First-Order Unification by Structural Recursion

Conor McBride
Department of Computer Science
University of Durham
c.t.mcbride@durham.ac.uk

Abstract

First-order unification algorithms (Robinson, 1965) are traditionally implemented via general recursion, with separate proofs for partial correctness and termination. The latter tends to involve counting the number of unsolved variables and showing that this total decreases each time a substitution enlarges the terms. There are many such proofs in the literature, for example, (Manna & Waldinger, 1981; Paulson, 1985; Coen, 1992; Rouyer, 1992; Jaume, 1997; Bove, 1999).

This paper shows how a dependent type can relate terms to the set of variables over which they are constructed. As a consequence, first-order unification becomes a structurally recursive program, and a termination proof is no longer required. Both the program and its correctness proof have been checked using the proof assistant Lego (Luo & Pollack, 1992; McBride, 1999).

Capsule Review

Using first-order unification as an example, McBride proselytizes for a style of programming using structural recursion over indexed families of datatypes. Although this style is not Turing complete, McBride shows that guaranteed termination and the expressiveness of dependent types, is a useful and beautiful paradigm.

1 Introduction

Inductive datatypes give more than just a machine representation for data. They equip it with a natural mode of inspection—case analysis—and a natural mode of terminating computation—structural recursion. Case analysis allows a function over an inductive datatype to decompose its input by stripping off the datatype's constructor symbols, and structural recursion gives access to the outputs for the components so exposed. The structure of the computation is explained in terms of the structure of each input, and the well-foundedness of the latter ensures the termination of the former.

By contrast, functions defined by general recursion can make whatever recursive calls they like, regardless of the structure of their input. There is no *a priori* guarantee that such functions will ever deliver an output. We may need this extra freedom in order to write the programs we want, but we pay with the extra work required to convince ourselves that the structures we exploit, no longer those of the data, really are well-founded. Such programs are still 'effective procedures' in the sense that we can order a computer to execute them. They do not, however, fully express the understanding we have in our minds. What makes general recursion necessary is a 'failure of language': if we could find the structures we need amongst those we can express, we could give the machine not just orders, but an 'effective explanation'.

Gödel's incompleteness theorems tell us that this failure is sometimes inevitable for languages in which all programs terminate: a language restricted to structural recursion cannot encode its own interpreter. However, despite the fact that we cannot always write programs via structural recursion, there is no need to conceal the structural explanations of many common algorithms which do not involve such spectacular bootstrapping. Indeed, the languages we have today are already enough to make structural many algorithms where general recursion is popularly considered essential. The classic example is Quicksort, a non-structural recursion on lists. As David Turner points out (Turner, 1995), the real structure of Quicksort is the tree-like call structure between whose branches the partitioning operation distributes the list elements. If we turn this into a data structure, we acquire exactly the tree-sort algorithm with which Rod Burstall introduced us to case analysis and structural recursion (Burstall, 1969).

Nonetheless, there are many algorithms where the structure in play is not a simple inductive one. For example, first-order unification (Robinson, 1965) is not structurally recursive on terms, because the terms grow larger each time a variable is substituted. Typical implementations use general recursion. The termination proof is usually quite subtle, relying on the 'occur-check' to ensure that each substitution reduces the number of distinct variables remaining in the problem. The usual representation of terms does not take account of the number of variables involved, hence this key piece of structure is hidden.

Dependent type systems give us a much richer language in which to describe data. In particular, we can stratify the type of terms into a family of types, $\mathsf{Term}\ n$, representing terms over a set of n variables. This paper presents a first-order unification algorithm which is structurally recursive on this n: its correctness proof is available online. Both program and proof have been coded and checked in the Extended Calculus of Constructions, as implemented in the proof assistant LEGO (Luo & Pollack, 1992; McBride, 1999). The point is this: dependent types make more recursion structural because dependent types can express more structure.

2 The Programming Language in this Paper

The programs in this paper are all real terms in a real dependent type theory. The full LEGO development is available online. However, LEGO was not designed for presenting programs, so for reasons of legibility, I have disguised this type theory as an imaginary functional language. In doing so, I have borrowed from Haskell and from Standard ML, as well as exploiting the cosmetic potential of LaTeX. Some less superficial aspects deserve more thorough discussion.

2.1 Datatype Declarations and Function Definitions

Where, in Haskell, we might write

$$\text{data}\ \mathbb{N} \;=\; \mathsf{0} \;\mid\; \mathsf{s}\,\mathbb{N}$$

this paper presents datatypes by a formation rule and constructors, in natural deduction style:

$$\text{data}\quad \frac{}{\mathbb{N}\ :\ \mathsf{Type}} \quad \text{where} \quad \frac{}{\mathsf{0}\ :\ \mathbb{N}} \quad \frac{n\ :\ \mathbb{N}}{\mathsf{s}n\ :\ \mathbb{N}}$$

The more familiar notation suggests the form of a typical constructor application, using a type as a placeholder for its inhabitants—$\mathsf{s}\,\mathbb{N}$, for example. In a simply typed setting, there is no risk of mistaking $\mathsf{s}\,\mathbb{N}$ for an actual application, because terms and types inhabit separate worlds. Dependent types break this separation, so there is perhaps less potential for misunderstanding if we avoid writing constructor declarations which resemble ill-typed expressions. The natural deduction style follows the same 'typical application' idea, but with variables as the placeholders. The s constructor may still be used in partially applied form: its actual type is $\mathbb{N} \to \mathbb{N}$, as one might expect.

Another benefit of illustrating typical usage in a declaration is that we can indicate that an argument is to be left implicit—and inferred by the machine—by leaving it as a schematic variable in a rule. Consider, for example

$$\text{data}\quad \frac{T\ :\ \mathsf{Type}}{\mathsf{Maybe}\ T\ :\ \mathsf{Type}} \quad \text{where} \quad \frac{}{\mathsf{no}\ :\ \mathsf{Maybe}\ T} \quad \frac{t\ :\ T}{\mathsf{yes}\ t\ :\ \mathsf{Maybe}\ T}$$

Here, the actual type of no is a dependent function space: $\forall T : \mathsf{Type}.\ \mathsf{Maybe}\ T$, where the return type depends on the type passed as an argument. However, this dependency enables the machine to infer the argument of no, given a desired return type. Similarly, the type of yes is $\forall T\!:\!\mathsf{Type}.\ T \to \mathsf{Maybe}\ T$. Here, $T \to \mathsf{Maybe}\ T$ may be used as syntactic sugar for $\forall t\!:T.\,\mathsf{Maybe}\ T$, as t does not occur in its scope. Correspondingly, we must supply t as an explicit argument when we apply yes, but we may still hide T. Schematic variables not explicitly declared in the premises of a rule yield $\forall$'s in the type of the symbol declared in the conclusion, but the corresponding arguments are left implicit in applications. This notation only gives explicit types to the explicit objects. Arguments which are usually implicit will be written as subscripts if they deserve attention.

