Higher-Order Symb Comput (2008) 21: 377–409
DOI 10.1007/s10990-008-9038-0

Adapting functional programs to higher order logic

Scott Owens · Konrad Slind

Published online: 30 October 2008

Abstract Higher-order logic proof systems combine functional programming with logic, providing functional programmers with a comfortable setting for the formalization of programs, specifications, and proofs. However, a possibly unfamiliar aspect of working in such an environment is that formally establishing program termination is necessary. In many cases, termination can be automatically proved, but there are useful programs that diverge and others that always terminate but have difficult termination proofs. We discuss techniques that support the expression of such programs as logical functions.

Keywords Higher order logic · Recursive definition · Termination · Well-foundedness · Regular expression pattern matching

1 Introduction

Many higher-order logics are based on Church's simple type theory [9]. These logics support general mathematical reasoning, which is not surprising since simple type theory was proposed as a foundation for mathematics. Contemporary interactive proof assistants typically implement extensions of simple type theory. For example, HOL [21] adds type variables, Isabelle/HOL [46] provides type classes, PVS [50] adds predicate subtypes, Coq [6] is a constructive logic with dependent types, and IMPS [14] supports partial functions. Each of these systems provides a library of verified mathematics to base proofs on, plus a collection of automated proof tools such as decision procedures and simplifiers.

Either directly or via definitional extension, the above logics support fundamental aspects of typed functional programming: polymorphic simple types, algebraic datatypes, higher-order functions, pattern matching, and recursive definitions. Thus, a user of one of these

Electronic supplementary material The online version of this article (http://dx.doi.org/10.1007/s10990-008-9038-0) contains supplementary material, which is available to authorized users.

S. Owens · K. Slind (✉)
School of Computing, University of Utah, Salt Lake City, USA
e-mail: slind@cs.utah.edu

systems and a functional programmer define and reason about many of the same constructs. In light of this connection, a higher-order logic verification environment could be thought of as a functional programming environment in which functional programs, their specifications, their executions, and their correctness proofs can be explored in a unified, semantically powerful setting. The essential question is: *is higher-order logic adequate to the task*?

In order to meaningfully answer this, we must refine the question. Higher-order logic can define the syntax and operational or denotational semantics of a programming language, thereby supporting the formalization of individual programs. However, such formalizations are bulky and awkward to deal with mathematically. Instead, functional programs can be directly mapped to the native, built-in functions of higher-order logic: no domain theory and no operational semantics required. In this way, functional programs live in the same milieu as the mathematics they are specified with. For example, a factorial program is modeled as a mathematical function on numbers, not as a function on the domain of lifted numbers, and not as a syntax tree animated by an operational semantics.

We can now ask: *are the native functions of higher-order logic adequate to the task of formalizing functional programs*? The answer is, of course, negative: most higher-order logics express total functions, while a functional programming language expresses computable functions. For example, extensional equality on functions—a heavily used feature of classical higher order logic—is not computable. On the other hand, functional languages can easily define partial functions by non-terminating recursion. The functions provided by each setting do not subsume each other. At first sight, this difference in expressivity is dismaying; however, experience over many years has shown that a vast number of useful and important algorithms can be directly defined as total functions.

There still remain important algorithms that are difficult to directly formalize as total functions. Often, the difficulty stems from the termination proof requirements of a logic of total functions. Termination proofs cannot be avoided in this setting, for without a termination proof for each recursive definition, the logic is inconsistent. Thus, termination is the fundamental obstacle to overcome. This paper explores, via examples, some termination-related difficulties with representing functional programs in higher-order logic, and shows how they can be surmounted, and in some cases, solved.

Proving termination for each recursive function seems like a painful burden; fortunately, proof automation can often deal with the termination of a proposed definition, particularly for structurally recursive functions. Hence, the (numerous) cases that present the functional programmer with no termination problems also present no such problems to the theorem-proving system user. A marked difference arises when considering programs whose termination is not obvious, or programs that might diverge on some inputs. In these situations, a functional programmer can establish termination in a number of ways: with the application of the mechanically unsupported intellect, with testing, with a paper-and-pencil proof, or even with a formal proof. In contrast, the higher-order logic user must always perform a formal termination proof. (We are slightly overstating our case here: some functions satisfying recursive specifications may be defined without proving termination. This will be discussed in Sect. 5.)

We explore this requirement with three examples: depth-first traversal in directed, possibly cyclic, graphs; an unfold; and regular expression pattern matching. The first two examples illustrate ways of formalizing partial functions in a logic with only total functions, while the last shows that, sometimes, an apparently partial function can be replaced by a total function that works just as well. Although we emphasize termination, we have also included a few correctness proofs: these may be safely skipped by the reader, but they serve as ultimate justifications of the definitions, and illustrate how mathematical reasoning about programs is performed in this context.

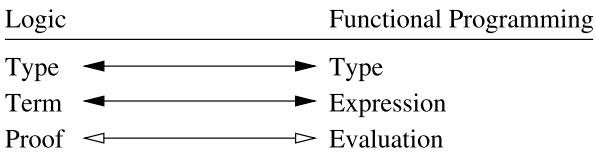

Fig. 1 A correspondence between functional programming and higher-order logic

To help focus the discussion, we use the higher-order logic of the HOL system [21]. This logic (also called HOL) supports a direct, although imprecise, correspondence between functional programs and logical functions. Figure 1 illustrates the relationship: logical types are similar to, but not the same as, ML types and logical terms resemble ML expressions. Executing a program on an argument can be carried out by performing a proof. Our approach therefore has nothing to do with the Curry-Howard correspondence, which precisely relates types in functional programming to propositions in logic, and expressions in functional programming to proofs.

We have performed all of our proofs in the HOL-4 verification system [48]. The formalizations can be found at the webpage http://dx.doi.org/10.1007/s10990-008-9038-0.

1.1 Notation and basic definitions

Detailed knowledge of the HOL logic is not needed to understand this paper. Syntactically, HOL resembles a conventional predicate logic that allows **λ**-notation and quantification over functions, sets, and relations; semantically, it resembles ordinary set-theoretic mathematics. Gordon and Melham [21] present a formal description of the logic.

Throughout this paper, we use *italicized font* for variables, constants are written in bold, sans-serif font, and keywords are in `typewriter font`. The following notation is also used:

- Standard logical notation: $\vee$, $\wedge$, $\neg$, $=$, $\Rightarrow$, $\Leftrightarrow$, $\forall$, $\exists$, true, false, and $(\cdot, \ldots, \cdot)$ (tuples). Also Hilbert's indefinite description operator: $\varepsilon x.\ P\ x$ denotes some a such that $P(a)$ holds; if P holds of nothing, then $\varepsilon x.\ P\ x$ denotes some arbitrary object with the same type as x.
- List notation: :: (cons), [] (empty list), @ (append), mem (member), map, filter, and flat (flatten), all_distinct (duplicate checking), foldr, and null (emptyness predicate).
- Set notation: $\in$, $\|\cdot\|$ (cardinality), $\setminus$ (set difference), $\{\cdot|\cdot\}$ (set comprehension), Finite (finiteness predicate), RTC (reflexive-transitive closure), ListToSet, and $f|_S$ (restriction of the domain of f to set S).
- Lambda: **λ**.
- ML keywords: `case`, `datatype`, `let`, and `if − then − else`.
- ML notation for types: $\rightarrow$ (function space), α, β (type variables), bool, α list, and α option. The num type represents $\mathbb{N}$.

The polymorphic constant ARB : α denotes a fixed but arbitrary element that inhabits all types. A form of function restriction can be defined using ARB as follows:

$$f|_S\ x = \texttt{if}\ x \in S\ \texttt{then}\ f\ x\ \texttt{else}\ \mathsf{ARB}$$

Thus, a function restricted to a set is not a partial function but is instead a total function returning a fixed value when applied to a value not in the set. In some cases, this modelling trick allows true partiality to be avoided. Logics with dependent types, or predicate subtypes such as PVS, can avoid this approach, but it is a standard method for modelling partiality in logics such as HOL and ACL2 [31].

A binary relation, $\prec\colon \alpha \to \alpha \to \mathsf{bool}$, is *wellfounded*, written $\mathsf{WF}(\prec)$, if it contains no infinite decreasing chains or, equivalently, if every non-empty set has a $\prec$-minimal element. The principle of wellfounded induction arises from the definition of wellfoundedness.

Definition 1 (Wellfoundedness)

$$\mathsf{WF}(R) \Leftrightarrow \forall P.\ (\exists w.\ P\ w) \Rightarrow \exists \mathit{min}.\ P\ \mathit{min} \wedge \forall b.\ R\ b\ \mathit{min} \Rightarrow \neg P\ b$$

Theorem 1 (Wellfounded induction)

$$\mathsf{WF}(R) \Rightarrow (\forall x.\ (\forall y.\ R\ y\ x \Rightarrow P\ y) \Rightarrow P\ x) \Rightarrow \forall x.\ P\ x$$

The lexicographic combination $<_{lex}$ of wellfounded relations $<_1$ and $<_2$ is itself wellfounded.

Definition 2 (Lexicographic comparison)

$$(x', y') <_{lex} (x, y) \Leftrightarrow x' <_1 x \vee (x' = x \wedge y' <_2 y)$$

Lexicographic combination can be iterated to treat tuples with more than two components.

2 Defining recursive functions

In logic, a recursive definition requires proof of the unique existence of a function satisfying the given recursion equations. Certain constraints on the style of recursion, such as primitive recursion, guarantee the unique existence of the specified function. Indeed, most higher-order logic implementations automatically prove the primitive recursion theorem for algebraic datatypes, and use it to justify the definition of functions on such types. However, for many functional programs the syntactic constraints of primitive recursion are too restrictive. The more general mechanism of *wellfounded recursion* can accept any recursion equation, provided it satisfies a semantic criterion—termination. A wellfounded relation $\prec$ on the function's domain must witness termination of the function for all inputs, *i.e.* proving $x' \prec x$ for all recursive calls $f(x')$ that can be invoked from a call $f(x)$ establishes termination of f for all inputs. Informally, if the arguments must get smaller by $\prec$ for each call and there is a finite limit (by wellfoundedness) on the number of smaller things, then the function terminates.

2.1 Wellfounded recursion

In untyped lambda calculus, the Y combinator supports recursive function definitions due to the fixed-point equation $\mathsf{Y}\ F = F\ (\mathsf{Y}\ F)$ which implies that $\mathsf{Y}\ F$ satisfies the recursion equation giving rise to the functional F. However, higher-order logic cannot express the Y combinator, for it would allow the definition of the function $f(x) = f(x) + 1$, from which one can immediately get $0 = 1$ by subtracting $f(x)$ from both sides. Higher-order logic can, however, define a wellfounded recursion combinator, WFREC, which behaves as a controlled version of Y. The WFREC combinator takes two arguments: a binary relation $\prec$ and a functional F. If $\prec$ is wellfounded, a Y-like fixed-point equation holds.

Theorem 2

$$\mathsf{WF}(\prec) \Rightarrow (\mathsf{WFREC}_{\prec}\ F)\ x = F\ ((\mathsf{WFREC}_{\prec}\ F)\ |_{\{y \mid y \prec x\}})\ x$$

Although $\mathsf{Y}\ F = F^n(\mathsf{Y}\ F)$ for any number n, each time the above equation is applied for WFREC, the domain of $(\mathsf{WFREC}_{\prec}\ F)$ is restricted to values smaller by $\prec$. Therefore, due to the wellfoundedness of $\prec$, the WFREC combinator cannot be unrolled indefinitely and so the function must terminate. If the recursion equation satisfies the condition that all recursive calls have smaller arguments, then the restriction is redundant, and the fixed-point equation becomes:

$$\mathsf{WFREC}_{\prec}\ F = F\ (\mathsf{WFREC}_{\prec}\ F)$$

The length function on lists provides a simple example of a definition by wellfounded recursion (of course, primitive recursion can also be used to define length).

$$\begin{array}{l}\mathsf{length}\ [\,] = 0\\ \mathsf{length}\ (h :: t) = 1 + \mathsf{length}\ t\end{array}$$

Abstracting the length name yields the following functional:

$$L = \lambda \mathit{length}\ \ell.\ \mathtt{case}\ \ell\ \mathtt{of}\ [\,] \Rightarrow 0 \mid h :: t \Rightarrow 1 + \mathit{length}\ t$$

The argument to length shrinks by the following wellfounded relation for each recursive call.

$$x \prec y \Leftrightarrow \exists a.\ (a :: x) = y$$

Thus $\mathsf{WFREC}_{\prec}\ L$ uniquely satisfies the recursion equation for length and we may define $\mathsf{length} = \mathsf{WFREC}_{\prec}\ L$ and derive the specified recursion equations for length as a logical theorem. These steps have been mechanized for HOL-4 and Isabelle/HOL [55]. For further discussion of wellfounded recursion in theorem provers, see [28, 51].

