BENL, BERGER, SCHWICHTENBERG, SEISENBERGER, ZUBER

PROOF THEORY AT WORK: PROGRAM DEVELOPMENT IN THE MINLOG SYSTEM

1. INTRODUCTION

The old idea that proofs are in some sense functions, has been made precise by the Curry-Howard-correspondence between proofs in natural deduction and terms in typed λ-calculus. Since the latter can be viewed as an idealized functional programming language, this amounts to an interpretation of proofs as functional programs. This concept and related ones going back to work of Gentzen, Gödel, Kleene and Kreisel are implemented in MINLOG, an interactive proof system designed for generating proof terms and exploring their algorithmic content. Besides tools for interactive proof generation, MINLOG has automatic devices

- to search for purely logical (sub)proofs,
- to check the correctness of a proof,
- to remove detours in a proof,
- to make a nonconstructive proof constructive,
- to read off witnesses from a constructive proof,
- to adapt an already existing proof to special cases,
- to produce a legible verbalization from a formal proof.

The motivation for the development of MINLOG is to use the proofs-as-programs paradigm to let program development go hand-in-hand with program verification. MINLOG's path to correct programs can be divided into three steps:

1. Specification of the desired properties $A[input, output]$.
2. Interactive proof M of $\forall x \exists y A[x,y]$ (supported by automatic proof search).
3. Extraction $[\![\cdot]\!]$ of computational content of M, i.e. a program $[\![M]\!]$ satisfying $A[x,[\![M]\!](x)]$ for all inputs x.

The advantage of this way of composing programs is that once you have checked your proof-checker correct — and this is easy for a small one like MINLOG's — proof theory guarantees (in principle) the correctness of all extracted programs. Furthermore one can regard a proof as an extremely detailed comment on its extracted program.

Since functional programs can be represented directly by terms in the language, MINLOG is also well-suited for program verification.

As MINLOG is based on natural deduction and generates proof terms in a λ-calculus style, it is closely related to implementations of constructive type theory such as Alf, Coq, Lego, NuPrl, or Typelab. In particular it shares many features with the specification and verification environment Typelab described in I.3.15 which – like MINLOG – is tailored for practical use on concrete programming problems. However, whereas Typelab and other systems mainly exploit the rich type theoretic specification language, we have chosen for MINLOG a very simple language (first-order formulas over a simply typed λ-calculus) which needs no 'expert knowledge' to understand. The conceptual simplicity of MINLOG allows the user to quickly access non-trivial proving and programming problems and to use advanced proof theoretic proof manipulation techniques. MINLOG is an open system, where the user may easily add self-defined concepts. Another advantage is that the extracted programs are just simply typed λ-terms with higher type primitive recursion. In systems like NuPrl or Coq proofs and programs are not separated so clearly and in particular programs may have complicated dependent types. A system where proofs and programs are clearly distinguished is PX (Hayashi, 1990) which is based on Feferman's untyped theory of functions and classes. However, in PX arbitrary partial recursive programs may be extracted whereas in MINLOG programs extracted from proofs are always total.

We describe the logical background of MINLOG's kernel in section 2 and its concrete realization in section 3. Further important features will be explained by means of three non-trivial examples:

In our first example coming from practice (section 4) we will use MINLOG to interactively verify a train scheduler. Nondeterminism will lead us to higher order terms. Also we will see the use of an error object, introducing the notion of partial objects.

The next example (section 5) is well-known in computer science: quicksort. Here we see the proofs-as-programs paradigm in a concrete case. The algorithm is *extracted* from a proof of its specification. We will also see how to introduce inductive data types as an extension to MINLOG.

The final example about search in finite trees (section 6) shows two techniques of very-high-level program development:

The fact, that a finite tree fulfilling a property always has a minimal subtree fulfilling it too, has a trivial classical proof. MINLOG is able to automatically turn it into a constructive proof (using a refined A-translation) and to extract a search algorithm.

Finally, if we have some additional information on the data (and hence enrich the specification), we can reuse and shorten the proof, resulting in a different (possibly more efficient) algorithm which in general cannot be obtained by optimizing the original program.

2. Logic for computable functionals

The theory implemented in MINLOG is Heyting arithmetic in finite types (Troelstra and van Dalen, 1988), arranged around a kernel of minimal logic. Classical arithmetic is contained in it as the fragment without the constructive quantifier $\exists^*$. From the classical point of view this fragment is sufficient since existential statements can be expressed via $\neg\forall\neg$.

2.1. *The term-calculus*

As objects of computational interest we choose the terms of Gödel's System T, coming with types (even if the type information is mostly suppressed in this text). Besides the natural numbers, the booleans are predefined ground types.

Types. $\mathsf{boole} \mid \mathsf{nat} \mid \rho\times\sigma \mid \rho\to\sigma.$

Constants. $\mathsf{true} \mid \mathsf{false} \mid 0 \mid \mathsf{S} \mid \mathsf{R}_{\mathsf{boole},\rho} \mid \mathsf{R}_{\mathsf{nat},\rho}.$

The successor S is of type $\mathsf{nat}\to\mathsf{nat}$, the types of the recursors (for each type ρ) will be made precise later. From typed variables x and constants c terms are built according to the standard of simply typed λ-calculus.

Terms. $x \mid c \mid \langle r,s\rangle \mid \pi_0 r \mid \pi_1 r \mid \lambda x r \mid rs.$

General bracketing conventions: products associate to the left, arrows to the right. The scope of a binder is the maximal (minimal) possible one, whenever the bound variable is (not) followed by a dot.

Terms are computed according to the

Conversions.

$$
\begin{array}{rcl}
\pi_0\langle r,s\rangle & \mapsto & r,\\
\pi_1\langle r,s\rangle & \mapsto & s,\\
(\lambda x r)s & \mapsto & r_x[s],\\
\mathsf{R}_{\mathsf{boole},\rho} rs\,\mathsf{true} & \mapsto & r,\\
\mathsf{R}_{\mathsf{boole},\rho} rs\,\mathsf{false} & \mapsto & s,\\
\mathsf{R}_{\mathsf{nat},\rho} rs0 & \mapsto & r,\\
\mathsf{R}_{\mathsf{nat},\rho} rs(\mathsf{S}t) & \mapsto & st(\mathsf{R}_{\mathsf{nat},\rho} rst),
\end{array}
$$

where $r_x[s]$ (or $r[s]$ for short) means s substituted for x in r. These rules show that $\mathsf{R}_{\mathsf{boole},\rho}rst$ (r, s of type ρ) corresponds to **if** t **then** r **else** s, whereas $\mathsf{R}_{\mathsf{nat},\rho}rst$ is the usual recursor with base r (of type ρ) and loop s (of type $\mathsf{nat} \to \rho \to \rho$) to be passed through t-times. The first three rules are called β-conversions, the others we call R-conversions.

As is well known (Tait, 1967; Troelstra and van Dalen, 1988), the rewriting procedure stops for each term with a unique *normal form*. Hence the congruence given by the conversions is decidable. In our setting, congruent terms are identified. This *Poincaré-equality* mostly saves us from needing an equality-calculus. (However the user importing self-defined conversions is responsible for their termination together with β and R!)

2.2. *The logical framework*

Now the basis is ready to introduce the objects of interest for theorem proving: formulas. It is crucial but nevertheless easy to distinguish between a term t of type boole and the atomic formula $\mathsf{atom}(t)$ saying 't is true'. In particular we have the atomic formulas

$$\begin{aligned} \top &:= \mathsf{atom}(\mathsf{true}), \\ \bot &:= \mathsf{atom}(\mathsf{false}). \end{aligned}$$

From the atomic formulas (which are decidable since their characteristic terms are computable) we build the

Formulas. $\mathsf{atom}(t) \mid A \wedge B \mid A \to B \mid \forall x A \mid \exists^* x A$,

where $\exists^*$ is the *strong* or *constructive* existential quantifier — to distinguish from the classical $\exists$ which can be viewed as defined by

$$\begin{aligned} \neg A &:= A \to \bot, \\ \exists x A &:= \neg \forall x \neg A. \end{aligned}$$

Formulas without $\exists^*$ are called *classical*. For the moment we will concentrate on the classical part. We will meet $\exists^*$ later again.

In the spirit of the Curry-Howard-correspondence we regard proofs in (the $\wedge, \to, \forall$-fragment of) natural deduction as terms in a typed λ-calculus, where the type of a proof term corresponds to (more precisely: *is*) its end formula in natural deduction, and the types of its free variables *are* its free assumptions.

For the sake of clarity we annotate types as superscripts in the definition of proofs. To start with, an assumption A is introduced by an assumption-variable u^A, whereas an axiom A is introduced by a constant c^A. For the moment our only constant is the *truth axiom* $\mathsf{Truth}^{\top}$.

Proofs. $u^A \mid c^A \mid \langle M^A, N^B\rangle^{A\wedge B} \mid (\pi_0 M^{A\wedge B})^A \mid (\pi_1 M^{A\wedge B})^B \mid$
$(\lambda u^A M^B)^{A\to B} \mid (M^{A\to B} N^A)^B \mid (\lambda x^\rho M^A)^{\forall x^\rho A} \mid (M^{\forall x^\rho A} t^\rho)^{A_x[t]}$.

The usual restriction on the object-variable x applies when building $\lambda x M$.