Function definitions are introduced by the keyword 'let', with a type signature also in natural deduction style, then an equational presentation of the function. For example, here are two function-transforming operations capturing the monadic behaviour which Maybe supports:

$$\text{let} \quad \frac{f \; : \; S \to T}{\blacktriangleright f \; : \; S \to \mathsf{Maybe}\ T} \qquad\qquad \blacktriangleright f \;\mapsto\; \mathsf{yes} \cdot f$$

$$\frac{f \; : \; S \to \mathsf{Maybe}\ T}{f \blacktriangleleft \; : \; \mathsf{Maybe}\ S \to \mathsf{Maybe}\ T} \qquad \begin{array}{l} f \blacktriangleleft \; \mathsf{no} \quad\;\; \mapsto \mathsf{no} \\ f \blacktriangleleft (\mathsf{yes}\ s) \mapsto f\ s \end{array}$$

The $\cdot$ symbol stands for function composition. Less conventionally, I write $\mapsto$ to indicate a directed computation rule, keeping $=$ for equational propositions like this one:

$$\text{fact} \quad (f \cdot g) \blacktriangleleft s \; = \; ((f \blacktriangleleft) \cdot \blacktriangleright g \blacktriangleleft)\ s$$

This is a proposition—indeed, a type—which I assert with the keyword 'fact', and which we might prove for every f, g and s, but it is not a rule used for evaluating $\blacktriangleleft$. Dependent types give us ready access to a logic for reasoning about programs. I use this programming language to define logical notions and its type system to state theorems. The typechecker also checks proofs!

Every monad gives rise to a functor composing two 'half-lifting' operations—those defined for Maybe above. By careful choice of operator precedence, we may write the full functorial lifting of some f by 'bracketing' it thus: $\blacktriangleright f \blacktriangleleft$. It is also possible to generalise $\blacktriangleright \_ \blacktriangleleft$ to a $\blacktriangleright \_ \blacktriangleleft^n$ operation which lifts an n-ary function to its n-ary Maybe counterpart. The entire family can be defined once for all by recursion on n, but it is not worth the trouble here, as the largest n used in this paper is 2.

$$\text{let} \quad \frac{f \; : \; R \to S \to T}{\blacktriangleright f \blacktriangleleft^2 \; : \; \mathsf{Maybe}\ R \to \mathsf{Maybe}\ S \to \mathsf{Maybe}\ T} \qquad \cdots$$

2.2 Inductive Families, Pattern Matching

Dependent types systems allow us to talk about collections of types in a systematic way by creating type *families*, typically with a signature resembling this:

$$\frac{t \; : \; T}{F\ t \; : \; \mathsf{Type}}$$

We say that F is a family *indexed* by T. We may refer to a particular instance of the family by applying F to a particular t, yielding $F\ t$. We may abstract over the whole family by abstracting over all t's, as in $\forall t : T. F\ t \to \cdots$. We may abstract over a *section* of the family by applying F to an expression containing abstracted variables, thus $\forall s\!:\!S.F\ (g\ s) \ \rightarrow\ \cdots$.

Type families support fine analysis of data structures. Many common datatypes can be redesigned as carefully stratified families, with their operations acquiring more precise types. For example, indexing matrices by their dimensions allows a multiplier whose type requires that the column-count on the left equals the row-count on the right. The termination of unification relies on an analysis of the number of variables over which terms are constructed: later, we shall make that analysis explicit by indexing the representation of terms.

Let us first look more closely at how to declare type families and define operations over them. Most dependent type systems allow the introduction of families by inductive definition, although there is considerable variation in the styles supported. Systems like LEGO, COQ and ALF (Luo, 1994; Pfenning & Paulin-Mohring, 1990; Magnusson, 1994) adopt minor variations on the scheme set out in (Dybjer, 1994). They view an inductive family as a collection of mutually defined datatypes, with each choice of index yielding a branch of the definition. We declare a formation rule for the new family, together with constructors which are free to target and to source their recursive arguments from *any* section of it. For example, we may declare a family of finite types, $\mathsf{Fin}\ n$—the index n gives the size of each instance:

$$\text{data}\quad \frac{n\ :\ \mathbb{N}}{\mathsf{Fin}\ n\ :\ \mathsf{Type}}\quad \text{where}\quad \frac{}{\mathsf{f0}_n\ :\ \mathsf{Fin}\ \mathsf{s}n}\quad \frac{x\ :\ \mathsf{Fin}\ n}{\mathsf{fs}_n\ x\ :\ \mathsf{Fin}\ \mathsf{s}n}$$

The fs constructor embeds $\mathsf{Fin}\ n$ as the 'old' elements of $\mathsf{Fin}\ \mathsf{s}n$, whilst $\mathsf{f0}$ makes a 'new' element: the indices n are details which can be inferred mechanically, so I shall usually omit the arguments which I have subscripted above. Observe that both constructors target a restricted section of the family—the types with at least one element. $\mathsf{Fin}\ 0$ is quite rightly uninhabited.

The ability to restrict the target sections of constructors makes pattern matching over inductive families both more complex and more powerful than the standard notion found in Hindley-Milner languages. If we are defining a function whose domain is a particular section of a family, then, in principle, we need only write patterns for those constructors which target that section. The machine determines which these are by attempting to unify the target indices of each constructor with the indices of the section being matched. Unification is undecidable in general, but we still get a very powerful language when we restrict to decidable first-order problems. Whenever a constructor's target section does not overlap the function's domain, we do not need to write a pattern for it. For example, given that no constructor targets $\mathsf{Fin}\ 0$, we can define the following function with a type signature but no equations!

