Technical Report

UCAM-CL-TR-179
ISSN 1476-2986

Number 179

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Automating Squiggol

Ursula Martin, Tobias Nipkow

September 1989

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Automating Squiggol

Ursula Martin* and Tobias Nipkow†

Department of Computer Science, RHBNC, University of London
Egham, Surrey TW20 0EX, *UK*
uhm@cs.rhbnc.ac.uk
and
University of Cambridge, Computer Laboratory,
Pembroke Street, Cambridge CB2 3QG, *UK*
tnn@cl.cam.ac.uk

Abstract

The Squiggol style of progam development is shown to be readily automated using LP, an equational reasoning theorem prover. Higher-order functions are handled by currying and the introduction of an application operator. We present an automated verification of Bird's development of the maximum segment sum algorithm, and a similar treatment of a proof of the binomial theorem.

*Research partially supported by ESPRIT BRA grant 3104, PROCOS.
†Research supported by ESPRIT BRA grant 3245, Logical Frameworks.

1 Introduction

The Bird-Meertens calculus "Squiggol" has been developed for deriving functional programs by equational reasoning. LP [6] is an equational reasoning theorem prover developed at MIT by Steve Garland and John Guttag. It supports proofs by induction, rewriting and completion, and is used interactively for designing, coding and debugging proofs. The aim of this paper is to show that LP may be used almost as a pocket calculator for Squiggol verifications. We take as our main example the verification of Bird's derivation [1] of the maximum segment sum algorithm. We present a verification consisting of around 3 pages of input, which closely follows the hand proofs indicated in [1], and executes in just over a minute on a SUN3/60.

We begin with a brief introduction to LP, although we must refer the reader to [6] for a full account. The introduction is illustrated by a proof of the binomial theorem which is itself a simple example of the Squiggol style. Section 3 briefly introduces the logic of term rewriting. Term rewriting is usually associated with first order theorem proving, but in Section 4 we describe how higher order functions are treated equationally by currying. Sections 5 to 7 are deliberately modelled closely on Bird's paper [1], and show how the derivations of Horner's rule and the maximum segment sum algorithm presented there are all easily carried out in LP. In the last section we discuss some of the extensions which would have been useful for this work, and some of the other systems which could have been used.

2 An Introduction to LP and a Sample Proof

In this section we give a brief introduction to equational reasoning and the Larch Prover, using an annotated transcript of the full input, and some of the output, for a proof of the binomial theorem in LP. A full introduction to LP can be found in [6].

LP is based upon equational term-rewriting. That is, given a set of operators and some equations which they satisfy, each equation is oriented into a directed substitution or rewrite rule. The inference mechanism is that of replacing subterms which match the left hand side of a rule by the corresponding right hand side. As well as this proof by rewriting, LP also supports proof by cases and by induction.

Classical term rewriting, and theorem provers such as Reve [13] and RRL [20], have been much concerned with the completion algorithm. We do not use completion at all in our work, a point we return to in the next section.

The session begins by declaring generators for natural numbers and lists. In our formulation, natural numbers are generated by `0` and `s` and lists by the empty list `nil` and the infix cons-operator `&&`:

```
add-generators
0 : -> nat
s : nat -> nat

nil : -> list
&& : nat, list -> list
..
```

The .. indicates end of input. The sole point of this declaration is to establish the structural induction schemata

$$\frac{P(\mathtt{0}) \quad \forall \mathtt{x}.\, P(\mathtt{x}) \Rightarrow P(\mathtt{s(x)})}{P(\mathtt{x})}$$

and

$$\frac{P(\mathtt{nil}) \quad \forall \mathtt{e},\mathtt{z}.\, P(\mathtt{z}) \Rightarrow P(\mathtt{e\ \&\&\ z})}{P(\mathtt{z})}$$

for natural numbers and lists. Since we define all functions by primitive recursion, these are sufficient for our purposes.

