14 Formalization of the Development Process

David Basin[1] and Bernd Krieg-Brückner[2]

[1] Department of Computer Science, University of Freiburg, Freiburg, Germany
basin@informatik.uni-freiburg.de
[2] Department of Computer Science, University of Bremen, Bremen, Germany
bkb@informatik.uni-bremen.de

14.1 Introduction

14.1.1 What is development?

Software development encompasses many phases including requirements engineering, specification, design, implementation, verification or testing, and maintenance. In this chapter we concentrate on the intermediate tasks: the transition from requirements specification to verified design and design optimization, in particular, techniques for developing correct designs as opposed to ad hoc or *a posteriori* methods in which a postulated design is later verified or tested.

We view software development as the composition of correctness preserving refinement steps. Hence, a development is meaningful relative to a notion of refinement and correctness. There are many such notions (e.g., Chapter 6), each defining a type of problem transformation or reduction. For example, specification refinement from a specification SP to SP' through a refinement map ρ might be defined as correct when the images of ρ over realizations of SP' are realizations of SP. Here a refinement step represents a reduction to a hopefully simpler design objective and results in an incomplete program that expresses design decisions made during refinement. These steps can be combined into a development that constructs programs satisfying the initial specification.

Our view of development is very general and there is a strong analogy between program development in our sense and proof, as a "development" of some logical formula; both are forms of deduction. When we refer to *development* below, we often mean both. While we suggest that there is no conceptual difference, there may be good reasons in practice to differentiate between them; proof systems are often based on a fixed set of context-free and rather simple deduction rules and fixed built-in tactics, while development rules may be quite complex and context-sensitive.

Previous chapters have considered concrete formalisms for specification and deduction. Instead of focusing on a particular development methodology, we consider how methodologies could be formalized, and the advantages of formalized development. While our use of *development* is quite general, we use *formal* in a strong and restrictive sense: A development (or assertion) is

$$[] \mathbin{+\!\!+} x = x \tag{14.1}$$
$$h{:}t \mathbin{+\!\!+} x = h{:}(t \mathbin{+\!\!+} x) \tag{14.2}$$
$$x \mathbin{+\!\!+} [] = x \tag{14.3}$$
$$x \mathbin{+\!\!+} (y \mathbin{+\!\!+} z) = (x \mathbin{+\!\!+} y) \mathbin{+\!\!+} z \tag{14.4}$$

Fig. 14.1. Defining axioms and properties of $\mathbin{+\!\!+}$

formal when it is carried out (or stated) in an explicitly formalized calculus and is *informal* otherwise. Formal mathematics, as pioneered by the project of Automath [dB80], has been taken up with considerable enthusiasm by computer scientists. However, most arguments about program correctness and, in general, mathematics are informal in our strong sense. Before we consider what extra effort is required to formalize developments, let us first motivate why it might be worth it.

14.1.2 A motivating example

For motivation, we employ a simple example from functional program transformation. A well-studied formalism is unfold–fold, developed by Burstall and Darlington [BD77]. It is a good example because the formalism is simple, but yet rich enough to illustrate many of the advantages of formalized development that we present later. Moreover, it is well-known and serves as a basis for many kinds of transformational development.

Unfold–fold consists of a set of rules that are applied to sets of recursion equations producing new equations. Among the rules are: *Definition* which introduces a new, distinctly named recursion equation; *Instantiation* which introduces a substitution instance of a previous equation; *Unfold* which replaces an instance of a function name by its appropriately instantiated body; *Fold* which replaces an instance of a function body by a corresponding function call; and *Laws* which rewrites an expression using known equations about primitive functions. A refinement step is the application of a rule. A development is then a sequence of these rule applications. At the end of a development, a subset of the resulting equations is taken as the new function definitions.

Development in this setting is best illustrated through an example. Consider a theory of lists where lists are constructed with *nil* and *cons*, written as [] and infix ":" respectively. Here $[h]$ denotes the singleton list containing h, i.e., $h{:}[]$. The append function, $\mathbin{+\!\!+}$, is the infix concatenation function defined in the first two equations in Figure 14.1 by primitive recursion on its first argument.

1.	$qrev(l, m)$	$= rev(l) \mathbin{+\!\!+} m$	*definition (14.7)*
2.	$qrev([], m)$	$= rev([]) \mathbin{+\!\!+} m$	*instantiation with* []
2*a*.		$= [] \mathbin{+\!\!+} m$	*unfold with* rev, *(14.5)*
2*b*.		$= m$	*unfold with* ++, *(14.1)*
3.	$qrev(h{:}t, m)$	$= rev(h{:}t) \mathbin{+\!\!+} m$	*instantiation with* $h{:}t$
3*a*.		$= (rev(t) \mathbin{+\!\!+} [h]) \mathbin{+\!\!+} m$	*unfold with* rev, *(14.6)*
3*b*.		$= rev(t) \mathbin{+\!\!+} ([h] \mathbin{+\!\!+} m)$	*law (14.4), associativity of* ++
3*c*.		$= rev(t) \mathbin{+\!\!+} h{:}([] \mathbin{+\!\!+} m)$	*unfold with* ++, *(14.2)*
3*d*.		$= rev(t) \mathbin{+\!\!+} h{:}m$	*unfold with* ++, *(14.1)*
3*e*.		$= qrev(t, h{:}m)$	*fold with* qrev *(14.7)*

Fig. 14.2. An unfold–fold transformation of *qrev* from *rev*

Furthermore, suppose we are given the following standard, but inefficient, program for list reversal.

$$rev([]) = [] \tag{14.5}$$

$$rev(h{:}t) = rev(t) \mathbin{+\!\!+} [h] \tag{14.6}$$

Figure 14.2 gives a development of *qrev*, an accumulator-based version of reverse. The justification for each rule is given in the right-hand column. Since *rev* is defined by embedding into *qrev* (with $m = []$),

$$qrev(l, m) = rev(l) \mathbin{+\!\!+} m \tag{14.7}$$

$$qrev(l, []) = rev(l) \tag{14.8}$$

the development yields a function for list reversal that runs in linear (instead of quadratic) time.

$$qrev([], m) = m \tag{14.9}$$

$$qrev(h{:}t, m) = qrev(t, h{:}m) \tag{14.10}$$

The rules of unfold–fold comprise a simple calculus. We use it to motivate which aspects of development can be formalized and the advantages there might be in formalizing them.

Correctness. First, there is the question of correctness: what is the relationship between the initial equations and the extracted program definition? In general, unfold–fold derivations guarantee only *partially correct* programs: on those inputs where the defined function terminates, it is equal to the original definition. On the other hand, many derivations result in totally correct (always terminating) functions. For example, *qrev* defines a total function.[1]

[1] Evaluated in the left-to-right direction, the defining rewrite rules terminate on any given pair of inputs and compute a unique output. On the other hand, a

However, within unfold–fold itself we cannot express the relationship (neither partial nor total correctness) between the original and the synthesized program.

An alternative to this informal notion of correctness is to formalize unfold–fold within a metalogic capable of expressing program equivalence and establishing sufficient conditions for proofs to yield correct programs. Such analysis has been done for unfold–fold: [Kot78,San95] present restrictions on unfold–fold derivations that guarantee total correctness. However the analysis and restrictions are not formalized. This is in contrast to, for example, the use of a programming logic like LCF [GMW79,Pau87] where a model of computation may be specified, and properties such as equality and termination of programs may be expressed and reasoned about, relative to a desired computational model. LCF is a particularly good example of such a framework. It was developed to experiment with Scott's Logic of Computable Functions and is based on a first-order logic for domain theory. This logic has been used, for example, to verify algebraic laws of functional programs [Les81], correctness of program schemes for recursion removal [Coh80], and relationships between denotational and Hoare-style axiomatic semantics [Sok87]. Recently, various alternatives have been suggested to verify the correctness of development rules or nontrivial transformation rules, using formal frameworks such as Isabelle or LEGO [AB95,AB96,Bas94,WK95].

The major advantage of formalizing development rules and formally proving that they constitute correct refinement steps is a separation of concerns and the resulting economy of verification effort. Rules can be verified once, as correctness-preserving, perhaps under some schematic applicability conditions. When applied, any applicability conditions are instantiated as verification conditions for the particular application. While considerable effort may have to be invested in the initial correctness proof, the proof of the applicability of the rule, if any, usually requires much less effort than a formal *a posteriori* proof for a development step.

By embedding and developing a formalism like unfold–fold within a metalogic like LCF, we would be able to state and verify formally that developed designs are partially or totally correct. Even more is possible. Rather than just considering which designs are total, we could formalize classes of development that guarantee total (as opposed to partial) correctness. The work of Kott, and more recently Sands [Kot78,San95], is relevant for our example. Both have examined sufficient conditions for restricted unfold–fold derivations to be totally correct. If we imagine a formalization of unfold–fold where derivations are formalized by, say, a recursive data type of proofs, then if the application conditions of rules in the restricted calculus are recursively

simple (and well-known) example of a partially, but not totally, correct derivation is to start with the program definition $f(x) = 42$ and observe that for any instance, say 7, $f(7) = 42$. Using this, and the folding rule, allows us to derive the diverging program $f(x) = f(7)$.

Defining equations for f (left-hand side of the transformation rule)

$$A(x) \Rightarrow f(x) = B(x) \tag{14.11}$$

$$\neg A(x) \Rightarrow f(x) = K(f(H(x)), J(x)) \tag{14.12}$$

Defining equations for f' (right-hand side of the transformation rule)

$$f'(x) = g'(x, N) \tag{14.13}$$

$$A(x) \Rightarrow g'(x, y) = K(B(x), y) \tag{14.14}$$

$$\neg A(x) \Rightarrow g'(x, y) = g'(H(x), K(J(x), y)) \tag{14.15}$$

Applicability conditions:

$$K(x, N) = x \tag{14.16}$$

$$K(x, K(y, z)) = K(K(x, y), z) \tag{14.17}$$

Correctness:

$$f(x) = f'(x) \tag{14.18}$$

Fig. 14.3. Transformation rule for tail recursion introduction, presented as a program schema

checkable, then it should be possible to define a subtype of derivations based on development rules that only produce totally correct designs.

Rule extensibility. We have indicated how restricting developments may enable correctness proofs. Alternatively, a formalization can provide a basis for *extending* refinement in a correctness-preserving manner. Extensibility is important for scalability. The rules of unfold–fold, for example, are appropriate for constructing small derivations but inappropriate for transforming large programs because development steps are too small. This problem is common since designers of logics usually aim for minimality at the cost of usability. Hence large developments often require some kind of user extensibility.

One of the simplest kinds of extensibility is to admit new rule schemata that preserve correct refinement. For example, one popular approach to program transformation is based on rule schemata (also called transformation templates) that specify the form of programs (or program fragments) before and after transformation [HL78]. In particular, a *transformation rule* is specified by providing a pair of program schemata along with additional applicability conditions that must be satisfied. The schemata are applied using higher-order matching or unification and the conditions are checked on the generated instance. This may lead to more succinct derivations than are possible using only deductive transformation methods with fixed sets of rules, such as the unfold–fold rules.

Consider Figure 14.3, which presents a transformation rule taken from [HK93]. Under the appropriate semantic interpretation, we could formulate and prove the equality of the definitions for f and f' for all instances (for the schematic variables) that satisfy the stated applicability conditions. This approach was taken in the 1970s, for example, by Huet and Lang [HL78] who investigated the correctness of similar schemata relative to a denotational semantics within LCF. Once formalized, this schema may be applied using second-order matching to transform instances of the schema for f to instances of f'.

Returning to the *rev* example, let us give an alternative formalization of *rev* using conditional equations.

$$x = [] \Rightarrow rev(x) = [] \tag{14.19}$$
$$\neg(x = []) \Rightarrow rev(x) = rev(tl(x)) +\!\!+ [hd(x)] \tag{14.20}$$

Now, (14.19) second-order matches (14.11) and (14.20) matches (14.12) under the substitutions $f(x) \equiv rev(x)$, $A(x) \equiv x = []$, $B(x) \equiv []$, $K(x,y) \equiv x +\!\!+ y$, $H(x) \equiv tl(x)$, and $J(x) \equiv [hd(x)]$. Note that the variable N takes no part in matching (in Section 14.2.3 we show how it can be computed by applying proof rules using unification; in Section 14.3.4, N is picked up from the context), but augmenting the substitution with $N \equiv []$ and applying it to (14.13)–(14.15) yields the following program.

$$f'(x) = g'(x, []) \tag{14.21}$$
$$x = [] \Rightarrow g'(x,y) = [] +\!\!+ y \tag{14.22}$$
$$\neg(x = []) \Rightarrow g'(x,y) = g'(tl(x), [hd(x)] +\!\!+ y) \tag{14.23}$$

The programs f and f' are equivalent as the applicability conditions are valid: $[]$ is a right-identity of $+\!\!+$ and $+\!\!+$ is associative (see (14.3) and (14.4)); in other words, $+\!\!+$ and $[]$ form a monoid. The resulting program for g' can be further simplified by observing (as in Figure 14.2) that $[hd(x)] +\!\!+ y = hd(x){:}y$ (see (14.2) and (14.1)) and when $x = []$ that $x +\!\!+ y = y$ (see (14.1)). Performing this simplification yields

$$x = [] \Rightarrow g'(x,y) = y \tag{14.24}$$
$$\neg(x = []) \Rightarrow g'(x,y) = g'(tl(x), hd(x){:}y) \tag{14.25}$$

which is a trivial reformulation of *qrev* as given in (14.9) and (14.10).

This example indicates that if we formalize enough metatheory, then new development rules may be proved correct and used in subsequent developments. In this example, "enough metatheory" means the appropriate semantics of recursion equations. Perhaps surprisingly, one can often get by with much less. In Section 14.2.3 we show that it is possible to derive a specialized form of this schema, without any formalized semantics, directly within a first-order theory of lists. There we formally develop both the program schema and the transformation of *qrev* from *rev*.

Tactic extensibility. Deriving new schematic rules is a powerful way of extending refinement and hence development. Often though, refinement strategies for constructing developments cannot be captured by rules. A strategy may involve iteration, case analysis, and indeed, arbitrary computation. Returning to our unfold–fold example, Feather, for instance, has looked at strategies defining common patterns of reasoning in unfold–fold and defined a class of programs, called design tactics, that implement them [Fea82]. In his work each tactic implements a new kind of transformation step that performs a specific and useful task; e.g., the *combine* tactic replaces nested function calls with a call to a new function definition that can subsequently be optimized.