$$\text{let}\quad \frac{}{\textbf{empty}\ :\ \mathsf{Fin}\ 0\ \rightarrow\ T}$$

The designers of dependently typed languages continue to debate which notions of inductive family it is sensible to support. This paper exploits one particular choice, but it is reasonable to consider others. Both Cayenne and Agda (Augustsson, 1998; Coquand & Coquand, 1999) are at present less liberal. Whilst they still allow a family's constructors to source recursive arguments from any section, they demand that every constructor targets the whole family. This removes the complexity from pattern matching: no matter which section of the family we match over, every constructor always applies. However, it substantially restricts the families we can readily define. The above definition of Fin, for example, is no longer available. Fortunately, there is an alternative in this case—we may compute a type of size n, using Maybe and the empty type, $\perp$:

$$\text{let} \quad \frac{n \;:\; \mathbb{N}}{\mathsf{Fin}'\, n \;:\; \mathsf{Type}} \qquad \begin{array}{l} \mathsf{Fin}'\ \ 0 \ \ \mapsto \perp \\ \mathsf{Fin}'\,(\mathsf{s}n) \mapsto \mathsf{Maybe}\,(\mathsf{Fin}'\ n) \end{array}$$

One can also imagine systems which allow an even more liberal notion of inductive family than that used in this paper. *Inductive-recursive* definitions (Dybjer & Setzer, 1999) allow datatype families and operations over them to be mutally defined. We might even allow inductive families to be defined mutually with the types which index them. Making structure explicit and precise is habit forming—failure to express oneself quickly brings hunger for more language!

2.3 Structural Recursion

Pattern matching gives us access to the components from which a function's inputs are constructed. Structural recursion gives us access to the function's outputs for the components so exposed. It is, perhaps, helpful to view the constructors of a datatype as building larger 'new' elements from smaller 'old' ones. Structural recursion can be rationalised correspondingly as computing 'new' outputs for larger inputs, given that we can already see the 'old' outputs for smaller inputs. The fact that recursive calls are initiated after the call which requires them is a consequence of the 'just in time' mechanisms which sensible implementations employ. It is this shift, interpreting recursion not as 'seeing prior results' but as 'doing subsequent computations', which can also give a meaning to general recursion.

In (Burstall, 1987), Rod Burstall describes a variation of ML which restores the 'seeing' interpretation. Each pattern variable x standing for a subterm of the input is automatically accompanied by its 'inductive hypothesis', a variable $\$x$ bound to the output for x. There are no recursive calls—return values are built from objects already known and named. The resulting programs may look a little odd, but they make their structural behaviour, and hence guaranteed termination, clear to behold.

We can present structural recursion with a more conventional notation if we restrict the arguments on which recursive calls can be made. In particular, we must be able to identify an argument position for which the recursive calls on the right use only strict subterms exposed by pattern matching on the left. For example, the first argument in the following definition of 'reversing append':

$$\text{let}\quad \frac{xs, ys \;:\; \mathsf{List}\; T}{\mathbf{revapp}\; xs\; ys \;:\; \mathsf{List}\; T} \qquad \begin{array}{l} \mathbf{revapp}\;\; [\,] \qquad\;\; ys \mapsto ys \\ \mathbf{revapp}\; (x :: xs)\; ys \mapsto \mathbf{revapp}\; xs\; (y :: xs) \end{array}$$

We may extend this notion to cover datatypes with higher-order constructors (e.g., Brouwer ordinals, infinitely branching trees, etc.) by considering a functional pattern variable to expose everything in its image. We may liberalise further, allowing *nested* structural recursion. Here we identify a sequence of argument positions, where recursive calls may preserve the first n positions as long as the argument in position $n+1$ decreases—Ackermann's function is a typical example:

$$\text{let}\quad \frac{m, n \;:\; \mathbb{N}}{\mathbf{ack}\; m\; n \;:\; \mathbb{N}} \qquad \begin{array}{l} \mathbf{ack}\;\; \mathsf{0} \qquad\; n \;\; \mapsto \mathsf{s}n \\ \mathbf{ack}\; (\mathsf{s}m) \;\; \mathsf{0} \;\;\; \mapsto \mathbf{ack}\; m\; (\mathsf{s0}) \\ \mathbf{ack}\; (\mathsf{s}m)\; (\mathsf{s}n) \mapsto \mathbf{ack}\; m\; (\mathbf{ack}\; (\mathsf{s}m)\; n) \end{array}$$

Structural recursion thus delivers more functions than primitive recursion (in its original first-order sense). Further, allowing recursive calls on any exposed subterm loosens primitive recursion's requirement to peel off exactly one constructor at each step. The direct (but inefficient) definition of the Fibonacci function is thus permitted:

$$\text{let}\quad \frac{n \;:\; \mathbb{N}}{\mathbf{fib}\; n \;:\; \mathbb{N}} \qquad \begin{array}{l} \mathbf{fib}\quad \mathsf{0} \qquad \mapsto \mathsf{0} \\ \mathbf{fib}\quad (\mathsf{s0}) \quad \mapsto \mathsf{s0} \\ \mathbf{fib}\; (\mathsf{s}(\mathsf{s}n)) \mapsto \mathbf{fib}\; n + \mathbf{fib}\; (\mathsf{s}n) \end{array}$$

A dependent type system such as LEGO's, which only provides higher-order primitive recursion, can nonetheless support all the programs definable by pattern matching and structural recursion in the above sense. The full translation procedure is given in my thesis (McBride, 1999), but the basic idea is to use primitive recursion to build an auxiliary data structure memoizing all 'prior' outputs for each input—the original function's recursive calls are replaced by look-up operations on this structure. 'Doing' becomes 'seeing', just as in Burstall's language!

Dependent types add more than just detail to the class of structurally recursive function definitions. An inductive family indexed by a datatype has two notions of structural recursion: the constructors of the index type give one, and the constructors of the family itself give another. The two may have quite different behaviours, and they may be combined by nesting. Inductive families thus allow us to avoid general recursion in situations where we can see how to turn hidden recursive structure into the explicit inductive structure of an index type. That is how we shall write first-order unification by structural recursion.

3 First-Order Terms, Renaming and Substitution

Let us begin our development of unification with a type of terms over n variables:

$$\text{data} \quad \frac{n \,:\, \mathbb{N}}{\mathsf{Term}\; n \,:\, \mathsf{Type}}$$

$$\text{where} \quad \frac{x \,:\, \mathsf{Fin}\; n}{\iota\, x \,:\, \mathsf{Term}\; n} \qquad \frac{}{\mathsf{leaf} \,:\, \mathsf{Term}\; n} \qquad \frac{s, t \,:\, \mathsf{Term}\; n}{s\; \mathsf{fork}\; t \,:\, \mathsf{Term}\; n}$$

Here $\mathsf{Fin}\; n$ represents the variable names, embedded into $\mathsf{Term}\; n$ with the ι constructor. We may compute on $\mathsf{Term}\; n$ by nested recursion on n (allowing us to grow the terms if we eliminate a variable), then on the terms themselves. Building the number of variables into the definition of terms explains the structure by which unification operates.