We next declare the equations we are going to use. Addition, multiplication and exponentiation are axiomatised as follows

```
add
x+s(y) == s(x+y)
x+0 == x
s(x)*y == (x*y) + y
x*(y+u) == (x*y) + (x*u)
0*x == 0
1 == s(0)
2 == s(1)
(power.s(x)).y == y*((power.x).y)
(power.0).s(x) == 1
..
```

Thus $(1+x)^n$ is represented by `(power.n).s(x)`. Notice that we do not need to declare variable and operator names as we are using the LP default that `uvwxyzUVWXYZ` are variable prefixes and other prefixes denote operators. Here `power` is a constant and the `.` denotes explicit application, so that `(power.x).y` is the curried form of a term which might also be represented by a function with two arguments. We return to this in Section 4.

To use these equations for computation we order them into rules by declaring that they should all be oriented from left to right

```
set order left-to-right
order automatically
```

A term may now be normalised by matching subterms against instances of the rules and rewriting until no more rules apply. The declaration

```
set ac * +
```

ensures that associative-commutative matching is used for `*` and `+`. Thus for example

```
normalize  (power.2).(x+y)
```

returns `(x * x) + (x * y) + (x * y) + (y * y)` . The prove command attempts a proof by rewriting; thus

```
prove (power.2).(x+y) == ((power.2).x) + ((power.2).y) + (2*x*y)
qed
```

returns

```
Conjecture 11
(power.2).(x + y) == (2*x*y) + ((power.2).x) + ((power.2).y)
[] Proved by rewriting.
```

The qed command checks that no more proofs are in progress. It fails if any previous proof attempt has failed, that is if there are any remaining subgoals. It has been placed there because we know the proof succeeds.

The input for an induction proof takes the following form:

```
prove  (power.x).1 == 1 by induction  x nat
qed
```

The first line performs the proof. It is by induction on the variable x of type nat. The typing information selects the structural induction schema for natural numbers. The output from the prove command shown below indicates how the proof is performed.

```
The basis step in an inductive proof of Conjecture 12
    (power . x) . 1 == 1
involves proving the following lemma(s):

12.1: (power . 0) . 1 == 1
      [] Proved by normalization

The induction step in an inductive proof of Conjecture
    (power . x) . 1 == 1
uses the following equation(s) for the induction hypothesis:

Induct.1: (power . c1) . 1 == 1

The system now contains 1 equation and 9 rewrite rules.

Ordered equation Induct.1 into the rewrite rule:
  (power . c1) . s(0) -> s(0)

The system now contains 10 rewrite rules.
```

After the proof the normalised form of (power.x).1 == 1 is added as a new rule ordered from left to right.

To state the binomial theorem we need some further functions.

```
add
(pointsum.nil).z == z
(pointsum.z).nil == z
(pointsum.(x1&&z1)).(x2&&z2) == (x1+x2) &&((pointsum.z1).z2)
```

```
bin.0 == s(0) && nil
bin.(s(x)) == (pointsum.(0&&(bin.x))).(bin.x)
(seq.x).nil == 0
(seq.x).(y&&z) == y + x*((seq.x).z)
..
order automatically
```

The function pointsum adds up two lists elementwise to obtain a third. We use this to define the list of binomial coefficients bin.n, that is $[\binom{n}{0},\ldots,\binom{n}{r},\ldots,\binom{n}{n}]$, using the recursive definition from Pascal's triangle. Finally the function seq.x turns a list of natural numbers $[a_0, a_1, \ldots, a_n]$ into the polynomial $a_0 + a_1 * x + \cdots a_n * x^n$. Thus the binomial theorem, for any element x of any commutative ring and for any natural number n

$$(x+1)^n = \binom{n}{0} + \binom{n}{1} * x + \cdots + \binom{n}{n} * x^n,$$

becomes

```
(power.xn).s(x) == (seq.x).(bin.xn) .
```

To prove the theorem we need a lemma which states that pointsum distributes over seq; this has a straightforward double induction proof

```
prove   (seq.x).((pointsum.z1).z2) == ((seq.x).z1) + ((seq.x).z2)
resume by induction z1 list
resume by induction z2 list
qed
```

The theorem follows by a second induction

```
prove  (power.xn).s(x) ==  (seq.x).(bin.xn)  by induction xn nat
qed
```