This idea of using tactics to raise the level of refinement in development goes back to the LCF project. Tactics are programs that, given a specification (goal) SP, reduce it to subspecifications $SP_1, \ldots, SP_n$; that is, a tactic is simply a user-programmable rule. The correctness of tactics is guaranteed in an after-the-fact fashion: each tactic has an associated *validation*, which is a program that constructs a solution to SP from solutions to the SP_i, using only primitive rules of the underlying logic.

Correctness based on validation functions makes tactic construction sometimes unnatural and their execution slow. Consider, for example, proving two terms equal that contain some associative commutative operator such as $+$, e.g., $a + ((b + c) + d)$ and $(c + a) + (d + b)$. A formal proof might consist of applying axioms for associativity, commutativity, and congruence. Alternatively we can view terms as trees and test equality by checking if their fringes are permutations of each other. This could be done by sorting both fringes and checking that the results are identical, i.e., the lists $[a, b, c, d]$. A tactic must do the former; but often we want development rules like the latter. Their construction may be simpler and their execution faster.

If our development formalism is strong enough to express and reason about programs and their properties, then an alternative to the tactic paradigm is possible. Rather than having tactics construct validations with each application, a tactic could be verified correct once and for all. After all, the problem of developing trusted tactics (that do not require validations) is a special case of the general problem of developing trusted software. Returning to the above example we could verify that the "fringe sorting" program is a decision procedure for the appropriate theory. Once verified, it could then be executed without producing a primitive justification because the program has been formally proven correct.

The unfold–fold formalism is not strong enough to show that functional programs are equivalent to arbitrary, not necessarily executable, specifications. We will consider more powerful approaches in Section 14.3. However, it could be used for tactic optimization. Namely, if a tactic is a functional program represented as recursion equations, then unfold–fold could be used to develop a more efficient version of it. This kind of optimization will be discussed in Section 14.4.

14.1.3 Development rules, refinement

The discussion above may have given the reader the impression that formalizing development is difficult and time consuming. It need not be! The size of the task depends on what is to be formalized. It may be easy (e.g., hours or days as opposed to months) to derive new development rules within a suitable framework. Alternatively, more work may be required to formalize nonschematic rules computed by arbitrary functions, or to provide semantic justifications for new rules. Overall, the amount of effort depends on the kinds of rules employed and how their correctness is to be established.

Logicians have looked at the general question "What constitutes a rule?", and the answer is surprisingly complex (e.g., [Avr91,FHV92]). For our purpose, development means problem reduction and the nature of a development rule depends on the syntactic entities we reason about, and the semantic notion of refinement involved. This is an open ended concept; we can imagine heterogeneous kinds of developments where refinement rules are parameterized by data types, models, executable programs, and assertions or *judgements* [Mar82,Mar85] between objects of these syntactic categories. The following rule, taken from [SST92], is not untypical.

$$\frac{SP : \mathbf{Spec}(\Sigma) \quad \Phi' \subseteq Sen(\Sigma') \quad \sigma : \Sigma \to \Sigma'}{\mathbf{abstract}\ SP\ \mathbf{wrt}\ \Phi'\ \mathbf{via}\ \sigma : \mathbf{Spec}(\Sigma)} \tag{14.26}$$

It is impressive for sheer syntax alone as it combines algebraic signatures (e.g., Σ), signature morphisms (σ), sets of sentences, judgements about set inclusion, and judgements about the satisfaction relation between objects and specifications.

Different kinds of rules require different kinds of metatheories to formalize and reason about them. In this chapter, we will consider two paradigms:

1. a rule is a *partial function*; the type of its domain and range is governed by the judgements of the theory, and
2. a rule is a *relation*; it is a (usually binary) relation between the terms (formulas, proofs, etc.) of the theory.

The first view is the traditional one in logic; it leads to a tree-like structure of developments, i.e., proofs seen as applications of proof rules. The second view is the predominant one in term rewriting, the λ-calculus, rewrite logics, program transformation, and, last but not least, the internal view in many theorem provers.

For example, consider the simple setting of first-order logic. Let $\mathcal{F}$ denote the set of first-order formulas and let pr be a judgement-valued function, where $pr(A)$ represents the judgement that $A \in \mathcal{F}$ is provable. Then, following [HS86,Tro82], a rule is a partial function ϕ from n-tuples of judgements ($n \geq 0$) to judgements. The n-tuple $\langle pr(A_1), \ldots, pr(A_n)\rangle$ in the domain of ϕ contains the *premises*, and if $\phi(pr(A_1), \ldots, pr(A_n)) = pr(B)$, then $pr(B)$ is the corresponding *conclusion* and the expression

$$\frac{pr(A_1), \ldots, pr(A_n)}{pr(B)}$$

is an instance of the rule ϕ.

In the second view, similar to that of rewriting, a rule is specified by a pair written as $l \to r$. For a given context C, two terms t and s are *in the rewriting relation at* C, written $t \to_C s$, iff there is a substitution σ such that $t = C[\sigma(l)]$ and $s = C[\sigma(r)]$. Two terms are *in the rewriting relation* iff there is a context C such that they are in the rewriting relation at C. For example, we can convert the set of equations formed by (14.5) and (14.6) into a typical first-order rewrite system composed of the two rules:

$$rev([]) \to [] \tag{14.27}$$

$$rev(h{:}t) \to rev(t) +\!\!+ [h] \tag{14.28}$$

In Section 14.4.1 we will return to this equational/rewriting view of developments.

Sometimes rules, either expressed as functions or relations, involve applicability conditions that must be checked before the rule can be applied; Figure 14.3 contains one such example, namely showing that K is associative with right-identity N. Such preconditions can often be checked automatically (see Section 14.3.4). Rules may also be applied where matching or unification builds in equational or higher-order theories (see Section 14.3.3).

Taking the view of rules as functions or relations, a metalogic that supports the formalization of rules and the derivation of new rules must be a logic capable of formalizing and reasoning about these entities. In practice, however, many rules are of a particularly simple form where they are expressible as *schematic rules* given by (first or higher-order) rule schemata. As we will see in the following section, the standard proof rules of first-order logic can be so formulated. For example, as a natural deduction rule we might formulate $\wedge$-I (conjunction introduction) by the rule schema

$$\frac{pr(\phi) \qquad pr(\psi)}{pr(\phi \wedge \psi)} \tag{14.29}$$

where ϕ and ψ are not concrete formulas of the logic, but rather variables in a metalanguage, which range over the formulas in $\mathcal{F}$. These variables are often called *metavariables* or *schematic variables.* A more complicated example might be a rule expressing program equivalence. We can formalize (14.11)–(14.18) as the single schematic rule

$$\frac{pr(A_1) \quad pr(A_2) \quad \ldots \quad pr(A_7)}{pr(\forall x. f(x) = f'(x))} \tag{14.30}$$

where the A_i are the universal closure of the equations (14.11)–(14.17).

14.1.4 Composition

In general, we identify an elementary development step with the application of a development rule. A compound development step corresponds to the application of a series of rules, possibly by a tactic. To illustrate this, let us continue working in the first-order setting above. According to the partial function view of rules, (14.29) corresponds to the function

$$andIntro : \{x : \mathcal{F} | pr(x)\} \to \{y : \mathcal{F} | pr(y)\} \to \{z : \mathcal{F} | pr(z)\}$$

with

$$andIntro = \lambda x.\lambda y.x \wedge y\,.$$

As a consequence, function composition gives meaning to the usual composition of proof trees:

$$\dfrac{pr(A) \qquad \dfrac{pr(B) \qquad pr(C)}{pr(B \wedge C)}}{pr(A \wedge (B \wedge C))}$$

Moreover, when extending to the higher-order case, this functional view (which is extensively used in constructive type theory, e.g., [CH88,Mar82]), gives a precise meaning to rules involving assumptions and their discharge, for example, *modus ponens*:

$$\dfrac{pr(A) \qquad \begin{array}{c}[pr(A)]\\ \vdots \\ pr(B)\end{array}}{pr(B)}$$

which can be represented as the function *modusPonens* of type

$$\{x : \mathcal{F} | pr(x)\} \to (\{y : \mathcal{F} | pr(y)\} \to \{z : \mathcal{F} | pr(z)\}) \to \{r : \mathcal{F} | pr(r)\})\,,$$

which is defined by

$$modusPonens = \lambda x.\lambda F.F(x)\,.$$

According to the relational view, relational composition is the analog of functional composition and a development is a linear sequence of the form

$$t_1 \to_{C_1} \ldots \to_{C_{n-1}} t_n\,.$$

An alternative representation, which makes explicit use of the relational composition operator $\circ$, is

$$t_1 \;(\to_{C_1} \circ \cdots \circ \to_{C_{n-1}})\; t_n\,.$$

This representation makes explicit that under the relational view, a development is a simple kind of compound object.

There is often a good deal of flexibility as to whether developments are formulated functionally or relationally, with the different formalizations emphasizing different views or aspects of development. In Section 14.2.3 we will give a functional formulation of the *qrev* development based on a functional formalization of the transformation schema given in Figure 14.3. Under this view, unfold–fold-style development can be seen as constructing equality proofs using derived rules. Alternatively, Figure 14.2 suggests a relational formalization where each development step relates sets of equations and corresponds to an instantiation step, or the application of a single equation, interpreted as a rewrite rule from left to right as an unfold step, from right to left as a fold step, or either way as a law. In each transformation step, the focus of applicability of a rewrite rule is adjusted appropriately, e.g., by a matching process.

14.1.5 Organization of this chapter

The remainder of this chapter is organized as follows. In Section 14.2 we present how simple schematic rules can be formalized using first-order and higher-order syntax and be formally derived. To make this concrete, we work through a particular example in the ISABELLE system, first deriving a development rule and then using it for program transformation. This motivates a general discussion about the relationship between developments for verification, synthesis, and transformation. In Section 14.3 we consider development rules that cannot be directly expressed as simple schemata. The problem then becomes one of formalizing general functions or relations. We also consider how schematic rules can be generalized, for example, by building in equational theories. In Section 14.4 we consider how developments can be formalized and how formalization supports their verification and optimization. Finally, in Section 14.5, we draw conclusions.

14.2 Simple schematic development rules

14.2.1 Motivation

As indicated above, many rules can be expressed schematically; here we consider what this entails. We address both metatheoretic extensibility, where users may derive new schematic rules, and how such encodings support the use of a metalogic as a "logical framework". We also consider the possibility of *schematic developments* where not only are rules schematic, but developments themselves may contain metavariables. This allows certain kinds of program synthesis to be integrated into the development process.

Let us first consider what kind of metatheory is needed to support schematic rules and developments. Many different logics have been proposed as metalogics for formalizing rules and derivations. These include equational logic [GSHH92], rewrite logic [MM93], first-order logic [GT93], type theory [HHP93], and higher-order logic [Pau94b], among others. Where possible, we will avoid a commitment to a particular metalogic although we will indicate minimal requirements when necessary.

14.2.2 Formalization and supporting metatheory

Syntax. The idea of encoding the syntax of one language in another goes back at least to Gödel [Göd31] who proposed representing formulas of first-order logic as constants in a theory of arithmetic and Tarski [Tar36] who suggested encodings based on "structural descriptive names" where each formula is associated with a term that mimics its structure. For example, under Tarski's approach one might encode the formula $A \vee (B \wedge C)$ as the term *or(A,and(B,C))*. The importance of this approach has been highlighted by the development of logical frameworks on computers where users encode logics within a given metalogic. Most approaches in current use are refinements of Tarski's idea, usually couched within a typed logic.

The basic idea is simple. Given a specific object logic, for each of its syntactic categories we declare a type in the metalogic. Further, for each operator (i.e., connective or quantifier) in the object logic we declare a constant in the metalogic whose type reflects the syntactic categories of its arguments and the result. The following, for example, is a fragment of the syntax of a first-order theory of lists with equality over a given data type such as the naturals *Nat*.[2]

$$
\begin{array}{llll}
\textbf{datatype}\ \mathcal{T} = & nil : \mathcal{T} & \qquad \textbf{datatype}\ \mathcal{F} = & eq : \mathcal{T} \times \mathcal{T} \to \mathcal{F} \\
& \mid\ cons : Nat \times \mathcal{T} \to \mathcal{T} & & \mid\ not : \mathcal{F} \to \mathcal{F} \\
& \mid\ append : \mathcal{T} \times \mathcal{T} \to \mathcal{T} & & \mid\ and : \mathcal{F} \times \mathcal{F} \to \mathcal{F} \\
& \vdots & & \vdots
\end{array}
$$

Here we are using a simple data type syntax to declare that elements in $\mathcal{T}$ are lists of natural numbers built from constructors *nil* and *cons* and functions like *append* (there may be others, of course), and formulas in $\mathcal{F}$ are built from the standard constructors. Of course, different logics will have different syntax for declaring types and constants over them. As an example, $not(eq(nil, cons(0, nil)))$ is an element of $\mathcal{F}$. To simplify presentation we will use the standard syntax (used in the introduction) to represent these constructors, e.g., $\neg([] = 0{:}[])$ or even $[] \neq [0]$. It should always be clear from the context whether we are referring to object or metalevel syntax.

[2] We will ignore the issue of parameterization here. A logical framework like Isabelle supports parameterized types through polymorphic typing of formulas in the metatheory.

$$
\begin{array}{ll}
\wedge\text{-}I & \forall \phi\, \psi : \mathcal{F}.\, [\![pr(\phi); pr(\psi)]\!] \Longrightarrow pr(\phi \wedge \psi) \\
\wedge\text{-}El & \forall \phi\, \psi : \mathcal{F}.\, pr(\phi \wedge \psi) \Longrightarrow pr(\phi) \\
\wedge\text{-}Er & \forall \phi\, \psi : \mathcal{F}.\, pr(\phi \wedge \psi) \Longrightarrow pr(\psi) \\
\rightarrow\text{-}I & \forall \phi\, \psi : \mathcal{F}.\, (pr(\phi) \Longrightarrow pr(\psi)) \Longrightarrow pr(\phi \Rightarrow \psi) \\
\rightarrow\text{-}E & \forall \phi\, \psi : \mathcal{F}.\, [\![pr(\phi \Rightarrow \psi); pr(\phi)]\!] \Longrightarrow pr(\psi)
\end{array}
$$

Fig. 14.4. Typical proof rules for FOL

Rules. Proof rules, which are to be represented in the metatheory, may be organized around the concept of a judgement or assertion [Mar85], where a logic dictates a notion of proof (or, more generally, development) that justifies the judgements. Since judgements are assertions over elements of the logic's syntactic categories, we may formalize them as appropriately typed propositions in the metalogic. In first-order logic, for example, there is only one kind of judgement, provability, and we might write $pr(\phi)$ to indicate that the formula ϕ is provable. Martin-Löf has four judgements in his type theory (typehood, equality of types, membership in a type, and equalities of members) [Mar82]. If we were reasoning about Hoare logic then our judgement might represent validity of a triple of arguments, e.g., $\{S\}\, P\, \{Q\}$ where S and Q are first-order formulas and P a "while-loop" program.