We can represent a *renaming* by a function between variable sets, and a *substitution* by a function from one variable set to terms over another. Substitution has a monadic behaviour and renaming is the associated functor. Just as we did with Maybe, we may define a pair of half-lifting operators, with $\rangle$ turning a renaming into a substitution, and $\langle$ applying a substitution to a term. The functorial map is again given by 'bracketing', $\rangle\_\langle$.

$$\text{let} \quad \frac{r \,:\, \mathsf{Fin}\; m \to \mathsf{Fin}\; n}{\rangle r \,:\, \mathsf{Fin}\; m \to \mathsf{Term}\; n} \qquad \rangle r \mapsto \iota \cdot r$$

$$\frac{f \,:\, \mathsf{Fin}\; m \to \mathsf{Term}\; n}{f \langle \,:\, \mathsf{Term}\; m \to \mathsf{Term}\; n} \qquad \begin{array}{lll} f\langle & (\iota\, x) & \mapsto & f\; x \\ f\langle & \mathsf{leaf} & \mapsto & \mathsf{leaf} \\ f\langle & (s\; \mathsf{fork}\; t) & \mapsto & (f\langle s)\; \mathsf{fork}\; (f\langle t) \end{array}$$

The same substitution, in terms of effect, can have different functional encodings. We shall need to work with a *pointwise* (or *extensional*) equality for substitutions:

$$\text{let} \quad \frac{f, g \,:\, \mathsf{Fin}\; m \to \mathsf{Term}\; n}{f \doteq g \,:\, \mathsf{Type}} \qquad f \doteq g \;\mapsto\; \forall x\!:\!\mathsf{Fin}\; m.\, f\; x = g\; x$$

All of the reasoning in this paper relies only on the pointwise behaviour of substitutions. I shall omit the details and freely rewrite with 'equations' of the form $f \doteq g$.

Inspecting the definition of $\langle$, we can see that ι plays the rôle of the identity substitution. We may define the usual composition and check useful monadic properties:

$$\text{let} \quad \frac{f \,:\, \mathsf{Fin}\; m \to \mathsf{Term}\; n \qquad g \,:\, \mathsf{Fin}\; l \to \mathsf{Term}\; m}{f \diamond g \,:\, \mathsf{Fin}\; l \to \mathsf{Term}\; n} \qquad f \diamond g \;\mapsto\; (f\langle) \cdot g$$

$$\text{fact} \quad \iota \langle t \;=\; t \qquad (f \diamond g)\langle t = f\langle (g \langle t) \qquad f \diamond (\rangle r) \doteq f \cdot r$$

4 The Occur-Check, through thick and thin

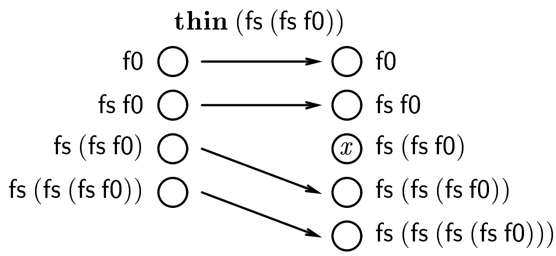
The notion of *occurrence* plays a key rôle in both the termination and correctness of first-order unification. If we are attempting to unify a variable x with a non-variable term t, then we must check whether x is used in t. If so, there is no unifier; if not, then the substitution 't for x' is not only a most general unifier, but it is also a means to eliminate x from the remaining parts of the problem: the corresponding reduction in the number of variables is sufficient to guarantee termination, even if the substitution enlarges the terms. The usual boolean occur-check does not expose this aspect of the algorithm: that is the crux of the problem this paper solves.

Our more stratified representation gives us the extra structure we need. If $x : \mathsf{Fin\,s}n$ and $t : \mathsf{Term\ s}n$, then any unifying substitution $f : \mathsf{Fin\ s}n \to \mathsf{Term}\ n$ would yield terms with structurally smaller indices. This section, presents an occur-check redesigned more positively and informatively as the search for such a substitution—the effective witness that x does not occur in t.

The boolean occur-check is built from the boolean equality test on variable names. A more informative occur-check needs a more informative equality test, returning a witness when its arguments are different. Hence we need a more concrete way to describe difference in variable sets. If $x : \mathsf{Fin\ s}n$, we may collect the other n elements of $\mathsf{Fin\,s}n$ as the image of an embedding, **thin** x from $\mathsf{Fin}\ n$, 'diluting' $\mathsf{Fin}\ n$ with x:

$$\text{let}\quad \frac{x : \mathsf{Fin\,s}n \qquad y : \mathsf{Fin}\ n}{\mathbf{thin}_n\ x\ y : \mathsf{Fin\,s}n} \qquad \begin{array}{llll} \mathbf{thin}_n & \mathsf{f0} & y & \mapsto \mathsf{fs}\ y \\ \mathbf{thin}_{\mathsf{s}n} & (\mathsf{fs}\ x) & \mathsf{f0} & \mapsto \mathsf{f0} \\ \mathbf{thin}_{\mathsf{s}n} & (\mathsf{fs}\ x) & (\mathsf{fs}\ y) & \mapsto \mathsf{fs}\ (\mathbf{thin}_n\ x\ y) \end{array}$$

The indices are highly significant in the pattern-matching behaviour of **thin**: in order to type the recursive call, we must ensure that there really is an s to strip from the index. However, there is no need for a $\mathbf{thin}_0\ (\mathsf{fs}\ x)\ldots$ case, as there is no such x. The following diagram illustrates **thin**'s behaviour:



The key properties of thinning are as follows:

$$\text{fact}\quad \frac{\mathbf{thin}\ x\ y\ =\ \mathbf{thin}\ x\ z}{y\ =\ z} \qquad \mathbf{thin}\ x\ y \neq x \qquad \frac{x\ \neq\ y}{\exists y'.\,\mathbf{thin}\ x\ y'\ =\ y}$$

In effect, every y apart from x is the image under **thin** x of a unique y'. Hence the terms t *without* x are exactly those expressible as $\rangle\mathbf{thin}\ x\langle t'$. The search for a unifying, variable-removing substitution thus reduces to the search for such a t'. This reduces in turn to finding the appropriate y' for each variable y we encounter in t. The refinement of boolean equality we need is thus the partial inverse to $\mathbf{thin}\,x$, here named $\mathbf{thick}\ x$.