The reader may object that defining the binomial coefficients in this way is not entirely honest! However we may also verify using LP that this definition is equivalent to the more usual recursive definition

$$\binom{n+1}{r+1} = \binom{n}{r} + \binom{n}{r+1}.$$

The function pick.n selects the n+1-st element of a list

```
(pick.x).nil == 0
(pick.0).(y&&z) == y
(pick.s(x)).(y&&z) == (pick.x).z
```

and thus the binomial coefficient b(x,y) is defined as

```
b(x,y) == (pick.y).(bin.x)
```

We need to prove that pick distributes over pointsum

```
prove (pick.x).((pointsum.z1).z2) == ((pick.x).z1) + ((pick.x).z2)
resume by induction z1 list
resume by induction z2 list
resume by induction x nat
```

and then the result is proved by rewriting

```
prove b(s(x),s(y)) == b(x,y) + b(x,s(y))
```

Further LP features will be discussed below.

This completes the input for our LP session. When input as a file of commands using the ex *<filename>* command it runs in under 30 seconds on a SUN3/60. However this was not the first draft of our LP proof; as is explained in [6], LP supports a proof style based on designing, coding and debugging where in particular proof attempts which are going to fail will fail quickly. For example our first attempt contained gaps (we had to work out an appropriate lemma on-line), bugs and infelicities (such as a less elegant definition of the sequence of binomial coefficients) which were fairly rapidly eliminated in the course of a hour or so in front of the prover.

We remark that one of the reasons our proof went through so smoothly was that we used the device, borrowed of course from Squiggol, of representing polynomials by a list of their coefficients, rather than working with them directly.

3 The Logic of Term Rewriting

Term-rewriting theory has been much concerned with the completion algorithm, which attempts to find a *complete* set of rules equivalent to a given set of equations, which give a solution to the word problem in the variety the equations define. A set of rules is called *complete* if it is *terminating*, that is no expression can be reduced infinitely often by repeatedly applying the substitutions, and *confluent*, which means that if two distinct expressions can be obtained from a given expression by repeated application of the rules then further applications of the rules can be found to reduce the two expressions to the same thing. Then each term has a unique normal form which can be found by repeatedly applying rules to it until it can be simplified no more, and we can decide whether two expressions are equal as a consequence of the given identities by seeing whether they have the same normal form or not. LP contains an implementation of the completion algorithm, and of several well-founded orderings on terms for proving termination. However obtaining a complete system, even when it is possible, is often computationally expensive and requires time and ingenuity from the user. As proof by rewriting is always sound with respect to equational logic, completion is only called for if a rewrite proof fails. In the few cases this happened to us, it was easier to prove one more equation to make the proof at hand succeed, than to complete the whole system.

A different question is whether LP's logic is suitable for program correctness proofs. The main problem is how to avoid the introduction of inconsistencies. Conceptually, there are two kinds of inconsistencies we would like to distinguish. On the one hand there are those that arise from carelessness, like adding `true == false` or defining both `f(x) == 0` and `f(x) == 1`. On the other hand there is the partial function problem: LP is based on a logic for total functions, and the introduction of truly partial functions can cause inconsistencies. Defining `f(x) == f(x)+1` is a typical example. Notice that termination orderings [4] alone do not protect against partial functions: the previous example would simply be ordered into `f(x)+1 -> f(x)`. Only in conjunction with syntactic constraints similar to those in the Boyer-Moore system [3] or functional programming languages do they enforce totality.

Consistency of complete term-rewriting systems can be proved by checking that certain terms like `true` and `false` or `0` and `1` are not equivalent, i.e. have distinct normal forms. In the inconsistency examples above, the systems are either not confluent or not terminating, depending on the orientation of the rules. Any attempt to complete those systems will eventually lead to the generation of an equality like `true == false` or `0 == 1`.

Our proofs are not accompanied by a formal demonstration of consistency. The reasons are those that lead us to reject completion as an approach to theorem proving. Even for a simple system like the one used for binomial coefficients in Section 2, LP cannot prove termination automatically because its termination orderings are not powerful enough. Assuming termination, LP can prove confluence and thus consistency. Unfortunately, LP's termination and confluence tests do not cover deduction rules since they lie outside the realm of equational logic.