2.2 Termination relations

There exists a wide variety of wellfounded relations and combinators, such as $<_{lex}$, to combine them. For example, every inductively defined datatype has a wellfounded relation analogous to the above relation for length, known as the immediate sub-term relation. Wellfounded recursion based on this relation corresponds exactly to primitive recursion. In practice however, most non-trivial termination proofs reason about size of arguments, and so they are most often based on wellfounded relations built by mapping into the $<$ relation for natural numbers using *measure* functions. Definition 3 formally justifies this practice.

Definition 3 (Measure functions)

$$\begin{array}{rl}\underline{\text{Definitions}}: & \mathsf{inv\_image}\ (\prec)\ f\ x\ y = (f\ x) \prec (f\ y)\\ & \mathsf{measure} = \mathsf{inv\_image}\ (<)\end{array}$$

$$\begin{array}{rl}\underline{\text{Consequences}}: & \mathsf{WF}(R) \Rightarrow \mathsf{WF}\ (\mathsf{inv\_image}\ R\ f)\\ & \forall f : \alpha \rightarrow \mathsf{num}.\ \mathsf{WF}\ (\mathsf{measure}\ f)\end{array}$$

Thus termination of a recursive function over a type τ can often be proved by showing that the recursive calls are in the relation $\mathsf{measure}(\mathit{size}_{\tau})$. For example, mergesort terminates because each recursive call receives a shorter list. Lexicographic combinations of wellfounded relations handle multiple argument functions that need to measure the size of more than one argument.

2.3 Induction principles

The most common induction principles are derived from datatype definitions. For example, the induction principles for numbers and lists are:

$$\forall P.\ P\ 0 \wedge (\forall x.\ P\ x \Rightarrow P\ (x+1)) \Rightarrow \forall x.\ P\ x$$
$$\forall P.\ P\ [\,] \wedge (\forall x\ \ell.\ P\ \ell \Rightarrow P\ (x :: \ell)) \Rightarrow \forall \ell.\ P\ \ell$$

However, just as primitive recursion can be too restrictive for definitions, datatype induction can be unsuitable for some correctness proofs, especially those involving a function that does not recur on sub-terms of its input. An example of such a function is variant (which finds application in symbolic algorithms that need to create *fresh* entities):

$$\mathsf{variant}\ x\ \ell = \mathtt{if}\ \mathsf{mem}\ x\ \ell\ \mathtt{then}\ \mathsf{variant}\ (x+1)\ \ell\ \mathtt{else}\ x$$

The variant function is somewhat unusual because it recurs with a larger number. Nevertheless it terminates, because the recursive calls eventually reach a number not contained in ℓ. The termination measure for variant counts the number of elements, y, in ℓ such that $y \geq x$.

Proving properties of variant by mathematical or complete induction is awkward. Fortunately, the wellfounded relation used to prove termination of a function can be used to derive an induction principle customized to the recursion structure of the function [7]. This principle allows one to prove a property P of a function by assuming P holds for each recursive call and then showing that P holds for the entire function. For example, the following 'variant-induction' principle mirrors variant's recursion structure.

$$\forall P.\ (\forall x\ \ell.\ (\mathsf{mem}\ x\ \ell \Rightarrow P\ (x+1)\ \ell) \Rightarrow P\ x\ \ell) \Rightarrow \forall x\ \ell.\ P\ x\ \ell$$

Proving that, for example, variant finds the smallest number n such that $n \geq x$ and $\neg\mathsf{mem}\ n\ \ell$ is almost trivial with variant-induction. We use similar custom induction theorems frequently in our proofs.

Given recursion equations specifying a function f and a termination relation $\prec$ showing that f terminates, the f-induction principle can be automatically derived from the well-founded induction theorem [54]. Consequently, both a function definition and its associated induction principle depend on a termination proof for the function.

3 Depth-first traversal

The depth-first traversal algorithm given in Fig. 2 folds a function f over a directed, possibly cyclic, graph G, represented by a function returning the children nodes of a given node.

$$\begin{aligned}
&\mathsf{DFTp} : (\alpha \to \alpha\ \mathsf{list}) \to (\alpha \to \beta \to \beta) \to \alpha\ \mathsf{list} \to \alpha\ \mathsf{list} \to \beta \to \beta\\
&\mathsf{DFTp}\ G\ f\ \mathit{seen}\ [\,]\ \mathit{acc} = \mathit{acc}\\
&\mathsf{DFTp}\ G\ f\ \mathit{seen}\ (\mathit{visit_now} :: \mathit{visit_later})\ \mathit{acc} =\\
&\quad \mathtt{if}\ \mathsf{mem}\ \mathit{visit_now}\ \mathit{seen}\\
&\qquad \mathtt{then}\ \mathsf{DFTp}\ G\ f\ \mathit{seen}\ \mathit{visit_later}\ \mathit{acc}\\
&\qquad \mathtt{else}\ \mathsf{DFTp}\ G\ f\ (\mathit{visit_now} :: \mathit{seen})\\
&\qquad\qquad\qquad (G\ \mathit{visit_now}\ @\ \mathit{visit_later})\\
&\qquad\qquad\qquad (f\ \mathit{visit_now}\ \mathit{acc})
\end{aligned}$$

Fig. 2 Depth-first traversal as a partial function

Notice that $G : \alpha \to \alpha\ \mathsf{list}$ must branch finitely because it returns the children in a list, whereas a choice such as $G : \alpha \to \alpha\ \mathsf{set}$ need not branch finitely. DFTp maintains a set of traversed nodes, *seen*, and a list of nodes that need to be visited. When DFTp visits a node already seen, it ignores the node and continues on. When visiting a new node, it applies f to the node and accumulator, *acc*, adds the node to *seen*, and adds all of the node's children to the visit list.

Although this function is perfectly acceptable in most functional programming languages, and all of the language constructs used in DFTp are available in HOL, the proof system does not accept the definition because DFTp may diverge. Consider the infinite graph $\lambda x.\ [x + 1]$, which can be pictured as:

$$\cdots \to 0 \to 1 \to 2 \to 3 \to \cdots \to n \to n + 1 \to \cdots$$

Depth-first traversal diverges when given an infinite number of reachable nodes, *e.g.*, consider the invocation DFTp $(\lambda x.\ [x + 1])\ f\ [\,]\ [0]\ x$. An inherently finite graph representation, such as an adjacency list, would prohibit such examples and ensure termination, but would also forgo the simplicity and abstraction offered by the functional graph representation.

Although DFTp is partial in general, it is total when restricted to input graphs and nodes from which only a finite number of nodes are reachable. Graph reachability can be formalized in higher-order logic as follows: let $\mathsf{R}_G : \alpha \to \alpha \to \mathsf{bool}$ be the single-step reachability relation for G, so that its reflexive-transitive closure is G's many-step reachability relation. The $\mathsf{reachlist}_G : \alpha\ \mathsf{list} \to \alpha \to \mathsf{bool}$ relation then captures reachability from a list of nodes.

Definition 4 (Reachability)

$$\begin{aligned}
&\mathsf{R}_G\ x\ y \Leftrightarrow \mathsf{mem}\ y\ (G\ x)\\
&\mathsf{reach}_G = \mathsf{RTC}\ R_G\\
&\mathsf{reachlist}_G\ \mathit{nodes}\ y \Leftrightarrow \exists x.\ \mathsf{mem}\ x\ \mathit{nodes} \wedge \mathsf{reach}_G\ x\ y
\end{aligned}$$

Unlike dependent types, simple types provide no good method of enforcing the finiteness of the set of reachable nodes in the type of DFTp. However, the semantic power of HOL allows the variant of DFT in Fig. 3 to itself check the finiteness constraint. On calls with infinitely many reachable nodes, DFT returns an arbitrary element, ARB. This is the same technique used to handle function restriction—indeed DFT is just DFTp restricted to inputs with finite reachability.

The Finite predicate is easy to define inductively; however, finiteness is an undecidable property. Thus, the definition of DFT mixes computable and non-computable elements: one can consider this a price to pay for totality or—more positively—consider DFT as a version of depth-first traversal which includes its requirements within the actual code.

Remark 1 The presentation of DFT is logically equivalent to the formulas in Fig. 4, so the finiteness requirement can be 'floated out' past the recursion equations, leaving a clean presentation of the function. It would be convenient to have the theorems in Fig. 4 directly derived without entwining the requirements of the function and its recursion equation, as in Fig. 3, but HOL-4's current automation does not support that functionality.

3.1 Termination of DFT

Given that the set of reachable nodes is finite, the termination proof for DFT requires only a small amount of ingenuity. Let $\prec$ be the lexicographic combination of the number of reachable nodes not yet seen and the number of nodes in *to_visit*.

```
DFT : (α → α list) → (α → β → β) → α list → α list → β → β
DFT G f seen to_visit acc =
  if Finite (reachlist_G to_visit)
    then case to_visit
           of [] ⇒ acc
           | (visit_now :: visit_later) ⇒
               if mem visit_now seen
                 then DFT G f seen visit_later acc
                 else DFT G f (visit_now :: seen)
                              (G visit_now @ visit_later)
                              (f visit_now acc)
    else ARB
```

Fig. 3 Depth-first traversal as a total function

```
DFT G f seen [] acc = acc

Finite (reachlist_G (visit_now :: visit_later))
⇒ (DFT G f seen (visit_now :: visit_later) acc =
     if mem visit_now seen
       then DFT G f seen visit_later acc
       else DFT G f (visit_now :: seen)
                    (G visit_now @ visit_later)
                    (f visit_now acc))
```

Fig. 4 Derived presentation of DFT

Definition 5

$$
\begin{aligned}
&(G, f, seen', to\_visit', acc') \prec (G, f, seen, to\_visit, acc)\\
&\quad \text{iff}\\
&(\|\mathsf{reachlist}_G\ to\_visit' \setminus \mathsf{ListToSet}\ seen'\|, \mathsf{length}\ to\_visit') <_{lex}\\
&(\|\mathsf{reachlist}_G\ to\_visit \setminus \mathsf{ListToSet}\ seen\|, \mathsf{length}\ to\_visit)
\end{aligned}
$$

In the first recursive call, the *visit_now* node has been previously seen, so the set of unseen reachable nodes does not change, and the *to_visit* list gets smaller. In the second call, the *visit_now* node is added to the *seen* list, and all of the nodes reachable from the children of *visit_now* are also reachable from *visit_now* itself. Thus the set of reachable nodes gets no additions, and since the addition to the *seen* list was previously reachable (recall that we use the reflexive-transitive closure operation), the size of the calculated set decreases. Since the cardinality measure only applies to finite sets, the termination proof crucially relies on the finiteness of the reachable nodes.

Once termination is established, the custom induction theorem for DFT can be used to prove properties of DFT. Recall that the shape of the induction theorem follows the recursive structure of the program.

Theorem 3 (DFT Induction)

$$\begin{array}{l}\forall P.\\ \left(\begin{array}{l}\forall G\ f\ s\ h\ t\ a.\\ \quad P\ G\ f\ s\ [\,]\ a \wedge\\ \quad \left(\begin{array}{l}(\mathsf{Finite}\ (\mathsf{reachlist}_G\ (h :: t)) \wedge \mathsf{mem}\ h\ s \Rightarrow P\ G\ f\ s\ t\ a) \wedge\\ (\mathsf{Finite}\ (\mathsf{reachlist}_G\ (h :: t)) \wedge \neg \mathsf{mem}\ h\ s\\ \qquad \Rightarrow P\ G\ f\ (h :: s)\ (G\ h\ @\ t)\ (f\ h\ a))\\ \Rightarrow P\ G\ f\ s\ (h :: t)\ a)\end{array}\right)\end{array}\right)\\ \quad \Rightarrow\\ \quad \forall v\ v_1\ v_2\ v_3\ v_4.\ P\ v\ v_1\ v_2\ v_3\ v_4\end{array}$$

3.2 Properties of DFT

The correctness of a fold on graphs may be informally summarized in the following claims:

- All reachable nodes are visited
- No unreachable nodes are visited
- No reachable node is visited twice

Formalizing these statements hinges on the meaning of *visits*. We capture this notion by using cons as the folding function given to DFT, so that the returned list is just the visited nodes. With this device, Theorem 4 states that DFT with folding function f is equal to gathering all the visited nodes and then folding f over the resulting list.

Theorem 4 (DFT Fold)

$$\begin{array}{l}\mathsf{Finite}\ (\mathsf{reachlist}_G\ \mathit{to\_visit}) \Rightarrow\\ \mathsf{DFT}\ G\ f\ \mathit{seen}\ \mathit{to\_visit}\ \mathit{acc} = \mathsf{foldr}\ f\ \mathit{acc}\ (\mathsf{DFT}\ G\ \mathsf{cons}\ \mathit{seen}\ \mathit{to\_visit}\ [\,])\end{array}$$

With this understanding, it suffices to prove that the invocation DFT G cons *seen to_visit* [ ] returns a list that contains no duplicate entries, contains each node reachable from *to_visit*, and contains no nodes not so reachable. The first property is expressed in Theorem 5 and the other two are embodied in Theorem 6.