In the terminology of [HHP93], proof rules represent judgements, which are *hypothetical* and *schematic*. If our metalogic contains implication[3], $\Longrightarrow$, then if J_1 and J_2 are judgements, the hypothetical judgement $J_1 \Longrightarrow J_2$ expresses consequence or logical entailment: J_2 is provable under the assumption of J_1. Alternatively, if C is a type representing a syntactic category, x a variable ranging over C, and $J(x)$ a judgement, then if the metalanguage contains universal quantification, $\forall$, $\forall x : C.\, J(x)$ is a schematic judgement expressing generality in rules and axioms, that is, $J(a)$ for all $a \in C$.

Given this view, it is straightforward to translate most rules into axioms in the metalogic by expressing them as hypothetical and schematic judgements. For example, the $\wedge$-I rule given by

$$\frac{pr(\phi) \qquad pr(\psi)}{pr(\phi \wedge \psi)}\,, \tag{14.31}$$

states that $pr(\phi \wedge \psi)$ follows from $pr(\phi)$ and $pr(\psi)$. Moreover, it is general because ϕ and ψ range over formulas. Hence we may formalize the rule as the first rule given in Figure 14.4. Other rules are axiomatized analogously.

We use $\boldsymbol{\forall}$ and $\Longrightarrow$ to represent universal quantification and implication in the metalogic. These operators (in **boldface font**) should not be confused with $\forall$ and $\Rightarrow$ in the declared object logic. Further, to simplify syntax, we write $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$ for $(\phi_1 \Longrightarrow (\ldots \Longrightarrow (\phi_n \Longrightarrow \phi) \ldots))$ when $n > 1$.

[3] This is not the only possibility, e.g., [GSHH92] encodes rules as equations in OBJ3.

Given a collection of axiomatized rules, new rules may be derived by deduction in the metatheory. For example, using the axioms in Figure 14.4 we might derive the new formula

$$\forall \phi\, \psi\, \gamma : \mathcal{F}.\, [\![pr(\phi); pr(\psi); pr(\gamma)]\!] \Longrightarrow pr(\phi \wedge (\psi \wedge \gamma)) \,, \tag{14.32}$$

which corresponds to adding

$$\frac{pr(\phi) \qquad pr(\psi) \qquad pr(\gamma)}{pr(\phi \wedge (\psi \wedge \gamma))} \tag{14.33}$$

to the set of rules. Its addition preserves correctness (in first-order logic, validity) since it is a derived rule.

Explicit axiomatizations of variables and substitution. So far we have not committed ourselves to a particular metalogic. The encoding above could be carried out within a sorted metatheory based on conditional equations or, at the other extreme, higher-order logic. However the alert reader will have noticed that we have ignored variables and quantifiers. In the past, two main approaches to their formalization have been pursued: explicit encoding and higher-order abstract syntax. Each has its advantages and disadvantages.

Explicitly axiomatizing operations on variables has the advantage that it is possible within weak metalogics (e.g., equational logic). Moreover, it makes syntactic properties explicit and this enables formal metareasoning about syntax itself. Let us reconsider our list theory example. To begin with, we augment the term and formula data types given on page 12. First, we add a new type $\mathcal{V}$, corresponding to the syntactic category of variables. Second, we extend the data type definition $\mathcal{T}$ by adding a new term constructor $var : \mathcal{V} \to \mathcal{T}$. And finally, we extend $\mathcal{F}$ with quantifiers $\forall$ and $\exists$, each of type $\mathcal{V} \times \mathcal{F} \to \mathcal{F}$.

With our extended syntax, we can formalize proof rules for quantifiers, for example, $\forall$-E,

$$\frac{pr(\forall x.\, \phi)}{pr(\phi[t/x])} \,. \tag{14.34}$$

This rule relates a judgement about ϕ to a substitution instance $\psi = \phi[t/x]$. To express this we must formalize substitution. This may be axiomatized as a ternary predicate, *subst*, in the metatheory over formulas, terms, and variables, where $subst(\phi, t, x) = \psi$ is provable precisely when $\phi[t/x] = \psi$. With this addition, $\forall$-E is axiomatizable as

$$\forall \phi, \psi : \mathcal{F}\, .\forall t : \mathcal{T}.\, \forall x : \mathcal{V}\, .[\![equal(subst(\phi, t, x), \psi); pr(\forall(x, \phi))]\!] \Longrightarrow pr(\psi) \,. \tag{14.35}$$

Here we have used two judgements. The new judgement *equal* over pairs of formulas is true when the two are provably equal. Note that we cannot simply

$$\begin{array}{llll}
FV_{\mathcal{T}}(var(x)) &= \{x\} & FV_{\mathcal{F}}(s = t) &= FV_{\mathcal{T}}(s) \cup FV_{\mathcal{T}}(t) \\
FV_{\mathcal{T}}([]) &= \{\} & FV_{\mathcal{F}}(\neg(p)) &= FV_{\mathcal{F}}(p) \\
FV_{\mathcal{T}}(h{:}t) &= FV_{\mathcal{T}}(h) \cup FV_{\mathcal{T}}(t) & FV_{\mathcal{F}}(\wedge(p,q)) &= FV_{\mathcal{F}}(p) \cup FV_{\mathcal{F}}(q) \\
FV_{\mathcal{T}}(s \mathbin{++} t) &= FV_{\mathcal{T}}(s) \cup FV_{\mathcal{T}}(t) & FV_{\mathcal{F}}(\forall(x,p)) &= FV_{\mathcal{F}}(p) - \{x\} \\
& \vdots & \vdots &
\end{array}$$

Fig. 14.5. Axiomatization of free term and formula variables

restate this as $pr(subst(\phi, t, x) = \psi)$ using the equality term constructor for the type $\mathcal{F}$ since its arguments must be members of $\mathcal{T}$.

There are a variety of approaches to explicitly formalizing theories of substitution and binding. The most direct approach is to take a standard textbook account of basic concepts, such as free and bound variables, capture, and equivalence under bound variable renaming, and translate them into an equational setting. As these may be defined by structural recursion on terms and formulas, the definitions are more or less straightforward. Figure 14.5 gives a flavor of this by equationally defining functions that, given terms ($FV_{\mathcal{T}}$) or formulas ($FV_{\mathcal{F}}$), return the set of their free variables. We have assumed a prior axiomatization of sets where $\{\}$ represents the empty set, $\{x\}$ represents the singleton set with element x, and, if A and B are sets, then $A \cup B$ is their union and $A - B$ their difference. In [MM93], Martí-Oliet and Meseguer have given equational encodings of several logics involving binding operators based on direct encodings of this form.

Less direct kinds of encoding have also been explored. One may change the representation of terms to avoid bound variables entirely. This was originally suggested by de Bruijn [dB72]: his idea was to avoid explicitly named binding variables and, instead, replace variables by a kind of pointer to the operator that binds them. This idea has been adopted, for example, in the work of Shankar [Sha88]. Another approach has been proposed by Talcott [Tal93], who has axiomatized a general theory of contexts (terms with "holes"), binding structure, and substitutions; any logic requiring these concepts should be able to import and build directly upon this theory. Another alternative, which is gaining some popularity, is based on extensions of the lambda calculus that formalize operators for explicit substitutions [ACCL91].

Explicit axiomatization has the advantage that one can state and prove theorems that depend on properties of variables, e.g., their names or whether their occurrences are free or bound. This is useful, for example, in formalizing transformation rules since we may wish to state explicit conditions on variables occurring in terms and their number of occurrences. For example, we may wish to make sure in the transformation rule of Figure 14.3, that f is linearly recursive in the source schema, i.e., does not occur in the schema variables A, B, K, H, or J. Another example would be the transformation rule for variable abstraction ("common subexpression elimination" [BW82, HK93]); a simplified version might be:

$K(e) \rightarrow let\ x = e\ in\ K(x)$

Here, we might want to insist that e occurs more than once in K.

Alternatively, there are decision procedures for formulas in restricted syntactic classes and these procedures must analyze variables and their occurrences. A simple example from propositional logic, due to Łukasiewicz, is that a propositional formula, built only from the equivalence connective ($\Leftrightarrow$), is provable if and only if every propositional variable in it occurs an even number of times. To formalize such a theorem requires a representation of syntax where we can reason about the names of propositional variables and their occurrences. This would not be possible under the approach we consider next.

Higher-order abstract syntax. An alternative to an explicit axiomatic theory of binding is the use of "higher-order abstract syntax" (henceforth HOAS), which was inspired by Church [Chu40] and Martin-Löf's system of arities [NPS90]. This formalization paradigm is supported by higher-order metalogics whose type system is at least as powerful as the simply typed λ-calculus, e.g., higher-order logic [Chu40], λ-Prolog [Mil90], and type theories like the Edinburgh LF [HHP93], Pfenning's related ELF system [Pfe91], the Calculus of Constructions [CH88], and NUPRL [C+86]. In this approach, variables in the metalogic are used to represent variables in the object logic. Moreover, as the metalogic is higher-order, operators that bind variables (e.g., $\forall$, $\exists$, λ) can be represented using constants whose domain is of function type. In this way, all binding in the object logic is reduced to binding in the λ-calculus and substitution is denoted by application and performed by β-reduction. This approach is very popular among designers of logical frameworks [HHP93,Pau94b]; its advantage is that, once α-conversion, substitution, and the like are implemented within the logical framework for the λ-calculus, these operations can be "inherited" in declared theories whose binding operators behave similarly. Hence there is a significant saving of effort over explicit encoding because a theory of binding and substitution has, operationally, already been implemented.

HOAS is best illustrated by returning to our list theory example: the $\mathcal{T}$ and $\mathcal{F}$ data types defined on page 12. The operators $\forall$ and $\exists$ are added to $\mathcal{F}$ by adding the new constructors $\forall : (\mathcal{T} \rightarrow \mathcal{F}) \rightarrow \mathcal{F}$ and $\exists : (\mathcal{T} \rightarrow \mathcal{F}) \rightarrow \mathcal{F}$. Binding is handled by representing the operators as higher-order functions with argument types $\mathcal{T} \rightarrow \mathcal{F}$. For example, the formula $\forall x.\, \exists y.\, x = y$ is encoded as $\forall(\lambda x : \mathcal{T}.\, \exists(\lambda y : \mathcal{T}.\, x = y))$. Observe that if our metalogic identifies α-convertible terms (in the λ-calculus), then the above term is equal, for example, to $\forall(\lambda z : \mathcal{T}.\, \exists(\lambda x : \mathcal{T}.\, z = x))$. This was not the case for our previously given formula type with explicitly named variables. There we would have had to define predicates on terms and formulas that axiomatize equality

$$
\begin{array}{ll}
\forall\text{-}E & \forall P : \mathcal{T} \to \mathcal{F}.\, \forall t : \mathcal{T}.\, pr(\forall x : \mathcal{T}.\, P(x)) \Longrightarrow pr(P(t)) \\
\forall\text{-}I & \forall P : \mathcal{T} \to \mathcal{F}.\, (\forall x : \mathcal{T}.pr(P(x))) \Longrightarrow pr(\forall x : \mathcal{T}.\, P(x)) \\
sub & \forall P : \mathcal{T} \to \mathcal{F}.\, \forall x\, y : \mathcal{T}.\, [\![pr(x = y); pr(P(x))]\!] \Longrightarrow pr(P(y)) \\
ind & \forall \phi : \mathcal{T} \to \mathcal{F}.\, [\![pr(\phi([])); \forall h : Nat.\, \forall x : \mathcal{T}.\, pr(\phi(x)) \Longrightarrow pr(\phi(h{:}x))]\!] \\
 & \qquad \Longrightarrow pr(\forall x : \mathcal{T}.\, \phi(x))
\end{array}
$$

Fig. 14.6. Typical HOAS proof rules

under bound variable renaming. The HOAS approach to representing syntax is very flexible. Examples in [AHMP92,HHP93,Pau94b] include higher-order logic, constructive type theories, modal logics, Hoare logic, LCF, and various lambda calculi.

HOAS eases the formalization of proof rules in that substitution, α-conversion, and side conditions on variables can also be "internalized". Let us return to the rule $\forall$-E, (14.34). We may represent this with the $\forall$-E axiom given in Figure 14.6. Note that, as syntactic sugaring, when Q is a variable binding operator, we shall write $Qx : T.\, \phi$ instead of $Q(\lambda x : T.\, \phi)$ (recall that we use a different font to distinguish $\forall$ in the metalogic from $\forall$ in the object logic). The $\forall$-E axiom has two parameters: the proposition-valued function P and the term t that instantiates it. So, for example, if we instantiate P with $\lambda x : \mathcal{T}.x \neq []$ and t with the list $[0]$, then, after β-reduction, this yields $pr([0] \neq [])$ under the assumption $pr(\forall x : \mathcal{T}.\, x \neq [])$. This indicates how substitution has been modeled by β-conversion. In the rule $\forall$-I, higher-order syntax has allowed us to model the variable occurrence condition (that x may not occur free in any undischarged assumptions), using universal quantification in the metalanguage and its associated side condition. The substitution rule indicates again how substitution can be modeled by application. As a final example, consider the following rule for structural induction over lists.

$$
\frac{pr(\phi[[]]) \qquad \begin{array}{c}[pr(\phi[x])] \\ \vdots \\ pr(\phi[h{:}x])\end{array}}{pr(\forall x : \mathcal{T}.\, \phi[x])}
$$

This rule has the side condition that x does not occur free in any assumption, other than those discharged by the application of the rule. This is axiomatized by the final rule in Figure 14.6. Reading this "backwards", it says that to prove some $\forall x.\phi[x]$, we must establish two assertions. First, we must prove $\phi[[]]$ and second, for some arbitrary (eigenvariables) h and x, we must show that $\phi[h{:}x]$ is provable under the assumption of $\phi[x]$.