2.3. *Reasoning in classical and intuitionistic arithmetic*

As we have said nothing special about $\bot$ so far, we have a calculus for minimal logic. To expand it to arithmetic and incorporate constructivity we add

Induction axioms.

$$\mathsf{Ind}_{\forall x^{\mathsf{boole}} A} : A_x[\mathsf{true}] \to A_x[\mathsf{false}] \to \forall x A,$$
$$\mathsf{Ind}_{\forall x^{\mathsf{nat}} A} : A_x[0] \to \forall x(A \to A_x[\mathsf{S}x]) \to \forall x A.$$

$\exists^*$ *axioms.*

$$\exists^{*+}_{x,A} : \forall x. A \to \exists^* x A,$$
$$\exists^{*-}_{x,A,B} : \exists^* x A \to \forall x(A \to B) \to B,$$

where in the latter x must not occur free in B. The notation $M{:}A$ is synonymous to M^A.

We formulated induction and existence as axioms since we regard them as extensions of our logical kernel. You might miss negation axioms. But they are already present:

Theorem. (*ex-falso-quodlibet.*) $\bot \to A$ is provable.

Proof. By induction on A. *Base.* We have to show $\bot \to \mathsf{atom}(t)$. This is an instance of $\forall x. \bot \to \mathsf{atom}(x)$. To prove the latter, we use the boolean induction axiom $\mathsf{Ind}_{\forall x. \bot \to \mathsf{atom}(x)}$. It forces us to prove both, $\bot \to \top$ and $\bot \to \bot$. Using an assumption u of type $\bot$, proofs are simply $\lambda u \mathsf{Truth}$ resp. $\lambda u u$. Summing up, $\mathsf{Ind}_{\forall x. \bot \to \mathsf{atom}(x)} \lambda u \mathsf{Truth}\, \lambda u u t$ is a proof of $\bot \to \mathsf{atom}(t)$. *Step.* By induction hypothesis we have a proof M of type $\bot \to A$. We get $\bot \to \exists^* x A$ as the type of $\lambda u. \exists^{*+}_{x,A} x(Mu)$. The other cases of the induction step are easy, too.

A similar consideration shows that *stability* $\neg\neg A \to A$ is provable for all classical formulas A. Nevertheless we often use ex-falso-quodlibet and stability as axioms in order to avoid myriads of boolean inductions.

Summing up, with the appropriate choice of a term calculus we get Heyting arithmetic, and restriction to the classical fragment yields Peano arithmetic.

2.4. *Semantics*

The denotational semantics kept in the design of MINLOG is Scott's domain theory (Scott, 1982). A variable is supposed to denote a partial continuous functional of its type, whereas a program constant is interpreted by a computable functional. This is reflected in our formalism where we can work with possibly undefined objects (cf. section 4). Since in some applications no partial objects appear, we in addition allow a second kind of variables ranging only over the subclass of total functionals. For the theory of totality in higher types we refer to (Berger, 1993b). Computability of the constants implies that atomic formulas are decidable and extracted programs are executable (See also section 5).

3. THE MINLOG SYSTEM

The MINLOG system is implemented in SCHEME, a LISP-dialect. Its size is about 1/2 MB, which is rather small compared with other theorem provers. It is an interactive system which however has quite elaborate components for automated proof search and term rewriting.

3.1. *Interactive theorem proving*

For proving a theorem in MINLOG the user first has to specify the framework consisting of a language and axioms. This can be done by declaring new variables and program constants of possibly user-defined types. We allow constants parametric in types and, corresponding to that, axioms parametric in formulas. Canonical examples are the recursion operator and the induction scheme. Further axioms or lemmas to be proven later can be introduced as so-called global assumptions. The user may take anything (hopefully valid) for granted; as long as a global assumption contains no computational information, it will not affect program extraction—as we will see in section 5.

In the sequel we want to give an idea how to work with the MINLOG system. Let us begin with a small example:

$$(\forall x.\, A[x] \to B[x]) \to \forall x A[x] \to \forall x B[x].$$

We first present a natural deduction proof of this formula in familiar tree notation:

$$\lambda u\,\cfrac{\lambda v\,\cfrac{\lambda x\,\cfrac{\cfrac{\cfrac{u:\forall x.A[x]\to B[x] \qquad x}{A[x]\to B[x]} \qquad \cfrac{v:\forall xA[x] \qquad x}{A[x]}}{B[x]}}{\forall xB[x]}}{\forall xA[x]\to\forall xB[x]}}{(\forall x.A[x]\to B[x])\to\forall xA[x]\to\forall xB[x]}$$

Below the protocol of the interactive proof is shown. The lines in parentheses are the commands given by the MINLOG user; those starting with a semicolon are the responses of the system, e.g. telling us current goals and actual contexts (some responses are dropped here).

We start with introducing an arbitrary type `arb` and declare a variable x of type `arb` as well as two program constants a and b of type `arb->boole`, displayed with capital letters, representing the formulas $A[x]$ and $B[x]$. The numerical argument 0 in the declaration of the program constants determines their degree of totality. This degree has impact on the execution of possible term rewriting discussed below.

After stating the goal the proof is built up in a backward reasoning fashion. For conciseness, the system displays the atomic formula $\mathsf{atom}(t)$ by the boolean term t (since it is clear from the context that a formula is meant). Hence we read `(A x)` for the internal `(atom (a x))`.

The symbols `?`, `?-KERNEL`, `?1` are labels to identify the (sub)goals. In I.3.15 the role of these 'metavariables', which are an official part of the formal language of Typelab, is thoroughly investigated, especially with respect to the problems arising in conjunction with β-reduction. In MINLOG, however, these problems do not occur, since we do not allow normalization for incomplete proofs.

With the command `assume` introduction rules for implication and universal quantification can be treated (backwards) in one step; the user only has to choose some new labels for the assumptions. The command `use-with` allows to prove the subgoal `?-KERNEL:` $B[x]$ by using the assumption u at x provided one can find a proof for the premise $A[x]$. This is the new goal `?1`, which of course can be proved with a further application of an elimination rule.

```
(add-ground-type 'arb 'x)
(add-program-constant 'a (cons-arrow 'arb 'boole) 0 "A")
(add-program-constant 'b (cons-arrow 'arb 'boole) 0 "B")

(set-goal '? (parse-formula "(all x (a x -> b x))  ->
                              all x (a x)  ->  all x (b x)"))
```

```
;?:(all x.A x -> B x) -> all x A x -> all x B x

(assume 'u 'v 'x)
;ok, under these assumptions we have the new goal
;?-KERNEL: B x from
;  u:all x.A x -> B x
;  v:all x A x
;  x

(use-with 'u 'x '?1)
;ok, ?-KERNEL can be obtained from
;?1: A x from
;  u:all x.A x -> B x
;  v:all x A x
;  x

(use-with 'v 'x)
;ok, ?1 is proved. Proof finished.
```

Now let us have a look at the proof term corresponding to that proof. It was generated interactively by editing an incomplete derivation term with holes in it which are the open goals. In our example the (partial) proof term develops as follows:

$$\begin{array}{l} \texttt{?} \\ \lambda u,v,x.\ \texttt{?-KERNEL} \\ \lambda u,v,x.\ ux\,\texttt{?1} \\ \lambda u,v,x.\ ux(vx). \end{array}$$

There are also some tools for forward reasoning. For instance, instead of the last two steps in the interactive proof, it is also possible to first instantiate the assumption v with x. Then only the context changes since we have a new assumption $v_i{:}A(x)$. To finish the proof we can use u at x and v_i. This different proof leads to different (partial) proof terms developing as follows:

$$\begin{array}{l} \texttt{?} \\ \lambda u,v,x.\ \texttt{?-KERNEL} \\ \lambda u,v,x.\ (\lambda v_i.\ \texttt{?-KERNEL-INST})(vx) \\ \lambda u,v,x.\ (\lambda v_i.\ uxv_i)(vx). \end{array}$$

Note that this proof term can be β-normalized to the proof term above (see subsection 3.4).

3.2. *Automated proof-search*

Of course our first example was an easy one that only deals with logic. A useful proof system should be able to prove such a formula without human interaction:

```
(set-goal '? (parse-formula "(all x (a x -> b x))  ->
                             all x (a x)  ->  all x (b x)"))
;?:(all x.A x -> B x) -> all x A x -> all x B x

(search)
;ok, ? is proved by minimal quantifier logic.
;Proof finished.
```

The automatically generated proof term, $\lambda u_{01}, u_{02}, x.\, u_{01}x(u_{02}x)$, is equal modulo bound renaming to the former one. It is obtained by breadth-first proof-search in the style of (Miller, 1991). Usually, this device is applied to subgoals provable by pure predicate logic (including the use of global assumptions).

3.3. *Proof presentation*

A proof term can be used to automatically produce a legible presentation, i.e. a TEX-script:

> **Proposition.** $(\forall x.A(x) \to B(x)) \to \forall x A(x) \to \forall x B(x)$.
>
> *Proof.* For $(\forall x.A(x) \to B(x)) \to \forall x A(x) \to \forall x B(x)$ assume $\forall x.A(x) \to B(x)$ [1], $\forall x A(x)$ [2] and x. Then we have to show $B(x)$. We derive this from $A(x) \to B(x)$ (as an instance of [1]) and have to show $A(x)$. That is an instance of [2].

This purely logical example was chosen to demonstrate the Curry-Howard-correspondence and the interactive use of the logical rules. MINLOG's emphasis, however, lies on concrete mathematical problems; data types and constants should have a fixed denotational semantics expressed operationally by suitable higher type rewriting rules.