Theorem 5 (DFT Distinct)

$$\mathsf{Finite}\ (\mathsf{reachlist}_G\ \mathit{to\_visit}) \Rightarrow \mathsf{all\_distinct}\ (\mathsf{DFT}\ G\ \mathsf{cons}\ \mathit{seen}\ \mathit{to\_visit}\ [\,])$$

Theorem 6 (DFT Reach)

$$\begin{array}{l}\mathsf{Finite}\ (\mathsf{reachlist}_G\ \mathit{to\_visit}) \Rightarrow\\ \forall x.\ \mathsf{reachlist}_G\ \mathit{to\_visit}\ x \Leftrightarrow \mathsf{mem}\ x\ (\mathsf{DFT}\ G\ \mathsf{cons}\ [\,]\ \mathit{to\_visit}\ [\,])\end{array}$$

In these proofs, reasoning about DFT and transitive closures is crucially intertwined, and the modelling of functional programs as ordinary mathematics makes this possible.

3.3 Alternative termination criteria

The DFT equations (Fig. 4 and Theorem 3) rely on the constraint

$$\mathsf{Finite}\ (\mathsf{reachlist}_G\ \mathit{to\_visit})$$

In many cases, simpler, but more restrictive, constraints that depend only on the graph itself are more appropriate. We examine three such constraints in the following subsections.

3.3.1 Finite parents

Define the parents of a graph to be those nodes that have children.

Definition 6 (Parents) $\mathsf{Parents}_G = \{x \mid G\ x \neq [\,]\}$

A graph with a finite parents has finitely many reachable nodes.

Theorem 7

$$\forall G.\ \mathsf{Finite}\ (\mathsf{Parents}_G) \Rightarrow \mathsf{Finite}\ (\mathsf{reachlist}_G\ \mathit{to\_visit})$$

The implication does not hold in the other direction. Consider the graph $\lambda x.[0]$: from any given list of nodes, only those nodes and 0 are reachable; however, the parents encompass the entire graph.

3.3.2 Finite types

Had we defined $\mathsf{Parents}_G$ as the entire set of nodes, instead of just the non-leaf nodes, DFT would only work when given a graph over a finite type. Our definition of $\mathsf{Parents}_G$ permits graphs with infinite isolated nodes, supporting standard graph constructions with natural numbers for nodes, for example the graph defined by

$$\lambda x.\ \mathtt{if}\ 0 < x < 3\ \mathtt{then}\ [x+1]\ \mathtt{else}\ [\,]$$

which can be pictured as

$$\cdots\ 0 \quad 1 \to 2 \to 3 \quad 4 \quad 5\ \cdots$$

Higher order logic allows the definition of a type constructor α finite which has the property that, for any type τ, the set of values of type τ finite is finite. Further, if the set of values of type τ is finite, then both sets have the same cardinality. By using $G : \alpha\ \mathsf{finite} \to \alpha\ \mathsf{finite\ list}$, DFT would always terminate. However, such types do not seem to be available in functional programming languages.[1]

[1] John Harrison showed us the possibility of type constructors such as finite.

```
DFT (toGraph A) f seen [] acc = acc
DFT (toGraph A) f seen (visit_now :: visit_later) acc =
  if mem visit_now seen
    then DFT (toGraph A) f seen visit_later acc
    else DFT (toGraph A) f (visit_now :: seen)
                           ((toGraph A) visit_now @ visit_later)
                           (f visit_now acc)
```

Fig. 5 Derived presentation of DFT for adjacency lists

3.3.3 Adjacency lists

The representation of a graph as a function is general enough to encapsulate graphs defined in other representations, *e.g.*, adjacency lists. An adjacency list $A : (\alpha \times \alpha \text{ list}) \text{ list}$ gives a listing of nodes paired with their children. The toGraph function converts an adjacency list into a graph.

Definition 7 (toGraph)

$$\begin{array}{l}\text{toGraph } a\ell\ n = \\ \quad \texttt{case filter } (\lambda(k, \_).\ (k = n))\ a\ell \\ \quad\quad \texttt{of } [] \to [] \\ \quad\quad\ \mid (\_, x) :: t \to x\end{array}$$

The following theorem guarantees that DFT always terminates on a graph built with toGraph.

Theorem 8

$$\forall a\ell.\ \text{Finite } (\text{Parents}(\text{toGraph } a\ell))$$

Figure 5 presents DFT specialized to adjacency lists. Thus Theorems 7 and 8 allow the derivation of unconstrained versions of DFT directly from the theorem in Fig. 4. The induction theorem for DFT can be similarly specialized. Other formalizations of graph traversals over adjacency lists may be found in [32, 47].

4 Unfold

One expects to find unfolds in a functional programming environment. However, defining an unfold in HOL is not completely straightforward since, unlike a fold, the termination behavior of the unfold depends on its parameters.

Definition 8 (Unfold for lists)

$$\begin{array}{l}\text{unfold} : (\alpha \to \text{bool}) \to (\alpha \to \beta) \to (\alpha \to \alpha) \to \alpha \to \beta \text{ list} \\ \text{unfold } d\ f\ g\ x = \texttt{if } d(x) \texttt{ then } [] \texttt{ else } f(x) :: \text{unfold } d\ f\ g\ (g\ x)\end{array}$$

In an instance of unfold where $d(x)$ is never true unfold will loop forever. The parameters d and g must satisfy termination constraints for the definition of unfold to be accepted. In particular, HOL requires a wellfounded relation $\prec$ such that whenever $d(x)$ holds, then $g(x) \prec x$. Such constraints can be automatically synthesized by analyzing the structure of the proposed recursion equations [56]. Consequently, defining unfold is similar to defining an ordinary recursive function in logic: the only difference is that the resulting definition is constrained by abstract termination requirements which are not provable until the d and g parameters are instantiated. However, one can still prove theorems at this abstract, pre-instantiation, level by using the constrained recursion equation and induction theorem for unfold:

$$\begin{aligned}&\mathsf{WF}(\prec) \wedge (\forall x.\ \neg d(x) \Rightarrow g(x) \prec x) \Rightarrow \\ &\quad \mathsf{unfold}\ d\ f\ g\ x = \mathtt{if}\ d(x)\ \mathtt{then}\ [\,]\ \mathtt{else}\ f(x) :: \mathsf{unfold}\ d\ f\ g\ (g\ x)\end{aligned}$$

$$\begin{aligned}&\mathsf{WF}(\prec) \wedge (\forall x.\ \neg d(x) \Rightarrow g(x) \prec x) \Rightarrow \\ &\quad \forall P.\ (\forall x.\ (\neg d(x) \Rightarrow P\ (g\ x)) \Rightarrow P\ x) \Rightarrow \forall v.\ P\ v\end{aligned}$$

Thus, one works with unfold by manipulating theorems in which applications of unfold are hedged about by abstract termination requirements. In this way, one can for example easily prove the (constrained) *fusion* theorems for fold/unfold combinations [52, 56] used to justify deforestation optimizations in compilers for functional languages. However, unfold is also useful as a general programming technique [17]. In the following, we show that working in HOL does not impede this kind of development.

Motivated by the breadth-first search example of Gibbons and Jones [17], we use unfold to define exhaustive breadth-first search in n-ary trees.

Definition 9 (Breadth-first search on n-ary trees)

$$\mathtt{datatype}\ \alpha\ \mathsf{tree} = \mathsf{Node}\ \mathsf{of}\ \alpha * (\alpha\ \mathsf{tree\ list})$$

$$\begin{aligned}&\mathsf{root}\ (\mathsf{Node}\ a\ \mathit{tlist}) = a \\ &\mathsf{subtrees}\ (\mathsf{Node}\ a\ \mathit{tlist}) = \mathit{tlist}\end{aligned}$$

$$\begin{aligned}&\mathsf{BFS} : (\alpha \to \mathsf{bool}) \to \alpha\ \mathsf{tree\ list} \to \alpha\ \mathsf{list\ list} \\ &\mathsf{BFS}\ P\ \ell = \mathsf{unfold}\ \underbrace{\mathsf{null}}_{d}\ \underbrace{(\mathsf{filter}\ P\ \mathsf{o}\ \mathsf{map\ root})}_{f}\ \underbrace{(\mathsf{flat}\ \mathsf{o}\ \mathsf{map\ subtrees})}_{g}\ \ell\end{aligned}$$

Most implementations of HOL fully automate definitions of algebraic datatypes such as the above definition of n-ary trees (note that constructors in HOL are curried by default).

The BFS function proceeds level-by-level through a list of trees. From this definition as an application of unfold, one can formally derive the recursion equation and induction theorem for BFS:

Theorem 9 (BFS recursion and induction)

$$\begin{aligned}&\mathsf{BFS}\ P\ \ell = \\ &\quad \mathtt{if}\ \mathsf{null}\ \ell\ \mathtt{then}\ [\,] \\ &\quad \mathtt{else}\ \mathsf{filter}\ P\ (\mathsf{map\ root}\ \ell) :: \mathsf{BFS}\ P\ (\mathsf{flat}\ (\mathsf{map\ subtrees}\ \ell))\end{aligned}$$

$$\forall P.\ (\forall \ell.\ (\neg \mathsf{null}\ \ell \Rightarrow P\ (\mathsf{flat}\ (\mathsf{map\ subtrees}\ \ell))) \Rightarrow P\ \ell) \Rightarrow \forall v.\ P\ v$$

The derivation amounts to proving that unfold, as instantiated, always terminates. The termination proof is quite simple, using a measure function tsize that counts the number of Nodes in a list of trees, omitting cons-cells.

Definition 10

$$\begin{aligned}
&\mathsf{tsize}(\mathsf{Node}\ x\ \mathit{tlist}) &&= 1 + \mathsf{ltsize}\ \mathit{tlist}\\
&\mathsf{ltsize}\ [\,] &&= 0\\
&\mathsf{ltsize}\ (h :: t) &&= \mathsf{tsize}\ h + \mathsf{ltsize}\ t
\end{aligned}$$

4.1 Program equivalence

While one could prove the correctness of BFS by proceeding along the same general lines as the correctness proof for DFT, we instead prove that BFS is equivalent to a queue-based breadth-first search.

Definition 11 (Queue-based BFS)

$$\begin{aligned}
&\mathsf{BFSq}\ :\ (\alpha \to \mathsf{bool}) \to \alpha\ \mathsf{tree\ list} \to \alpha\ \mathsf{list}\\
&\mathsf{BFSq}\ P\ [\,] = [\,]\\
&\mathsf{BFSq}\ P\ (\mathsf{Node}\ x\ \mathit{tlist} :: \mathit{rst}) =\\
&\quad \texttt{if}\ P\ x\ \texttt{then}\ x :: \mathsf{BFSq}\ P\ (\mathit{rst}\ @\ \mathit{tlist})\ \texttt{else}\ \mathsf{BFSq}\ P\ (\mathit{rst}\ @\ \mathit{tlist})
\end{aligned}$$

The proof that the two kinds of breadth-first traversal coincide is interesting, since they traverse their data in such different ways: BFS goes level-by-level through the list of trees, while BFSq descends into the tree at the head of the list, appending subtrees to the end of the queue.

Lemma 1

$$\begin{aligned}
\forall \ell_1 \ell_2.\ &\mathsf{BFSq}\ P\ (\ell_1 @ \mathsf{flat}\ (\mathsf{map\ subtrees}\ \ell_2)) =\\
&\quad \mathsf{filter}\ P\ (\mathsf{map\ root}\ \ell_1)\ @\\
&\quad \mathsf{BFSq}\ P\ (\mathsf{flat}\ (\mathsf{map\ subtrees}\ \ell_2\ @\ \mathsf{map\ subtrees}\ \ell_1))
\end{aligned}$$

Proof By list-induction on ℓ_1. □

Lemma 2

$$\forall \ell.\ \mathsf{BFSq}\ P\ \ell = \mathsf{filter}\ P\ (\mathsf{map\ root}\ \ell)\ @\ \mathsf{BFSq}\ P\ (\mathsf{flat}\ (\mathsf{map\ subtrees}\ \ell))$$

Proof By BFSq-induction on ℓ, using Lemma 1. □

Theorem 10 (Equivalence of BFS and BFSq)

$$\forall \mathit{tlist}.\ \mathsf{flat}\ (\mathsf{BFS}\ P\ \mathit{tlist}) = \mathsf{BFSq}\ P\ \mathit{tlist}$$

Proof By BFS-induction on *tlist*, using Lemma 2. □

Although the proof tools in HOL-4 provide enough automation so that the proofs of the above statements are short, phrasing the correct statements and choosing the right induction

theorems was somewhat difficult. By using BFS-induction for the main theorem we arranged a goal—essentially the statement of Lemma 2—that was purely about BFSq, considerably simplifying the development. Lemma 1 is the crucial technical lemma, relating BFSq to the recursion structure of BFS. We were unable to mentally conjure up the correct statement of Lemma 1 just by pondering the two recursive functions; instead the theorem prover led us to it in the course of trying to prove Lemma 2.