With some experience, the translation of natural deduction presentations of development rules into HOAS becomes natural. However, the question of whether an encoding of syntax and rules actually represents the intended logic is a difficult question, especially since HOAS mixes the syntax of the object and metalogics. In [Gar93,HHP93], considerable attention is devoted to this problem and methodologies are provided for formally establishing the *adequacy* of a representation, i.e., there is the intended correspondence between the syntax and proofs of the object logic and the metatheory. A proof of adequacy for an encoding of a first-order theory similar to that sketched here (first-order arithmetic formalized within the LF-metalogic) can be found in [HHP93].

14.2.3 Developments in a logical framework

One advantage of schematic rules, especially those formalized using HOAS, is that, within a logical framework that supports this style of encoding, it is easy to construct developments. Moreover, program verification, program synthesis, and the construction of new development rules are similar activities. In order to demonstrate this clearly, and also to support our contention that formalization is not difficult, we will present some simple examples carried out in the logical framework ISABELLE [Pau89b,Pau94b]. This is not the only possible system for such work, e.g., ELF [Pfe91] and λ-Prolog [FM88] also come to mind.

ISABELLE **background.** Before illustrating developments in ISABELLE, we provide some background necessary to understand our examples. We will omit many details and [Pau94b] should be consulted for a full account of ISABELLE. ISABELLE is a tactic-based theorem prover whose (meta)logic is the fragment of higher-order logic based on $\forall$ and $\Longrightarrow$. Within this logic, object logics are encoded. The proof rules given in Figures 14.4 and 14.6 are almost exactly as they would be input to ISABELLE; the only difference is that in ISABELLE one often omits the outermost quantifiers and types as they can be reconstructed by the system. Developments may be constructed in ISABELLE in a top-down refinement style. A rule has the form

$$\forall v_1, \ldots, v_m. [\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi \tag{14.36}$$

where the notation $[\![\phi_1; \ldots; \phi_n]\!] \Longrightarrow \phi$ is shorthand for the iterated implication $\phi_1 \Longrightarrow \ldots \Longrightarrow (\phi_n \Longrightarrow \phi)$. This also represents a proof state where ϕ is the goal to be established and the ϕ_i represent the subgoals to be proven. Given 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.

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 (14.36), then (treating the v_i as variables for unification) 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, as unification is used to apply rules, the proof state may contain metavariables itself. We will show that this supports program transformation and synthesis during development. Also note that, although rules are formalized in a natural deduction style, (14.36) can be read as an intuitionistic sequent where the ϕ_i are the hypotheses. ISABELLE has resolution tactics that apply rules in a way that maintains this illusion of working with sequents.

All proofs presented in this section are, with the exception of minor printing enhancements, taken directly from an ISABELLE session (given in `typewriter font`). Note that ISABELLE's displays $\forall$ and $\Longrightarrow$ as `!!` and `==>` respectively. Also note that, as our proofs are restricted to one kind of judgement (namely *pr*), we omit this "coercion" and display $pr(\phi)$ as simply ϕ; context will always disambiguate metalogic from object-logic syntax. Finally note that we have *lengthened* proofs to make them easier to follow.

Verification. A formalized metatheory may be straightforwardly applied to prove judgements. For example, in our first-order metatheory we can easily prove a theorem like the following.

$$pr(\forall A\, B\, C.\, A \Rightarrow B \Rightarrow C \Rightarrow (A \wedge B \wedge C))$$

This theorem can be proven automatically in ISABELLE using a simplification tactic that comes with the system. However, we shall go through an interactive proof to give a feel for such proof construction using a formalized metatheory, and also as a "warm-up" for more complicated examples.

We begin by giving this goal (in our first-order theory *thy*) to the system.

```
- goal thy "ALL A B C. A --> B --> C --> (A & B & C)";
ALL A B C. A --> B --> C --> A & B & C
 1. ALL A B C. A --> B --> C --> A & B & C
```

We have typed the first line. The following two are system output. On the second line ISABELLE prints the ultimate goal and the third line contains the subgoal (identical to the ultimate goal) required to establish it. We proceed by resolving the subgoal (numbered 1) against $\forall$-I given in Figure 14.6.

```
- by (resolve_tac [allI] 1);
ALL A B C. A --> B --> C --> A & B & C
 1. !!A. ALL B C. A --> B --> C --> A & B & C
```

In the new subgoal, the resolution step has converted quantification over A in the object level to quantification in the metalogic. ISABELLE is a tactic-based theorem prover and we can use standard LCF-style tacticals (combinators that combine tactics) to shorten proofs. For example, we use *REPEAT* to iterate application of $\forall$-I.

```
- by (REPEAT (resolve_tac [allI] 1));
ALL A B C. A --> B --> C --> A & B & C
 1. !!A B C. A --> B --> C --> A & B & C
```

This applies $\forall$-I twice. At this point we proceed by resolving with $\rightarrow$-I (Figure 14.4).

```
- by (REPEAT (resolve_tac [impI] 1));
ALL A B C. A --> B --> C --> A & B & C
 1. !!A B C. [| A; B; C |] ==> A & B & C
```

The effect has been to convert implication at the object level to implication at the metalevel. Read as a sequent, the subgoal states that we must prove the conjunction from the three hypotheses. We apply $\wedge$-I to decompose the leading conjunction, which yields two subgoals.

```
- by (resolve_tac [conjI] 1);
ALL A B C. A --> B --> C --> A & B & C
 1. !!A B C. [| A; B; C |] ==> A
 2. !!A B C. [| A; B; C |] ==> B & C
```

Subgoals in ISABELLE are maintained as a numbered list. We may solve the first subgoal using the rule *ba* (by assumption), which invokes the assumption rule of the metalogic.

```
-  ba 1;
ALL A B C. A --> B --> C --> A & B & C
 1. !!A B C. [| A; B; C |] ==> B & C
```

The remaining goals (in this case only one) are renumbered. Here, after one more application of $\wedge$-I and two of *ba*, we have completed the proof. ISABELLE responds with the following proven theorem.

```
ALL A B C. A --> B --> C --> A & B & C
No subgoals!
```

This example is artificially simple, but it demonstrates that, with the right metalevel support, constructing developments using a formal metatheory is not much different from directly using a specially built implementation. A significant difference though is that, with a metatheory, we can formalize and derive new development rules.

Development rules. In Figure 14.3 we presented a schema from [HK93] and suggested that it could be formalized and derived as a development rule. To verify the exact schema given requires a formalized semantics of functions. Here we illustrate a less general, but much simpler alternative, which is often just as useful. Namely, we formulate and prove a specialized version within our metatheory of lists. With slight modification we can restate the schema as a formula that is directly derivable by list induction.

```
- val assums = goal thy
  "[| f([]) = B;
      ALL h t. f(h:t) = K(f(t),J(h,t));
      ALL x.f'(x) = g'(x,N);
      ALL y.g'([],y) = K(B,y);
      ALL h t y.g'(h:t,y) = g'(t,K(J(h,t),y));
      ALL x. K(x,N) = x;
```

```
      ALL x y z. K(K(x,y),z) = K(x,K(y,z)) |]
          ==> ALL x. f(x) = f'(x)";
ALL x. f(x) = f'(x)
 1. ALL x. f(x) = f'(x)
```

The first nine lines (up to the closing double quote and semicolon) are our input, which specifies the theorem to be proven. Lines 2–8 contain the assumptions that can be used in establishing the theorem: Lines 2–3 provide the schema for f, lines 4–6 provide the schema for f' (in terms of the auxiliary function g'), and lines 7–8 specify additional properties of K and N. The list of these assumptions are bound to a variable *assums* and may be used in proving the goal $\forall x.\, f(x) = f'(x)$.

Proving this theorem is actually quite similar to proving the equivalence of *rev* and *qrev* (*assums* can be seen as axiomatizing the algebraic laws needed for that theorem) and we begin by proving a generalization: we resolve with $\forall$-*I* and then use *subgoal_tac* to apply *cut* in the metalogic to prove a more general theorem. Cut allows us to insert a new hypothesis, provided we first prove it.

```
- by (resolve_tac [allI] 1 THEN subgoal_tac "ALL y. K(f(x),y) = g'(x,y)" 1);
ALL x. f(x) = f'(x)
 1. !!x. ALL y. K(f(x), y) = g'(x, y) ==> f(x) = f'(x)
 2. !!x. ALL y. K(f(x), y) = g'(x, y)
```

This step results in two subgoals. First, we must show that the cut formula implies the original goal and second, we must prove the cut formula. We proceed with the first goal and apply $\forall$-*E* to the assumption (the universally quantified formula on the left-hand side of `==>`) and instantiate y with N.

```
- by (inst_var [allE] ("y","N") 1);
ALL x. f(x) = f'(x)
 1. !!x. K(f(x), N) = g'(x, N) ==> f(x) = f'(x)
 2. !!x. ALL y. K(f(x), y) = g'(x, y)
```

Now using the second and sixth assumption (in *assums*) we simplify both the hypothesis and conclusion of the first goal. ISABELLE's rewriter completely solves this goal. This command uses a version of *simp_tac*, which performs first-order rewriting using a given "simplification set". The simplification set *list_ss* is a standard collection of previously proven rewrite rules for lists augmented with our assumptions *assums*.

```
- by (asm_full_simp_tac (list_ss addsimps assums) 1);
ALL x. f(x) = f'(x)
 1. !!x. ALL y. K(f(x), y) = g'(x, y)
```

We are left with our strengthened goal. We resolve against our structural induction rule for lists (*ind* in Figure 14.6) and then apply the rewriter to the resulting base and step case.

```
- by ((resolve_tac [ind] 1) THEN
      (ALLGOALS (simp_tac (list_ss addsimps assums))));
ALL x. f(x) = f'(x)
 1. !!x xa xs.
       ALL y. K(f(xs), y) = g'(xs, y) ==>
       ALL y. K(f(xs), K(J(xa, xs), y)) = g'(xs, K(J(xa, xs), y))
```

We only see the (simplified) step case because the rewriter has completely proved the base case, which is $\forall y.K(f([]), y) = g'([], y)$. This case was proved automatically by simplifying both sides of the equality to $K(B, y)$. In the step case we proceed by using the induction hypothesis as a rewrite rule to simplify the conclusion. This completes the proof.

```
- by (asm_simp_tac list_ss 1);
ALL x. f(x) = f'(x)
No subgoals!
```

ISABELLE allows us to bind this derived rule to a constant (which we call *tail_opt*) for use in subsequent developments.

This example indicates that deriving new development rules in a logical framework like ISABELLE that supports proof construction in a metalogic (e.g., through resolution and simplification) is not difficult. With some experience, this kind of derivation can be carried out interactively in minutes.

Transformation. Let us now use our derived rule *tail_opt* to synthesize the *qrev* function from the *rev* specification. Our development will not only illustrate how we can apply derived development rules, but also how HOAS and higher-order unification support program synthesis.

Our stated goal is to prove the equivalence of *rev* and an unknown function *qrev* under assumptions (bound again to *assums*) that axiomatize *rev*.

```
- val assums = goal thy
  "[| rev([])=[]; ALL h t. rev(h:t)=rev(t) ++ [h] |]
      ==> ALL l. rev(l) = ?qrev(l)";
ALL l. rev(l) = ?qrev(l)
 1. ALL l. rev(l) = ?qrev(l)
```

The question-mark in front of *qrev* tells ISABELLE that *qrev* should be treated as a metavariable for unification (otherwise it would be interpreted as an undeclared constant, like *rev*). That is, we are allowed to prove an *instance* of this theorem and in our case we intend this instantiation to be the new program.

Our first step is to resolve against the previous derived *tail_opt* rule. The tactic *res_inst_tac* performs resolution after first partially instantiating *tail_opt* by assigning *rev* to f and *qrev* to f'.[4]

[4] Strictly speaking it is unnecessary to provide this assignment because higher-order unification would generate this instance. However, in this case there are many (inappropriate) unifiers so it is simpler to provide the substitution up front.

```
- by (res_inst_tac [("f", "rev"),("f'","qrev")] tail_opt 1);
ALL l. rev(l) = ?qrev(l)
 1. rev([]) = ?B
 2. ALL h t. rev(h:t) = ?K(rev(t), ?J(h, t))
 3. ALL x. ?qrev(x) = ?g'(x, ?N)
 4. ALL y. ?g'([], y) = ?K(?B, y)
 5. ALL h t y. ?g'(h:t, y) = ?g'(t, ?K(?J(h, t), y))
 6. ALL x. ?K(x, ?N) = x
 7. ALL x y z. ?K(?K(x, y), z) = ?K(x, ?K(y, z))
```

The seven goals correspond to the seven premises of the *tail_opt* rule. Note that the uninterpreted constants in the *tail_opt* rule (i.e., B, K, G', J, and N) were generalized to variables by ISABELLE after the rule was derived, and these can be instantiated here by unification; hence, unification will determine an instance of the rule. Our plan is to discharge goals 1 and 2 by resolving them with the assumptions that define *rev*. This will generate an instantiation for most of the program variables (e.g., K will become append); afterwards we prove 6 and 7 using properties of append. This leaves goals 3 to 5, which after simplification will constitute the definition for *qrev*.