3.4. *Normalization-by-evaluation*

The heart of MINLOG is the normalization-by-evaluation mechanism providing an efficient implementation of the operational semantics. Since terms are represented by SCHEME expressions it is quite natural to use SCHEME evaluation instead of explicitly programming a normalization procedure (Berger and Schwichtenberg, 1991; Berger, 1993a; Berger et al., 1998). Terms are normalized with respect to βR-conversion and additional higher order rewrite rules given by the user. Correctness and termination of the additional rules are under control of the user. The problem of termination and confluence of higher type rewrite systems has been analyzed e.g. in (van de Pol and Schwichtenberg, 1995; Nipkow, 1993).

The effect of term rewriting is shown in a second small example. We prove

$$\forall n\, \mathsf{even}(2{*}n)$$

where odd and even are defined as usual

$$\begin{array}{ll} \mathsf{odd}(0) = \mathsf{false} & \mathsf{even}(0) = \mathsf{true} \\ \mathsf{odd}(n{+}1) = \mathsf{even}(n) & \mathsf{even}(n{+}1) = \mathsf{odd}(n) \end{array}$$

by induction on n. The base case $\mathsf{even}(2{*}0)$, to be proven first, normalizes to the truth axiom. In the step case we also use term rewriting when applying the command `use-with`.

```
(set-goal '? (parse-formula "all n even(2*n)"))

(ind)
;ok, ? can be obtained from
;?-STEP: all n.even(2*n) -> even(2*(n+1))
;?-BASE: even(2*0)

(normalize-goal)
;ok, the normalized goal is
;?-BASE-NF: T

(prop)
;ok, ?-BASE-NF is proved by minimal propositional logic.
;The active goal now is
;?-STEP: all n.even(2*n) -> even(2*(n+1))

(assume 'n 'ih)
;ok, under these assumptions we have the new goal
;?-STEP-KERNEL: even(2*(n+1)) from
;  n  ih:even(2*n)

(use-with 'ih)
;ok, ?-STEP-KERNEL is proved. Proof finished.
```

Step by step we get the following (partial) proof terms:

$$?$$
$$\text{nat-ind-at}_{\forall n\, \mathsf{even}(n)}\ \texttt{?-BASE}\ \texttt{?-STEP}$$
$$\text{nat-ind-at}_{\forall n\, \mathsf{even}(n)}\ \mathsf{Truth}^{\top}\ (\lambda n, \mathrm{ih}.\, \mathrm{ih})$$

where nat-ind-at is the constant representing the induction axiom. Note that although the rewrite steps are not shown in the proof term, they are visible ($\searrow^{*}\swarrow$) in the automatically generated TEX-script.

Proposition. $\forall n\, \mathbf{even}(2 * n)$.

Proof. $\forall n\, \mathbf{even}(2 * n)$ is shown by induction on n. The base case follows from the axiom of truth where $\mathbf{even}(2 * 0) \searrow^{*}\swarrow \top$. Step case: Assume n and $\mathbf{even}(2 * n)$ [1] and use $\mathbf{even}(2 * n)$ [1] where $\mathbf{even}(2 * (n + 1)) \searrow^{*}\swarrow \mathbf{even}(2 * n)$.

Now we have described the basic system. The following sections are devoted to MINLOG's additional features and examples.

4. HIGHER TYPES FOR PROGRAM VERIFICATION: SCHEDULING

As a first example of how to prove something with the MINLOG system we pick a simple but somewhat practical one: a safety property for a distributed system. The example is mathematically quite trivial; however, it hopefully makes clear that in a similar way one can deal with more complex (and more realistic) examples. In particular, we want to demonstrate the following points.

1. Nondeterminism can be described using a choice function (or scheduler) as a parameter (as was done in (Boyer and Moore, 1988, p. 87) in a first order context).
2. To make proofs manageable it is essential to use rewrite rules to take care of the computational parts.
3. It is useful to give explicit definitions (as primitive recursive functionals) wherever possible. This turns many of the proof obligations into computational tasks (cf. 2).

4.1. *The problem*

Consider a track divided into segments, with an arbitrary number of trains moving back and forth. For simplicity let us assume that the track is linear and infinite, so the segments can be modelled by the integers. Trains can move in both directions, but they can enter another segment only if it is empty, i.e. not occupied by another train. We want to give a formal proof of a very simple safety property of this system, namely that no train can ever pass another one.

The system we want to describe is nondeterministic: at any time every train can try to move in any direction. We choose an interleaving model to describe the situation, i.e. we assume that the actions take place one after the other. To prove safety properties we do not loose anything by this view.

We model the nondeterministic system using a choice function (or scheduler) as a parameter. This choice function determines at every moment which action is to be taken. Hence the behaviour of the system is uniquely determined once the choice function is given. The description of the system then consists of giving a definition of the state of each agent by primitive recursion over the time (modelled discretely by the natural numbers), with the choice function as a parameter.

It is convenient to split up the choice function into two components: a function giving the active segment, and a function telling the direction in which an activated train can try to move.

4.2. *The language*

An appropriate language has four ground types (sorts), with variables

m,n	for natural numbers (used to model discrete time),
p,q	for booleans,
i	for segments,
j	for trains.

The predecessor and successor functions for segments are denoted by `prev` and `next`. We also use $=$, $<$, $\leq$ for segments.

It will be useful to allow an error object `undef` (read 'undefined') for every ground type. We use variables with a hat (i.e. $\hat{n}$, $\hat{p}$, $\hat{\imath}$, $\hat{\jmath}$) to range over arbitrary objects of their type, including `undef`. Variables without hat range over defined (or total) objects (cf. section 2).

Parameters
The system is determined by its state at time 0, given by the functions $\mathsf{initocc}\colon\mathsf{segment} \to \mathsf{train}$ resp. $\mathsf{initloc}\colon\mathsf{train} \to \mathsf{segment}$ describing the initial distribution of trains to the track, together with the two parts of the choice function: act (for active-segment) and back (or, if you prefer, forth), a boolean valued function: $\mathsf{back}(n)$ telling whether the direction to move at time n is 'back'. For notational convenience we will suppress these four parameters.

Occupancy
Our first perspective is standing at a certain segment of the track, looking back and forth, waiting for a train. The program constant occ (for occupancy) yields for given (parameters,) time n and segment i the train occupying this segment at this time. If there is no train in the segment, the value is to be undefined. We want to define occ in such a way that the following property holds: *A train leaves the actual segment in the actual direction or will reach a segment moving in the actual direction from its neighbour being actual, only if its goal segment is not occupied; and nothing happens besides.* Together with the initialization this results in six clauses, in which $\mathsf{def}(\mathsf{occ}(n,i))$ means that at time n, there is a train that occupies segment i, and $\mathsf{occ}(n+1,i) = \mathsf{undef}^{\mathsf{train}}$ means that at time $n+1$, no train occupies segment i.

$$
\begin{aligned}
&\mathsf{occ}(0,i) = \mathsf{initocc}(i),\\
&\mathsf{back}(n) \to \mathsf{act}(n) = i \to \neg\mathsf{def}(\mathsf{occ}(n,\mathsf{prev}(i))) \to\\
&\quad \mathsf{occ}(n+1,i) = \mathsf{undef}^{\mathsf{train}},\\
&\mathsf{back}(n) \to \mathsf{act}(n) = \mathsf{next}(i) \to \neg\mathsf{def}(\mathsf{occ}(n,i)) \to\\
&\quad \mathsf{occ}(n+1,i) = \mathsf{occ}(n,\mathsf{next}(i)),\\
&\mathsf{back}(n) \to \mathsf{act}(n) = i \to \mathsf{def}(\mathsf{occ}(n,\mathsf{prev}(i))) \to\\
&\quad \mathsf{occ}(n+1,i) = \mathsf{occ}(n,i),\\
&\mathsf{back}(n) \to \mathsf{act}(n) = \mathsf{next}(i) \to \mathsf{def}(\mathsf{occ}(n,i)) \to\\
&\quad \mathsf{occ}(n+1,i) = \mathsf{occ}(n,i),\\
&\mathsf{back}(n) \to \mathsf{act}(n) \neq \mathsf{next}(i) \to \mathsf{act}(n) \neq i \to\\
&\quad \mathsf{occ}(n+1,i) = \mathsf{occ}(n,i).
\end{aligned}
$$

The clauses for the other direction are similar. So we define occ accordingly, by primitive recursion using the above case distinctions. The recursion equations can then be used as rewrite rules.

Location
Now let us assume, we are sitting in a train, looking where we move. The program constant loc (for location) yields for given (parameters,) time n and

train j the segment where the train is located at this time. For loc we require the following property: *The train being in the active segment moves one segment in the actual direction only if this segment is not* occ*upied(!), and nothing happens besides.* In clauses:

$$\begin{aligned}
&\mathsf{loc}(0,j) = \mathsf{initloc}(j),\\
&\mathsf{act}(n) = \mathsf{loc}(n,j) \to \mathsf{back}(n) \to \neg\mathsf{def}(\mathsf{occ}(n,\mathsf{prev}(\mathsf{loc}(n,j)))) \to\\
&\quad \mathsf{loc}(n+1,j) = \mathsf{prev}(\mathsf{loc}(n,j)),\\
&\mathsf{act}(n) = \mathsf{loc}(n,j) \to \mathsf{back}(n) \to \mathsf{def}(\mathsf{occ}(n,\mathsf{prev}(\mathsf{loc}(n,j)))) \to\\
&\quad \mathsf{loc}(n+1,j) = \mathsf{loc}(n,j),\\
&\mathsf{act}(n) \neq \mathsf{loc}(n,j) \to \mathsf{loc}(n+1,j) = \mathsf{loc}(n,j).
\end{aligned}$$

The clauses for the other direction are similar; we define loc accordingly, by primitive recursion.