Again, the termination constraints on unfold allow us to remain in a set-theoretic setting, rather than forcing us to build our formalization on top of domain theory, as the approach of Gibbons and Jones [17] would mandate. Our approach supports the free mixture of ordinary mathematics and programs, avoiding the tedious business of lifting elementary types like lists and trees into domains.

Remark 2 Many functional programs can be viewed as instantiations of very general patterns of recursion known as *recursion schemes* [30]. A scheme represents a class of terminating programs each of which is obtainable by instantiating parameters in the scheme and then proving termination. For example, the class of programs described by *linear* recursion is captured by the following scheme:

$$\begin{aligned}&\mathsf{linRec} : (\alpha \to \mathsf{bool}) \\ &\qquad \to (\alpha \to \alpha) \to (\alpha \to \beta) \to (\beta \to \gamma \to \beta) \to (\alpha \to \gamma) \to \alpha \to \beta \\ &\mathsf{linRec}\ d\ e\ f\ g\ h\ (x) = \\ &\quad \texttt{if}\ d(x)\ \texttt{then}\ f(x)\ \texttt{else}\ g\ (\mathsf{linRec}\ d\ e\ f\ g\ h\ (e\ x))\ (h\ x)\end{aligned}$$

The unfold function, itself a recursion scheme, is an instance of linRec:

$$\mathsf{unfold}\ d\ f\ g = \mathsf{linRec}\ \underbrace{d}_{d}\ \underbrace{g}_{e}\ \underbrace{(\lambda x.\ [\,])}_{f}\ \underbrace{(\lambda x\ y.\ y :: x)}_{g}\ \underbrace{f}_{h}$$

Recursion schemes have been used to justify high-level program transformations such as equivalences between recursive and tail-recursive presentations of a function, or fusion theorems. Shankar [52] and Slind [56] showed how recursion schemes can be supported in higher-order logic mechanizations.

An interesting connection between program schemes and functional programming is illustrated in Lewis *et al.* [36] wherein free variables are used to support a degree of dynamic scoping in a statically scoped functional programming language. A program is treated as being parameterized by its free variables: in essence, although Lewis *et al.* do not mention it, such a program is a program scheme.

We have now discussed two examples in which partial functions are adapted to fit in a world of total functions. In the depth-first search example, a constraint on the graph sufficient to ensure termination needed to be formulated and included in the definition. In the unfold example, the mechanization automatically synthesized abstract termination constraints in order to make a stand-alone definition. These constraints had to be eliminated in order to reason about the breadth-first traversal instantiation of unfold. Thus the main technique is the application of additional constraints that ensure totality.

In the next two sections, we deal with unconstrained recursion: one recursive function that apparently needs no termination argument; and another which is total without constraint, obtained by adding a termination check to a non-terminating program.

5 While loops

It is a curious fact that higher-order logic, although a logic of total functions, allows the definition of functions that don't seem total, at least from a computational perspective. A trivial instance of this is the theorem

$$\forall f\ x.\ f(x) = f(x)$$

which is just an instance of reflexivity. Read as a function definition, this would seem to allow an (admittedly useless) non-terminating recursion. A more interesting example is WHILE-loops.

Theorem 11 (WHILE)

$$\mathsf{WHILE}\ P\ g\ x = \mathtt{if}\ P\ x\ \mathtt{then}\ \mathsf{WHILE}\ P\ g\ (g\ x)\ \mathtt{else}\ x$$

Clearly, if P in this theorem was instantiated to $\lambda x.\mathsf{true}$, the resulting instance of WHILE would 'run forever' if executed. Why is such an "obviously" partial function definable in higher-order logic?

The answer is originally due to J Moore, in the context of ACL2. In higher-order logic, Moore's insight can be formalized in a subtle definition for WHILE. Consider the following total and non-recursive function:

$$\begin{array}{l}\lambda x.\ \mathtt{if}\ \exists n.\ \neg P\ (g^n\ x)\\ \qquad \mathtt{then}\ g^{(\varepsilon n.\ \neg P\ (g^n\ x)\ \wedge\ \forall m.\ m<n \Rightarrow P\ (g^m\ x))}\ x\\ \qquad \mathtt{else}\ \mathsf{ARB}\end{array}$$

This function does a case analysis on the iterations of function g: if there is an n such that $P(g^n\ x)$ fails to hold, it picks the least such n, *i.e.*, the n at which iteration stops, and returns $g^n\ x$. On the other hand, infinite iterations are mapped to ARB. This function is used as the witness for f in the proof of the following theorem:

$$\forall P\ g.\ \exists f.\ \forall x.\ f\ x = \mathtt{if}\ P\ x\ \mathtt{then}\ f\ (g\ x)\ \mathtt{else}\ x$$

which justifies the introduction of a constant WHILE satisfying the recursion equation of Theorem 11.

In other words, some recursive functions may be defined in higher-order logic by constructions that avoid the wellfounded recursion theorem, and thus seem to avoid consideration of termination.

However, two facts prevent this from being an effective panacea for those who wish to ignore termination issues. First, the trick used in the modelling only applies to tail-recursive functions. Second, any inductive reasoning about a function defined in this manner will still require a proof of termination in order to instantiate the wellfounded induction theorem, for example to prove a Hoare-style WHILE rule. So termination cannot be avoided, if one wants to reason about such functions after all. Further discussion of termination-free tail recursion may be found in [3, 43].

6 Regular expression matching

In this section we examine several algorithms for regular expression matching. The majority of the discussion concerns a matcher we developed to overcome a termination-related issue

$$
\begin{array}{lll}
\mathsf{re} = \varepsilon & & \text{empty string (Epsilon)} \\
\mid \mathsf{C} & & \text{character set, (Charset)} \\
\mid (\mathsf{re} \mid \mathsf{re}) & & \text{union of regular expressions (Or)} \\
\mid (\mathsf{re} \cdot \mathsf{re}) & & \text{concatenation of regular expressions (Then)} \\
\mid \mathsf{re}^* & & 0 \text{ or more repetitions (Repeat)}
\end{array}
$$

$$
\frac{}{\mathsf{sem}\ \varepsilon\ [\,]} \qquad \frac{\mathsf{mem}\ c\ \mathsf{C}}{\mathsf{sem}\ \mathsf{C}\ [c]} \qquad \frac{\mathsf{sem}\ r_1\ w}{\mathsf{sem}\ (r_1 \mid r_2)\ w} \qquad \frac{\mathsf{sem}\ r_2\ w}{\mathsf{sem}\ (r_1 \mid r_2)\ w}
$$

$$
\frac{\mathsf{sem}\ r_1\ w_1 \qquad \mathsf{sem}\ r_2\ w_2}{\mathsf{sem}\ (r_1 \cdot r_2)\ (w_1\ @\ w_2)} \qquad \frac{\mathsf{sem}\ r\ w_1 \wedge \ldots \wedge \mathsf{sem}\ r\ w_n}{\mathsf{sem}\ (r^*)\ (w_1\ @\ \ldots\ @\ w_n)}
$$

Fig. 6 Regular expression syntax and semantics

raised in a paper by Harper [27]. For contrast we have also included a simple algorithm from Brzozowski [8], which operates along quite different principles.

Harper [27] makes a case for *proof-directed debugging* using the example of an elegant recursive function for matching regular expressions. The matcher diverges for some expressions, but terminates on an easily identified subset, so Harper adds a preliminary normalization pass that converts expressions into the subset. Inspired by this example, we sought to design and formally verify a total regular expression matching function, *i.e.*, one that works without a normalization pass. In our previous examples, we added constraints to partial functions in order to define them in HOL. In this example, we seek a correct total function—and one of the principal difficulties is dealing with termination. We discuss Harper's matcher further in Sect. 6.2.1 and fully in Sect. 6.5.1.

6.1 Syntax and semantics

A regular expression is a member of the re datatype defined in Fig. 6. Regular expressions are matched against strings, represented as lists of characters. We write (sem r w) to mean that expression r matches string w (see Fig. 6).

Because the semantic rules do not specify how to divide w in the Then and Repeat cases, they do not define an algorithmic function. We aim to modify the semantics by removing the guesswork from these cases. Because the string w can be partitioned into w_1 and w_2 in a finite number of ways, we could implement the Then case by exhaustively searching all possible partitions. The Repeat case could be implemented with a similar strategy. Unfortunately this approach leads to somewhat complex code and a very slow function, so our matcher will instead process w character by character.

6.2 Constructing the matcher

We proceed in three phases, first removing the existential quantification from the Then case, then removing it from the Repeat case, and finally modifying the resulting function to ensure termination. This process parallels our original development of the function.

6.2.1 Then

Consider the subset of the regular expressions having no Or or Repeat operations.

$$
\begin{aligned}
&\mathsf{match} : \mathsf{re'\ list} \to \mathsf{char\ list} \to \mathsf{bool}\\
&\mathsf{match}\ [\,]\ [\,] = \mathsf{true}\\
&\mathsf{match}\ [\,]\ \_ = \mathsf{false}\\
&\mathsf{match}\ (\varepsilon :: t)\ w = \mathsf{match}\ t\ w\\
&\mathsf{match}\ (\mathsf{Charset}\ C :: t)\ [\,] = \mathsf{false}\\
&\mathsf{match}\ (\mathsf{Charset}\ C :: t)\ (c :: w) = \mathsf{mem}\ c\ C\ \wedge\ \mathsf{match}\ t\ w\\
&\mathsf{match}\ ((r_1 \cdot r_2) :: t)\ w = \mathsf{match}\ (r_1 :: r_2 :: t)\ w
\end{aligned}
$$

Fig. 7 Preliminary matcher

Definition 12

$$\mathsf{re'} = \varepsilon \mid \mathsf{C} \mid (\mathsf{re'} \cdot \mathsf{re'})$$

A naive, structurally recursive approach does not work for implementing a matcher for re' expressions. When matching $(r_1 \cdot r_2)$ against string w, structural recursion suggests dividing w into w_1 and w_2 and matching r_1 with w_1 and r_2 with w_2; however, this leads to difficulties: for re' expressions one could compute the length of the string that r_1 should match and partition w accordingly, but this approach fails once we add Or expressions, since then r_1 could match strings of many different lengths and we want to avoid checking multiple partitions of w (the approach taken by Thompson [59]).

Harper [27] solves this problem with a continuation-passing-style (CPS) function that matches r_1 against w and builds a function to match r_2 against whatever string the match of r_1 leaves over. This approach preserves structural recursion, and provides a nice example of CPS. In contrast, we adopt a worklist approach that avoids CPS, but is otherwise similar. By using a worklist, our proofs can directly inspect all of the arguments to each recursive call, whereas in the CPS approach, the continuation function hides this data.

A regular expression of type re' is a tree with εs and character sets at its leaves. A string matched by such a regular expression can be obtained by the sequence of non-ε leaves in left-to-right order. To match a regular expression of type re' we flatten the tree on the fly into a list of Charsets. As the first element of the list becomes a Charset, and not a concatenation, we test the head of the string for inclusion in the Charset and then proceed with the flattening. Thus we will be matching a list of regular expressions against a string: matching $[r_1, r_2, \ldots, r_n]$ is the same as matching $(r_1 \cdot r_2 \cdot \cdots \cdot r_n)$. By using the list representation, we can limit our operations to its head: if the head is a Then it will be further flattened, if it is an ε it is ignored, and if it is a Charset, the head of the string will be tested for inclusion. The tail-recursive function in Fig. 7 implements this algorithm. The intended correctness property is $\mathsf{match}\ [r]\ w \Leftrightarrow \mathsf{sem}\ r\ w$.

Remark 3 Instead of matching a list of expressions, we could flatten Thens with the following rule:

$$\mathsf{match}\ ((r_1 \cdot r_2) \cdot r_3)\ w = \mathsf{match}\ (r_1 \cdot (r_2 \cdot r_3))\ w$$

However, looking ahead, the size measures in Sect. 6.3 need to distinguish between original and introduced Thens, so we continue to use the cons representation.