Let us begin by resolving the first two goals against the equations for *rev* in our assumptions *assums*. These resolution steps produce the substitutions $K(x,y) \equiv x +\!\!+\ y$ and $J(h,t) \equiv [h]$. These substitutions are propagated to the remaining subgoals, which are then renumbered.

```
- by (OnHyps (resolve_tac assums) [1,2] );
ALL l. rev(l) = ?qrev(l)
 1. ALL x. ?qrev(x) = ?g'(x, ?N)
 2. ALL y. ?g'([], y) = [] ++ y
 3. ALL h t y. ?g'(h:t, y) = ?g'(t, [h] ++ y)
 4. ALL x. x ++ ?N = x
 5. ALL x y z. (x ++ y) ++ z = x ++ y ++ z
```

We now turn our attention to goals 4 and 5, which have been instantiated to be assertions about append. Goal 5 may be proved by rewriting (the simplification set *list_ss* contains the lemma, already proved, that append is associative). Note that $+\!\!+$ is printed infix as a right-associative operator. Goal 4 may be proved by resolving with the lemma *append_Nil2*, which states that $x +\!\!+ [] = x$. We combine these steps as follows.

```
- by (simp_tac list_ss 5 THEN
      ((resolve_tac [allI] 4 THEN resolve_tac [append_Nil2] 4)));
ALL l. rev(l) = ?qrev(l)
 1. ALL x. ?qrev(x) = ?g'(x, [])
 2. ALL y. ?g'([], y) = [] ++ y
 3. ALL h t y. ?g'(h:t, y) = ?g'(t, [h] ++ y)
```

Notice that resolution with the lemma $x +\!\!+ [] = x$ generates the substitution $N \equiv []$ that is propagated to the first subgoal. Now we are almost done: we may take the first goal as the definition of *qrev* under the auxiliary function g' given by the second and third definitions. Before doing this, we simplify these remaining goals.

```
- by (ALLGOALS (simp_tac (list_ss addsimps assums)));
ALL l. rev(l) = ?qrev(l)
 1. ALL x. ?qrev(x) = ?g'(x, [])
 2. ALL y. ?g'([], y) = y
 3. ALL h t y. ?g'(h:t, y) = ?g'(t, h:y)
```

Finally, (using *uresult*) we can discharge these goals by making them assumptions of the theorem we are proving. ISABELLE returns the following proven theorem, which states the equality of *rev* and *qrev* under the given recurrence equations.

```
- val qsynth = uresult();
val qsynth =
  "[| ?rev([]) = []; ALL h t. ?rev(h:t) = ?rev(t) ++ [h];
      ALL x. ?qrev(x) = ?g'(x, []);
      ALL y. ?g'([], y) = y;
      ALL h t y. ?g'(h:t, y) = ?g'(t, h:y) |] ==> ALL l. ?rev(l) = ?qrev(l)"
```

Let us review what we have achieved. First, we began by giving a simple axiomatization of list theory in ISABELLE's metalogic. Next, within this theory we formally derived a development rule for recursion optimization. Finally, we used this rule to prove a theorem that states the equality between a given function *rev* and an unknown function *qrev*. During the development we generated a specialized definition of *qrev*, which reverses lists in linear, instead of quadratic, time. This example illustrates how higher-order unification can be used to synthesize programs during their correctness proofs. We will explore this development paradigm further in the next section.

14.2.4 Relating verification and synthesis

Our last example illustrated a schematic development: the development itself contained metavariables. Such developments facilitate a simple but powerful kind of program construction where unification can solve for unknown functions. In this section we will try to give more insight into this kind of development by briefly surveying some other approaches to synthesis and transformation, and indicating how they can be understood or recast as schematic development.

Our starting point is to observe that schematic development blurs the distinction between *verification*, which is proving that some object (e.g., program) has some desired property (e.g., specification), and *synthesis*, which is constructing an object with a desired property. The *rev* to *qrev* example suggests that a calculus designed for one of these activities can be used for both. We could use the *tail_opt* schema either to verify the equality of two definitions or, using metavariables, to delay commitment to *qrev* and synthesize it during the development.

We shall use an even simpler example to bring out this relationship between verification and synthesis. We shall make an analogy with Prolog proofs. Consider a simple Prolog theory about "happy people". We might state that rich and healthy people are happy. Written as a Prolog clause this is

$$happy(X) \leftarrow rich(X), healthy(X)\,.$$

Further, we might have facts in our theory that a certain individual, say *bob*, is both healthy and rich, i.e., *healthy(bob)* and *rich(bob)*. We might choose to verify in this theory an assertion like *happy(bob)*. Alternatively we might try to synthesize some expression X satisfying the specification *happy(X)*.

$$\frac{\dfrac{}{rich(bob)} \quad \dfrac{}{healthy(bob)}}{happy(bob)} \qquad \frac{\dfrac{}{rich(X)}\{bob/X\} \quad \dfrac{}{healthy(X)}\{X = bob\}}{happy(X)}\{X = bob\}$$

On the left we have given the verification proof that a Prolog interpreter would (implicitly) construct in answering the query *happy(bob)*. On the right is a synthesis proof. If we assume that goals are solved from left-to-right (as is the case in Prolog) then, in the synthesis proof, after resolving with the clause defining *happy*, we have a resolution step with the ground clause *rich(bob)*. This generates a substitution *bob* for X that is propagated to the remaining subgoal (as well as the original goal), which is then solved with a ground resolution step. This extremely simple example illustrates that these two proofs are essentially the same: from the perspective of resolution they are identical modulo the base case of unification! The only difference is that on the left we always unify ground terms against ground terms, whereas on the right we also unify variables against ground terms, which produces an answer substitution.

The situation is analogous for proof construction in our metalogic. In Prolog, one begins with an axiomatized Horn clause theory. In our case, we begin with an axiomatized metatheory of rules that we may formally extend. In Prolog, proofs are constructed using a uniform search procedure based on resolution (i.e., SLD resolution). In our case, proofs are constructed semi-interactively using higher-order resolution. In both, unification can be used either to verify properties of given objects or to construct them during proof. In our setting though, we require higher-order unification because we wish to solve for functions, which are second-order objects in our declared metatheory.

Relationship to resolution-based development. This idea of unifying verification and synthesis by using unification to apply rules is old. It goes back at least to Green's use of resolution, not only for checking answers to queries, but also for synthesizing programs. In [Gre69], Green suggests using the "answer predicate trick" in resolution[5] to find unifiers using resolution

[5] The resolution rule provides no means to keep track of substitution instances. The trick is to extend the goal clause with the literal *answer(X)* where X is

to answer questions. In this setting, as in Prolog, object-level variables stand in for metavariables and are assigned values, possibly incrementally, using resolution.

Let $R(x,y)$ be a first-order proposition. Green showed how resolution can serve as a basis for answering the following types of questions:

Problem	Question	Desired Answer
checking	$R(a,b)$	yes or no
simulation	$\exists x.\, R(a,x)$	yes $x = b$ or no
verifying	$\forall x.\, R(x, g(x))$	yes or no $x = c$
programming	$\forall x.\, \exists y.\, R(x,y)$	yes $y = f(x)$ or no $x = c$

The first three are straightforward, but the last may be surprising; it is closely related to our *qrev* synthesis example. To see if $\forall x.\, \exists y.\, R(x,y)$ follows from a set of formulas, Green negates it, turns it into a clause, and searches for a refutation. Negated and clausified, it becomes $\{\neg R(x_0, y)\}$, and if we can prove inconsistency we get a unifier for y, e.g., $y \equiv hd(x_0)$. Hence we have y as a "function" of x_0, and only first-order variables and unification have been used in the proof process. First-order variables and unification suffice because skolemizing removes binding operators and, at the same time, (implicitly) extends the signature (with x_0) in which the theorem is proved. Green is not really synthesizing a function, but rather a first-order term. However, since x_0 is arbitrary, he can generalize afterwards.

We see here the essence of the schematic development idea in a simple first-order setting: the same resolution steps can be used either to verify that a function is correct, or to construct the function.

Relationship to deductive synthesis. There are a number of approaches loosely classified under the heading of deductive synthesis that also have close ties to schematic developments, even if they are formalized otherwise. For example, in Manna and Waldinger's deductive tableau, 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" [MW80]. Although it is not expressed this way, y plays the role of a metavariable and is incrementally instantiated during subsequent proof steps. This is quite similar to Green's approach, except that the goal is not negated, since proof construction is not refutation-based. At the end of the proof, if variables have been instantiated to ground terms meeting certain side conditions, the tableau yields a completed program.

It is not difficult to draw a precise, formal correspondence between deductive tableau proofs and natural deduction proofs with metavariables. These

the variable whose instance we want. Then, rather than halting when resolution produces the empty clause, we may halt with a clause *answer(t)* where t contains the generated substitution instance for X. See also [CL73].

details have been worked out in [Aya95,AB96] and it is shown how, in the spirit of the derivation of *tail_opt*, one can derive program construction rules that can be used to simulate deductive tableau proofs. For example, corresponding to program generation by case splitting, we may derive the rule

$$[\![pr(A) \Longrightarrow pr(P(s)); pr(\neg A) \Longrightarrow pr(P(t))]\!] \Longrightarrow pr(P(\mathit{if}(A, s, t)))\,.$$

Now, if we have a goal $pr(R(x, ?y))$ we can resolve this goal against the rule above, which instantiates $?y$ to $\mathit{if}(?A, ?s, ?t)$ and yields the subgoals $pr(?A) \Longrightarrow pr(R(x, ?s))$ and $pr(\neg ?A) \Longrightarrow pr(R(x, ?t))$. The metavariables $?s$, $?t$, and $?A$ will be further instantiated as the cases are proven.

A number of other researchers, e.g., [Cle92,Coe92,HRS90,KBB93], have also observed that different approaches to deductive synthesis may be implemented by providing support for metavariables in developments.

Relationship to constructive synthesis. A distinction is sometimes made in the literature between deductive and constructive synthesis methods (see e.g., [DL93]). The latter are typically based on some kind of constructive logic or type theory equipped with a notion of realizability, e.g., Martin-Löf's type theory [Mar82], NUPRL [C⁺86], or the Calculus of Constructions [CH88]. In such logics one proves that a term t meets a specification T by demonstrating that t has the type T, denoted $t \in T$. For example, if we show that $t \in (\forall x : T_1.\, \exists y : T_2.\, R(x, y))$ then, according to the semantics of these logics, t must be an executable function where for any x in T_1, $t(x)$ evaluates to a pair, whose first component, $f(x)$, belongs to T_2, and second component belongs to $R(x, f(x))$.

Should one view a type theory as a theory for program verification, synthesis, or both? In the NUPRL system, for example, the program t may or may not be given *a priori*, and the system contains two similar sets of rules depending on whether t is present (i.e., the goal is $t \in T$) or not (i.e., the goal is simply T). In the former case, proofs verify programs. In the latter, they say how to construct or synthesize programs, and the system may *extract* the inhabiting term t. The extraction, however, may be understood in a very simple way: the term t may be represented initially by a metavariable stored "behind the scenes" and each development rule application elaborates the structure of t. In [Pau94b], Paulson documents an implementation in ISABELLE of a similar type theory, where metavariables are "first-class" objects and may appear directly in proofs. Rules are applied with higher-order resolution (as we have described) and one proves the judgement $t \in T$, where t may be a specific 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 left unspecified (see [HBS92] for uses of such hybrid verification/synthesis).

It is worth emphasizing that the idea of schematic developments suffices to explain program construction in such theories. A common explanation of program extraction 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 be misleading. Unification during schematic developments is enough to account for extraction in type theories like Martin-Löf's or NUPRL's. Constructivity does not play a role in the *extraction* of a program from a proof but rather in its *execution* and *meaning*. That is, because the logic is constructive, the terms extracted can be effectively executed and their evaluation behavior agrees with the semantics of the type theory. For example, see [How91] for a semantics for a classical type theory where programs correspond to oracles that cannot be executed.

Admittedly this discussion takes us a long way from the topic of formalizing developments. However it does help to illustrate that, by choosing the proper development framework, one can formalize developments in ways that support not just program verification, but also transformation and synthesis.

14.3 General development rules

14.3.1 Motivation

The previous section has examined developments based on schematic rules. Unfortunately, not all rules are so easily formalized. Recall from Section 14.1.3 the more general view of a rule as a partial function. Below we consider two examples of rules that are not formalizable as simple schematic rules because computing an instance requires more than just substitution into a schema.

"Regular" schemata. In our formalized first-order theory, for all sequences of formulas ϕ_i and ϕ, it is provable that

$$(\phi_1 \Rightarrow (\ldots \Rightarrow (\phi_n \Rightarrow \phi)\ldots)) \Leftrightarrow ((\phi_1 \wedge \ldots \wedge \phi_n) \Rightarrow \phi)\,.$$

Now suppose we wished to formalize a rule that allowed us to reason backwards from a formula in uncurried form (the right-hand side) to curried form (the left). We might formulate this, say, for $n = 3$, as

$$\forall \phi_1\, \phi_2\, \phi_3\, \phi : \mathcal{F}.\, pr(\phi_1 \Rightarrow (\phi_2 \Rightarrow (\phi_3 \Rightarrow \phi))) \Longrightarrow pr((\phi_1 \wedge \phi_2 \wedge \phi_3) \Rightarrow \phi)\,.$$

Unfortunately, while we can formalize this rule for any particular n, we cannot capture this pattern for every n, e.g.,

$$\forall \phi_1\, \ldots\, \phi : \mathcal{F}.\, pr(\phi_1 \Rightarrow (\ldots \Rightarrow (\phi_n \Rightarrow \phi)\ldots)) \Longrightarrow pr((\phi_1 \wedge \ldots \wedge \phi_n) \Rightarrow \phi)\,.$$

The problem is that the " ... " have only an informal meaning: " ... " expresses a set of schemata. This set can in turn be described, for example, by a context-free language and recognized by a simple parsing algorithm. Hence, one way to formalize this rule is to formalize such an algorithm; but then establishing the correctness of the rule requires reasoning about the given algorithm or grammar (see Section 14.3.3).

Decision procedures. Another useful class of rules are those that implement decision procedures for logics or sublogics. A simple example of this is the theorem of Łukasiewicz that we alluded to earlier: a propositional formula built only from the equivalence connective is provable when all its propositional variables occur an even number of times.

Let us consider the formalization of this. This is a theorem about propositional logic, so let us assume we have a type $\mathcal{F}$ of propositions constructed from variables (over a type $\mathcal{V}$) with a constructor $var : \mathcal{V} \to \mathcal{F}$ and at least the connective (there could be others) $Iff : \mathcal{F} \times \mathcal{F} \to \mathcal{F}$ that builds equivalences. Further, assume we have a type $Bool$ in the metatheory with operators like & defining conjunction. Now, we may define a function $parity : \mathcal{F} \to Bool$,

$$parity(\phi) = iff\_form(\phi)\ \&\ even\_occ(\phi),$$

that checks that (1) ϕ is only built from variables and the Iff operator and that (2) all variables occurring in ϕ have an even number of occurrences. The first condition, *iff_form*, might be formalized as follows.

$$\begin{aligned}
\forall x : \mathcal{V}.\ iff_form(var(x)) &= true \\
\forall \phi_1\ \phi_2 : \mathcal{F}.\ iff_form(Iff(\phi_1, \phi_2)) &= iff_form(\phi_1)\ \&\ iff_form(\phi_2) \\
\forall \phi_1\ \phi_2 : \mathcal{F}.\ iff_form(and(\phi_1, \phi_2)) &= false \\
&\vdots
\end{aligned}$$

The function *even_occ*, which checks the second condition, could be formalized, for example, as the composition of a function which converts the fringe of the formula, viewed as a tree, to a list and one which checks that each variable in the list occurs an even number of times. Finally, let Eq be a judgement over Booleans, where $Eq(b_1, b_2)$ is true when b_1 and b_2 have the same Boolean value. Now, we may formalize Łukasiewicz's theorem in the metatheory as

$$\forall \phi : \mathcal{F}.\ Eq(parity(\phi), true) \Longrightarrow pr(\phi)\,. \tag{14.37}$$

This theorem may be proved by induction over the size of formulas as we now sketch. Given a $\phi \in \mathcal{F}$ such that $parity(\phi) = true$, then ϕ is a term built from Iff and must contain at least two occurrences of the same variable, call it a. One case is $\phi = Iff(a, a)$, but $pr(Iff(a, a))$ may be proved using rules for Iff. In the second case, ϕ must contain more than one Iff connective. Since Iff is associative and commutative, we may reorder variables in ϕ and "shuffle" together two occurrences of a from the remainder, ϕ'. Hence we must have

$$pr(Iff(Iff(a, a), \phi')) \Longrightarrow pr(\phi).$$

Furthermore, we can "cancel" the formula $Iff(a, a)$ yielding $pr(\phi') \Longrightarrow pr(\phi)$. Since $parity(\phi') = true$ and ϕ' is simpler (e.g., fewer variables) than ϕ, we have $pr(\phi')$ by the induction hypothesis. Hence, we can conclude $pr(\phi)$.