There is one important point to be made here. The occ-function and the whole first perspective could have been avoided if we had used an existential quantifier in the equations above, as follows:

$$\begin{aligned}
&\mathsf{act}(n) = \mathsf{loc}(n,j) \to \mathsf{back}(n) \to \neg\exists j_1 \mathsf{loc}(n,j_1) = \mathsf{prev}(\mathsf{loc}(n,j)) \to\\
&\quad \mathsf{loc}(n+1,j) = \mathsf{prev}(\mathsf{loc}(n,j)),\\
&\mathsf{act}(n) = \mathsf{loc}(n,j) \to \mathsf{back}(n) \to \exists j_1 \mathsf{loc}(n,j_1) = \mathsf{prev}(\mathsf{loc}(n,j)) \to\\
&\quad \mathsf{loc}(n+1,j) = \mathsf{loc}(n,j).
\end{aligned}$$

However, since we want to shift as much as possible of the burden of proving something to our rewrite apparatus, we make this existential quantifier explicit by introducing an appropriate function. This is a general pattern: whenever in a given problem there is a functional dependency of some data from others, introduce a function to describe this dependency in an explicit form. This function can then be used in a rewriting mechanism, which generally makes proving much simpler.

4.3. *The specification and its proof*

Now we can formulate the safety property we want to prove, namely that no train can ever pass another one:

$$\begin{aligned}
&\mathsf{initocc}(\mathsf{initloc}(j_1)) = j_1 \to \mathsf{initocc}(\mathsf{initloc}(j_2)) = j_2 \to\\
&\mathsf{initloc}(j_1) < \mathsf{initloc}(j_2) \to \forall n\, \mathsf{loc}(n,j_1) < \mathsf{loc}(n,j_2).
\end{aligned}$$

Rather than describing the interactive construction of the proof in the MINLOG system, we restrict ourselves to some hints: The proof is by induction on n. In order to keep it comprehensible, it is crucial to think of appropriate lemmas first. In the present example, beside some basic arithmetical facts, the following lemmas are proved separately and then used:

$$\begin{aligned}
&\mathsf{prev}(\mathsf{loc}(n,j)) \leq \mathsf{loc}(n+1,j),\\
&\mathsf{loc}(n+1,j) \leq \mathsf{next}(\mathsf{loc}(n,j)),\\
&\mathsf{back}(n) \to \mathsf{loc}(n+1,j) \leq \mathsf{loc}(n,j),\\
&\neg\mathsf{back}(n) \to \mathsf{loc}(n,j) \leq \mathsf{loc}(n+1,j),\\
&\mathsf{initocc}(\mathsf{initloc}(j)) = j \to \mathsf{occ}(n,\mathsf{loc}(n,j)) = j.
\end{aligned}$$

The size of the proof term finally obtained is about two pages, and the effort to generate it (without using any automated search) was 42 interactions. The amount for each lemma was approximately the same.

4.4. *Final remarks on the scheduling example*

1. The constants occ and loc are defined by primitive recursion with function parameters, i.e. within Gödel's system T. Since we only have used the recursion equations as rewrite rules, termination as well as preservation of the values is guaranteed.
2. The possibility to use e.g. i as well as $\hat{\imath}$ provides additional expressive power, which can be quite useful: cf. the use of $\mathsf{def}(\mathsf{occ}(n,i))$ for 'is occupied by a (real) train'.
3. The ground type $\mathsf{segment}$ was described axiomatically, by means of properties of prev and next like $\mathsf{prev}(\mathsf{next}(i)) = i$. If instead we have a fixed ground type like the integers for segments and constants for particular segments, we obtain an executable specification. It can be executed for completely concrete data, but also for partially given ones (partial evaluation).

5. Program extraction Quicksort

In the previous section, we showed how MINLOG can be used to verify that a given program (loc) meets its specification. Now we will explain how to obtain correct programs in MINLOG using program extraction. From a constructive proof one can read off a witness, i.e. a term that realizes the proven formula. First, we will explain what it means exactly that a term (a program) realizes a formula (we use the modified realizability interpretation (Kreisel, 1959)) and how a realizing term can be obtained. Then we will use this technique to get the well-known quicksort algorithm. This example also shows another extension of the basic MINLOG system at work: the Simultaneous Free Algebras. The SFA implementation allows the user to add inductively defined data types (in this example lists of natural numbers) as new ground types to the system and automatically provides the corresponding induction schemes and recursors, as well as an interface for the (otherwise fairly complicated) creation of functionals defined by recursion over this type. Similar considerations concerning program extraction and its application to sorting problems can already be found in (Turner, 1991). In the present section we want to show that such a task can be carried out rather elegantly in the MINLOG system.

5.1. *Program extraction*

A *Harrop formula* is a formula not containing the constructive existential quantifier $\exists^*$ strictly positive. In the following let A, B stand for Harrop formulas and C, D for non–Harrop formulas. First of all we define the type $\tau(C)$ of potential realizers of C:

$$\begin{array}{rcl}
\tau(\exists^* x^\rho A) & := & \rho, \\
\tau(\exists^* x^\rho C) & := & \rho \times \tau(C), \\
\tau(\forall x^\rho C) & := & \rho \to \tau(C), \\
\tau(A \to C) & := & \tau(C), \\
\tau(D \to C) & := & \tau(D) \to \tau(C), \\
\tau(A \wedge C) & := & \tau(C), \\
\tau(C \wedge A) & := & \tau(C), \\
\tau(C \wedge D) & := & \tau(C) \times \tau(D).
\end{array}$$

Now we can define the *modified realizability interpretation* $t^{\tau(C)}\, \mathbf{mr}\, C$ ('t realizes C') and $\varepsilon\, \mathbf{mr}\, A$:

$$\begin{array}{rcl}
t\, \mathbf{mr}\, \exists^* x A & := & \varepsilon\, \mathbf{mr}\, A_x[t], \\
t\, \mathbf{mr}\, \exists^* x C & := & \pi_1 t\, \mathbf{mr}\, C_x[\pi_0 t], \\
\varepsilon\, \mathbf{mr}\, \forall x A & := & \forall x.\, \varepsilon\, \mathbf{mr}\, A, \\
t\, \mathbf{mr}\, \forall x C & := & \forall x.\, t x\, \mathbf{mr}\, C, \\
\varepsilon\, \mathbf{mr}\, (A \to B) & := & \varepsilon\, \mathbf{mr}\, A \to \varepsilon\, \mathbf{mr}\, B, \\
\varepsilon\, \mathbf{mr}\, (C \to A) & := & \forall x.\, x\, \mathbf{mr}\, C \to \varepsilon\, \mathbf{mr}\, A, \\
t\, \mathbf{mr}\, (A \to C) & := & \varepsilon\, \mathbf{mr}\, A \to t\, \mathbf{mr}\, C, \\
t\, \mathbf{mr}\, (D \to C) & := & \forall x.\, x\, \mathbf{mr}\, D \to t x\, \mathbf{mr}\, C, \\
\varepsilon\, \mathbf{mr}\, (A \wedge B) & := & \varepsilon\, \mathbf{mr}\, A \wedge \varepsilon\, \mathbf{mr}\, B,
\end{array}$$

$$
\begin{aligned}
t\ \mathbf{mr}\ (A \wedge C) &:= \varepsilon\ \mathbf{mr}\ A \wedge t\ \mathbf{mr}\ C, \\
t\ \mathbf{mr}\ (C \wedge A) &:= t\ \mathbf{mr}\ C \wedge \varepsilon\ \mathbf{mr}\ A, \\
t\ \mathbf{mr}\ (C \wedge D) &:= \pi_0 t\ \mathbf{mr}\ C \wedge \pi_1 t\ \mathbf{mr}\ D.
\end{aligned}
$$

Finally, the most important part: how to get a realizing term from a constructive proof M^C. Informally, the structure of the extracted term $[\![M]\!]$ is *exactly* the same as that of the constructive part of the proof term M. In other words, to get the extracted term, we simply 'cut out' those parts of the proof term which prove Harrop formulas and replace the constants (representing axioms in our logic system) in the remaining part with corresponding extracted terms. For example induction axioms are replaced by recursors. Then it is easy to show $[\![M]\!]\ \mathbf{mr}\ C$.