4 Higher-Order Functional Programming in LP

Term rewriting is usually associated with first-order terms. In this section we want to show how functional programming can be emulated by first-order rewriting systems. The key idea is rather trivial: application, which is usually part of the meta-language, is made explicit on the object level by the introduction of an *application operator*. In LP an infix "`.`" is used for that purpose. Thus there are now two ways of defining functions. In addition to the customary `f(x) == ...` we can also write `f.x == ...`. Of course a term `f(a)` matches only the left-hand side of the first rule, and `f.a` only the left-hand side of the second rule. The advantage of the second version is that `f` is now a constant which can be passed as an argument to a function, such as in `f.f`, or be returned as a result, as in `g(x) == f`. It is the latter version that is used almost exclusively throughout this paper. In some cases we use both forms to make things more legible: in addition to the constant `f` representing a binary function, we introduce the infix operator `+` and define `(f.x).y == x+y`. We can now write `(x+y)+z == x+(y+z)` instead of `(f.((f.x).y)).z == (f.x).((f.y).z)`. If `f` is also commutative, we can declare `+` to be an AC operator in LP, which we could not do with `f`.

It should be noted that although the introduction of an apply operator may look like a hack, it merely brings to the surface the internal representation of functional terms in systems like LCF [16] or HOL [8], or in the implementation of functional programming languages [18].

However, one fundamental concept of functional programming is not covered by our extension: λ-abstraction. Fortunately, Squiggol does not use it. Otherwise all occurrences of λ have to be eliminated by a technique called "λ-lifting" [12].

LP offers a simple type system which is not expressive enough for our application.

Therefore we have not used this feature and work in an untyped calculus permitting us to write things like `f.f`. In a polymorphic type system, the type of the application operator would be $Fun(\alpha, \beta) * \alpha \rightarrow \beta$, where Fun is an uninterpreted type constructor. Elements of Fun are introduced either explicitly by declaration or by type inference: writing `f.x == x+1` would force `f` to be of type $Fun(int, int)$.

The rest of the section presents the definition of some common laws and combinators dealing with functions in LP. We start the LP session by defining that all names beginning with "F", "G", "H", or "u" through "z" are variables:

```
set var-prefix FGHuvwxyz
```

Although LP cannot make the distinction, we use "F", "G", and "H" as function variables and "u" through "z" as variables of base type. In addition to function application, we define an infix composition operator `@`:

```
(F @ G).x == F.(G.x)
```

One of the fundamental laws of higher-order calculi is extensionality,

$$(\forall u.\ f(u) = g(u)) \;\Rightarrow\; f = g$$

which goes beyond even conditional equational logic. Fortunately, LP's *deduction rules* can express such axioms:

```
set name ext
add-deduction when (forall u) F.u == G.u yield F == G
```

The `set name` command enables us to refer to the rule following it by `ext`. The logical meaning of the deduction rule is obvious. Operationally it acts like a trigger: if an equation `f.x == g.x` is added to the system, the corresponding rule `f == g` is added automatically. We often prove some theorem of the form `f.x == g.x`, because the argument to `f` and `g` is necessary to unfold their definitions. But after the proof has succeeded, `ext` is triggered, and the more useful rule `f == g` is added, thus reducing the original theorem to a triviality.

This scheme fails if the theorem is of the form `f1.(f2.x) == g1.(g2.x)` because it does not match the premise of `ext`. An explicit instantiation is necessary:

```
instantiate F by F1 @ F2, G by G1 @ G2 in ext
```

adds the deduction rule

```
when (forall u) F1.(F2.u) == G1.(G2.u) yield F1 @ F2 == G1 @ G2
```

A different example of explicit instantiation is the derivation of associativity of `@` from `ext`:

```
instantiate F by (F @ G) @ H, G by F @ (G @ H) in ext
```

triggers the addition of the rule

```
(F @ G) @ H == F @ (G @ H)
```

because the premise of `ext` reduces to an identity.