6.2.2 Repeat

Having motivated the basic computational strategy of our algorithm, we now add Or and Repeat. Dealing with Or is quite simple: $(r_1 \mid r_2)$ matches w if either r_1 matches w or r_2 matches w. Adding Repeat seems at first equally simple. The identity $r^* = \varepsilon \mid (r \cdot r^*)$

$$
\begin{aligned}
&\mathsf{match} : \mathsf{re\ list} \to \mathsf{char\ list} \to \mathsf{bool}\\
&\mathsf{match}\ [\,]\ [\,] = \mathsf{true}\\
&\mathsf{match}\ [\,]\ \_ = \mathsf{false}\\
&\mathsf{match}\ (\varepsilon :: t)\ w = \mathsf{match}\ t\ w\\
&\mathsf{match}\ (\mathsf{Charset}\ C :: t)\ [\,] = \mathsf{false}\\
&\mathsf{match}\ (\mathsf{Charset}\ C :: t)\ (c :: w) = \mathsf{mem}\ c\ C\ \wedge\ \mathsf{match}\ t\ w\\
&\mathsf{match}\ ((r_1 \cdot r_2) :: t)\ w = \mathsf{match}\ (r_1 :: r_2 :: t)\ w\\
&\mathsf{match}\ ((r_1 \mid r_2) :: t)\ w = \mathsf{match}\ (r_1 :: t)\ w\ \vee\ \mathsf{match}\ (r_2 :: t)\ w\\
&\mathsf{match}\ (r^* :: t)\ w = \mathsf{match}\ t\ w\ \vee\ \mathsf{match}\ (r :: r^* :: t)\ w
\end{aligned}
$$

Fig. 8 Regular expression matcher as a partial function

provides enough semantic information to directly implement matching for the Repeat case. The matcher for full regular expressions in Fig. 8 uses this idea. This matcher simply and directly expresses our intuition regarding how to match a regular expression against a string. Unfortunately, it is too good to be true: the recursion in the Repeat clause may diverge. One can easily verify that match loops when matching either ε^* or $(\varepsilon \mid a)^*$ against a non-empty string. Simply put: if match recursively processes $(r \cdot r^*)$ while processing r^*, it may diverge if r can match the empty string.

We have arrived back at our starting point: the matcher in Fig. 8 fails to terminate on the same set of inputs as Harper's CPS function, for the same reason. (Section 6.5.1 discusses restoration of termination for the CPS function.) Unlike our previous two examples, we shall eliminate all divergence from match instead of constraining it to terminating cases.

6.2.3 Restoring termination

Our approach to the termination of match is inspired by the depth-first traversal algorithm of Sect. 3, which tracks nodes as they are visited to ensure that no node is processed multiple times. Provided that the graph is finite, the algorithm must then terminate.

The non-termination observed in Sect. 6.2.2 comes from a cycle in the graph of recursive calls, and not from an infinite number of different calls, so we apply the depth-first traversal methodology and keep a list of the arguments to all the recursive calls. The matcher will detect calls that have happened previously and prune those branches of the computation. Because it only needs to detect cycles, and not nodes revisited due to sharing in the graph, the matcher does not need a global record of the arguments with which the function has been invoked. A *stop-list* of the arguments along the path from the start of computation to the current node suffices.

Scanning the path that the function has taken from the root on each recursive call is a brute force approach to detecting cycles in its execution. However, three observations allow a more elegant and efficient implementation.

Observation 1 Every cycle must include the Repeat case since the matcher without this case is easily proven to terminate. So the path is only checked when entering the Repeat case, and therefore the matcher only needs to keep a stop-list containing the history of the entries into the Repeat case.

Observation 2 Whenever the function matches a character from the string, the computation cannot revisit a node, as none of the recursive calls allow the string to grow. Thus the stop-list need not track the string argument; the stop-list can simply be reset whenever the function

matches a character. If the current regular expression is in the stop-list, then the string must not have changed between the two calls and the function has entered a cycle. We have now concluded that any cycle can be detected by keeping a stop-list comprising only the first argument to the Repeat clause, *i.e.*, the stop-list will be a list of $(r^* :: t)$ entries.

Observation 3 A careful analysis of the Repeat case shows that the tail of the stop-list is never considered. Suppose that matching $r^* :: t$ against w causes a cycle such that $r^* :: t$ will be matched against w again. The following diagram illustrates the path of recursive calls (even though only calls starting with a Repeat would be recorded in the list):

$$(r^* :: t) \to (r :: r^* :: t) \to \cdots \to (r^* :: t)$$

If r contains no Repeat expressions, only the front of the stop-list will be needed, because the next Repeat encountered is the cycle. As an example, let $r = \varepsilon \mid \mathsf{a}$.

$$\begin{aligned}(\varepsilon \mid \mathsf{a})^* :: t &\to (\varepsilon \mid \mathsf{a}) :: (\varepsilon \mid \mathsf{a})^* :: t \\ &\to \varepsilon :: (\varepsilon \mid \mathsf{a})^* :: t \\ &\to (\varepsilon \mid \mathsf{a})^* :: t\end{aligned}$$

If r does contain a Repeat expression, say p^*, then the sequence can be refined as

$$\begin{aligned}r^* :: t &\to r :: r^* :: t \\ &\xrightarrow{*} p^* :: \cdots :: r^* :: t \\ &\to p :: p^* :: \cdots :: r^* :: t \\ &\xrightarrow{*} r^* :: t\end{aligned}$$

and upon reaching the second $(r^* :: t)$, r^* will not be the head of the stop-list. However, in

$$p :: p^* :: \cdots :: r^* :: t \xrightarrow{*} r^* :: t$$

one transition must drop p (because our function only considers the head of the regular expression list), before anything following p can be dropped. Thus the function must pass through $p^* :: ... :: r^* :: t$ a second time, and a cycle will be detected there. For a concrete example, let $r = \varepsilon^*$.

$$\begin{aligned}(\varepsilon^*)^* :: t &\to \varepsilon^* :: (\varepsilon^*)^* :: t \\ &\to \varepsilon :: \varepsilon^* :: (\varepsilon^*)^* :: t \\ &\to \varepsilon^* :: (\varepsilon^*)^* :: t\end{aligned}$$

Thus, cycles can be 'stacked up' and by examining only the top of the stack, we can always detect cycles. Therefore, only the top of the stack need be kept. When the matcher detects a cycle, it immediately returns false. Because all branches in the computation tree are combined with $\vee$, a false return value should not affect the computation. Figure 9 gives the final, correct, algorithm. Notice that this matcher has exactly the same structure as the matcher of Fig. 8, but has an extra *stop* argument which is only used to detect cycles.

Remark 4 (Equality) We have left the type of a regular expression's characters unspecified. The algorithm places a constraint on this choice: that the characters in a Charset admit an equality test. Further, cycle detection relies on the ability to test lists of regular expressions for equality ($new\_stop = stop$), forcing re list to support equality as well. For concreteness, we have chosen a list representation for Charsets; others would suffice such as a tree-based

```
match : re list → char list → re list option → bool
match [ ] [ ] _ = true
match [ ] _ _ = false
match (ε :: t) w stop = match t w stop
match (Charset C :: t) [ ] _ = false
match (Charset C :: t) (c :: w) _ = mem c C ∧ match t w NONE
match ((r1 · r2) :: t) w stop = match (r1 :: r2 :: t) w stop
match ((r1 | r2) :: t) w stop = match (r1 :: t) w stop ∨ match (r2 :: t) w stop
match (r* :: t) w stop =
    let new_stop = SOME (r* :: t) in
      if new_stop = stop then false
      else match t w stop ∨ match (r :: r* :: t) w new_stop
```

Fig. 9 Regular expression matcher as a total function

set representation. However, the elegant representation of Charset by a predicate (*i.e.*, a function of type $\alpha \to$ bool) does not support the computable equality required by a programming language, although it would be fine in HOL. Thus, moving terminating pure functional programs into HOL is usually straightforward; in contrast, transporting a HOL function to a functional programming language can be more problematic, since HOL functions freely rely on extensional equality. This subtlety is another instance of imprecision in the connection between functional programming and higher-order logic.

6.3 Formalizing termination

We formally prove termination by exhibiting a wellfounded relation $\prec$ over the arguments to match. Part of the $\prec$ relation corresponds to the finiteness of the graph mentioned in the previous section and part of it corresponds to the impossibility of a cycle. It will be instructive to step through the construction of $\prec$ instead of simply defining it and showing the proof.

The Charset case recurs with a shorter string. Certainly, when the string shrinks, the function is closer to termination. The Epsilon and Or cases make the first argument, a regular expression list, smaller. To take both into account, let $\prec$ be a lexicographic combination of the two measures. In the Repeat case, the string size must be more significant, because the string never gets larger, whereas the regular expression might. Thus, we have (for the moment) the relation

$$(r\,\ell', w', stop') \prec (r\,\ell, w, stop)$$
$$\text{iff}$$
$$(\text{length } w', \text{size } r\,\ell') <_{lex} (\text{length } w, \text{size } r\,\ell)$$

We have not yet made *size* precise. The size function of a data structure sometimes counts all of the constructors. Since match operates on a list of regular expressions, that would mean counting the :: occurrences, along with Epsilon, Charset, Or, Then and Repeat constructors. (An alternative approach omits leaf nodes from the count, *e.g.*, the Epsilon constructor would not be counted for the same reason [ ] is omitted from the length measure on a list; however, inspection of the Epsilon case of the algorithm reveals the need to count leaf constructors.) The concatenation case removes one Then but adds two occurrences of cons. Thus the size

of the regular expression list should count only the regular expression constructors and not the conses in the list (reminiscent of BFS's size measure).

$$\mathsf{size}\ [r_1, \ldots, r_n] = \mathsf{size}_{re}\ r_1 + \cdots + \mathsf{size}_{re}\ r_n$$

It is easy to verify that all but one recursive call shrinks by $\prec$ with size defined in this way. The unfolding step in the Repeat clause increases the size of the expression without shortening the string. Since the current $\prec$ does not inspect the *stop* argument, and the *stop* argument is the only reason the function always terminates for this case, $\prec$ should not be expected to work yet.

The next step is inspired by the termination of β reduction in the simply typed lambda calculus: β steps may make a term larger, but they reduce the depth of arrows in the types of sub-terms. Similarly, we hoped that the star height[2] of the list of regular expressions would be reduced. Unfortunately, in going from $r^* :: t$ to $r :: r^* :: t$, the star height of the list cannot go down. However, the star height of r is less than that of r^* (by exactly one), so the star height of the head of the regular expression list goes down. We add this as a third component to $\prec$, between the existing two:

$$\begin{array}{l}(r\ell', w', stop') \prec (r\ell, w, stop)\ \mathsf{iff}\\ (\mathsf{length}\ w', \mathsf{headStarHeight}\ r\ell', \mathsf{size}\ r\ell') <_{lex}\\ (\mathsf{length}\ w, \mathsf{headStarHeight}\ r\ell, \mathsf{size}\ r\ell)\end{array}$$

Alas, this prospective $\prec$ no longer works in the Epsilon case, which takes the tail of the list. Since we know nothing of the star height of the head of this tail, we cannot prove that it is not larger. Of course, this attempted $\prec$ was doomed since it still ignores the stop argument. We must consider the star height of at least some prefix of the list, but not all of the list since $r :: r^* :: t$ has the same star height as $r^* :: t$. Now we turn to the *stop* argument, for it records where $r^* :: t$ starts. We thus consider the star heights of the regular expression list only up to the portion of the list that equals the stop argument (if there is one). Replacing headStarHeight with frontStarHeight finally yields a working $\prec$.

Definition 13 (Termination relation for match)

$$\begin{array}{l}\mathsf{frontStarHeight} : \mathsf{re\ list} \to \mathsf{re\ list\ option} \to \mathsf{num}\\ \mathsf{frontStarHeight}\ [\,]\ s\ell = 0\\ \mathsf{frontStarHeight}\ (h :: t)\ s\ell =\\ \quad \mathtt{if}\ s\ell = \mathsf{SOME}\ (h :: t)\ \mathtt{then}\ 0\\ \quad \mathtt{else}\ \mathsf{max}\ (\mathsf{starHeight}\ h)\ (\mathsf{frontStarHeight}\ t\ s\ell)\end{array}$$

$$\begin{array}{l}(r\ell', w', stop') \prec (r\ell, w, stop) \Leftrightarrow\\ (\mathsf{length}\ w', \mathsf{frontStarHeight}\ r\ell'\ stop', \mathsf{size}\ r\ell') <_{lex}\\ (\mathsf{length}\ w, \mathsf{frontStarHeight}\ r\ell\ stop, \mathsf{size}\ r\ell)\end{array}$$

The $\prec$ relation is wellfounded because it is the lexicographic combination of wellfounded relations (each is the $<$ relation on natural numbers). Given $\prec$, HOL can prove the termination of match with a few term rewriting commands. We now consider each case, where we give the arguments to the call, followed by the corresponding right hand side.

[2]The *star height* of a regular expression is the maximum nesting depth of Repeats in the expression; the star height of a list of regular expressions is then the maximum of the star heights of the elements.

– $(\mathsf{Charset}\ C :: t, c :: w, stop) \Rightarrow \mathsf{mem}\ c\ C\ \wedge\ \mathsf{match}\ t\ w\ \mathsf{NONE}$
  In this case, the string shrinks from $c :: w$ to w.
– $(\varepsilon :: t, w, stop) \Rightarrow \mathsf{match}\ t\ w\ stop$
  $((r_1 \cdot r_2) :: t, w, stop) \Rightarrow \mathsf{match}\ (r_1 :: r_2 :: t)\ w\ stop$
  $((r_1 \mid r_2) :: t, w, stop) \Rightarrow \mathsf{match}\ (r_1 :: t)\ w\ stop \vee$
  $\mathsf{match}\ (r_2 :: t)\ w\ stop$

  In these cases, the string is unchanged, the frontStarHeight stays the same, and the size of the regular expression list decreases.

