Dependently Typed Functional Programs and their Proofs

Conor McBride

Doctor of Philosophy
University of Edinburgh
1999

Abstract

Research in dependent type theories [M-L71a] has, in the past, concentrated on its use in the presentation of theorems and theorem-proving. This thesis is concerned mainly with the exploitation of the computational aspects of type theory for programming, in a context where the properties of programs may readily be specified and established. In particular, it develops technology for programming with dependent inductive families of datatypes and proving those programs correct. It demonstrates the considerable advantage to be gained by indexing data structures with pertinent characteristic information whose soundness is ensured by typechecking, rather than human effort.

Type theory traditionally presents safe and terminating computation on inductive datatypes by means of elimination rules which serve as induction principles and, via their associated reduction behaviour, recursion operators [Dyb91]. In the programming language arena, these appear somewhat cumbersome and give rise to unappealing code, complicated by the inevitable interaction between case analysis on dependent types and equational reasoning on their indices which must appear explicitly in the terms. Thierry Coquand's proposal [Coq92] to equip type theory directly with the kind of pattern matching notation to which functional programmers have become used over the past three decades [Bur69, McB70] offers a remedy to many of these difficulties. However, the status of pattern matching relative to the traditional elimination rules has until now been in doubt. Pattern matching implies the uniqueness of identity proofs, which Martin Hofmann showed underivable from the conventional definition of equality [Hof95]. This thesis shows that the adoption of this uniqueness as axiomatic is sufficient to make pattern matching admissible.

A datatype's elimination rule allows abstraction only over the whole inductively defined family. In order to support pattern matching, the application of such rules to specific instances of dependent families has been systematised. The underlying analysis extends beyond datatypes to other rules of a similar second order character, suggesting they may have other roles to play in the specification, verification and, perhaps, derivation of programs. The technique developed shifts the specificity from the instantiation of the type's indices into equational constraints on indices freely chosen, allowing the elimination rule to be applied.

Elimination by this means leaves equational hypotheses in the resulting subgoals, which must be solved if further progress is to be made. The first-order unification algorithm for constructor forms in simple types presented in [McB96] has been extended to cover dependent datatypes as well, yielding completely automated solution to a class of problems which can be syntactically defined.

The justification and operation of these techniques requires the machine to construct and exploit a standardised collection of auxiliary lemmas for each datatype. This is greatly facilitated by two technical developments of interest in their own right:

- a more convenient definition of equality, with a relaxed formulation rule allowing elements of different types to be compared, but nonetheless equivalent to the usual equality plus the axiom of uniqueness;
- a type theory, OLEG, which incorporates incomplete objects, accounting for their 'holes' entirely within the typing judgments and, novelly, not requiring any notion of explicit substitution to manage their scopes.

A substantial prototype has been implemented, extending the proof assistant LEGO [LP92]. A number of programs are developed by way of example. Chiefly, the increased expressivity of dependent datatypes is shown to capture a standard first-order unification algorithm within the class of structurally recursive programs, removing any need for a termination argument. Furthermore, the use of elimination rules in specifying the components of the program simplifies significantly its correctness proof.

Acknowledgements

Writing this thesis has been a long hard struggle, and I could not have done it without a great deal of friendship and support. I cannot thank my official supervisor Rod Burstall enough for his constant enthusiasm, even when mine was straining at times. Supervisions with Rod, whatever else they achieved, always managed to make me happy. It has also been a pleasure to be part of the LEGO group, and indeed, the LFCS as a whole. My slightly less official supervisors, successively James McKinna, Healf Goguen and Martin Hofmann, deserve my warmest gratitude. I hope I have done them justice. A word of thanks, too, must go to Randy Pollack for his implementation of LEGO—if it were not for his code, I could not have built mine.

My friends and family have been a constant source of love and encouragement. Thank you, all of you. Finally, I cannot express how much I appreciate the flatmates I have lived with over the last four years. Phil, Carsten, Melanie, even Firkin the cat, your forbearance is something for which I shall always be profoundly grateful. Your friendship is something I treasure.

Declaration

I declare that this thesis is my own work, not submitted for any previous degree.