5 The Theory of Lists

In the next three sections we develop a full verification of Bird's derivation of Horner's rule and the maximum segment sum algorithm. Our proof is very close to that of [1]. Our input consists solely of the definitions used there and the rules for higher order functions from the previous section. We derive lemmas, proved either by structural induction or rewriting, which are essentially the same as Bird's. The main results are obtained in the form of deduction rules, and the derivation of the maximum segment sum is by instantiating deduction rules. We do not give the full input to LP but merely state all the lemmas that were proved.

This section presents a fragment of the theory of lists developed in the first 6 sections of [1]. Without ado, we give the definition of some of the most common combinators on lists:

```
nil++y == y
(x&&z1)++z2 == x&&(z1++z2)
(app.z1).z2 == z1++z2

(map.F).nil == nil
(map.F).(x&&z) == (F.x)&&((map.F).z)

((foldr.F).x).nil == x
((foldr.F).x).(y&&z) == (F.y).(((foldr.F).x).z)

concat == (foldr.app).nil

tails.nil == nil&&nil
tails.(x&&z) == (x&&z)&&(tails.z)

(scanr.F).x == (map.((foldr.F).x))@tails

heads.z == nil&&(hds.z)
hds.nil == nil
hds.(x&&z) == (map.(cons.x)).(heads.z)
(cons.z).x == z&&x
```

The meaning of all these functions should be clear from their names and the defining equations. Most of them are identical to the ones in [1].

The following simple lemmas are all proved by structural induction over z, as are all subsequent results:

```
(z++z2)++z3 == z++(z2++z3)
(map.F).(z++z2) == ((map.F).z)++((map.F).z2)
((map.F)@(map.G)).z == (map.(F@G)).z
((map.F)@concat).z == (concat@(map. (map.F))).z
```

The last two are called *map distributivity* and *map promotion* in [1].

However, not all lemmas are purely equational. Certain transformations depend on properties of the operators involved. For example the law called *fold promotion* in [1] asserts that

```
((foldr.F).xa) @ concat == ((foldr.F).xa) @ (map.((foldr.F).xa))
```

provided F is associative and xa is a left and right-identity for F. Although we can formulate this proposition as a deduction rule, we cannot prove deduction rules in LP. Therefore the proposition is first proved with constants f and a in the presence of the additional rules

```
(f.x).y == x+y
(x+y)+z == x+(y+z)
x+a == x
a+x == x
```

Now we can show that f distributes over app,

```
((foldr.f).x).((app.z).z2) == (((foldr.f).a).z)+(((foldr.f).x).z2)
```

which suffices for the proof of fold promotion to go through:

```
(((foldr.f).a) @ concat).z == (((foldr.f).a) @ (map.((foldr.f).a))).z
```

Having proved this version of fold promotion, the axioms and lemmas involving f, + and a are removed from the system and the deduction rule for fold promotion is added:

```
set name fold_promo
add-deduction-rule
when (forall x,y,z) (F.((F.x).y)).z == (F.x).((F.y).z)
                    (F.x).xa == x
                    (F.xa).x == x
yield ((foldr.F).xa) @ concat == ((foldr.F).xa) @ (map.((foldr.F).xa))
```

The method for proving deduction rules just outlined is applied repeatedly in subsequent sections. However, it is an insecure device as there is no formal connection between the theorem proved and the actual deduction rule added. The ability to prove deduction rules directly appears to be a simple and desirable extension of LP.

Finally we need a lemma known as *fold-scan-fusion*:

```
((foldr.F).x) @ ((scanr.G).y)) ==
fst @ ((foldr.((dot.F).G)).p((F.y).x,y))
```

where fst and dot are defined as

```
fst.p(x,y) == x
(((dot.F).G).x).p(u,v) == p( (F.((G.x).v)).u, (G.x).v )
```

The function p constructs *pairs* and fst projects onto the first component of a pair. In [1] dot is the binary $\odot$ and the dependence on the two functions it combines remains implicit. In our formulation those two functions are the additional arguments F and G.