Once proved, this theorem can be used as a new development rule; its application reduces proving $pr(\phi)$ to proving $Eq(parity(\phi), true)$. The resulting goal can be proved by symbolic evaluation using the equations defining *parity*; this evaluates to *true* when ϕ meets the precondition *parity*. This evaluation can proceed either by rewriting in the metatheory using the equations defining *parity* or by evaluation in the theorem prover, if the prover implements a metatheory based on a programming language (e.g., type theories like [C+86] or pure Lisp like [BM79]). Evaluation is often more efficient than proving the original goal with a tactic. The metatheorem need not construct a validation since it has been formally verified to be correct, just like the rules from the previous section.

14.3.2 Supporting metatheory

In Section 14.2.2 we showed how the syntactic categories of a logic are specified as data types in a metatheory. Formalizing and reasoning about functions over these types is a special case of the more general problem examined in this book: how should programs be formalized (e.g., using equations, Horn clauses, etc.) and what formalisms are adequate to specify and establish their properties? Rather than surveying such possibilities again, we will restrict ourselves to pointing out some of the options that have been successfully applied to formal metatheory in the past.

First-order metatheories. In the last section we considered using first-order metatheories for explicitly encoding syntax. The same kinds of metatheories suffice to reason about functions represented as equations or Horn clauses. For example, the definition for *iff_form* used in *parity* could be directly formalized in a metalogic based on sorted equations. In the last section, rule definitions did not involve recursion. Here the difference is that to reason about rules defined by recursive functions, or containing recursively specified applicability conditions, we require induction principles for our data types. In the *parity* example, we performed induction over formulas, which may be semantically justified by restricting ourselves to considering the initial model of this data type [GTW78].

This approach to developing rules as verified metatheorems in an equational setting has been carried out using the OBJ3 theorem prover, which is based on a term-rewriting implementation of equational logic. [GSHH92] presents the use of OBJ3 as a metalogic where object logics are encoded in equational logic in the explicit style we have presented. There, a rule like the Łukasiewicz example can be directly formalized as we have done. Once proved, it may be applied, and OBJ3's rewriter can be used to evaluate the preconditions of *parity*. A similar investigation of the use of first-order metatheories for the derivation of rewrite rules has been carried out using Maude [MM94], which is a logical framework based on rewrite logic.

Higher-order logics and type theories. In higher-order settings, e.g., based on higher-order logic or some kind of type theory, we can carry out similar developments by defining data types and axiomatizing induction and computation principles for these types. For example, [BC93] presents a kind of higher-order abstract data type based on existential types in a type theory like Martin-Löf's. The definition of data type constructors is similar to the algebraic setting, however each data type is equipped with a structural induction rule and a rule for defining functions by primitive recursion. These principles suffice to define functions like *parity* and prove their properties using derived induction principles. Most type theories support some form of inductive definition suitable for these kinds of proofs (e.g., [BCMS89,Pau89a]). A notable exception is the type theory of the Edinburgh LF, which is very weak and neither supports the definition of functions by primitive recursion nor induction over defined signatures. However, Pfenning has been carrying out an ambitious program of formalized development, where programs are expressed as logic programs using LF signatures and induction is carried out indirectly through a kind of *schema checking* [PR92].

Within a sufficiently expressive metalogic (e.g., HOL, or Calculus of Constructions), an alternative to axiomatizing induction is formally to define data types as inductive definitions within the metatheory, and then derive rules like induction within this theory. One such approach, formalized within the ISABELLE system by Paulson [Pau94a] is based on an internalized account of the Knaster–Tarski fixpoint theorem within higher-order logic. This theorem is used to justify inductive definitions and recursion over definitions given by least and greatest fixpoint equations.

There are advantages in using a constructive type theory as a metalogic as opposed to classical logics. In particular, as proofs construct programs, the verification of rules often has useful computational content. Consider Łukasiewicz's theorem. A proof of this theorem yields a function f such that, for any formula ϕ that satisfies the *parity* predicate, $f(\phi)$ evaluates to a member of $pr(\phi)$, which constitutes a proof of ϕ. This function plays a role analogous to the validation of a tactic (except here proofs are terms within the metalogic, instead of derivations constructed using the metalogic). But f need not be executed to verify the correctness of the development step since the rule itself has been formally verified. On the other hand, f could be executed if we wanted to construct a proof term showing $pr(\phi)$. [BC93] contains a further discussion of these points. Related work also includes [CH90,How88a] in which term rewriters, matchers, and normalizers, are implemented and formally verified within the NUPRL type theory (see also Section 14.4.1) and the work of [Kre93] in formulating and verifying rules for developing "global search algorithms" based on schemata due to Smith [Smi91].

Specially designed frameworks. The final possibility we consider is the use of a metalogic explicitly designed as a formal metatheory, such as Feferman's FS_0 [Fef88]. FS_0 was designed as a minimal logic for allowing formal metareasoning (it is a conservative extension of primitive recursive arithmetic). The syntax of the object logic is explicitly encoded using S-expressions (as in Lisp) in FS_0. For each syntactic category of the object logic, an inductive class is defined that contains all S-expressions representing members of that category. FS_0 allows computation over S-expressions by primitive recursion and reasoning by structural induction. In [MSB93] an example is given showing how the predicate calculus may be formalized and how one may reason about explicitly encoded bound variables in the context of a prenex normal form theorem for predicate calculus. Note that proofs in the object logic are also defined in FS_0 by inductively defined classes. This allows the development of rules that are admissible, but not necessarily derivable (for the distinction, see [HS86,Tro82]). Furthermore, this facilitates a formal development of proof transformations such as cut elimination [Mat94].

14.3.3 Generalized unification and matching

The advantage of schematic rules over rules specified using arbitrary programs is that the former may be directly applied using matching or unification. With the latter, a development step requires the user to provide appropriate instances, "execute" the program, and apply the result. There is a useful middle ground between these two extremes: the limitations of schemata in expressing rules may often be overcome by extending matching and unification to incorporate equational theories. With this approach, the meaning of functions may be declared equationally and incorporated into the matching or unification algorithm used to apply rules.

The problem of equational matching and unification has been extensively studied and various algorithms for $\mathcal{E}$-unification (unification modulo a set of equations) have been developed. In particular, techniques based on *narrowing* have been developed and proved complete in solving unification problems associated with convergent term-rewriting systems [BHS89,GS88, Hul80]. Narrowing has been used as the major computational mechanism underlying functional logic languages such as EQLOG [GM86] and SLOG [Fri85], however it has rarely been used in program synthesis and program transformation, or for large scale theorem proving in systems like ISABELLE.

Let us illustrate how the equational approach can provide a direct solution to the problem of formalizing ellipsis for the kind of schemata we considered earlier, e.g.,

$$(\phi_1 \Rightarrow (\ldots \Rightarrow (\phi_n \Rightarrow \phi)\ldots)) \Leftrightarrow ((\phi_1 \wedge \ldots \wedge \phi_n) \Rightarrow \phi)\,. \tag{14.38}$$

We may specify these schemata by defining in Figure 14.7 equations that define curried and uncurried formulas. We assume that our signature for formulas contains functions *and* and *imp*, which build conjunctions and implications. These equations, oriented from left to right, define a canonical

$$\begin{aligned}
curry(p, []) &= p \\
curry(p, x{:}xs) &= imp(x, curry(p, xs)) \\
uncurry(p, []) &= p \\
uncurry(p, x{:}xs) &= imp(aux(x{:}xs), p) \\
aux(x{:}[]) &= x \\
aux(x{:}x'{:}xs) &= and(x, aux(x'{:}xs))
\end{aligned}$$

Fig. 14.7. Axioms defining *curry* and *uncurry*

rewrite system. Now (14.38) can be formalized as the following rule in our metalogic

$$\forall \phi : \mathcal{F}. \forall l : \mathcal{F}\ List.\ pr(curry(\phi, l)) \Longrightarrow pr(uncurry(\phi, l)) . \tag{14.39}$$

Now, given a particular goal like

$$pr((\psi_1 \wedge \psi_2 \wedge \psi_3) \Rightarrow \psi) ,$$

we can (equationally) match this against the right-hand side of our rule, which results in an instance where $\phi \equiv \psi$ and $l \equiv [\psi_1, \psi_2, \psi_3]$. If we had applied this proof rule using resolution incorporating the equations above, then with this substitution instance, our goal would be reduced to proving

$$pr(\psi_1 \Rightarrow (\psi_2 \Rightarrow (\psi_3 \Rightarrow \psi))) .$$

It is actually possible to specify (14.39) in a more modular way because the desired schemata are simple and "regular" enough that we can express the rule using a higher-order recursive function *foldr* as a kind of control combinator:

$$\begin{aligned}
foldr\ op\ z\ [] &= z \\
foldr\ op\ z\ (x{:}xs) &= op\ x\ (foldr\ op\ z\ xs)
\end{aligned}$$

Direct computation shows that the left-hand side of (14.38) is equal to

$$foldr\ imp\ \phi\ [\phi_1, \ldots, \phi_n]$$

and the right-hand side is

$$imp\ (foldr\ and\ true\ [\phi_1, \ldots, \phi_n])\ \phi$$

where *true* is the neutral element for conjunction. Hence, in a metatheory that contains the definition of *foldr*, our rule for uncurrying can be directly expressed as a formula in the metalogic, namely

$$\forall \phi : \mathcal{F}. \forall l : \mathcal{F}\ List.\ (foldr\ imp\ \phi\ l) \Longrightarrow imp((foldr\ and\ true\ l), \phi) .$$

The use of a combinator like *foldr* is possible since some of the parameters are known; *curry* is essentially the partial parameterization *foldr imp* ϕ. The use of such combinators represents a kind of *control abstraction* and facilitates the reuse of development rules and their verification.

It is well-known that $\mathcal{E}$-unification is complex and inefficient in general; there are often many solutions, even to simple unification problems, and a combination of equations usually requires a new unification algorithm. This is also true for the general narrowing approach. These problems have hindered the incorporation of general equational unification procedures into program synthesis and transformation systems. Recent work by Shi [Shi94] presents algorithms that can often efficiently solve matching problems formalized using a class of recursive functions, called *matching combinators.* The combinators *foldr* and *curry* are examples of higher-order and first-order matching combinators, respectively. Under this approach, second-order matching against formulas in the metalogic with *foldr* is decidable and also unitary if it contains a concrete function (e.g., $\Rightarrow$ in the example above). This work is currently being integrated into the ISABELLE system and promises considerable support for both formalizing and using schematic rules.

14.3.4 Context sensitivity

Many rules have applicability conditions that formalize assumptions about the context in which the rule must be applied. According to the partial function view of rules, which gives rise to tree-shaped deductions, such assumptions take the form of additional premises, e.g., the equations about the monoid property in our example in Figure 14.3 and its later variants. According to the relational view of rules, applicability conditions give rise to verification conditions, which are generated after matching the left-hand side of a transformation rule; these conditions must be discharged before the rule can be safely applied.

In both cases, we can roughly classify conditions based on how they may be checked.

1. Structural or static semantic conditions that can be discharged automatically by some inference mechanism built into the transformation system. Such mechanisms include type inference and incremental evaluation of attributes decorating the abstract syntax tree [HK93]. Examples of such conditions include information on types, occurrences, scope information, etc.
2. Conditions that can be discharged by user-provided tactics.
3. Conditions that have to be proved interactively by the user.

Depending on the representation of context information in the system, our rule for tail recursion may fall into any of the above categories:

1. The context may provide access to all locally applicable axioms and theorems (this "local theory" was provided as an extra attribute in [HK93]) such that associativity can be checked against an appropriate axiom and the value of the neutral element N can be picked up from an axiom as well.
2. The monoid axioms can be proved and the value of N deduced by some other verification tactic.
3. The user has to provide the value of N as an input parameter and prove associativity and that N is the identity.

Note that only if contextual conditions are computable (the first two cases) can rules be easily composed or used in complex application strategies, which, for example, apply a rule deep down in some recursion process (see Section 14.4).

14.4 Developments

We have seen how individual proof rules can be specified and derived. Here we shall consider how development steps can be composed, i.e., how we may reason about the composition of rules and the strategies that combine them. In particular, we consider how combinators can be used to build developments in a uniform way that facilitates formal specification of development strategies and their verification. In order to illustrate this concretely, we turn to the relational view of developments and examine developments based on equational reasoning and the formalization of rewrite proofs.

14.4.1 Formalizing equational developments

In the setting of algebraic development, equational reasoning forms a central component of formal correctness proofs (see Chapter 8) and is implemented in systems that support rewriting. Traditionally, rewriting is not formalized (although see [MM93]). System support may consist of some kind of completion procedure that converts a rule set into a canonical rewrite system that can be used as a decision procedure [Gan91,KSZ86]. Alternatively, in tactic-based theorem proving systems, the user typically provides the system with a set of rewrite rules, which are exhaustively applied (as was the case in our ISABELLE examples), perhaps under some user-specified strategy. The rewrite theory need not be confluent or even terminating.

Below we consider a standard approach for implementing rewriting in a tactic-based setting in which rewriting can be programmed in a structured and modular way. The idea goes back to the implementation of rewriting functions called *conversions* in the LCF system [Pau83,Pau87]. To ease our presentation, we will give a slightly simplified account of conversions; implementations in the spirit described here have been carried out in the LCF, HOL, NUPRL, and ISABELLE systems.