To define the extracted term more formally, assume we have assigned to any assumption variable u^C a new variable $x_u^{\tau(C)}$.

$$
\begin{aligned}
[\![u^C]\!] &:= x_u^{\tau(C)}, \\
[\![\lambda x^\rho M^C]\!] &:= \lambda x [\![M]\!], \\
[\![\lambda u^A M^C]\!] &:= [\![M]\!], \\
[\![\lambda u^C M^D]\!] &:= \lambda x_u^{\tau(C)} [\![M]\!], \\
[\![\langle M^C, N^A \rangle]\!] &:= [\![M]\!], \\
[\![\langle N^A, M^C \rangle]\!] &:= [\![M]\!], \\
[\![\langle M^C, N^D \rangle]\!] &:= \langle [\![M]\!], [\![N]\!] \rangle, \\
[\![M^{\forall x^\rho C} t^\rho]\!] &:= [\![M]\!]\, t, \\
[\![M^{A \to C} N^A]\!] &:= [\![M]\!], \\
[\![M^{D \to C} N^D]\!] &:= [\![M]\!]\, [\![N]\!], \\
[\![\pi_1(M^{A \wedge C})]\!] &:= [\![M]\!], \\
[\![\pi_0(M^{C \wedge A})]\!] &:= [\![M]\!], \\
[\![\pi_i(M^{C \wedge D})]\!] &:= \pi_i([\![M]\!]).
\end{aligned}
$$

Now all that is left is to find the extracted terms for the constructive axioms:

$$
\begin{aligned}
[\![(\exists^{*+}_{x^\rho, A})]\!] &:= \lambda x x, \\
[\![(\exists^{*+}_{x^\rho, C})]\!] &:= \lambda x, y^{\tau(C)}. \langle x, y \rangle, \\
[\![(\exists^{*-}_{x^\rho, A, C})]\!] &:= \lambda x, y^{\rho \to \tau(C)}. y x, \\
[\![(\exists^{*-}_{x^\rho, D, C})]\!] &:= \lambda y^{\rho \times \tau(D)}, z^{\rho \to \tau(D) \to \tau(C)}. z (\pi_0 y)(\pi_1 y), \\
[\![\mathsf{Ind}_{\forall p^{\mathsf{boole}} C}]\!] &:= \mathsf{R}_{\mathsf{boole}, \tau(C)}, \\
[\![\mathsf{Ind}_{\forall n^{\mathsf{nat}} C}]\!] &:= \mathsf{R}_{\mathsf{nat}, \tau(C)}.
\end{aligned}
$$

5.2. *Simultaneous free algebras*

Using SFAs, we can add new ground types defined by simultaneous induction. Formally, an SFA is defined from a list of new types $\iota_1, \dots, \iota_n$ and for each ι_i a list of constructors $c_{i1}, \dots, c_{in_i}$ together with their types. Each constructor c_{ij} has a type $\rho_{ij1} \to \dots \to \rho_{ijn_{ij}} \to \iota_i$ with each ρ_k being constructed using $\to$ only and with the ι_k occurring only strictly positive in them. The objects of the types $\iota_1, \dots, \iota_n$ are freely generated from these constructors, which is expressed by the induction schemes that are automatically added to the system together with these types. Here are some examples:

– The type seq of lists of natural numbers is generated from the constructors ε: seq and $\frown$: nat $\to$ seq $\to$ seq and is introduced by

```
(introduce-algebras
 '((seq ((empty)
        (add (n nat) (s seq))))))
```

The induction axiom for a formula $A[s^{\mathsf{seq}}]$ reads

$$A[\varepsilon] \to \forall n, s\,(A[s] \to A[n^\frown s]) \to \forall s A[s].$$

– To define the type tree of finitely branching trees, we have to simultaneously define the type treelist. The constructors are $\{\cdot\}$: treelist $\to$ tree , ε: treelist and $\frown$: tree $\to$ treelist $\to$ treelist. It is introduced by

```
(introduce-algebras
 '((treelist ((empty)
             (add (t tree) (l treelist))))
   (tree ((tr (l treelist))))))
```

and the induction principle for formulas $A[t^{\mathsf{tree}}]$ and $B[l^{\mathsf{treelist}}]$ is

$$\forall l\,(B[l] \to A[\{l\}]) \to B[\varepsilon] \to \forall t, l\,(A[t] \to B[l] \to B[t^\frown l]) \to \\ \forall t A[t] \wedge \forall l B[l].$$

– Finally the type inftree of trees branching over the natural numbers uses the constructors ε: inftree and lim: (nat $\to$ inftree) $\to$ inftree.

```
(introduce-algebras
 '((inftree ((empty)
            (lim (f (c-arrow 'nat 'inftree)))))))
```

The induction axiom for $A[t^{\mathsf{inftree}}]$ is

$$A[\varepsilon] \to \forall f^{\mathsf{nat} \to \mathsf{inftree}}\,(\forall n A[f n] \to A[\lim f]) \to \forall t A[t].$$

Functionals defined by structural recursion over an inductive type are usually given as a set of equations, one equation for each constructor. For example, the length of an element of the type seq would be defined:

$$\begin{aligned}\mathsf{len}(\varepsilon) &:= 0,\\ \mathsf{len}(n^\frown s) &:= 1+\mathsf{len}(s).\end{aligned}$$

We could enter a term that does exactly the computation given above using the appropriate recursor. The term would look like this:

```
(((alg-rec-at 'seq '(seq) '(nat)) 0)
(lambda (n^) (lambda (s^) (lambda (m^)
((plus-nat m^) 1)))))
```

This certainly is not a convenient way of entering the equations above. Therefore, the SFA extension also provides a simple interface for entering recursion equations. The command

```
(def-rec 'len (c-arrow 'seq 'nat) 0)
```

tells MINLOG that we want to define a constant `len` of type $\mathsf{seq} \to \mathsf{nat}$.

```
; Please feed in (using define-vars) new variables
; for the following types:
; nat
```

The interface asks for one or more variable names that will be used to represent step values in the recursion equations.

```
(define-vars 'll)
; len empty =? C[]
```

Now we are asked to enter the right-hand side of the first equation.

```
(define-rhs (parse-term "0"))
; len(add n^ s^) =? C[n^,s^,ll^]
; where ll^ = len s^
```

Finally, we have to give the right-hand side of the second equation. Here we can use the free variables $\hat{n}$,$\hat{s}$ (for the arguments of the constructor) and $\hat{ll}$ (for the step value).

```
(define-rhs (parse-term "ll^ + 1"))
; OK, COMPLETED; You have added the following
; recursion equations:
; len empty = 0
; len(add n^ s^) = (len s^)+1
```

From now on, the constant `len` will be used in all in- and outputs to represent the rather lengthy term given above.

5.3. *The quicksort example*

We will now use seq to get the quicksort algorithm from a proof of its specification. To this end, we introduce some more recursively defined functionals that we will need:

- $s_1 * s_2$ concatenates the two lists s_1 and s_2.
- $\#_m(s)$ counts the occurrences of m in s.
- $s \geq m$ checks whether m is a lower bound for the elements in s:

$$\begin{aligned} \varepsilon \geq m \quad &:= \quad \mathsf{true}, \\ (n^\frown s) \geq m \quad &:= \quad \textbf{if} \quad m \leq n \quad \textbf{then} \quad s \geq m \quad \textbf{else} \quad \mathsf{false} \quad \textbf{fi}. \end{aligned}$$

- Likewise, $s < m$ checks whether m is a proper upper bound.
- $s_{\geq m}$ computes the sublist of s of elements $\geq m$:

$$\begin{aligned} \varepsilon_{\geq m} \quad &:= \quad \varepsilon, \\ (n^\frown s)_{\geq m} \quad &:= \quad \textbf{if} \quad m \leq n \quad \textbf{then} \quad n^\frown(s_{\geq m}) \quad \textbf{else} \quad s_{\geq m} \quad \textbf{fi}. \end{aligned}$$

- $s_{<m}$ is defined correspondingly.
- $\mathsf{sorted}(s)$ checks whether the list s is sorted:

$$\begin{aligned} \mathsf{sorted}(\varepsilon) \quad &:= \quad \mathsf{true}, \\ \mathsf{sorted}(n^\frown s) \quad &:= \quad \textbf{if} \quad s \geq n \quad \textbf{then} \quad \mathsf{sorted}(s) \quad \textbf{else} \quad \mathsf{false} \quad \textbf{fi}. \end{aligned}$$

Using the structural induction scheme, we can easily prove the following fundamental properties of these functionals:

1. $\mathsf{len}(s) \leq 0 \to s = \varepsilon$
2. $\mathsf{len}(s_{<n}) \leq \mathsf{len}(s)$
3. $\mathsf{len}(s_{\geq n}) \leq \mathsf{len}(s)$
4. $(\forall m \#_m(s_1) = \#_m(s_2)) \to s_1 \geq n \to s_2 \geq n$
5. $(\forall m \#_m(s_1) = \#_m(s_2)) \to s_1 < n \to s_2 < n$
6. $s_{\geq n} \geq n$
7. $s_{<n} < n$
8. $\mathsf{sorted}(s_1) \to s_1 \geq n \to \mathsf{sorted}(s_2) \to s_2 < n \to \mathsf{sorted}(s_2 * s_1)$
9. $s \geq n \to \mathsf{sorted}(s) \to \mathsf{sorted}(n^\frown s)$
10. $\#_n(s) = \#_n(s_{<m}) + \#_n(s_{\geq m})$
11. $\#_n(s_1 * s_2) = \#_n(s_1) + \#_n(s_2)$

Now we formulate the specification of the desired algorithm: we want to show that every list of natural numbers can be sorted, i.e. for every list s there is a list t such that t is sorted and every number n occurs in t as often as in s. Formally:

$$\forall s\, \exists^* t.\, \mathsf{sorted}(t) \wedge \forall n\, \#_n(s) = \#_n(t).$$