$$\text{let}\quad \frac{x, y \ :\ \mathsf{Fin}\,\mathsf{s}n}{\mathbf{thick}_n\ x\ y\ :\ \mathsf{Maybe}\,(\mathsf{Fin}\ n)} \qquad \begin{array}{llll} \mathbf{thick}_n & \mathsf{f0} & \mathsf{f0} & \mapsto \mathsf{no} \\ \mathbf{thick}_n & \mathsf{f0} & (\mathsf{fs}\ y) & \mapsto \mathsf{yes}\ y \\ \mathbf{thick}_{\mathsf{s}n} & (\mathsf{fs}\ x) & \mathsf{f0} & \mapsto \mathsf{f0} \\ \mathbf{thick}_{\mathsf{s}n} & (\mathsf{fs}\ x) & (\mathsf{fs}\ y) & \mapsto \rangle \mathsf{fs}_n \langle\, (\mathbf{thick}_n\ x\ y) \end{array}$$

An induction on x shows that **thick** really is the partial inverse of **thin**:

$$\text{fact}\quad \frac{\mathbf{thick}\ x\ y\ =\ r}{\bigvee \left\{ \begin{array}{r} y = x \wedge r = \mathsf{no} \\ \exists y'.\ y = \mathbf{thin}\ x\ y' \wedge r = \mathsf{yes}\ y' \end{array} \right.}$$

We can see $\mathbf{thick}\ x$ as a 'failure-prone renaming'. The occur-check just propagates $\mathbf{thick}\ x$ monadically through the structure of terms:

$$\text{let}\quad \frac{x\ :\ \mathsf{Fin}\,\mathsf{s}n \quad t\ :\ \mathsf{Term}\,\mathsf{s}n}{\mathbf{check}\ x\ t\ :\ \mathsf{Maybe}\,(\mathsf{Term}\ n)} \qquad \begin{array}{lll} \mathbf{check}\ x & (\iota\ y) & \mapsto \rangle \iota \langle\, (\mathbf{thick}\ x\ y) \\ \mathbf{check}\ x & \mathsf{leaf} & \mapsto \mathsf{yes}\ \mathsf{leaf} \\ \multicolumn{3}{l}{\mathbf{check}\ x\ (s\ \mathsf{fork}\ t) \mapsto} \\ \multicolumn{3}{l}{\quad (\mathbf{check}\ x\ s) \rangle \mathsf{fork} \langle^2\, (\mathbf{check}\ x\ t)} \end{array}$$

If $\mathbf{check}\ x\ t$ returns no, then x occurs in t; if we get $\mathsf{yes}\ t'$, then $\rangle \mathbf{thin}\ x \langle\ t' \ =\ t$. We may now define the corresponding unifer, $t'\ \mathbf{for}\ x$.

$$\text{let}\quad \frac{x\ :\ \mathsf{Fin}\,\mathsf{s}n \quad t'\ :\ \mathsf{Term}\ n}{t'\ \mathbf{for}\ x\ :\ \mathsf{Fin}\,\mathsf{s}m\ \to\ \mathsf{Term}\ n} \qquad (t'\ \mathbf{for}\ x)\ y\ \mapsto\ \begin{array}{lll} \text{case} & \multicolumn{2}{l}{\mathbf{thick}\ x\ y} \\ \text{of} & \mathsf{yes}\ y' & \Rightarrow\ \iota\ y' \\ & \mathsf{no} & \Rightarrow\ t' \end{array}$$

As t' **for** x 'thickens' everything apart from x itself, we may show that

$$\text{fact}\quad t'\ \mathbf{for}\ x \cdot \mathbf{thin}\ x \doteq \iota$$

Hence, t' **for** x really does unify x with t:

$$\text{fact}\quad \frac{\mathbf{check}\ x\ t\ =\ \mathsf{yes}\ t'}{\begin{array}{l} t'\ \mathbf{for}\ x \langle\, t = \ t'\ \mathbf{for}\ x \langle\, (\rangle \mathbf{thin}\ x \langle\, t')\ =\ t'\ \mathbf{for}\ x \cdot \mathbf{thin}\ x \langle\, t'\ =\ t' \\ \qquad\qquad = \ (t'\ \mathbf{for}\ x)\ x \end{array}}$$

5 A Refinement of Substitution

In order to exploit recursion on n when unifying over $\mathsf{Term}\ n$, we need to know that each step of the substitution we accumulate really does get rid of a variable. It is not obvious how a functional substitution breaks down into individual steps, but we can define an inductive representation of substitutions built by solving for a sequence of variables in the manner described above.

$$\text{data}\quad \frac{m, n \,:\, \mathbb{N}}{\mathsf{AList}\; m\; n \,:\, \mathsf{Type}}$$

$$\text{where}\quad \frac{}{\mathsf{anil} \,:\, \mathsf{AList}\; n\; n} \qquad \frac{\sigma \,:\, \mathsf{AList}\; m\; n \quad t' \,:\, \mathsf{Term}\; m \quad x \,:\, \mathsf{Fin}\; \mathsf{s}m}{\sigma\; \mathsf{asnoc}\; t'/x \,:\, \mathsf{AList}\; \mathsf{s}m\; n}$$

Looking just at the indices, $\mathsf{AList}\; m\; n$ behaves very like $m \geq n$. Looking at the data, we see that $\mathsf{AList}\; m\; n$ stores association lists, linking each x with a t' over the set with x removed, until the number of variables has been whittled down from m to n. The family $\mathsf{AList}\; m\; n$ captures exactly the subset of $\mathsf{Fin}\; m \rightarrow \mathsf{Term}\; n$ we shall actually use, and in a form which admits structural decomposition, not just application. We can easily recover the functional form:

$$\text{let}\quad \frac{\sigma \,:\, \mathsf{AList}\; m\; n}{\mathbf{sub}\, \sigma \,:\, \mathsf{Fin}\; m \rightarrow \mathsf{Term}\; n} \qquad \begin{array}{lll} \mathbf{sub} & \mathsf{anil} & \mapsto \iota \\ \mathbf{sub} & (\sigma\; \mathsf{asnoc}\; t'/x) & \mapsto \mathbf{sub}\, \sigma \diamond t'\; \mathbf{for}\; x \end{array}$$