Primitive conversions. We shall assume that a tactic programming language, such as ML or some other functional programming language, is given. Conversions are programs that implement rewriting; they are partial functions of type $\mathcal{T} \to \mathcal{T} \times \mathit{tactic}$. A conversion should, on input t, return a pair $\langle s, T\rangle$ where the tactic T proves the equation $pr(t = s)$ in the appropriate theory, or returns a failure token (an *exception* in the ML programming language). With conversions, to rewrite a term t, we execute $c(t)$ and afterwards execute the resulting tactic T to prove $pr(t = s)$.

We begin by describing primitive conversions, which correspond to simple kinds of rewriting steps. Initially, there is the identity conversion *IdConv*, defined by the function $\lambda t.\, \langle t, \mathit{Refl}\rangle$, that simply returns the input term paired with a tactic *Refl* that proves $pr(t = t)$ for all terms t. Primitive conversions may also be defined from known equalities, i.e., $pr(l = r)$. We can write a conversion-valued function that, given an equation $pr(l = r)$, returns a conversion c such that $c(t)$ executes by matching l against t, which either returns the substitution σ or fails if the terms do not match, and afterwards returns the pair $\langle \sigma(r), T\rangle$ where the tactic T proves $pr(\sigma(l) = \sigma(r))$ using the given equation.

Composite conversions. Primitive conversions comprise basic development steps: rewriting at the root of a term with the identity or an equation. These basic steps can be combined by introducing choice (branching) between conversions, iteration, and extension to rewriting within contexts. Composition of steps is programmed by *conversionals*, which are analogous to tacticals in combining tactic-based developments. Indeed, the analogy is very close; for example, in the ISABELLE development in Section 14.2.3 we presented the tacticals *THEN* and *REPEAT* for composing proof steps and, here, the conversionals *ThenC* and *RepeatC* play an analogous role in composing rewrites. For both tactics and conversions, the use of combinators gives rise to a programming calculus that may in turn be formalized as an algebra. The modular construction of tactics and conversions raises their abstraction, eases reasoning about their correctness, and facilitates reuse.

There are two basic conversionals for sequencing conversions: *OrelseC* and *ThenC*. The former provides selective composition and the latter a method of sequentially composing conversions. *OrelseC* works by testing if conversions succeed or fail (return an exception). Suppose the expression $e_1\ ?\ e_2$ computes the value of e_1 and, if that fails, it computes the value of e_2. We use this to define $(c_1\ \mathit{OrelseC}\ c_2)(t)$ as $(c_1\, t)\ ?\ (c_2\, t)$. This conversional can be used to pick a successful rewrite strategy out of a set of conversions, that is, it provides a way of trying different strategies. On the other hand, the (infix) conversional *ThenC* uses transitivity,

$$\forall t_1, t_2, t_3 : \mathcal{T}.\, [\![pr(t_1 = t_2); pr(t_2 = t_3)]\!] \Longrightarrow pr(t_1 = t_3)\,,$$

to compose conversions sequentially. That is, the conversion c_1 *ThenC* c_2, when applied to a term t, first computes

$$c_1\, t = \langle t_1, T_1 \rangle$$
$$c_2\, t_1 = \langle t_2, T_2 \rangle$$

and returns $\langle t_2, T \rangle$, where the tactic T proves $pr(t = t_2)$ by backchaining through the transitivity rule above and proving the subgoals $pr(t = t_1)$ and $pr(t_1 = t_2)$ using T_1 and T_2, respectively.

Using sequencing we can directly program iteration. For example, we can define recursively the conversional *RepeatC*, where *RepeatC*(c) repeatedly applies the conversion c until failure, i.e.,

$$\mathit{RepeatC}(c) = (c\ \mathit{ThenC}\ (\mathit{RepeatC}(c)))\ \mathit{OrelseC}\ \mathit{IdConv}\,.$$

Subterm traversal. In the case of tactics, the objects operated upon are goal trees, and tacticals apply tactics to the fringes of these trees (with the exception of transformation tactics in the sense of NUPRL [C+86]). In the case of rewriting, the objects are terms, and developments must specify a subterm or context where simplification occurs. The conversions we have previously described allow simplification at the root of the input term. We may extend them to subterms with *homomorphic extension* functionals, which extend the effect of a (local) transformation over larger contexts (see also [HK93, Kri89]).

Rewriting subterms is justified by the fact that equality is a congruence, i.e., for each n-ary function f, we have

$$\forall t_1 \ldots t_n\, s_1 \ldots s_n : \mathcal{T}.\, [\![pr(t_1 = s_1); \ldots ; pr(t_n = s_n)]\!] \\ \Longrightarrow pr(f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n))$$

The conversion *SubConv*(c), when applied to a term $f(t_1, \ldots, t_n)$ produces a pair $\langle f(s_1, \ldots, s_n), T \rangle$ by first applying c to each subterm t_i, yielding the pair $\langle t_i', T_i \rangle$, or, if it fails, the pair $\langle t_i, \mathit{Refl} \rangle$. If all n conversions fail, then *SubConv* fails. The tactic T is constructed that applies the congruence rule above, which reduces proving the equation $pr(f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n))$ to proving the subgoals $pr(t_i = t_i')$. T then proves each such subgoal by T_i.

SubConv provides the basis for traversing terms recursively. We may now write operators which recursively traverse and rewrite subterms. For example, the operator *Depth*(c) that recursively rewrites all subterms of a term in depth-first order may be recursively defined as follows:

$$\mathit{Depth}(c) = ((\mathit{SubConv}(\mathit{Depth}(c)))\ \mathit{ThenC}\ (\mathit{RepeatC}(c))) \\ \mathit{OrelseC}\ (\mathit{RepeatC}(c))\,.$$

Similarly, top-down rewriting is accomplished by *Top*(c) whose recursive definition is[6]

$$\mathit{Top}(c) = (\mathit{ProgressC}(\mathit{RepeatC}(c)))\ \mathit{OrelseC}\ (\mathit{SubConv}(\mathit{Top}(c)))\,.$$

[6] *ProgressC*(c) is a conversion that fails when c behaves like *IdConv*.

With a small set of combinators, we can define powerful, indeed arbitrary, rewrite strategies for applying a given set of rewrite rules.

14.4.2 Verified development strategies

Tactics and conversions are programs that construct developments. They are used as flexible programming languages for expressing development strategies and how developments can be composed. In both cases, a validation is required to guarantee the correctness of these steps. A conversion, when given an input t, simultaneously computes a rewritten term s and a tactic T that justifies the equality. The execution of T is necessary since any function of type $\mathcal{T} \to \mathcal{T} \times tactic$ is a conversion and users can program (maliciously or by accident) a conversion where $pr(t = s)$ is not provable.

Call a conversion c correct if, when $c(t) = \langle s, T \rangle$, T will succeed in proving $pr(t = s)$. If we restrict the way in which conversions are built, we can establish the correctness of conversions and their composition, namely, only use conversions built from (the previously described) primitive conversions and the conversionals. If we denote the set of such conversions $\mathcal{C}$, then we can show that every $c \in \mathcal{C}$ is correct by induction on the structure of c: primitive conversions are easily seen to be correct and each conversional preserves correctness.

We have previously considered the formalization and verification of rules expressed by programs. If our metalanguage for writing conversions is part of our formalized metatheory, then we can formally establish the correctness of these development strategies. The advantage of formally establishing correctness is that for $c \in \mathcal{C}$, if $c(t) = \langle s, T \rangle$ we can assert $pr(t = s)$ without executing T. This is analogous to the way in which decision procedures can be verified, as discussed in Section 14.3. Actually, if conversions are verified, then, since T need not be executed, we need not construct it in the first place. Instead, conversions can be defined as partial functions of type $\mathcal{T} \to \mathcal{T}$ and primitive conversions and conversionals may be defined that do not build tactics. As before, a subset of these corresponding to $\mathcal{C}$ can be proved correct and serve as a basis for composing development steps (rewriting steps) in a flexible and verified way.

The kind of verification of rewriting developments described here has been carried out by Howe [How88b] in the NUPRL system. There Howe specified a type of correct conversions (analogous to the notion of correctness given above) and defined and showed correct a class of conversions similar to those considered here.[7] Term rewriting based on a similar idea has been formalized in the OBJ3 system [GSHH92]. In this work, OBJ3 is used as a metalanguage for itself, and a first-order algebra of rewriting combinators is defined.

[7] Note that Howe's work is significantly more involved because he was also concerned with *reflection*, that is, how verified rewriting in NUPRL could be applied to reason about NUPRL proofs themselves (see also [ACHA90]). If one maintains the meta/object logic distinction, the problem is considerably simpler.

14.4.3 Metaprogram transformation and optimization

As observed earlier, the development of correct metaprograms (i.e., tactics) is a special case of developing correct programs. Similarly, when metaprograms are defined in a modular way, for example, based on higher-order combinators, then we can reason about the combinator algebra to derive more efficient tactics using transformation rules at the metalevel.

As a concrete example, consider homomorphic extension. Many functions on lists can be described using the following homomorphic extension functional *hom*.

$$\begin{aligned} \mathit{hom}\ \mathit{op}\ e\ h\ [\,] &= e \\ \mathit{hom}\ \mathit{op}\ e\ h\ (x{:}xs) &= \mathit{op}\ (h\ x)\ (\mathit{hom}\ \mathit{op}\ e\ h\ xs) \end{aligned}$$

Examples of this functional include the reverse function

$$\mathit{rev} = \mathit{hom}\ (\lambda x\, y.y \mathbin{+\!\!+} x)\ [\,]\ \lambda x.[x]\,,$$

the right folding functional *foldr*, where $\mathit{foldr}\ \mathit{op}\ e = \mathit{hom}\ \mathit{op}\ e\ \mathit{id}$, or the mapping functional *map*, where $\mathit{map}\ f = \mathit{hom}\ \mathbin{+\!\!+}\ [\,]\ f$.

Now, provided *op* and *e* form a monoid, we can optimize *hom* by introducing tail recursion.

$$\begin{aligned} \mathit{hom}\ \mathit{op}\ e\ h\ xs &= h_2\ \mathit{op}\ e\ h\ xs\ e \\ h_2\ \mathit{op}\ e\ h\ [\,]\ r &= \mathit{op}\, r\ e \\ h_2\ \mathit{op}\ e\ h\ (x : xs)\ r &= h_2\ \mathit{op}\ e\ h\ xs\ (\mathit{op}\, r\ (h\ x)) \end{aligned}$$

In doing so, we have optimized all instantiations of *hom* (where *op* and *e* form a monoid); this includes those functions that are used at the metalevel.

This kind of optimization is very common; we can perform similar kinds of optimizations for homomorphisms over trees and this could be used to improve term-rewriting tactics requiring subterm traversal (e.g., *SubConv* in Section 14.4.1). In the case of rewriting, we can also imagine reasoning about the equivalence of rewrite strategies and showing that some particular inefficient rewrite strategy is equivalent to a more efficient one. For example, we might prove that repeated outermost rewriting with each rule is equivalent to one bottom-up sweep, where every rule is tried once at every applicable position [HK93,KKLT91,Kri89].

14.4.4 Development methods = rules + tactics

The construction and transformation of programs, either interactively or automatically, requires more than the simple blind application of some set of transformation rules. In practice, a designer of transformations must go further and build *development methods* that combine both verified development rules and tactics that embody strategies for applying them. Both aspects are necessary as a rule cannot formalize all aspects of program development and one rule often gives rise to various strategies. For example, in the relational view of transformation rules, a tactic may say how to automatically check context conditions, generate verification conditions, or request user input, such as how to instantiate metavariables. In the partial function view, when there are multiple subgoals, the tactic may chose to attack the subgoals in a particular order and this may correspond to different approaches to program development.

A concrete example of this is provided in the ISABELLE derivation we gave of *qrev* in Section 14.2.3. After applying the transformation rule we had seven goals. We chose to discharge them in a particular order: we proved goals 1 and 2 first, which provided instantiations for most of the metavariables in the resulting proof state, before moving on to goals 6 and 7, which expressed properties of some of these metavariables. As a result, 6 and 7 were almost fully instantiated when we proved them. If the order were reversed, we would have had to solve goals 6 and 7 with more degrees of freedom. In particular, there might be other possibilities for K and N in our background theory (or theories), which together form a monoid and we would have to try these various possibilities. This is, of course, wasteful since goal 2 forces a unique assignment for K, which further determines N. It is easy to imagine similar degrees of freedom in algebraic data type implementations, where it may be advantageous to state the target first, and then develop a corresponding abstraction homomorphism, or to develop the homomorphism first, or to start by describing the induced congruence relation, etc.

Our view of combining rules with tactics leads to a natural hierarchical structure of development theories (see, too, the discussion of *development graphs* that follows). At the foundation is a logic or set of related logics. The next layer consists of derived rules, which are proved correct in these logics relative to the intended semantics of the programs and data types. At the highest level, the rules are combined with tactics into methods, which specify strategies for applying the rules and resolving subsequent deductive tasks. This view leads to a methodology for structuring development theories and clarifies conceptually the roles of the underlying theory, derived rules, and tactics.

14.4.5 System support

Formal development requires adequate system support. The kinds of development and transformation methodologies described here can best be carried out with machine assistance for tasks such as theory organization, application of proof rules, checking applicability conditions, and the like. Moreover, support should not simply be for rule application, but should center around all aspects of development including specification, verification of rules, reuse, and modification.

It is possible that all of these development activities may be carried out within the same system; in this chapter we gave such an example using Isabelle. However this is not the only possibility. Different activities (e.g., verifying, as opposed to simply applying, rules) require different skills and thus different kinds of user support. Moreover, considerable structuring facilities are required since numerous kinds of documents are related to each other in a formal development: specifications, program modules, proof obligations and proofs, proof and transformation rules, tactics, development scripts, etc. All these form part of a *project development graph* [KMR+94,PW+95]; their relations are formal since they refer to the underlying semantics and methodology. Moreover, these relationships carry usable information; for example, inheritance of properties between specifications may aid reusability.

Orthogonal to a configuration composed of specification or program components, the most important relation in a development graph is the refinement relation between, say, a requirement specification SP and a design specification SP'. The edge in the graph carries, as specialized information, the incremental development script from SP to SP'; in turn, the script, which is a single application of a transformation rule, or tactic, or a composition of these, relates to verification conditions, their proofs, lemmata that arose during such proofs, and so on. Such information is important for the purposes of certification of a development and is vital for any kind of reuse. In fact, each development step corresponds to a development graph transformation that introduces such items and the relation between them [PW+95].