– $((r^* :: t), w, stop) \Rightarrow$
  `let` $new\_stop = \mathsf{SOME}\ (r^* :: t)$ `in`
  `if` $new\_stop = stop$ `then` false
  `else` $\mathsf{match}\ t\ w\ stop\ \vee\ \mathsf{match}\ (r :: r^* :: t)\ w\ new\_stop$

  In the first recursive call, the string stays the same, but it throws away an r^*, which looks like it could make frontStarHeight larger if $r^* :: t$ was the stop argument and therefore frontStarHeight was 0. Inspection of the if expression assures us that in that case, the recursive call is never invoked. Thus frontStarHeight stays the same (or shrinks) and the regular expression list gets smaller. In the second recursive call frontStarHeight shrinks because it used to be at least $\mathsf{starHeight}(r^*)$ (again, if it was 0, the recursive call would not be reached) and is now exactly $\mathsf{starHeight}(r)$ due to the new stop argument.

6.4 Correctness proof

We now turn to the proof that the terminating match algorithm correctly implements the semantic specification of regular expressions.

Theorem 12

$$\mathsf{sem}\ r\ w = \mathsf{match}\ [r]\ w\ \mathsf{NONE}$$

This theorem states that match, when called with appropriate arguments, has the same result as the semantics. We split the equality into two implications.[3]

Lemma 3

$$\mathsf{match}\ \ell\ w\ s \Rightarrow \mathsf{sem}\ (\mathsf{fromList}\ \ell)\ w$$

Lemma 3 follows easily from the match function's induction principle.

Lemma 4

$$\mathsf{sem}\ r\ w \Rightarrow \mathsf{match}\ [r]\ w\ \mathsf{NONE}$$

Lemma 4 is significantly more difficult. Our attempts to prove it by induction, either on the definition of sem or on the definition of match, failed because we were unable to find an appropriately strengthened induction hypothesis (or equivalently, a strong enough invariant).

[3] $\mathsf{fromList}(\ell)$ converts a list of regular expressions ℓ into a single regular expression *i.e.*, $\mathsf{fromList} = \mathsf{foldr}\ (\cdot)\ \varepsilon$.

The case in match that detects a cycle and returns false confounds the local reasoning of inductive approaches because it returns false at a point that may actually lead to a match if allowed to continue (recall that the semantic specification places no constraints on evaluation order). Induction cannot properly account for the stop accumulator because the accumulator remembers state from before the previous iteration.

To begin a new approach, we note that since all branches in the match function are combined with $\vee$ operations, the matcher returns true iff one of the leaves returns true. If matching gets cut off prematurely in one branch by the cycle detector, we only need to show that it will still match along a different path. To construct this path we use an auxiliary data structure (in the proof only, we do not modify the match function) we call a *match sequence*.

Intuitively, a match sequence corresponds to a path through the computation tree of the partial matcher of Fig. 8 ending at a true answer. We formally define a match sequence as a list where each element is related to the following element by a one-step relation $\rightsquigarrow$ and the last element is $([\,], [\,], s)$ for some s. The relation $\rightsquigarrow$ (written as an infix operator on triples) relates arguments of match, $(r\ell, w, stop)$ to $(r\ell', w', stop')$ if and only if match $r\ell'\ w'\ stop'$ can be a recursive call from match $r\ell\ w\ stop$.

Definition 14 (Match sequence)

$$\begin{aligned}\rightsquigarrow\colon\ & (\mathsf{re\ list} * \mathsf{char\ list} * \mathsf{re\ list\ option}) \to \\ & (\mathsf{re\ list} * \mathsf{char\ list} * \mathsf{re\ list\ option}) \to \mathsf{bool}\end{aligned}$$

$$\begin{aligned}(\varepsilon :: t, w, s) &\rightsquigarrow (t, w, s) \\ (\mathsf{C} :: t, c :: w, s) &\rightsquigarrow (t, w, \mathsf{NONE}) \quad \text{if } (\mathsf{mem}\ c\ \mathsf{C}) \\ ((r_1|r_2) :: t, w, s) &\rightsquigarrow (r_1 :: t, w, s) \\ ((r_1|r_2) :: t, w, s) &\rightsquigarrow (r_2 :: t, w, s) \\ ((r_1 \cdot r_2) :: t, w, s) &\rightsquigarrow (r_1 :: r_2 :: t, w, s) \\ (r^* :: t, w, s) &\rightsquigarrow (t, w, s) \\ (r^* :: t, w, s) &\rightsquigarrow (r :: r^* :: t, w, \mathsf{SOME}(r^* :: t))\end{aligned}$$

A match sequence is then a list of linked single steps.

$$\mathsf{match\_seq} : (\mathsf{re\ list} * \mathsf{char\ list} * \mathsf{re\ list\ option})\ \mathsf{list} \to \mathsf{bool}$$

$$\frac{}{\mathsf{match\_seq}\ [\,]} \qquad \frac{}{\mathsf{match\_seq}\ [\,([\,], [\,], s)\,]} \qquad \frac{s_1 \rightsquigarrow s_2 \quad \mathsf{match\_seq}\ (s_2 :: t)}{\mathsf{match\_seq}\ (s_1 :: s_2 :: t)}$$

A match sequence serves as a witness bridging the semantics and the matching algorithm. Match sequences closely correspond to the transitive closure of $\rightsquigarrow$, but the list representation supports typical functional programming over match sequences, unlike the transitive closure representation.

Three more steps finish the proof of correctness.

Step 1. Given a regular expression and string that match, we first construct a match sequence from the semantics. If sem $r\ w$, we can build a match sequence that starts with the desired initial arguments to match.

$$\mathsf{sem}\ r\ w \Rightarrow \exists ms.\ \mathsf{match\_seq}\ (([r], w, \mathsf{NONE}) :: ms)$$

This lemma implies that the partial matcher of Fig. 8 matches whenever the semantics do (provided it terminates). The proof proceeds by induction on the structure of regular expressions and hinges upon our ability to sequentially compose match sequences. If

sem $(r_1 \cdot r_2)$ $(w_1 \mathbin{@} w_2)$, then we get two match sequences ms_1 and ms_2 by inductive hypothesis, the first from sem r_1 w_1 and the second from sem r_2 w_2. The needed match sequence corresponds intuitively to $(ms_1 \mathbin{@} ms_2)$. As such, this list does not satisfy the definition of match sequence, but a somewhat more complicated composition function does produce a match sequence.

We would now like to use the match sequence as a witness for the match function. However, the witness may contain elements of the form $(r^* :: t,\ w,\ \mathsf{SOME}(r^* :: t))$. Such elements cause match to return false although the match sequence continues on. The construction of match sequences does not easily allow us to preclude generating such elements, so we remove them in the next step.

Step 2. The match sequence is normalized to remove any useless cycles, as codified by hasCycle.

Definition 15

$$\mathsf{hasCycle}(ms) = \exists r\ t\ w.\ \mathsf{mem}\ (r^* :: t,\ w,\ \mathsf{SOME}(r^* :: t))\ ms$$

The hasCycle predicate exactly captures the arguments that the match function must avoid, and is so named because these bad arguments arise from the function entering into a cycle. The NS function normalizes the match sequence by repeatedly applying the following transformation. A sequence

$$\begin{aligned} &\cdots \rightsquigarrow (r\ell_1, w_1, stop_1) \\ &\quad \rightsquigarrow \underbrace{(r\ell_2, w_2, stop_2) \rightsquigarrow \cdots \rightsquigarrow (r\ell_2, w_2, \mathsf{SOME}\ r\ell_2)}_{\text{cycle}} \\ &\quad \rightsquigarrow (r\ell_3, w_3, stop_3) \rightsquigarrow \cdots \end{aligned}$$

becomes

$$\cdots \rightsquigarrow (r\ell_1, w_1, stop_1) \rightsquigarrow (r\ell_2, w_2, stop_2) \rightsquigarrow (r\ell_3, w_3, stop_3) \rightsquigarrow \cdots$$

Normalization preserves the property of being a match sequence.

$$\mathsf{match\_seq}(ms) \Rightarrow \mathsf{match\_seq}(\mathsf{NS}\ ms)$$

A normalized match sequence will not have a cycle because $r\ell$ is only in the *stop* argument when it has already been encountered.

$$\mathsf{match\_seq}(ms) \Rightarrow \neg\mathsf{hasCycle}\ (\mathsf{NS}\ ms)$$

Step 3. Finally, the normalized match sequence is used to guide the proof that the match algorithm works. If we have a match sequence ms that starts at $(r\ell, w, s)$, and if ms has no cycles, then match $r\ell$ w s returns true.

$$\begin{pmatrix} \mathsf{match\_seq}\ ms \wedge \\ \mathsf{hd}(ms) = (r\ell, w, s) \wedge \\ \neg\mathsf{hasCycle}\ ms \end{pmatrix} \Rightarrow \mathsf{match}\ r\ell\ w\ s$$

Because the match sequence closely follows the structure of the match function, this theorem is proved with a simple induction on the match sequence.

6.5 Further challenges

A careful reader might ask if Observation 3 in Sect. 6.2.3 can be used to justify carrying at most one regular expression as the *stop* argument, which would allow the final clause of the definition to be

$$\begin{array}{l}\mathsf{match}\ (r^* :: t)\ w\ stop = \\ \quad \texttt{let}\ new\_stop = \mathsf{SOME}\ r\ \texttt{in} \\ \quad\quad \texttt{if}\ new\_stop = stop\ \texttt{then}\ \mathsf{false} \\ \quad\quad \texttt{else}\ \mathsf{match}\ t\ w\ stop\ \vee\ \mathsf{match}\ (r :: r^* :: t)\ w\ new\_stop\end{array}$$

Although we conjecture that this alteration of the algorithm is correct, the proof seems more difficult than the proof just given.

6.5.1 Harper's matcher

As previously mentioned, our motivation for approaching the regular expression problem comes from Harper [27]. Figure 10 contains his algorithm translated to our notation. The matchCPS function uses a continuation function to accumulate the regular expressions that it needs to match in the future, whereas our match uses an explicit list of regular expressions. As noted previously, matchCPS terminates only for a certain class of regular expression inputs. To formally prove termination of matchCPS, one would need to restrict it to this special class of regular expressions using the same technique as the finiteness check in DFT. Alternately, matchCPS could be modified to explicitly detect cycles with the following change to the Repeat case:

$$\begin{array}{l}\mathsf{matchCPS}\ (r^*)\ w\ k = \\ \quad k\ w\ \vee \\ \quad \mathsf{matchCPS}\ r\ w\ (\boldsymbol{\lambda} x.\texttt{if}\ x = w\ \texttt{then}\ \mathsf{false}\ \texttt{else}\ \mathsf{matchCPS}\ (r^*)\ x\ k)\end{array}$$

Furthermore, the termination proof of the modified matchCPS is essentially the same as Harper's proof for his original function. Both proofs rely mainly on the fact that the continuation k cannot be invoked with a longer string (or an equally-sized string, depending on the case) than the string argument, w, when the continuation was created.

Although we can intuitively understand that this will hold, it seems difficult to capture this intuition with a formal termination relation. Interestingly, Xi's DML system [62] proves that the following slight modification of the function terminates.

$$\begin{array}{l}\mathsf{matchCPS}\ (r^*)\ w\ k = \\ \quad k\ w\ \vee \\ \quad \mathsf{matchCPS}\ r\ w\ (\boldsymbol{\lambda} x.\texttt{if}\ \mathsf{length}\ x = \mathsf{length}\ w\ \texttt{then}\ \mathsf{false} \\ \qquad\qquad\qquad\qquad \texttt{else}\ \mathsf{matchCPS}\ (r^*)\ x\ k)\end{array}$$

Our HOL-based approach can easily prove termination when the comparison is $\mathsf{length}\ x \geq \mathsf{length}\ w$ but seems to struggle with Xi's formulation with $\mathsf{length}\ x = \mathsf{length}\ w$. Therefore, this style of program provides an interesting point of comparison between a syntactically based approach, such as Xi's, and a denotational one, such as ours.