The idea is to take, given a list $s = n'^\frown s'$, the first element n' and split s' up into $s'_{<n'}$ and $s'_{\geq n'}$, sort these shorter lists and put the results together in the right order. In other words, we prove the goal by induction over the *length* of the given list, so we first prove

$$\forall m, s.\, \mathsf{len}(s) \leq m \to \exists^* t.\, \mathsf{sorted}(t) \wedge \forall n\, \#_n(s) = \#_n(t)$$

by induction on m and then get the original goal by instantiating m with $\mathsf{len}(s)$. The proof is simple; we show it anyway to demonstrate that this example can be carried out in MINLOG without running into technical difficulties.
Base $m = 0$. Let s with $\mathsf{len}(s) \leq 0$ be given. Then $s = \varepsilon$ by lemma 1. Let $t := \varepsilon$, then $\mathsf{sorted}(t) \searrow^* \swarrow \top$, we have to show $\forall m\, \#_m(s) = \#_m(t)$. This follows from $s = \varepsilon = t$.
Step $m \to m+1$. Let s with $\mathsf{len}(s) \leq m+1$ be given.
Note: we make a case distinction here ($s = \varepsilon$ or $s = n'^\frown s'$). Formally, this is achieved by using the structural induction scheme for seq, without making use of the induction hypothesis in the step case.
Case $s = \varepsilon$. See $m = 0$.
Case $s = n'^\frown s'$. Now $\mathsf{len}(s') \leq m \searrow^* \swarrow \mathsf{len}(s) \leq m+1$, which we have just assumed. Using lemma 2 and 3 we conclude $\mathsf{len}(s'_{<n'}) \leq m$ and $\mathsf{len}(s'_{\geq n'}) \leq m$. Using the induction hypothesis for $s'_{<n'}$ and $s'_{\geq n'}$ we get t_1 and t_2 with (1) $\mathsf{sorted}(t_1)$, (2) $\forall n\, \#_n(s'_{<n'}) = \#_n(t_1)$, (3) $\mathsf{sorted}(t_2)$ and (4) $\forall n\, \#_n(s'_{\geq n'}) = \#_n(t_2)$. Let $t := t_1 * (n'^\frown t_2)$. We have to show $\mathsf{sorted}(t)$ and $\forall n\, \#_n(s) = \#_n(t)$.
$\mathsf{sorted}(t)$: lemma 7 yields $s'_{<n'} < n'$ and hence, using lemma 5 and (2), we get (5) $t_1 < n'$. In the same way, using lemma 6 and 4 instead of 7 and 5, we get $t_2 \geq n'$ and conclude (6) $n'^\frown t_2 \geq n'$ using normalization again, as well as (7) $\mathsf{sorted}(n'^\frown t_2)$ by lemma 9 and (3). Finally with lemma 8, (7), (6), (1) and (5) we get $\mathsf{sorted}(t_1 * (n'^\frown t_2))$.
To show $\forall n\, \#_n(s) = \#_n(t)$, let n be given.
Case $n = n'$: $\#_n(n'^\frown s') \searrow^* \swarrow \#_n(s') + 1 \overset{\text{L.10}}{=} \#_n(s'_{<n'}) + \#_n(s'_{\geq n'}) + 1$
$\overset{(2),(4)}{=} \#_n(t_1) + \#_n(t_2) + 1 \searrow^* \swarrow \#_n(t_1) + \#_n(n'^\frown t_2) \overset{\text{L.11}}{=} \#_n(t_1 * n'^\frown t_2)$.
Case $n \neq n'$: $\#_n(n'^\frown s') \searrow^* \swarrow \#_n(s') \overset{\text{L.10}}{=} \#_n(s'_{<n'}) + \#_n(s'_{\geq n'})$
$\overset{(2),(4)}{=} \#_n(t_1) + \#_n(t_2) \searrow^* \swarrow \#_n(t_1) + \#_n(n'^\frown t_2) \overset{\text{L.11}}{=} \#_n(t_1 * n'^\frown t_2)$.
Here is the program extracted from this proof M. We show an edited version of the original SCHEME expression (we have split the expression into three

parts and renamed the bound variables):

$$\begin{aligned}
[\![M]\!] &= \lambda s.\, g\,\mathsf{len}(s) s\,, \qquad \text{where} \\
g &= \mathsf{R}_{\mathsf{nat},\mathsf{seq}\to\mathsf{seq}}(\lambda s\,\varepsilon)(\lambda n\, h) \qquad \text{and} \\
h &= \lambda f^{\mathsf{seq}\to\mathsf{seq}}.\, \mathsf{R}_{\mathsf{seq},\mathsf{seq}}\,\varepsilon\,(\lambda n\,\lambda s_1\,\lambda s_2.\,(f(s_{1<n})) * (n^\frown(f(s_{1\geq n})))).
\end{aligned}$$

$(g\,n)\colon \mathsf{seq} \to \mathsf{seq}$ is a quicksort function able to deal with lists of length up to n and is defined by recursion over n:

$$\begin{aligned}
g\,0\,s &= \varepsilon \\
g(n+1)s &= h(g\,n)s.
\end{aligned}$$

If f is a function that can sort lists of length up to n, then $(h\,f)$ is a function that sorts lists of length up to $n+1$:

$$\begin{aligned}
h\,f\,\varepsilon &= \varepsilon \\
h\,f(n^\frown s) &= (f(s_{<n})) * (n^\frown(f(s_{\geq n}))).
\end{aligned}$$

6. Advanced features: A-translation and pruning

There are a number of proofs of $\forall\exists$–statements which — although very short and elegant — do not immediately yield a program, since they contain non-constructive arguments. However, it is well-known that such classical proofs under certain circumstances can be translated into constructive proofs, and hence yield algorithms via program extraction (cf. section 5). There is a substantial literature on that subject, and the MINLOG system supports a variant, known as 'A-translation', which goes back to work of Friedman (1978) and Leivant (1985). We will explain this translation by means of an example concerning minimization on finite binary trees. We will also use this example to discuss a second technique for program development: the 'pruning' operation going back to Goad (1980).

6.1. *Search through binary trees*

In our example we consider finite binary trees with leaves labelled by integers:

$$\mathsf{leaf}(n) \mid \langle s_1, s_2 \rangle.$$

We let P be a decidable property of trees. We wish to prove constructively that for every tree t with property $P[t]$ there is a subtree s of t satisfying $P[s]$

which is minimal, i.e. for no proper subtree s' of s, $P[s']$ holds. Of course we are interested in the program extracted from such a proof.

Let $s \subseteq t$ mean that s is a subtree of t and define

$$\begin{aligned} Q[s] \quad &:\equiv \quad \forall s' \subseteq s\,.\, s' \neq s \to \neg P[s'], \\ A_0[t,s] \quad &:\equiv \quad s \subseteq t \wedge P[s] \wedge Q[s]. \end{aligned}$$

Then our goal can be stated formally as

$$\forall t\,.\, P[t] \to \exists^* s A_0[t,s].$$

6.2. *From classical proofs to programs*

We will first prove the goal classically, i.e. deduce from $P[t]$ the classical existence statement

$$\neg\forall s \neg A_0[t,s]$$

and then apply Friedman's A-translation which gives us a constructive proof containing a quite clever algorithm.

The classical proof

Here and in 6.2.2 and 6.2.3 we fix a tree t and suppress it notationally, for sake of better readability. Hence

$$A_0[s] :\equiv A_0[t,s].$$

Roughly, the classical proof goes as follows: assume $P[t]$ and $\forall s \neg A_0[s]$. We have to derive a contradiction. Using the second assumption we can easily prove $\forall s\,.\, s \subseteq t \to Q[s]$ by induction on s. Now, setting $s := t$, we obtain $Q[t]$ and hence $A_0[t]$ ($\equiv A_0[t,t]$) using the assumption $P[t]$. But this contradicts our assumption $\forall s \neg A_0[s]$.

A-translation

In order to apply the A-translation to this proof we first have to look at the shape of the proof in some detail. Note that we only used induction on trees and the following facts about P, Q and $\subseteq$:

$$\begin{aligned} \mathrm{ax}_1&: \quad \forall n\,.\, Q[\mathsf{leaf}(n)] \\ \mathrm{ax}_2&: \quad \forall s_1, s_2\,.\, \neg P[s_1] \to \neg P[s_2] \to Q[s_1] \to Q[s_2] \to Q[\langle s_1, s_2 \rangle] \\ \mathrm{ax}_3&: \quad \forall s\,.\, s \subseteq s \\ \mathrm{ax}_4&: \quad \forall s_1, s_2, s\,.\, \langle s_1, s_2 \rangle \subseteq s \to s_1 \subseteq s \wedge s_2 \subseteq s. \end{aligned}$$

If we view Q as a primitive predicate then these are Π-formulas, i.e. universal formulas with a quantifier-free kernel. Moreover, the kernel of our goal, $A_0[s]$, is quantifier-free. Now, the A-translation works for a situation like this. Let us briefly explain its crucial idea at our classical proof.

Recall that above we have proved $\bot$ from the axioms $\mathrm{ax}_1, \ldots, \mathrm{ax}_4$, the assumptions $P[t]$ and $\forall s\,.\,A_0[s] \to \bot$. Observe also that the symbol $\bot$, for falsity, didn't play a special role in the proof (e.g. we used neither ex-falso-quodlibet nor stability); in particular the *proof* was entirely constructive (although the *formula* proven was, of course, a non-constructive existence statement). Therefore we may replace everywhere in the proof the formula $\bot$ by our constructive goal-formula

$$A :\equiv \exists^* s A_0[s]$$

and obtain a correct and constructive derivation of the formula A from the assumptions $P[t]$ and $\forall s\,.\,A_0[s] \to A$. But: the latter formula is an instance of an $\exists^{*+}$ axiom. Hence we get A constructively from the axioms and $P[t]$ alone.