The fold-scan-fusion law is already quite complex, but its proof requires a further generalisation:

```
((foldr.((dot.F).G)).p((F.y).x,y)).z
== p( (((foldr.F).x) @ ((scanr.G).y)).z, ((foldr.G).y).z )
```

which can be proved by a simple induction on `z`. The actual fold-scan-fusion law is an equational consequence of this generalisation.

6 Horner's Rule

This section corresponds to the section of the same title in Bird [1]. *Horner's rule*

$$\sum_{i=0}^{n} a_i * x^i = a_0 + x * (a_1 + x * (a_2 + x * (\ldots)\ldots),$$

an efficient scheme for the evaluation of polynomials, has the following close relative:

$$\sum_{i=0}^{n} \prod_{j=1}^{i} x_i = 1 + x_1 * (1 + x_2 * (1 + x_3 * (\ldots)\ldots).$$

Bird not only realised that the latter rule can be stated quite succinctly in the theory of lists but also that it holds in many more structures than the real numbers. In the general case, it takes the form

```
((foldr.f).a) @ (map.((foldr.g).xb)) @ heads ==
(foldr.(((hdot.f).g).xb)).xb
```

where hdot is defined as

```
((((hdot.F).G).xb).x).y == (F.xb).((G.x).y)
```

and `f`, `g` and `a` satisfy

```
(f.((g.z).x)).((g.z).y) == (g.z).((f.x).y)
(f.x).a == x
```

This means that `a` is a right-identity for `f`, and `f` distributes over `g` on the left. In the sequel we refer to this Squiggol law as Horner's rule. Its proof proceeds like the one for fold promotion: it is first carried out with constants `f`, `g` and `a` and the requirements as axioms. Afterwards those axioms are deleted and the following version of Horner's rule as a deduction rule is added:

```
when (forall x,y,z) (F.((G.z).x)).((G.z).y) == (G.z).((F.x).y)
                    (F.x).xa == x
yield ((foldr.F).xa) @ (map.((foldr.G).xb)) @ heads ==
      (foldr.(((hdot.F).G).xb)).xb
```

This rule is given the name horner.

The following lemma found in [1], and again proved by list induction,

```
((foldr.f).a).((map.(g.y)).(x&&z)) == (g.y).(((foldr.f).a).(x&&z))
```

is the last stepping stone in the proof of Horner's rule:

```
((foldr.(((hdot.f).g).xb)).xb).z ==
(((foldr.f).a) @ (map.((foldr.g).xb)) @ heads).z
```

Although Bird [1] proofs both propositions by a simple case split on whether z is empty or not, in LP it is easier to use induction.

A surprising application of Horner's rule is presented in the next section.

7 Maximum Segment Sum

In this section we verify Bird's [1] solution to the maximum segment sum problem using Horner's rule. A partial specification of this problem is given by mss:

```
mss == max @ (map.sum) @ segs
max == (foldr.max2).neginf
sum == (foldr.plus).0
segs == concat @ (map.heads) @ tails

(max2.((max2.x).y)).z == (max2.x).((max2.y).z)
(max2.x).neginf == x
(max2.neginf).x == x
(plus.x).((max2.y).z) == (max2.((plus.x).y)).((plus.x).z)
```

Although we have used the suggestive names plus, max2, neginf, and 0, these functions are not specified any further. Only the relationships required for the application of Horner's rule are stated. Hence the theorem we are about to prove is again valid for any interpretation of these four functions meeting the requirements. Over the non-negative reals, product, minimum, 0, and 1 qualify. Further nontrivial applications remain to be discovered.

If plus and max2 are constant time functions, the above executable specification of mss has time complexity $O(n^3)$ where n is the length of the input list. Horner's rule yields a linear algorithm for mss which can actually be computed by LP, provided we give it a few more hints:

```
instantiate F by max2, xa by neginf, G by plus, xb by 0 in horner
instantiate F by max2, xa by neginf in fold_promo
```

The axioms for max2, neginf and plus discharge the hypotheses of both deduction rules and their two conclusions are added as rewrite rules:

```
((foldr . max2) . neginf) @ ((map . ((foldr . plus) . 0)) @ heads)
== (foldr . (((hdot . max2) . plus) . 0)) . 0
((foldr . max2) . neginf) @ (((foldr . app) . nil) @ z)
== ((foldr . max2) . neginf) @ ((map . ((foldr . max2) . neginf)) @ z)
```

Explicit instantiation is required because both deduction rules have more than one premise.