Observe that anil gives a concrete name to the identity substitution (for each n). We can also define composition:

$$\text{let}\quad \frac{\rho \,:\, \mathsf{AList}\; m\; n \quad \sigma \,:\, \mathsf{AList}\; l\; m}{\rho \mathbin{+\!\!+} \sigma \,:\, \mathsf{AList}\; l\; n}$$

$$\begin{array}{ll} \rho \mathbin{+\!\!+} \mathsf{anil} & \mapsto \rho \\ \rho \mathbin{+\!\!+} (\sigma\; \mathsf{asnoc}\; t/x) & \mapsto (\rho \mathbin{+\!\!+} \sigma)\; \mathsf{asnoc}\; t/x \end{array}$$

It is easy to check that **sub** respects $\mathbin{+\!\!+}$ in the appropriate way:

$$\text{fact}\quad \mathbf{sub}\,(\rho \mathbin{+\!\!+} \sigma) \doteq \mathbf{sub}\, \rho \diamond \mathbf{sub}\, \sigma$$

That is, the $\mathsf{AList}\; m\; n$ form a category whose objects are just those types in the image of Term and whose collections of arrows are a subset of the usual function spaces over those objects—just those functions in the image of $(\!|\,|\!) \cdot \mathbf{sub}$.

The most we can say in advance about the unifying substitution we hope to find for two terms in $\mathsf{Term}\; m$ is that it inhabits $\mathsf{AList}\; m\; n$ for *some* n. We can express this existential notion via *dependent pairing*, which generalizes the cartesian product, $\times$, in the same way that $\forall$ generalizes $\rightarrow$.

$$\text{data}\quad \frac{T \,:\, S \rightarrow \mathsf{Type}}{\exists T \,:\, \mathsf{Type}} \quad \text{where}\quad \frac{s \,:\, S \quad t \,:\, T\, s}{\langle s, t\rangle \,:\, \exists T}$$

If we wish to express existential propositions, we may regard the $\exists x : S.\; T$ as syntactic sugar for $\exists(\lambda x\!:\!S.\; T)$. The non-sugared syntax is also useful: $\exists(\mathsf{AList}\; m)$ is exactly the type of 'substitutions for *some* target'.

Let us overload $\_\mathsf{asnoc}\,\_/\_$ with its 'uncurried' counterpart, extending substitutions paired with their targets:

$$\text{let} \quad \frac{a \;:\; \exists(\mathsf{AList}\; m) \qquad t' \;:\; \mathsf{Term}\, m \qquad x \;:\; \mathsf{Fin}\, \mathsf{s}m}{a\; \mathsf{asnoc}\; t'/x \;:\; \exists(\mathsf{AList}\, \mathsf{s}m)}$$
$$\langle n, \sigma\rangle\; \mathsf{asnoc}\; t'/x \;\mapsto\; \langle n, \sigma\; \mathsf{asnoc}\; t'/x\rangle$$

The effect of $\exists$ here is to hide AList's 'target' index. Would it not be simpler, one might reasonably ask, to define AList with just a 'source' index in the first place? Given such a definition, we could still compute the target, should we need it:

$$\text{data} \quad \frac{m \;:\; \mathbb{N}}{\mathsf{AList}'\; m \;:\; \mathsf{Type}}$$

$$\text{where} \quad \frac{}{\mathsf{anil}' \;:\; \mathsf{AList}'\; m} \qquad \frac{\sigma \;:\; \mathsf{AList}'\; m \qquad t' \;:\; \mathsf{Term}\, m \qquad x \;:\; \mathsf{Fin}\, \mathsf{s}m}{\sigma\; \mathsf{acons}'\; t'/x \;:\; \mathsf{AList}'\; \mathsf{s}m}$$

$$\text{let} \quad \frac{\sigma \;:\; \mathsf{AList}'\; m}{\mathbf{targ}\, \sigma \;:\; \mathbb{N}} \qquad \begin{array}{lll} \mathbf{targ} & \mathsf{anil}'_m & \mapsto m \\ \mathbf{targ} & (\sigma\; \mathsf{acons}'\; t'/x) & \mapsto \mathbf{targ}\; \sigma \end{array}$$

However, this would make types which do not involve targets marginally less complex at the cost of complicating those which do:

$$\frac{\sigma \;:\; \mathsf{AList}'\, \sigma}{\mathbf{sub}'\, \sigma \;:\; \mathsf{Fin}\, m \;\to\; \mathsf{Term}\; (\mathbf{targ}\, \sigma)} \qquad \frac{\rho \;:\; \mathsf{AList}'\; (\mathbf{targ}\, \sigma) \qquad \sigma \;:\; \mathsf{AList}'\; m}{\rho \mathbin{+\!\!+}' \sigma \;:\; \mathsf{AList}' m}$$

Moreover, commuting interpretation with composition now changes the type of the resultant substitution:

$$\begin{array}{r} \mathbf{sub}'\, (\rho \mathbin{+\!\!+}' \sigma) : \; \mathsf{Fin}\; m \;\to\; \mathsf{Term}\; (\mathbf{targ}\; (\rho \mathbin{+\!\!+}' \sigma)) \\ \mathbf{sub}'\, \rho \diamond \mathbf{sub}'\, \sigma : \; \mathsf{Fin}\; m \;\to\; \mathsf{Term}\; (\mathbf{targ}\, \rho) \end{array}$$

It requires a proof by induction to show that the targets will be equal for all values of ρ and σ. Not all programs exploit the same structural aspects of a datatype. It costs us little to discard explicit structure with $\exists$; it costs rather more to recover structure which we have failed to express.

6 First-Order Unification

We are now ready to write the first-order unification algorithm. It takes two terms s and t and attempts to find a most general unifying substitution f. That is, f must satisfy:

1. $f \lessdot s \;=\; f \lessdot t$
2. If $g \lessdot s \;=\; g \lessdot t$ then, for some h, $g \doteq h \diamond f$

In particular, if $s, t : \mathsf{Term}\; m$, then f should be $\mathbf{sub}\, \sigma$ where $\sigma \;:\; \mathsf{AList}\; m\; n$ for some n. Hence we may give unification the type signature

$$\text{let} \quad \frac{s, t \;:\; \mathsf{Term}\; m}{\mathbf{mgu}_m\; s\; t \;:\; \mathsf{Maybe}\; (\exists(\mathsf{AList}\; m))}$$

One might wonder whether this type for **mgu** is a little tight, in that it apparently forces s and t to 'have the same number of variables', but that is not what the type means. The point is that s and t are expressed over a common set of unknowns, not all (or indeed any) of which they are obliged to mention. It only makes sense to unify expressions from a common language.