We expect a correctness proof for matchCPS to mirror the correctness proof of match, the primary difficulty being the false return upon cycle detection.

$$
\begin{aligned}
&\mathsf{matchCPS} : \mathsf{re} \to \mathsf{char\ list} \to (\mathsf{char\ list} \to \mathsf{bool}) \to \mathsf{bool}\\
&\mathsf{matchCPS}\ \varepsilon\ w\ k = k\ w\\
&\mathsf{matchCPS}\ (\mathsf{Charset}\ C)\ [\,]\ k = \mathsf{false}\\
&\mathsf{matchCPS}\ (\mathsf{Charset}\ C)\ (c :: w)\ k = \mathtt{if}\ \mathsf{mem}\ c\ C\ \mathtt{then}\ k\ w\ \mathtt{else}\ \mathsf{false}\\
&\mathsf{matchCPS}\ (r_1 \cdot r_2)\ w\ k = \mathsf{matchCPS}\ r_1\ w\ (\lambda w'.\ \mathsf{matchCPS}\ r_2\ w'\ k)\\
&\mathsf{matchCPS}\ (r_1 \mid r_2)\ w\ k = \mathsf{matchCPS}\ r_1\ w\ k\ \vee\ \mathsf{matchCPS}\ r_2\ w\ k\\
&\mathsf{matchCPS}\ (r^*)\ w\ k = k\ w\ \vee\ \mathsf{matchCPS}\ r\ w\ (\lambda x.\ \mathsf{matchCPS}\ (r^*)\ x\ k)
\end{aligned}
$$

Fig. 10 Harper's matcher

6.6 Matching extended regular expressions

After the lengthy proofs above, it may be reassuring to know that there are also simple algorithms for regular expression matching. For example, Brzozowksi's classic paper on derivatives of regular expressions [8] introduces an elegant method for computing a minimal deterministic automaton equivalent to a given *extended* regular expression. To obtain the extended regular expressions, Brzozowski added intersection and complementation to the usual operations, thus obtaining all boolean operations on regular expressions. In passing, the paper also describes a very simple matcher.

In Fig. 11 we provide the new constructors for the type (named rex) and their semantics. Notice that the type rex allows the *empty* regular expression $\emptyset$ to be defined as $\neg((c_1 \mid c_2 \mid \cdots \mid c_n)^*)$, where $\mathsf{char} = \{c_1, \ldots, c_n\}$. The notion of a regular expression being *nullable*, *i.e.*, having ε in its language is also needed.

Definition 16 $\mathsf{nullable} : \mathsf{rex} \to \mathsf{bool}$

$$
\begin{aligned}
&\mathsf{nullable}\ (\varepsilon) = \mathsf{true}\\
&\mathsf{nullable}\ (\mathsf{Charset}\ \_) = \mathsf{false}\\
&\mathsf{nullable}\ (\mathsf{Not}\ r) = \neg\mathsf{nullable}\ r\\
&\mathsf{nullable}\ (r_1 \mid r_2) = \mathsf{nullable}\ r_1 \vee \mathsf{nullable}\ r_2\\
&\mathsf{nullable}\ (r_1\ \ \&\ \ r_2) = \mathsf{nullable}\ r_1 \wedge \mathsf{nullable}\ r_2\\
&\mathsf{nullable}\ (r_1 \cdot r_2) = \mathsf{nullable}\ r_1 \wedge \mathsf{nullable}\ r_2\\
&\mathsf{nullable}\ (r^*) = \mathsf{true}
\end{aligned}
$$

$$\mathsf{nullable}\ r \Leftrightarrow \mathsf{sem}\ r\ [\,]\ .$$

The *derivative*[4] of a regular expression with respect to a string of characters can be defined by primitive recursion.

Fig. 11 Extended regular expression syntax and semantics

$$
\begin{array}{lll}
\mathsf{rex} = \ldots & & \text{standard operators}\\
\quad \mid\ \neg\mathsf{rex} & & \text{complement (Not)}\\
\quad \mid\ (\mathsf{rex}\ \ \&\ \ \mathsf{rex}) & & \text{intersection of regular expressions (And)}
\end{array}
$$

$$\frac{\neg\mathsf{sem}\ r\ w}{\mathsf{sem}\ (\neg r)\ w} \qquad \frac{\mathsf{sem}\ r_1\ w \qquad \mathsf{sem}\ r_2\ w}{\mathsf{sem}\ (r_1\ \ \&\ \ r_2)\ w}$$

[4]The name *derivative* is used because some of the equations for D are formally similar to the rules for differentiation.

Definition 17 Derivative : char list → rex → rex

$$\begin{aligned}
&\mathsf{Derivative}\ [\,]\ r = r\\
&\mathsf{Derivative}\ (h :: t)\ r = \mathsf{Derivative}\ t\ (\mathsf{D}\ h\ r)\\
\\
&\mathsf{D}\ c\ \varepsilon = \emptyset\\
&\mathsf{D}\ c\ (\mathsf{Charset}\ C) = \mathtt{if}\ \mathsf{mem}\ c\ C\ \mathtt{then}\ \varepsilon\ \mathtt{else}\ \emptyset\\
&\mathsf{D}\ c\ (\mathsf{Not}\ r) = \mathsf{Not}(\mathsf{D}\ c\ r)\\
&\mathsf{D}\ c\ (r_1 \mid r_2) = (\mathsf{D}\ c\ r_1 \mid \mathsf{D}\ c\ r_2)\\
&\mathsf{D}\ c\ (r_1\ \&\ r_2) = (\mathsf{D}\ c\ r_1\ \&\ \mathsf{D}\ c\ r_2)\\
&\mathsf{D}\ c\ (r_1 \cdot r_2) = ((\mathsf{D}\ c\ r_1) \cdot r_2) \mid (\mathtt{if}\ \mathsf{nullable}\ r_1\ \mathtt{then}\ \mathsf{D}\ c\ r_2\ \mathtt{else}\ \emptyset)\\
&\mathsf{D}\ c\ (r^*) = (\mathsf{D}\ c\ r) \cdot r^*
\end{aligned}$$

Intuitively, $\mathsf{D}\ c\ r$ represents the set of strings that r can match after it has matched c. The essential property of D is easily proved by induction on the structure of rex:

$$\mathsf{sem}\ (\mathsf{D}\ c\ r)\ w = \mathsf{sem}\ r\ (c :: w)$$

From this it is a quick step to the correctness of Derivative:

$$\mathsf{sem}\ r\ w \Leftrightarrow \mathsf{nullable}(\mathsf{Derivative}\ w\ r)$$

So a matcher can be built that takes the derivative of r with respect to the given string w and then checks to see if the resulting regular expression is nullable.

This algorithm shows that there are, at least, two fundamentally different ways to match regular expressions containing occurrences of Kleene star. Both approaches are ultimately based on the identity $r^* = \varepsilon \mid r \cdot r^*$. However, the approach taken by match and its associates—although natural—leads to a subtle termination problem, and relatively formidable correctness proofs. In contrast, the derivative-based matcher has a trivial termination problem and an easy correctness proof.

7 Related work

This paper touches on four areas of research: total functional languages, functional programming in proof assistants, termination proofs, and implementations of regular expression matching in functional languages.

7.1 Total functional languages

Perhaps the work closest to ours in the field of functional programming is the notion of *Strong Functional Programming* [60], which emphasizes total functions; since higher-order logics are based on total functions, they can be seen as environments for strong functional programming.

Charity [10] is a category-theory based programming language which builds on ideas in Hagino's thesis [26]. Charity is explicitly founded on folds and unfolds, so its syntax disallows general recursion.

The *Cayenne* system [2] incorporates a powerful (undecidable) type system so that functional programs may be specified by their types. Unlike HOL, it does not require termination

proofs for recursive definitions. As the Cayenne documentation notes, this makes the type system unusable as a proof system. However, in principle, enough proof power is available to prove program termination, should one wish to do so.

7.2 Functional programming in proof assistants

Formalization in higher-order logic typically uses a mixture of recursive functions, recursive datatypes, inductively defined relations, and other mathematics as required. Thus users of proof assistants such as ACL2, HOL, Isabelle/HOL, PVS, Nuprl, Coq, LEGO, *etc*. write formal pure functional programs quite often in the course of building formalizations. Indeed, the original LCF system, for which ML was invented, was aimed at verifying functional programs using Scott's PPλ logic [22]. Similarly, the NQTHM system of Boyer and Moore [7] was also originally aimed at automating induction proofs of pure LISP programs.

ACL2 superseded NQTHM. It provides a healthy subset of Common Lisp to program in and supports a close connection between the object language of the ACL2 logic and the metalanguage in which the implementation is coded (Common Lisp). For example, object-language functions can be compiled and executed as native code. This facility has been used to advantage in large hardware and software formalizations using ACL2 [24, 25]: formal descriptions of the system under investigation can be executed at 'near-C' speeds, and can have theorems proved about them as well. PVS, HOL-4, and Isabelle/HOL [3, 53, 57] also provide ground execution facilities based on the execution engine supplied by the metalanguage (Lisp for PVS, SML for HOL-4 and Isabelle/HOL). Recently, support for testing has been added to higher-order logic environments [4, 13]. As well, some support for polytypism is appearing [57]. Thus it can be seen that interactive proof environments are increasingly taking on features that appear in functional programming environments.

Kreitz [33] has developed a lightweight embedding of a subset of OCAML, plus libraries, in Nuprl in order to import Ensemble [38] programs into Nuprl so that the formal reasoning power of the proof system can be applied to build and optimize network stacks. The embedding uses a state-passing style in order to interpret `ref`-cells, which are used in the imported Ensemble code. Other techniques for dealing with impure aspects of functional languages may be found in the thesis work of Filliâtre [15] and in Krstić and Matthews [34].

7.3 Termination

The ACL2 system attempts to synthesize correct termination relations and automatically prove termination for functions being defined, as do HOL-4 and Isabelle/HOL. However, in any system complex definitions can require an explicitly given termination relation and the termination proof can require substantial guidance. This is an impediment, for it requires prospective users to learn termination notions, the concept of measure functions at least. Even worse, termination problems can be arbitrarily hard, and it follows that users would ultimately be forced to also become knowledgeable about theorem proving. Therefore, more advanced support for termination automation is needed to push onerous termination requirements further away from an average user. Related work in Type Theory frameworks includes Abel and Altenkirch [1] as well as Xi [62]. Some of the most advanced work on termination for functional programs comes from term rewriting systems [18, 19]. The recent approach to automatically proving termination of Lee *et al.* [35] is also promising, and has been incorporated into a termination tool for ACL2 definitions [39].

7.4 Regular expression matchers

Regular expression matchers are, of course, quite well investigated. The standard way to match regular expressions translates them to non-deterministic finite state automata and performs on-the-fly reachability analysis [58]. Nipkow [45] proves the correctness of the (standard) translations of regular expressions to non-deterministic state machines, and then to deterministic machines as part of a formal development of a lexer. In [16] the matching problem is approached from the perspective of using matches to produce structured data.

Brzozowski's work, as developed by Berry and Sethi [5], has been incorporated into a regular-expression matching library in OCAML [40]. Derivative-based matchers are also finding use in XML schema validation [61] and lexer generators. For example, [49] find that the derivative-based approach performs very well in a lexer generator for the DrScheme environment.

The earliest CPS matcher we are aware of appears in a paper by Danvy and Filinski [11]. That program was incorrect, for the same reason that Harper's first matcher failed (it lacked a progress test for the Kleene star). In later work, Danvy and Nielsen [12] present informal correctness proofs of a CPS matcher and a first-order matcher.

Recent applications of regular expressions in functional programming include elegant solutions to classic puzzles [42] and pattern-matching for XML [29].

8 Discussion

The depth-first traversal, unfold, While-loop, and regular expression examples each illustrates different aspects of formalizing functional programs. The DFT program was adapted to higher-order logic by having a (non-computable) user-defined constraint included in the function specification. The unfold recursion scheme had its termination conditions automatically computed by the system. Instantiating the unfold to a particular program, breadth-first search (BFS), required instantiating abstract termination conditions and solving the ensuing concrete termination conditions. The regular expression matcher match illustrates another approach to partiality: sometimes a computable check for divergence can be incorporated into the algorithm itself. Finally, in some cases, it may be possible to avoid termination issues by defining functions with an unrestricted WHILE loop, although reasoning about such functions still requires termination.

Because higher-order logic has a set-theoretic semantics, we used only ordinary mathematics in the termination and correctness proofs for these examples; nowhere did we need to appeal to domain-theoretic constructions, or operational semantics. Additionally, the HOL-4 proof system supported the mechanical verification of the proofs, ensuring their validity.

This paper's approach to functional program verification relies on the similarities between programs in an ML-like language and terms in higher-order logic. Difficulties arise from the differences between the two settings, primarily from higher-order logic's inability to handle partial functions. Our examples illustrate how to overcome this difficulty in practical situations. Indeed, discovering the termination relations themselves was the most difficult part of embedding the programs into HOL, yet termination proof is an inherent challenge in verifying program correctness in any setting. Thus, the true overhead in embedding functional programs in HOL is quite small compared to the benefits of mechanically-checkable mathematical reasoning about them.

What about our vision of using higher-order logic theorem provers as functional programming environments? The examples we have discussed, along with the related work,

show that pure functional programming is starting to be well-supported in such systems. However, that conception of functional programming is too restrictive for many applications. Programs that use imperative features, such as exceptions, reference cells, and I/O, do not usually correspond as closely to HOL terms as our examples. Although the usual transformations, *e.g.*, continuation-passing and store-passing styles, can translate many imperative programs into purely applicative ones, these are *global* transformations and so increase the differences between the program as verified and the program as executed. Furthermore, the meaning of an imperative program often depends upon a defined ordering of evaluation, but logic has no notion of evaluation ordering.