For development "in the large", it is important to have a system architecture that manages global correctness and integration between all of the heterogeneous components. An interesting observation is that the notions of refinement, development, and their formalization arise for program development, transformation development ("metadevelopment"), and system development (i.e., integration of new tools, updates) in much the same way. Indeed their implementation can be reused at different levels for different users [KPO+95,HK93,KKLT91].

14.5 Conclusions

We have taken a general view of development and considered it as the composition of correctness-preserving refinement steps. Given this view, we have examined how, and to what degree, development can be formalized; this has included rules, compositions of rules, their combination with tactics and the like. We have also presented the kinds of advantages that formalized development offers. In particular, formalization offers not only some assurance of correctness of development calculi themselves, but also a concrete foundation for formally extending them and optimizing their rules.

Although much of this chapter was a survey of relevant ideas and literature, we have tried to use a concrete example of a classical transformation rule (linear to tail recursion) to illustrate many of the principles that we covered. One particular contribution, we hope, is to show that, with the appropriate foundations and support, formal development of rules and their application is not just a desirable goal, but a realistic proposition.

References

[AB95] Penny Anderson and David Basin. Deriving and applying logic program transformers. In *Algorithms, Concurrency and Knowledge (1995 Asian Computing Science Conference)*, volume 1023 of *Lecture Notes in Computer Science*, pages 301–318. Springer, 1995.

[AB96] Abdelwaheb Ayari and David Basin. Generic system support for deductive program development. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 313–328. Springer, 1996.

[ACCL91] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[ACHA90] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William E. Aitken. The semantics of reflected proof. In *Symposium on Logic in Computer Science*. Computer Society Press of the IEEE, 1990.

[AGM92] S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum. *Handbook of Logic in Computer Science, Vols. 1–6*. Clarendon Press, Oxford, 1992.

[AHMP92] Arnon Avron, Furio Honsell, Ian Mason, and Randy Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9:309–352, 1992.

[Avr91] Arnon Avron. Simple consequence relations. *Information and Computation*, 92(1):105–139, 1991.

[Aya95] Abdelwaheb Ayari. A reinterpretation of the deductive tableaux system in higher-order logic. Master's thesis, Universität Saarbrücken, 1995.

[Bas94] David A. Basin. Logic frameworks for logic programs. In *4th Intl. Workshop on Logic Program Synthesis and Transformation (LOPSTR'94)*, volume 883 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1994.

[BC93] David Basin and Robert Constable. Metalogical frameworks. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 1–29. Cambridge University Press, 1993.

[BCMS89] Roland Backhouse, Paul Chisholm, Grant Malcolm, and Erik Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1(1):19–84, 1989.

[BD77] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, 1977.

[BHS89] Hans-Jürgen Bürckert, Alexander Herold, and Manfred Schmidt-Schauß. On equational theories, unification, and decidability. *Journal of Symbolic Computation*, 8:3–50, 1989.

[BM79] Robert S. Boyer and J. Strother Moore, editors. *A Computational Logic*. Academic Press, 1979.

[BW82] Friedrich L. Bauer and Hans Wössner, editors. *Algorithmic Language and Program Development*. Springer, 1982.

[C+86] Robert L. Constable et al., editors. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, 1986.

[CH88] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, pages 95–120, 1988.

[CH90] Robert L. Constable and Douglas J. Howe. Implementing metamathematics as an approach to automatic theorem proving. In *A source book of formal approaches to A.I.* North Holland, 1990.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Symbolic Logic*, 5(1):56–68, 1940.

[CL73] Chin-Liang Chang and Richard Char-Tung Lee, editors. *Symbolic logic and mechanical theorem proving.* Academic Press, 1973.

[Cle92] Tim Clement. Using metavariables in natural deduction proofs. In Cliff B. Jones, Roger C. Shaw, and Tim Denvir, editors, *5th Refinement Workshop*, Workshops in computing, London, pages 255–271. Springer, 1992.

[Coe92] Martin David Coen. Interactive program derivation. Technical Report 272, Cambridge University Computer Laboratory, 1992.

[Coh80] Avra Cohn. *Machine Assisted Proofs of Recursion Implementation.* PhD thesis, University of Edinburgh, 1980.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[dB80] N. G. de Bruijn. A survey of the project Automath. In *Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[DL93] Yves Deville and Kung-Kiu Lau. Logic program synthesis. *Journal of Logic Programming*, 12, 1993.

[Fea82] Martin S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 4:1–20, 1982.

[Fef88] Solomon Feferman. Finitary inductively presented logics. In *Logic Colloquium '88.* North-Holland, 1988.

[FHV92] Ronald Fagin, Joeseph Y. Halpern, and Mosche Y. Vardi. What is an inference rule? *Journal of Symbolic Logic*, 57(3):1018–1045, 1992.

[FM88] Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *9th Intl. Conference on Automated Deduction, Argonne, Illinois (USA)*, 1988.

[Fri85] L. Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *IEEE Symposium on Logic Programming, Boston (MA)*, 1985.

[Gan91] Harald Ganzinger. A completion procedure for conditional equations. *Journal of Symbolic Computation*, 11:51–81, 1991.

[Gar93] Philippa A. Gardner. A new type theory for representing logics. In A. Voronkov, editor, *Proc. 4th Intl. Conference on Logic Programming and Automated Reasoning*, volume 698 of *Lecture Notes on Artificial Intelligence.* Springer, 1993.

[GG94] Dov Gabbay and Franz Guenthner, editors. *What is a Logical System?* Oxford University Press, 1994.

[GM86] Joseph A. Goguen and José Meseguer. EQLOG: Equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 259–363. Prentice Hall, 1986. An earlier version appeared in *Journal of Logic Programming*, 1(2):179–210, 1984.

[GMW79] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science.* Springer, 1979.

[Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der *Principia Mathematica* und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–98, 1931.

[Gre69] Cordell Green. Application of theorem proving to problem solving. In *Intl. Joint Conference on Artificial Intelligence '69*, pages 219–239, 1969.

[GS88] Jean Gallier and Wayne Snyder. Complete sets of transformations for general e-unification. *Theoretical Computer Science*, 67:203–260, 1988.

[GSHH92] Joseph A. Goguen, Andrew Stevens, Keith Hobley, and Hendrik Hilberdink. 2OBJ, a metalogical framework based on equational logic. *Philosophical Transactions of the Royal Society, Series A*, 339:69–86, 1992. Also in *Mechanized Reasoning and Hardware Design*, edited by C.A.R. Hoare and M.J.C. Gordon, Prentice-Hall, 1992, pages 69–86.

[GT93] Fausto Giunchiglia and Paolo Traverso. Reflective reasoning with and between a declarative metatheory and the implementation code. In *Intl. Joint Conference on Artificial Intelligence '91, Sydney (Australia)*, pages 111–117, 1993.

[GTW78] Joseph A. Goguen, James W. Thatcher, and Eric G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In Raymond Yeh, editor, *Current Trends in Programming Methodology, IV*, pages 80–149. Prentice-Hall, 1978. Also as Report RC 6487, IBM T.J. Watson Research Center, 1976.

[HBS92] Jane Hesketh, Alan Bundy, and Alan Smaill. Using middle-out reasoning to transform naive programs into tail recursive ones. In *Proceedings of CADE-11*, 1992.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[HK93] Berthold Hoffmann and Bernd Krieg-Brückner, editors. *Program Development by Specification and Transformation*, volume 680 of *Lecture Notes in Computer Science*. Springer, 1993.

[HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, pages 31–55, 1978.

[How88a] Douglas J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory*. PhD thesis, Cornell University, 1988.

[How88b] Douglas J. Howe. Computational metatheory in Nuprl. In *9th Intl. Conference on Automated Deduction, Argonne, Illinois (USA)*, pages 238–257, 1988.

[How91] Douglas J. Howe. On computational open-endedness in Martin-Löf's type theory. In *Sixth Annual Symposium on Logic in Computer Science, Amsterdam (The Netherlands)*, 1991.

[HRS90] Maritta Heisel, Wolfgang Reif, and Werner Stephan. Tactical theorem proving in program verification. In *10th Intl. Conference on Automated Deduction, Kaiserslautern (Germany)*, pages 117–131, 1990.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.

[Hul80] J.-M. Hullot. Canonical forms and unification. In W. Bibel and R. Kowalski, editors, *Proc. 5th Intl. Conference on Automated Deduction (CADE'80)*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer, 1980.

[KBB93] Ina Kraan, David Basin, and Alan Bundy. Middle-out reasoning for logic program synthesis. In *10th Intl. Conference on Logic Programming (ICLP'93), Budapest (Hungary)*, pages 441–455, 1993.

[KKLT91] B. Krieg-Brückner, E. W. Karlsen, J. Liu, and O. Traynor. The PROSPECTRA methodology and system: Uniform transformational (meta-) development. In S. Prehn and W. J. Toetenel, editors, *VDM, Formal Software Development Methods, Proc. 4th Intl. Symposium of VDM Europe; Vol. 2: Tutorials*, volume 552 of *Lecture Notes in Computer Science*, pages 363–397. Springer, 1991.

[KMR+94] B. Krieg-Brückner, W. Menzel, W. Reif, H. Ruess, Th. Santen, D. Schwier, G. Schellhorn, K. Stenzel, and W. Stephan. System architecture framework for KORSO. In Bernd Krieg-Brückner, editor, *Programmentwicklung durch Spezifikation und Transformation – Bremer Beiträge zum Verbundprojekt KORSO (Korrekte Software)*, Informatik Bericht Nr. 1/94. Universität Bremen, 1994.

[Kot78] Laurent Kott. About a transformation system: A theoretical study. In *3rd Intl. Symposium on Programming, Paris (France)*, pages 232–247, 1978.

[KPO+95] Bernd Krieg-Brückner, Jan Peleska, Ernst-Rüdiger Olderog, D. Balzer, and Alexander Baer. Universelle Entwicklungsumgebung für formale Methoden (uniform workbench). Informatik Bericht Nr. 8/95, Universität Bremen, 1995.

[Kre93] Christoph Kreitz. Meta-synthesis: Deriving programs that develop programs. Technical Report AIDA-93-03, Technische Hochschule Darmstadt, 1993. Habilitation.

[Kri89] B. Krieg-Brückner. Algebraic specification and functionals for transformational program and meta-program development. In J. Diaz and F. Orejas, editors, *Proc. TAPSOFT'89 Vol. 2*, volume 352 of *Lecture Notes in Computer Science*, pages 36–59. Springer, 1989.

[KSZ86] Deepak Kapur, G. Sivakumar, and H. Zhang. RRL: A rewrite rule laboratory. In *8th Intl. Conference on Automated Deduction, Oxford (UK)*, pages 691–692, 1986.

[Les81] J. Leszczyłowski. Theory of FP systems in Edinburgh LCF. In *Formalization of Programming Concepts*, volume 107 of *Lecture Notes in Computer Science*, pages 374–386. Springer, 1981.

[Mar82] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth Intl. Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175. North-Holland, 1982.

[Mar85] Per Martin-Löf. On the meaning of the logical constants and the justifications of the logical laws. Technical report, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Universita di Siena, 1985.

[Mat94] Sean Matthews. A theory and its metatheory in FS_0. In Gabbay and Guenthner [GG94].

[Mil90] Dale Miller. Abstractions in logic programs. In P. Odifreddi, editor, *Logic and Computer Science*, pages 329–359. Academic Press, 1990.

[MM93] Narciso Martí-Oliet and José Meseguer. Rewriting logic as a logical and semantic framework. Technical Report SRI-CSL-93-05, Computer Science Laboratory, SRI International, 1993.

[MM94] Narciso Martí-Oliet and José Meseguer. General logics and logical frameworks. In Gabbay and Guenthner [GG94].

[MSB93] Sean Matthews, Alan Smaill, and David Basin. Experience with FS_0 as a framework theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 61–82. Cambridge University Press, 1993.

[MW80] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*, volume 7 of *The International Series of Monographs on Computer Science*. Oxford Science Publications, 1990.

[Pau83] Lawrence C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[Pau87] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.

[Pau89a] Christine Paulin-Mohring. Inductive definitions in the calculus of constructions. In *The Calculus of Constructions, Documentation and users's guide*. Project Formel, 1989.

[Pau89b] Lawrence C. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[Pau94a] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In *Proc. 12th Intl. Conference on Automated Deduction (CADE-12)*, volume 814 of *Lecture Notes on Artificial Intelligence*. Springer, 1994.

[Pau94b] Lawrence C. Paulson. *Isabelle : a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[PR92] Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In *Proc. 11th Intl. Conference on Automated Deduction (CADE-11)*, pages 537–551. Springer, 1992.

[PW+95] Peter Pepper, Martin Wirsing, et al. Korso, a methodology for the development of correct software. In Manfred Broy and Stefan Jähnichen, editors, *Correct Software by Formal Methods*, Lecture Notes in Computer Science. Springer, 1995.

[San95] David Sands. Correctness of recursion-based automatic program transformations. In *TAPSOFT'95*, volume 915 of *Lecture Notes in Computer Science*, pages 681–965. Springer, 1995.

[Sha88] Natarayan Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the Association for Computing Machinery*, 35(3):475–522, 1988.

[Shi94] Hui Shi. *Extended Matching with Applications to Program Transformation*. PhD thesis, Universität Bremen, 1994.

[Smi91] Doug R. Smith. KIDS – a knowledge-based software development system. In Michael R. Lowry and Robert D. Mcartney, editors, *Automating Software Design*, pages 483–514. AAAI Press/The MIT Press, 1991.

[Sok87] Stefan Sokołowski. Soundness of Hoare's logic: an automatic proof using LCF. *ACM Transactions on Programming Languages and Systems*, 9:100–120, 1987.

[SST92] Donald Sannella, Stefan Sokołowski, and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica*, 29(8):689–736, 1992.

[Tal93] Carolyn Talcott. A theory of binding structures and applications to rewriting. *Theoretical Computer Science*, 112:99–143, 1993.

[Tar36] Alfred Tarski. Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, 1:261–405, 1936.

[Tro82] Anne S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer, 1982.

[WK95] Burkhart Wolff and Kolyang. Development by refinement revisited: Lessons learnt from a case study. In *Proc. Softwaretechnik '95*, Software-Technik Trends. Springer, 1995.