Let us employ the usual technique of introducing an accumulator:

$$\text{let} \quad \frac{s, t \;:\; \mathsf{Term}\; m \qquad acc \;:\; \exists(\mathsf{AList}\; m)}{\mathbf{amgu}_m\; s\; t\; acc \;:\; \mathsf{Maybe}\; (\exists(\mathsf{AList}\; m))}$$

$$\mathbf{mgu}_m\; s\; t \;\mapsto\; \mathbf{amgu}_m\; s\; t\; \langle m, \mathsf{anil}\rangle$$

The idea behind the accumulator technique is to proceed *optimistically*, guessing that the unifier is the most general solution to the subproblems examined thus far. We thus initialize the accumulator with the identity substitution and extend it only when disillusioned. This optimism is justified: once we have a most general unifier for part of the problem, we can show that extending it just enough to solve the rest yields a most general unifier for the whole thing. An abstract justification of *optimistic optimization* can be found in my thesis. However, the correctness proof, available online, illustrates the technique.

As suggested above, **amgu** works by nested structural recursion, first on m, then on s. There is no reason why **amgu** should perform its *case analyses* in that order. Traditional primitive recursors and fold operators make a tight connection between recursion and case analysis, in that they force recursion on some x to be conducted via case analysis on x exactly once and first of all. In fact, **amgu** does case analysis on s and t, and then only if a variable is encountered will it examine acc. If acc is $\langle n, \sigma\; \mathsf{asnoc}\; r/z\rangle$, then we know we can get rid of z. That is, although our outermost recursion is on m, we never examine m directly—the constructor types of AList tell us that nontrivial substitutions have nonempty domains.

I have written **amgu** with *prioritized* patterns, saving space by combining clauses of the case analysis which operate uniformly. The algorithm is fairly straightforward, dividing unification problems into three classes:

- *Rigid-rigid* problems: each term begins with leaf or fork. Like constructors yield subproblems for corresponding subterms; otherwise, unification fails.
- *Flexible* problems: at least one term is a variable and the accumulator holds anil. The appropriate helper function solves these immediately.
- *Postponed* problems: at least one term is a variable, but the accumulated σ might instantiate that variable. We peel off one step of σ and expand the terms—removing one variable justifies a recursive call. If successful, we glue the step back on.

$$\text{let}\quad \frac{s, t \ : \ \mathsf{Term}\ m \qquad acc \ : \ \exists(\mathsf{AList}\ m)}{\mathbf{amgu}_m\ s\ t\ acc \ : \ \mathsf{Maybe}\ (\exists(\mathsf{AList}\ m))}$$

$$\begin{array}{llllll}
\mathbf{amgu}_m & \mathsf{leaf} & \mathsf{leaf} & acc & \mapsto \mathsf{yes}\ acc \\
\mathbf{amgu}_m & \mathsf{leaf} & (t_1\ \mathsf{fork}\ t_2) & acc & \mapsto \mathsf{no} \\
\mathbf{amgu}_m & (s_1\ \mathsf{fork}\ s_2) & \mathsf{leaf} & acc & \mapsto \mathsf{no} \\
\mathbf{amgu}_m & (s_1\ \mathsf{fork}\ s_2) & (t_1\ \mathsf{fork}\ t_2) & acc & \mapsto \\
 & \multicolumn{4}{l}{\mathbf{amgu}_m\ s_2\ t_2 \blacktriangleleft \mathbf{amgu}_m\ s_1\ t_1\ acc} \\
\mathbf{amgu}_m & (\iota\ x) & (\iota\ y) & \langle m, \mathsf{anil}\rangle & \mapsto \mathsf{yes}\ (\mathbf{flexFlex}_m\ x\ y) \\
\mathbf{amgu}_m & (\iota\ x) & t & \langle m, \mathsf{anil}\rangle & \mapsto \mathbf{flexRigid}_m\ x\ t \\
\mathbf{amgu}_m & t & (\iota\ x) & \langle m, \mathsf{anil}\rangle & \mapsto \mathbf{flexRigid}_m\ x\ t \\
\mathbf{amgu}_{\mathsf{s}m} & s & t & \langle n, \sigma\ \mathsf{asnoc}\ r/z\rangle & \mapsto \\
 & \multicolumn{4}{l}{\blacktriangleright \lambda\sigma.\ \sigma\ \mathsf{asnoc}\ r/z \blacktriangleleft \mathbf{amgu}_m\ (r\ \mathbf{for}\ z \triangleleft s)\ (r\ \mathbf{for}\ z \triangleleft t)\ \langle n, \sigma\rangle}
\end{array}$$

$$\frac{x, y \ : \ \mathsf{Fin}\ m}{\mathbf{flexFlex}_m\ x\ y \ : \ \exists(\mathsf{AList}\ m)}$$

$$\begin{array}{llll}
\mathbf{flexFlex}_{\mathsf{s}m}\ x\ y \ \mapsto & \text{case} & \multicolumn{2}{l}{\mathbf{thick}\ x\ y} \\
 & \text{of} & \mathsf{yes}\ y' & \Rightarrow \ \langle m, \mathsf{anil}\ \mathsf{asnoc}\ \iota\ y'/x\rangle \\
 & & \mathsf{no} & \Rightarrow \ \langle \mathsf{s}m, \mathsf{anil}\rangle
\end{array}$$

$$\frac{x \ : \ \mathsf{Fin}\ m \qquad t \ : \ \mathsf{Term}\ m}{\mathbf{flexRigid}_m\ x\ t \ : \ \mathsf{Maybe}\ (\exists(\mathsf{AList}\ m))}$$

$$\begin{array}{llll}
\mathbf{flexRigid}_{\mathsf{s}m}\ x\ t \ \mapsto & \text{case} & \multicolumn{2}{l}{\mathbf{check}\ x\ t} \\
 & \text{of} & \mathsf{yes}\ t' & \Rightarrow \ \mathsf{yes}\ \langle m, \mathsf{anil}\ \mathsf{asnoc}\ t'/x\rangle \\
 & & \mathsf{no} & \Rightarrow \ \mathsf{no}
\end{array}$$

Prioritizing the patterns has saved us six lines in all: the two lines which call **flexRigid** apply whether the non-variable argument is a leaf or a fork, whilst the last line applies in each of the five *postponed* cases.

flexFlex and **flexRigid** each take a variable x (perforce from a nonempty set) and attempt to express their other argument as an image of $\triangleright\mathbf{thin}\ x\triangleleft$. If successful, we have found a unifier. Otherwise, in the **flexFlex** case, the two variables were already the same, so the identity substitution suffices, and in the **flexRigid** case, the occur-check has failed, so there is no unifier.