Miraculously, normalising mss yields the expected algorithm:

```
normalize mss
The sequence of term reductions leading to the normal form of the term is:
1.  mss
2.  fst @ ((foldr.((dot.max2).(((hdot.max2).plus).0))).p(0, 0))
```

To recover Bird's formulation, one merely has to notice that the subexpression

```
(dot . max2) . (((hdot . max2) . plus) . 0)
```

is equivalent to $\odot$ as defined in Section 8 of [1].

8 Conclusions

We hope that we have convinced the reader that equational reasoning theorem provers such as LP are easy to use, and support a proof style which is close to that used in the hand verification of Squiggol derivations. Although term rewriting is usually associated with first order theorem proving, we have shown that equational reasoning can be adapted to higher order reasoning by the simple device of currying. Completion and proof of termination are the most difficult parts of term rewriting theorem proving to automate, but they are not necessary in our work. In addition to the two examples presented, we have also verified Bird's and Hughes' [2] development of the α-β algorithm.

We describe now some of the problems we had, and some of the other theorem provers we might have used.

Some Problems

The most annoying problems encountered during the proofs were caused by the absence of λ's and higher-order unification [9], or rather, matching. A typical example is the impossibility of rewriting the term `f @ (g @ h)` with a rule `f @ g == t` because the former is bracketed the wrong way round. It means that we first have to derive the corollary `f @ (g @ H) == t @ H`, which is trivial but tedious. In λ-notation function composition becomes $\lambda F, G, x.F(G(x))$ and higher-order unification matches $\lambda x.f(g(x))$ with a subterm of $\lambda x.f(g(h(x)))$ because it is β-equivalent to $\lambda x.(\lambda x.f(g(x)))(h(x))$.

Although it would cure a symptom, not the cause, the inclusion of associative *matching* in LP would dispose of this problem. LP does currently not support associativity because it grew out of the completion-oriented prover Reve [13] which requires *unification* rather than just matching. Associative unification poses a difficult decision problem [14] and may result in an infinite number of unifiers [19].

A related problem occurs when trying to apply extensionality (see Section 4): the premise `F.u == G.u` does not match the rule `f.x == g.(h.x)`. However, higher-order matching of $F(u) = G(u)$ and $f(x) = g(h(x))$ produces the substitution $\{F \mapsto f, G \mapsto \lambda x.g(h(x))\}$.

The lack of higher-order unification is also partly responsible for our emphasis of program proofs rather than derivations or synthesis. Program transformation in the style of [10] without higher-order matching leads to the same tedious mismatches just discussed.

Another issue that came up during the proofs is *modularity*. Although LP is intended as a prover for the Larch specification language [7], it does currently not support any of Larch's module constructs. Apart from naming specifications and arranging them in a hierarchical fashion, the feature we missed most was theory parametrisation, which, in Larch, is expressed via the "assumes" construct. Deduction rules can serve a similar purpose as parametrised theories. But instead of our unsafe method of proving deduction rules outlined in Section 5, one would work safely in the parameterised theory.

Some other Theorem Provers

LP is by no means the only possible choice of theorem prover for Squiggol. For a glimpse of the variety of systems and features one can choose from, we look briefly at some other systems. In selecting a particular system one has to strike a compromise between expressiveness and ease of use.

RRL [20] is similar to LP, but supports more general forms of induction. Therefore it can handle inductive proofs involving functions that are not defined by primitive recursion, such as Quicksort. On the other hand RRL does not offer deduction rules.

The Boyer-Moore [3] is also based on rewriting. Two of its key features are well-founded induction, which makes it very powerful, and the restriction to total functions. Since the system requires a proof of termination for every function that is introduced, consistency cannot be violated. On the other hand the totality requirement enforces a restricted syntax for function definitions. In particular it is impossible to work with uninterpreted functions `f` and `g` which are only related by some algebraic laws, as in Section 6.

