for the most part centered around approaches based on deductive transformation. This loose classification includes techniques like unfold-fold [7], LOPS [6], the work of Hogger [13], Clark [9, 8], among others. In these approaches a program is represented in first-order logic and manipulated by rules (e.g., allowing partial evaluation, folding, replacement of a formula by an equivalent or stronger formula, etc.) and the result is a program with identical meaning, relative to some semantics, but with hopefully improved runtime performance.

Template based transformation can, of course, also be seen as a kind of deductive transformation. In our work, templates are deduced and then applied as rules. The above mentioned work on logic programs centers on the explicit derivation and manipulation of concrete programs; perhaps, in the appropriate logic or metalogic, the derived equivalences could be subsequently generalized and used as transformation templates. Perhaps the research closest to ours is that of Waldau [22] who suggests an intuitionistic logic for reasoning about properties of logic programs. Like us, he is concerned with appropriate logical foundations for verifying the equivalence of transformation schemata and is motivated by their subsequent use in transformation. Our work differs in the logics we use (he uses a specialized intuitionistic logic, while we use standard higher-order logic), and in the kinds of schemata we reason about (he uses primarily schemata specialized over datatypes and reasons about them by induction over the datatype). Waldau does not discuss machine implementation or how schemata would be applied, but our approaches are compatible in the sense that his approach could be implemented in the manner we suggest here.

References

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

[2] Martin Aronsson, Lars-Henrik Eriksson, Lars Hallnäs, and Per Kreuger. A survey of GLCA: A definitional approach to logic programming. In *Extensions of Logic Programming*, LNCS-475, pages 49–100. Springer-Verlag, 1991.

[3] CIP System Group: F. L. Bauer et al. *The Munich Project CIP, Volume II: The Program Transformation System CIP-S*, volume 292 of *Lecture Notes in Computer Science*. Springer-Verlag, 1987.

[4] F.L Bauer and H. Wössner. *Algorithmic Language and Program Development.* Springer-Verlag, 1982.

[5] Yves Bertot and Ranan Fraer. Reasoning with executable specifications. In *International Joint Conference of Theory and Practice of Software Development (TAPSOFT/FASE)*, volume 915. Springer-Verlag LNCS, May 1995.

[6] W. Bibel. Syntax-directed, semantics-supported program synthesis. *Artificial Intelligence*, 14:243–261, 1980.

[7] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, 1977.

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

[9] K.L. Clark. Predicate logic as a computational formalism. Technical Report TOC 79/59, Imperial College, 1979.

[10] Masami Hagiya and Takafumi Sakurai. Foundation of logic programming based on inductive definition. *New Generation Computing*, 2:59–77, 1984.

[11] P. Hill and J. Lloyd. The Gödel Report. Technical Report TR-91-02, Department of Computer Science, University of Bristol, March 1991. Revised in September 1991.

[12] Berthold Hoffmann and Bernd Krieg-Brückner (Eds.). *Program Development by Specification and Transformation.* Springer LNCS 680, 1993.

[13] C.J. Hogger. Derivation of logic programs. *JACM*, 28(2):372–392, April 1981.

[14] Gérard Huet. A unification algorithm for typed lambda-calculus. *Theoretical Computer Science*, pages 27–57, 1975.

[15] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, pages 31–55, 1978.

[16] Richard O'Keefe. *The Craft of Prolog.* MIT Press, Cambridge, Massachusetts, 1990.

[17] Helmut A. Partsch. *Specification and Transformation of Programs.* Springer-Verlag, 1990.

[18] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In *Proc. of 12th International Conference On Automated Deduction (CADE-12)*, Nancy, France, June 1994. Springer-Verlag.

[19] Lawrence C. Paulson. *Isabelle : a generic theorem prover; with contributions by Tobias Nipkow.* LNCS-828. Springer, Berlin, 1994.

[20] Lawrence C. Paulson and Andrew W. Smith. Logic programming, functional programming, and inductive definitions. In *Extensions of Logic Programming*, LNCS-475, pages 283–310. Springer-Verlag, 1991.

[21] M.H. van Emden and R.A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23:733–42, 1976.

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