The online supplement to this paper contains a proof that the answer returned by this program, either a substitution or an indication of failure, is correct. It follows the same lines as the traditional proofs of partial correctness found in many papers about first-order unification, but with two modest novelties:

- The step cases of the algorithm are justified by a proof technique addressing *optimistic* optimization.
- The notion of *occurrence* has a *data* representation, inspired by Huet's notion of 'zipper' (Huet, 1997). This allows a more direct characterization of **check**,

showing exactly its relationship to unification, and comparing favourably with the collection of variable-counting lemmas previously required.

The major novelty is that there is no need for a termination proof: by expressing the structure which unification exploits, **mgu** becomes a recognizable program in a language of total functions.

7 Discussion

This development of unification is another in a long line. From Zohar Manna and Richard Waldinger's pioneering hand-synthesis (Manna & Waldinger, 1981), through Larry Paulson's machine verification in LCF (Paulson, 1985) to the more recent work in diverse proof systems (Coen, 1992; Rouyer, 1992; Jaume, 1997; Bove, 1999), all have faced the same inherent problem of explaining programs which simply could not and did not make all of the sense which their makers had in mind. Understanding the size of the variable signature over which terms are constructed is crucial to the termination of first-order unification. All the above treatments have been forced to fill in this missing structure after the fact, with complex lemmas relating substitution, the occur-check and the function which counts the number of distinct variables used in a term.

Ana Bove's treatment (Bove, 1999), like mine, yields a unification function which is an executable term in a strongly normalising type theory. She sets out to show that a standard Haskell function using general recursion can be imported systematically into type theory, acquiring an extra 'termination' argument (a proof of termination for the actual input) and a proof obligation (that there is such a proof for each input). The extra argument belongs to an inductive family whose constructors and indices reflect exactly the case analyses and recursive calls in the algorithm: the function is thus a structural recursion on termination proofs. The original Haskell code survives almost intact, but the real work now lies in the proof obligation, and it is here that the usual variable-counting lemmas reappear. The 'program' is still missing its explanation, but Bove's inductive presentation of termination proofs adds it very cleanly, with the bonus of providing exactly the right induction principle with which to prove partial correctness!

A dependent type system thus serves well as an 'explanation language' for a standard, simply-typed, general recursive program. However, this paper gives strong evidence that dependent types can be used to improve the language of programs themselves, reducing the amount of subsequent explanation required by making more structure explicit in advance. This is not just the shifting of a comparable burden from proof to program: it is genuinely easier to exploit structure which is present in data than to recover structure which is absent.

What I hope this paper shows is that the apparent need for general recursion is evidence not that structural recursion is a flawed ideal, but rather that we need a better language with which to express structure. With dependent types, we can express our *understanding*, not just our *procedure*. That is the very purpose of declarative programming—to make it more likely that we mean what we say by improving our ability to say what we mean.

Acknowledgements

This work was supported in part by grants from the UK EPSRC (GR/N 24988/01) and from the Leverhulme Trust. The original development was supervised by Rod Burstall, Healfdene Goguen and James McKinna. Many thanks also to Ana Bove, Paul Callaghan and the anonymous referees for helpful discussions, advice and encouragement.

References

Augustsson, Lennart. 1998 (Sept.). Cayenne—a language with dependent types. *ACM International Conference on Functional Programming*. ACM.

Bove, Ana. (1999). *Programming in Martin-öf Type Theory. Unification: A non-trivial Example*. Licentiate Thesis, Chalmers University of Technology, Sweden.

Burstall, Rod. (1969). Proving properties of programs by structural induction. *Computer journal*, **12**(1), 41–48.

Burstall, Rod. (1987). Inductively Defined Functions in Functional Programming Languages. *Journal of computer and system sciences*, **34**, 409–421.

Coen, Martin. (1992). *Interactive Program Derivation*. Ph.D. thesis, University of Cambridge.

Coquand, Catarina, & Coquand, Thierry. (1999). Structured Type Theory. *Workshop on Logical Frameworks and Metalanguages*.

Dybjer, Peter. (1994). Inductive families. *Formal Aspects of Computing*, **6**, 440–465.

Dybjer, Peter, & Setzer, Anton. (1999). A finite axiomatization of inductive-recursive definitions. *Pages 129–146 of: Proceedings of Typed Lambda Calculi and Applications*. LNCS. Springer-Verlag.

Huet, Gérard. (1997). The Zipper. *Journal of Functional Programming*, **7**(5), 549–554.

Jaume, M. (1997). Unification : a Case Study in Transposition of Formal Properties. *Pages 79–93 of:* Gunter, E.L., & Felty, A. (eds), *Supplementary Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics: Poster session TPHOLs'97*.

Luo, Zhaohui. (1994). *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press.

Luo, Zhaohui, & Pollack, Robert. 1992 (May). *LEGO Proof Development System: User's Manual*. Tech. rept. ECS-LFCS-92-211. Laboratory for Foundations of Computer Science, University of Edinburgh.

Magnusson, Lena. (1994). *The implementation of ALF—A Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitutiton*. Ph.D. thesis, Chalmers University of Technology, Göteborg.

Manna, Zohar, & Waldinger, Richard. (1981). Deductive Synthesis of the Unification Algorithm. *Science of Computer Programming*, **1**, 5–48. North-Holland.

McBride, Conor. (1999). *Dependently Typed Functional Programs and their Proofs.* Ph.D. thesis, University of Edinburgh.

Paulson, Lawrence. (1985). Verifying the Unification Algorithm in LCF. *Science of Computer Programming*, **5**, 143–169. North-Holland.

Pfenning, Frank, & Paulin-Mohring, Christine. (1990). Inductively Defined Types in the Calculus of Constructions. *Pages 209–228 of:* Main, M., Melton, A., Mislove, M., & Schmidt, D. (eds), *Mathematical Foundations of Programming Semantics 1989.* LNCS, vol. 442. Springer-Verlag.

Robinson, Alan. (1965). A Machine-oriented Logic Based on the Resolution Principle. *Journal of the ACM*, **12**, 23–41.

Rouyer, Joseph. 1992 (Nov.). *Développement de l'algorithme d'unification dans le Calcul des Constructions avecs types inductifs.* Tech. rept. 1795. INRIA-Lorraine.

Turner, David. (1995). Elementary Strong Functional Programming. *Functional Programming Languages in Education, First International Symposium.* LNCS, vol. 1022. Springer-Verlag.