LCF [16] was one of the first theorem provers dedicated to recursive functions, and still is one of the few dealing with partiality. It features full first-order logic, higher-order, partial and even non-strict functions, fixpoint induction, a polymorphic type system, and a powerful meta-language with user definable proof tactics. In principle LCF is ideally suited for reasoning about functional programs in general and Squiggol in particular. However, its rich theory and minimal interface are obstacles in coming to grips with it. In particular one of its main features, the treatment of partiality, complicates reasoning about total functions.

All systems mentioned so far are based on first-order unification. One of the distinguishing features of Isabelle [17] is higher-order unification. In [15] it is shown how Isabelle can be used for the verification and transformation of sorting algorithms, including Quicksort. At the transformation stage higher-order unification plays a major role.

Acknowledgements

The Larch Prover is being developed at MIT by Steve Garland and John Guttag and we are grateful to them for letting us have a copy.

References

[1] R. Bird: *Algebraic Identities for Program Calculation*, The Computer Journal, Vol.

32, No. 2, 1989, 122-126.

[2] R.S. Bird, J. Hughes: *The Alpha-Beta Algorithm: An Exercise in Program Transformation*, Information Processing Letters 24 (1987), 53-57.

[3] R.S. Boyer, J S. Moore: *A Computational Logic Handbook*, Academic Press (1988).

[4] N. Dershowitz: *Termination of Rewriting*, Journal of Symbolic Computation 3 (1987), 69-117.

[5] S.J. Garland, J.V. Guttag: *Inductive Methods for Reasoning about Abstract Data Types*, Proc. 15th ACM Symposium on Principles of Programming Languages (1988), San Diego, 219-228.

[6] S.J. Garland, J.V. Guttag: *An Overview of LP, The Larch Prover*, Proc. Rewriting Techniques and Applications, 3rd Intl. Conference, LNCS 355 (1989), 137-151.

[7] J.V. Guttag, J.J. Horning: *Report on the Larch Shared Language*, Science of Computer Programming 6 (1986), 103-134.

[8] Michael J.C. Gordon: *HOL: A Proof Generating System for Higher-Order Logic*, in: Graham Birtwistle and P.A. Subrahmanyam, editors, VLSI Specification, Verification and Synthesis, Kluwer Academic Publishers (1988), 73-128.

[9] G. Huet: *A Unification Algorithm for the Typed λ-Calculus*, Theoretical Computer Science 1 (1975), 27-57.

[10] G. Huet, B. Lang: *Proving and Applying Program Transformations Expressed with Second-Order Patterns*, Acta Informatica 11 (1978), 31-55.

[11] G. Huet, D.C. Oppen: *Equations and Rewrite Rules - A Survey*, in: Formal Languages: Perspectives and Open Problems, R. Book (ed.), Academic Press (1982).

[12] T. Johnsson: *Lambda Lifting: Transforming Programs to Recursive Equations*, Proc. Conference on Functional Programming Languages and Computer Architecture, LNCS 201 (1985), 190-203.

[13] P. Lescanne: *REVE: A Rewrite Rule Laboratory*, Proc. 8th Intl. Conference on Automated Deduction, LNCS 230 (1986), 695-696.

[14] G.S. Makanin: *The Problem of Solvability of Equations in a Free Semigroup*, Math. USSR Sb. 32 (1977).

[15] T. Nipkow: *Term Rewriting and Beyond – Theorem Proving in Isabelle*, to appear in FACS.

[16] L.C. Paulson: *Logic and Computation*, Cambridge University Press (1987).

[17] L.C. Paulson: *The Foundation of a Generic Theorem Prover*, Journal of Automated Reasoning (1989), in press.

[18] S.L. Peyton Jones: *The Implementation of Functional Programming Languages*, Prentice-Hall International (1987).

[19] G.D. Plotkin: *Building-in Equational Theories*, Machine Intelligence, Vol. 7, Halsted Press (1972), 73-90.

[20] H. Zhang, D. Kapur, M.S. Krishnamoorthy: *A Mechanizable Induction Principle for Equational Specifications*, Proc. 9th Intl. Conference on Automated Deduction, LNCS 310 (1988).