Unfortunately there are some complications: (1) Of course, we want to be allowed to use ex-falso-quodlibet ($\bot \to A$) and stability ($\neg\neg A \to A$) in our proofs and (2) the translated proof also uses the translated axioms; for instance

$$\mathrm{ax}_2' \colon \forall s_1, s_2\,.\,(P[s_1] \to A) \to (P[s_2] \to A) \to Q[s_1] \to Q[s_2] \to Q[\langle s_1, s_2 \rangle]$$

which is not provable, and, in general, will even be false. A way out of these problems is to first replace in the classical proof every atomic formula by its double negation (Gödel's negative translation) and afterwards replace $\bot$ by A. Then (1) ex-falso-quodlibet and stability are translated into formulas provable in minimal logic and (2) each assumption ax_i is translated into a formula which follows constructively from ax_i. Of course, this modified translation (i.e. Gödel's translation followed by the replacement $\bot \mapsto A$) also affects the formula A_0, but still transforms the formula $\forall s\,.\,A_0[s] \to \bot$ into a provable formula.

Remarks: 1. Friedman's original translation (Friedman, 1978) replaces every atomic formula R by $R \vee A$ and not, as we did, by $(R \to A) \to A$. But clearly the formulas $R \vee A$ and $(R \to A) \to A$ are constructively equivalent assuming decidability of R. We have chosen the latter variant, since in MINLOG we prefer reasoning with implications rather with disjunctions.

2. Another way to see the A-translation, is to say that in constructive logic we may pass from a proof of $\neg\forall x \neg D[x]$ (D quantifier-free), or equivalently $\neg\neg\exists^* D[x]$, to a proof of $\exists^* x D[x]$. This is known as Markov's rule.

The translated proof
In MINLOG we have implemented a refinement of the A-translation which does not replace all atomic formulas R by $(R \to A) \to A$. In our example this is only necessary for formulas $Q[\cdot]$; in general, it can be decided easily whether an atomic formula has to be replaced or not. For more information on this refinement we refer to (Berger and Schwichtenberg, 1995).

Now, the MINLOG system transforms our classical proof into a constructive one. We show this automatically generated proof in tree form below. Due to lack of space we graphically contracted consecutive applications of elimination rules to one rule. Similarly for consecutive introduction rules. A double line means that the conclusion follows from the premises by some elimination rules. The subproofs named M, M' etc. will play a role in section 6.2.4 only.

To make the proof tree easier to understand we give also an informal description: assume $P[t]$ and let

$$B[s] :\equiv s \subseteq t \to (Q[s] \to A) \to A$$

(again we suppressed t). We first prove $\forall s B[s]$ by induction on s. The base, $B[\mathsf{leaf}(n)]$, is easy. To prove the step we assume $B[s_1]$ and $B[s_2]$. In order to show $B[\langle s_1, s_2\rangle]$ we further assume $\langle s_1, s_2\rangle \subseteq t$ and $Q[\langle s_1, s_2\rangle] \to A$. We have to show A. *Case* $P[s_1]$: then by i.h., $B[s_1]$, we have $(Q[s_1] \to A) \to A$. Hence it suffices to show $Q[s_1] \to A$. So assume $Q[s_1]$. We have $s_1 \subseteq t$, $P[s_1]$ and $Q[s_1]$, i.e. $A_0[s_1]$. Hence A, by existence introduction. *Case* $\neg P[s_1]$: *Sub-case* $P[s_2]$: similar to case $P[s_1]$, but using i.h., $B[s_2]$. *Sub-case* $\neg P[s_2]$: we use again the i.h., $B[s_1]$. Hence it suffices to prove $Q[s_1] \to A$. So assume $Q[s_1]$ and show A. Using the i.h., $B[s_2]$, we see that in fact we may also assume $Q[s_2]$. Now we have $\neg P[s_1]$, $\neg P[s_2]$, $Q[s_1]$ and $Q[s_2]$. Hence $Q[\langle s_1, s_2\rangle]$, by ax_2. Now we remember that $Q[\langle s_1, s_2\rangle] \to A$ holds. Hence A. This completes the inductive proof of $\forall s B[s]$. Setting $s := t$ we get $(Q[t] \to A) \to A$. So it finally suffices to prove $Q[t] \to A$. So assume $Q[t]$. Since we assumed $P[t]$ and also $t \subseteq t$ holds, we have $A_0[t]$ ($\equiv A_0[t,t]$). Hence A, by existence introduction. This completes the proof.

Note that in the proof above $\exists$-introduction and case analysis on P occur which both were not used in the classical proof. The explanation is that $\exists$-introduction is used for proving the translation of the wrong assumption, $\forall s \neg A_0[s]$, and case analysis is needed for proving ax_2 from its translation.

$$\dfrac{u_4 \qquad \dfrac{\mathrm{ax}_2\ \ s_1\ \ s_2\ \ u_9\ \ u_{10}\ \ u_{11}\ \ u_{12}}{Q[\langle s_1, s_2\rangle]}}{N'\colon A}$$

$$\dfrac{\mathrm{ih}_1 \quad \dfrac{\mathrm{ax}_4, u_3}{s_1 \subseteq t} \quad \dfrac{\dfrac{\mathrm{ih}_2 \quad \dfrac{\mathrm{ax}_4, u_3}{s_2 \subseteq t} \quad \dfrac{N'\colon A}{Q[s_2] \to A}\, u_{12}}{A}}{Q[s_1] \to A}\, u_{11}}{N\colon A}$$

$$\dfrac{\dfrac{\dfrac{\mathrm{ih}_2 \quad \dfrac{\mathrm{ax}_4, u_3}{s_2 \subseteq t} \quad \dfrac{\dfrac{\dfrac{\dfrac{\mathrm{ax}_4, u_3}{s_2 \subseteq t} \quad u_7 \quad u_8}{A_0[s_2]}}{A}}{Q[s_2] \to A}\, u_8}{A}}{P[s_2] \to A}\, u_7 \qquad \dfrac{N\colon A}{\neg P[s_2] \to A}\, u_{10}}{M'\colon A}\ \mathrm{cases}_{P[s_2]}$$

$$\dfrac{\mathrm{ih}_1 \quad \dfrac{\mathrm{ax}_4, u_3}{s_1 \subseteq t} \quad \dfrac{\dfrac{\dfrac{\dfrac{\mathrm{ax}_4, u_3}{s_1 \subseteq t} \quad u_5 \quad u_6}{A_0[s_1]}}{A}}{Q[s_1] \to A}\, u_6}{M''\colon A}$$

$$\dfrac{\dfrac{M''\colon A}{P[s_1] \to A}\, u_5 \qquad \dfrac{M'\colon A}{\neg P[s_1] \to A}\, u_9}{M\colon A}\ \mathrm{cases}_{P[s_1]}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{u_2 \quad \dfrac{\mathrm{ax}_1\ \ n}{Q[\mathsf{leaf}(n)]}}{A}}{\forall n . B[\mathsf{leaf}(n)]}\, n, u_1, u_2 \qquad \dfrac{M\colon A}{\forall s_1, s_2 . B[s_1] \to B[s_2] \to B[\langle s_1, s_2\rangle]}\, s_1, s_2, \mathrm{ih}_1, \mathrm{ih}_2, u_3, u_4}{\forall s . B[s]}\ \mathrm{ind} \quad t \quad \dfrac{\mathrm{ax}_3\ \ t}{t \subseteq t}}{(Q[t] \to A) \to A} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\mathrm{ax}_3\ \ t}{t \subseteq t} \quad \mathrm{ax}_0 \quad u_0}{A_0[t]}}{A}}{Q[t] \to A}\, u_0$$

$$\dfrac{(Q[t] \to A) \to A \qquad Q[t] \to A}{A}$$

$A_0[s]$	$:\equiv$	$s \subseteq t \wedge P[s] \wedge Q[s]$
A	$:\equiv$	$\exists^* s\, A_0[s]$
$B[s]$	$:\equiv$	$s \subseteq t \to (Q[s] \to A) \to A$

ax_0: $P[t]$

ax_1: $\forall n \,.\, Q[\mathsf{leaf}(n)]$

ax_2: $\forall s_1, s_2 \,.\, \neg P[s_1] \to \neg P[s_2] \to Q[s_1] \to Q[s_2] \to Q[\langle s_1, s_2\rangle]$

ax_3: $\forall s \,.\, s \subseteq s$

ax_4: $\forall s_1, s_2, s \,.\, \langle s_1, s_2\rangle \subseteq s \to s_1 \subseteq s \wedge s_2 \subseteq s$

$u_0 : Q[t]$
$u_1 : \mathsf{leaf}(n) \subseteq t$
$u_2 : Q[\mathsf{leaf}(n)] \to A$

$u_3 : \langle s_1, s_2\rangle \subseteq t$	$u_5 : P[s_1]$	$u_7 : P[s_2]$	$u_9 : \neg P[s_1]$	$u_{11} : Q[s_1]$
$u_4 : Q[s_1, s_2] \to A$	$u_6 : Q[s_1]$	$u_8 : Q[s_2]$	$u_{10} : \neg P[s_2]$	$u_{12} : Q[s_2]$

The extracted program
From the translated proof we obtain the following program:

$$\begin{array}{rcl} f(t) & = & g(t,t) \\ g(\mathsf{leaf}(n),t) & = & t \\ g(\langle s_1,s_2\rangle,t) & = & \begin{array}[t]{ll} \textbf{if} & P[s_1] \\ \textbf{then} & g(s_1,s_1) \\ \textbf{else} & \begin{array}[t]{lll} \textbf{if} & P[s_2] & \\ \textbf{then} & g(s_2,s_2) & \\ \textbf{else} & g(s_1,g(s_2,t)) & \textbf{fifi}. \end{array} \end{array} \end{array}$$