Given the above problems with connecting higher-order logic directly with imperative features, a pure functional language, such as Haskell, might appear to mesh more easily with HOL. Yet Haskell diverges from HOL in another important aspect: its recursive datatypes describe potentially infinite structures whereas the inductive datatypes commonly used in HOL describe only finite structures. The HOLCF extension of Isabelle/HOL formalizes domain theory and also provides a datatype package that supports the definition of lazy datatypes [44]. However, what if one wants to avoid domains? A non-domain-theoretic theory of lazy lists has been built in HOL by Michael Norrish, based on work by John Matthews [41]. These lazy lists can faithfully mirror Haskell's lists, and the usual operations on lazy lists can also be defined. The remaining challenge is automation: at present it seems that the only proof system automating the definition of such co-datatypes and associated co-recursive functions is Coq [20].

Formalizing programs in a logic of total functions has impressed upon us the virtue of totality. Various factors drive programmers when writing code: first and foremost there is, or ought to be, a drive for correctness, but also other considerations are important, such as maintainability, speed, and efficient use of resources. To this list we would like to add *totality*, when it is attainable. Hunting down and eliminating partiality from programs—whether or not they will ever be verified—is simply good programming practice and can even, as in the match example, lead to the invention of interesting new programs.

There is a deep, continuously evolving, connection between higher-order logic and functional programming. This must be so, since they are both rooted in simple type theory. Termination is of great importance in this connection: it is the price a functional program pays to enter the logic so that simple mathematics can be used to reason about it. We think that higher-order logic proof systems have the potential to be fruitful environments for functional programming; in particular, they embody the notion of semantically-based programming environments. As we have seen, that allows formal, mechanically checked proofs of program termination and correctness. However, much more should be possible since program transformation, partial evaluation, and even compilation [23, 37] can be viewed as theorem proving procedures.

Acknowledgements Matthew Flatt, Michael Norrish, Alexander Krauss, and Steven Obua scrutinized drafts of the paper and made useful comments. Mike Gordon pointed us to Harper's algorithm. Thanks also to Olivier Danvy for pointers to CPS regular expression matchers.

References

1. Abel, A., Altenkirch, T.: A predicative analysis of structural recursion. J. Funct. Program. **12**(1), 1–41 (2002)
2. Augustsson, L.: Cayenne—a language with dependent types. In: International Conference on Functional Programming, pp. 239–250 (1998)

3. Berghofer, S., Nipkow, T.: Executing higher order logic. In: Callaghan, P., Luo, Z., McKinna, J., Pollack, R. (eds.) Types for Proofs and Programs (TYPES 2000). Lecture Notes in Computer Science, vol. 2277, pp. 24–40. Springer, New York (2002)
4. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: Second IEEE International Conference on Software Engineering and Formal Methods (SEFM 2004). IEEE Computer Society Press, Silver Spring (2004)
5. Berry, G., Sethi, R.: From regular expressions to deterministic automata. Theor. Comput. Sci. **48**(1), 117–126 (1986)
6. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science, An EATCS Series. Springer, New York (2004)
7. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic, New York (1979)
8. Brzozowski, J.: Derivatives of Regular Expressions. J. ACM **11**(4), 481–494 (1964)
9. Church, A.: A formulation of the simple theory of types. J. Symb. Log. **5**(2), 56–68 (1940)
10. Cockett, R., Fukushima, T.: About charity. Technical Report TR 92/480/18, Department of Computer Science, University of Calgary (1992)
11. Danvy, O., Filinski, A.: Abstracting control. In: Proceedings of the 1990 ACM Conference on LISP and Functional Programming, Nice, New York, NY, pp. 151–160. ACM, New York (1990)
12. Danvy, O., Nielsen, L.R.: Defunctionalization at work. Technical Report RS-01-23, BRICS (2001). Extended version of an article appearing in 3rd International Conference on Principles and Practice of Declarative Programming, PPDP'01 Proceedings, pp. 162–174 (2001)
13. Dybjer, P., Haiyan, Q., Takeyama, M.: Combining testing and proving in dependent type theory. In: Basin, D., Wolff, B. (eds.) Proceedings of TPHOLs 2003. Lecture Notes in Computer Science, vol. 2758, pp. 188–203. Springer, New York (2003)
14. Farmer, W., Guttman, J., Thayer, J.: IMPS: an interactive mathematical proof system. In: Stickel, M. (ed.) Tenth International Conference on Automated Deduction (CADE). Kaiserslautern, pp. 653–654 (1990)
15. Filliâtre, J.-C.: Verification of non-functional programs using interpretations in type theory. J. Funct. Program. **13**(4), 709–745 (2003)
16. Frisch, A., Cardelli, L.: Greedy regular expression matching. In: ICALP 2004. Lecture Notes in Computer Science, vol. 3142, pp. 618–629. Springer, New York (2004)
17. Gibbons, J., Jones, G.: The under-appreciated unfold. In: Proceedings of the third ACM SIGPLAN International Conference on Functional Programming, pp. 273–279. ACM Press, Baltimore (1998)
18. Giesl, J.: Termination analysis for functional programs using term orderings. In: Proceedings of the Second International Symposium on Static Analysis, pp. 154–171. Springer, New York (1995)
19. Giesl, J.: Termination of nested and mutually recursive algorithms. J. Autom. Reason. **19**(1), 1–29 (1997)
20. Giménez, E.: Structural recursive definitions in type theory. In: Proceedings of ICALP'98. Lecture Notes in Computer Science, vol. 1443. Springer, New York (1998)
21. Gordon, M., Melham, T.: Introduction to HOL, a Theorem Proving Environment for Higher Order Logic. Cambridge University Press, Cambridge (1993)
22. Gordon, M., Milner, R., Wadsworth, C.: Edinburgh LCF: A Mechanised Logic of Computation. Lecture Notes in Computer Science, vol. 78. Springer, New York (1979)
23. Gordon, M., Iyoda, J., Owens, S., Slind, K.: Automatic formal synthesis of hardware from higher order logic. In: Proceedings of Fifth International Workshop on Automated Verification of Critical Systems (AVoCS). ENTCS, vol. 145 (2005)
24. Greve, D., Wilding, M., Hardin, D.: High-speed, analyzable simulators. In: Kaufmann, M., Manolios, P., Moore, J. (eds.) Computer-Aided Reasoning Case Studies, pp. 113–135. Kluwer Academic, Dordrecht (2000)
25. Greve, D.A., Kaufmann, M., Manolios, P., Moore, J.S., Ray, S., Ruiz-Reina, J.L., Sumners, R., Vroon, D., Wilding, M.: Efficient execution in an automated reasoning environment. J. Funct. Program. **18**(1), 15–46 (2008)
26. Hagino, T.: A categorical programming language. Ph.D. thesis, University of Edinburgh (1987). Also published as ECS-LFCS-87-38
27. Harper, R.: Proof-directed debugging. J. Funct. Program. **9**(4), 463–470 (1999)
28. Harrison, J.: Inductive definitions: automation and application. In: Schubert, E.T., Windley, P.J., Alves-Fos, J. (eds.) Proceedings of the 1995 International Workshop on Higher Order Logic Theorem Proving and its Applications, Aspen Grove, Utah, pp. 200–213. Springer, New York (1995)
29. Hosoya, H., Pierce, B.: Regular expression pattern matching for XML. J. Funct. Program. **13**(6), 961–1004 (2003)
30. Huet, G., Lang, B.: Proving and applying program transformations expressed with second-order patterns. Acta Inf. **11**, 31–55 (1978)

31. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer Academic, Dordrecht (2000)
32. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: Case Studies. Kluwer Academic, Dordrecht (2000)
33. Kreitz, C.: Building reliable, high-performance networks with the nuprl proof development system. J. Funct. Program. **14**(1), 21–68 (2004)
34. Krstić, S., Matthews, J.: Verifying BDD algorithms through monadic interpretation. In: Cortesi, A. (ed.) Verification, Model Checking and Abstract Interpretation: Third International Workshop (VMCAI 2002). Lecture Notes in Computer Science, vol. 2294. Springer, New York (2002)
35. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 81–92 (2001)
36. Lewis, J.R., Shields, M.B., Meijer, E., Launchbury, J.: Implicit parameters: dynamic scoping with static types. In: Reps, T. (ed.) ACM Symposium on Principles of Programming Languages, Boston, Massachusetss, USA. ACM Press, New York (2000)
37. Li, G., Owens, S., Slind, K.: Structure of a proof-producing compiler for a subset of higher order logic. In: ESOP 2007. Lecture Notes in Computer Science, vol. 4421. Springer, New York (2007)
38. Liu, X., Kreitz, C., Renesse, R., Hickey, J., Hayden, M., Birman, K., Constable, R.: Building reliable, high-performance communication systems from components. In: Proceedings of the 17th ACM Symposium on Operating System Principles. ACM Press, New York (1999)
39. Manolios, P., Vroon, D.: Termination analysis with calling context graphs. In: Ball, T., Jones, R. (eds.) Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 4144, pp. 401–414. Springer, New York (2006)
40. Marché, C.: A simple library for regular expressions. Regexp library for OCaml, available at http://www.lri.fr/~marche/regexp/ (2002)
41. Matthews, J.: Recursive definition over coinductive types. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Thery, L. (eds.) Proceedings of the 12th International Conference on Theorem Proving in Higher Order Logics, TPHOLs'99, Nice. Springer, New York (1999)
42. McIlroy, D.: Enumerating the strings of regular languages (Functional Pearl). J. Funct. Program. **14**(5), 503–518 (2004)
43. Moore, J., Manolios, P.: Partial functions in ACL2. J. Autom. Reason. **31**(2), 107–127 (2003)
44. Müller, O., Nipkow, T., Oheimb, D.v, Slotosch, O.: HOLCF = HOL + LCF. J. Funct. Program. **9**, 191–223 (1999)
45. Nipkow, T.: Verified lexical analysis. In: Grundy, J., Newey, M. (eds.) Theorem Proving in Higher Order Logics. Lecture Notes in Computer Science, vol. 1479, pp. 1–15. Springer, New York (1998). Invited talk
46. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL—A Proof Assistant for Higher-Order Logic. Lecture Notes in Computer Science, vol. 2283. Springer, New York (2002)
47. Nishihara, T., Minamide, Y.: Depth first search. Entry in the Isabelle Archive of Formal Proofs (2004)
48. Norrish, M., Slind, K.: HOL-4 manuals. Available at http://hol.sourceforge.net/ (1998–2005)
49. Owens, S., Flatt, M., Shivers, O., McMullan, B.: Parsing tools in scheme. In: Proceedings of the 2004 Scheme Workshop (2004)
50. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS System Guide. SRI Computer Science Laboratory. Available at http://pvs.csl.sri.com/documentation.shtml (2001)
51. Rudnicki, P., Trybulec, A.: On equivalents of well-foundedness. J. Autom. Reason. **23**(3), 197–234 (1999)
52. Shankar, N.: Steps towards mechanizing program transformations using PVS. In: Moeller, B. (ed.) Mathematics of Program Construction, Third International Conference (MPC'95), Kloster Irsee, Germany, pp. 50–66 (1995)
53. Shankar, N.: Static analysis for safe destructive updates in a functional language. In: Pettorossi, A. (ed.) Logic Based Program Synthesis and Transformation, 11th International Workshop, LOPSTR 2001. Lecture Notes in Computer Science, vol. 2372, pp. 1–24. Springer, New York (2001)
54. Slind, K.: Derivation and use of induction schemes in higher order logic. In: Theorem Proving in Higher Order Logics, Murrary Hill, New Jersey, USA, pp. 275–291. Springer, New York (1997)
55. Slind, K.: Reasoning about terminating functional programs. Ph.D. thesis, Institut für Informatik, Technische Universität München. Available at http://tumb1.biblio.tu-muenchen.de/publ/diss/in/1999/slind.html (1999)
56. Slind, K.: Wellfounded schematic definitions. In: McAllester, D. (ed.) Proceedings of the Seventeenth International Conference on Automated Deduction CADE-17, vol. 1831. Pittsburgh, Pennsylvania, pp. 45–63. Springer, New York

57. Slind, K., Hurd, J.: Applications of polytypism in theorem proving. In: Basin, D., Wolff, B. (eds.) Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rome, Italy, Proceedings. Lecture Notes in Computer Science, vol. 2758, pp. 103–119. Springer, New York (2003)
58. Thompson, K.: Programming techniques: regular expression search algorithm. Commun. ACM **11**(6), 419–422 (1968)
59. Thompson, S.: Regular expressions and automata using Haskell. Technical Report 5-00, Computing Laboratory, University of Kent. Available at http://www.cs.ukc.ac.uk/pubs/2000/958 (2000)
60. Turner, D.A.: Elementary strong functional programming. In: Functional Programming Languages in Education. Lecture Notes in Computer Science, vol. 1022, pp. 1–13. Springer, New York (1995)
61. van der Vlist, E.: Relax NG. O'Reilly (2003)
62. Xi, H.: Dependent types for program termination verification. J. Higher-Order Symb. Comput. **15**, 91–131 (2002)