6.3. *Pruning*

The idea of Goad's pruning operation is the following: consider the constructive proof above obtained from the A-translation. At many points of this proof we have derived the formula A. Suppose M is such a subproof deriving A, and assume that M contains again a subproof M' deriving A, too. Now it is tempting to simplify the whole proof by replacing M by the smaller proof M'. However, in general this will be impossible since M' may depend on an assumption u^B which is bound in M by an $\to$-introduction. In λ-term notation

$$M = M[\lambda u^B M'[u^B]].$$

Now assume that we are interested only in trees t satisfying $P[t]$ and some additional condition $C[t]$, i.e. we look for a proof of

$$\forall t\,.\,P[t] \wedge C[t] \to \exists^* s A_0[t,s]$$

and expect from this hopefully simpler proof a hopefully optimized program adapted to the restriction $C[t]$. Such a simplified proof may be obtained as follows: in the subproof $M'[u^B]$ considered above we replace the assumption u^B by a proof K^B which uses the additional assumption $C[t]$ (and possibly other assumptions valid at the particular occurrence of u), and then replace the modified proof $M[\lambda u^B M'[K^B]]$ by $M'[K^B]$. Of course, there is no guarantee that $M'[K^B]$ is indeed simpler than $M[\lambda u^B M'[u^B]]$, but in most cases it will be.

In the following we will consider three pruning conditions $C_1[t]$, $C_2[t]$, $C_3[t]$, and study their effect on the proof and the extracted program. It will turn out that not only are the new programs simpler than the one in 6.2.4, but they also yield different results. This shows that it is essential that the pruning operation is done on *proofs*, since optimized *programs* will always compute the same result on the restricted inputs.

First example
Assume we enrich our specification by the additional information

$$C_1[t] :\equiv \forall s_1, s_2 . \langle s_1, s_2 \rangle \subseteq t \to P[s_1] \to P[s_2],$$

i.e. if P holds for a left subtree, than also for the right one. (The choice of the somewhat arbitrary formula C_1 is motivated by the effect it later will have.) Remember that in the constructive proof described informally in 6.2.3 we had a *case* $\neg P[s_1]$ with the goal A. Then there was a *sub-case* $\neg P[s_2]$ again with goal A. In this sub-case we used the assumption $\neg P[s_1]$. This assumption can now be proved using the pruning condition $C_1[t]$ and the assumptions $\neg P[s_1]$ and $\langle s_1, s_2 \rangle \subseteq t$ which are both valid at that point. This means that the case analysis according to whether $P[s_1]$ holds can be removed from the proof, since we have shown that in fact $\neg P[s_1]$ holds. In the proof tree the simplified proof is obtained by replacing the subproof M by M', where the assumption u_9 is replaced by a proof using $C_1[t]$, u_3 and u_{10}.

The program extracted from the simplified proof is the following:

$$\begin{array}{rcll} f(t) & = & g(t,t) & \\ g(\mathsf{leaf}(n),t) & = & t & \\ g(\langle s_1, s_2 \rangle, t) & = & \textbf{if} & P[s_1] \\ & & \textbf{then} & g(s_1, s_1) \\ & & \textbf{else} & g(s_1, g(s_2, t)) \quad \textbf{fi}. \end{array}$$

Second example
Next consider

$$C_2[t] :\equiv \forall s . s \subseteq t \to P[s].$$

This has an extreme effect on the proof, since already the outer case analysis on $P[s_1]$ is decided positively. Hence we get a proof without case analysis. This means that in the proof tree we replace the subproof M by M'', where the assumption u_5 is replaced by a proof using $C_2[t]$, u_3 and ax_4. The extracted program is simply

$$\begin{array}{rcl} f(t) & = & g(t,t) \\ g(\mathsf{leaf}(n),t) & = & t \\ g(\langle s_1, s_2 \rangle, t) & = & g(s_1, s_1). \end{array}$$

Third example
Our last pruning condition is

$$C_3 :\equiv \forall s\,.\, \neg P[s] \to Q[s]$$

which is in fact an extra condition on P. We look again at the *case* $\neg P[s_1]$ and therein at the *sub-case* $\neg P[s_2]$. To prove A we used both induction hypotheses, $B[s_1]$ and $B[s_2]$. We ended up in a situation where we had to prove A under the extra assumptions $Q[s_1]$ (from ih_1) and $Q[s_2]$ (from ih_2). Now from $\neg P[s_1]$ and $\neg P[s_2]$ and the pruning condition C_3 we can *prove* $Q[s_1]$ and $Q[s_2]$. Hence we do not need to use the induction hypotheses at that point. For the proof tree this means that the subproof N is replaced by N', where the assumptions u_{11} and u_{12} are replaced by a proof using C_3, u_9 and u_{10}. The effect on the extracted program is that the nested recursive call disappears:

$$\begin{array}{rcl}
f(t) & = & g(t,t) \\
g(\mathsf{leaf}(n),t) & = & t \\
g(\langle s_1,s_2\rangle,t) & = & \begin{array}[t]{ll} \textbf{if} & P[s_1] \\ \textbf{then} & g(s_1,s_1) \\ \textbf{else} & \begin{array}[t]{ll} \textbf{if} & P[s_2] \\ \textbf{then} & g(s_2,s_2) \\ \textbf{else} & t \quad \textbf{fifi}. \end{array} \end{array}
\end{array}$$

ACKNOWLEDGEMENTS

We are grateful to Felix Joachimski, Karl-Heinz Niggl and Klaus Weich for their contributions to the MINLOG system which were used in many places of this presentation.

REFERENCES

Berger, U.: 1993a, 'Program extraction from normalization proofs'. In: M. Bezem and J. Groote (eds.): *Typed Lambda Calculi and Applications*, Vol. 664 of *Lecture Notes in Computer Science*. pp. 91–106.
Berger, U.: 1993b, 'Total Sets and Objects in Domain Theory'. *Annals of Pure and Applied Logic* **60**, 91–117.

Berger, U., M. Eberl, and H. Schwichtenberg: 1998, 'Normalization by evaluation'. Submitted to: B. Möller and J.V. Tucker (eds.): *Prospects for hardware foundations*. NADA volume, *Lecture Notes in Computer Science*.

Berger, U. and H. Schwichtenberg: 1991, 'An inverse of the evaluation functional for typed λ–calculus'. In: R. Vemuri (ed.): *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*. pp. 203–211.

Berger, U. and H. Schwichtenberg: 1995, 'Program Extraction from Classical Proofs'. In: D. Leivant (ed.): *Logic and Computational Complexity, LCC '94*, Vol. 960 of *Lecture Notes in Computer Science*. pp. 77–97.

Boyer, R. S. and J. S. Moore: 1988, *A Computational Logic Handbook*, Vol. 23 of *Perspectives in Computing*. Academic Press, Inc.

Friedman, H.: 1978, 'Classically and intuitionistically provably recursive functions'. In: D. Scott and G. Müller (eds.): *Higher Set Theory*, Vol. 669 of *Lecture Notes in Mathematics*. pp. 21–28.

Goad, C. A.: 1980, 'Computational uses of the manipulation of formal proofs'. Ph.D. thesis, Stanford University. Stanford Department of Computer Science Report No. STAN–CS–80–819.

Hayashi, S.: 1990, 'An introduction to PX'. In: G. Huet (ed.): *Logical Foundations of Functional Programming*. Addison–Wesley, pp. 432–486.

Kreisel, G.: 1959, 'Interpretation of analysis by means of constructive functionals of finite types'. In: A. Heyting (ed.): *Constructivity in Mathematics*. North–Holland, Amsterdam, pp. 101–128.

Leivant, D.: 1985, 'Syntactic Translations and Provably Recursive Functions'. *The Journal of Symbolic Logic* **50**(3), 682–688.

Miller, D.: 1991, 'A logic programming language with lambda–abstraction, function variables and simple unification'. *Journal of Logic and Computation* **1**(4), 497–536.

Nipkow, T.: 1993, 'Orthogonal Higher–Order Rewrite Systems are Confluent'. In: M. Bezem and J. Groote (eds.): *Typed Lambda Calculi and Applications*, Vol. 664 of *Lecture Notes in Computer Science*. pp. 306–317.

Scott, D.: 1982, 'Domains for denotational semantics'. In: E. Nielsen and E. Schmidt (eds.): *Automata, Languages and Programming*, Vol. 140 of *Lecture Notes in Computer Science*. pp. 577–613.

Tait, W. W.: 1967, 'Intensional Interpretation of Functionals of Finite Type I'. *The Journal of Symbolic Logic* **32**(2), 198–212.

Troelstra, A. S. and D. van Dalen: 1988, *Constructivism in Mathematics. An Introduction*, Vol. 121, 123 of *Studies in Logic and the Foundations of Mathematics*. North–Holland, Amsterdam.

Turner, R.: 1991, *Constructive Foundations for functional languages*. McGraw–Hill.

van de Pol, J. and H. Schwichtenberg: 1995, 'Strict functionals for termination proofs'. In: M. Dezani-Ciancaglini and G. Plotkin (eds.): *Typed Lambda Calculi and Applications*, Vol. 902 of *Lecture Notes in Computer Science*. pp. 350–364.