Table of Contents

Chapter 1

Introduction

> *'The philosophers have merely interpreted the world in various ways. The point, however, is to change it.'* (Marx and Engels)

Computer programs are not expected to make sense. In fact, they are seldom expected to work, which is as much as to say that computer programmers are not expected to make sense either. This is understandable—programming is primarily a form of giving orders.

Nonetheless, there are grounds for optimism. This is because programmers do not really want genuinely stupid orders to be obeyed, and we understand that the more sense we are able to make, the shorter our orders need be. The benefit comes by taking the sense within the programmer's mind and manifesting it explicitly in the program.

From named variables and looping constructs through to functional abstraction and method encapsulation, the evolution of programming languages has greatly facilitated the programmer who actively seeks to make sense. In particular, type systems now allow so much sense to be made that they are even becoming compulsory in some industrial programming languages. Where the purpose of typing in C is to indicate the number of bits left an array subscript should be shifted, strongly typed languages like Java genuinely reduce the gullibility with which machine faces human.

It is with the objective of promoting sense in programs that I have pursued the research documented in this thesis. Its main purpose is to show the advantage a **dependent** type system lends to the cause of principled programming.

Briefly, the principal contributions are these:

- OLEG, a type theory with 'holes' (or 'metavariables') standing for the missing parts of constructions explained entirely within the judgments of the calculus—the state of a theorem prover may thus be represented as a valid judgment

- the identification of what must be added to conventional type theories (such as those underlying LEGO or COQ) to facilitate pattern matching for dependent types (as implemented in ALF)
- a systematic view of elimination rules, leading to the use of *derived* elimination rules to characterise and indeed specify programs in a compact and powerful way

1.1 overview

This thesis records my development of technology to support functional programming over dependent datatypes by pattern matching and structural recursion in an intensional type theory. This technology also suggests novel tools and techniques for reasoning about such programs. Let me give an overview, identifying the innovations.

I open with an account of a theorem proving in a type theory, OLEG,[1] which is based on Luo's ECC [Luo94], but includes an account of 'holes'[2] in terms. There is a lot of theorem proving in this thesis. Some of it is done by hand. Much of it is done by machines, manufacturing and exploiting standard equipment for working with datatypes and equational problems. I therefore feel obliged to give a precise treatment not only of theorems but also theorem proving.

The novelty is that holes are handled much as other variables and accounted for by binding entirely within the judgments of the system. This system is workable because the core calculus of terms is embedded in a 'development calculus', which is where the hole-bindings are to be found—a core term in the scope of a hole may nonetheless refer to that hole. The effect of the separation is to prevent troublesome interaction between computation and holes. Consequently, terms (called 'partial constructions') in the development calculus enjoy the property that one may safely be replaced by another of the same type—remarkably good behaviour for a dependent type system.

As a result, theorem proving in OLEG consists exactly of editing OLEG judgments in ways which are guaranteed to preserve their derivability. Although OLEG is more

[1]The name 'OLEG' is a tribute to Randy Pollack's proof assistant LEGO. The new treatment of partial proofs required only a minor rearrangement.

[2]also known as 'metavariables', 'existential variables', 'question marks' and many other names besides

restrictive than systems with explicit substitution, those restrictions will not hinder us in the slightest.

The inductive datatypes we shall be concerned with are much like those of LEGO, COQ[Coq97] or ALF [Mag94]. Their elements are introduced by constructor symbols whose recursive arguments satisfy a strict positivity condition. Recursive computation and inductive proof are provided in the old-fashioned 'elimination rule' style. This necessitated the innovation of principled tactical support for such rules, documented in chapter three. However, the technology is not restricted to elimination rules arising from datatypes.

The contribution from this thesis to the methodology of program verification lies in the use of derived elimination rules to capture the leverage exerted by a given piece of information on an arbitrary goal. The abstraction of the predicate in an induction principle or the return type in a datatype fold operator point the way. Given a piece of information, we have been indoctrinated to ask what we can deduce from it—we should rather ask how we can deduce *what we want* from it. The tactics of chapter three were developed to support datatype elimination rules, but they allow us to exploit a wide class of rules which similarly abstract the type of their conclusions. I give numerous examples capturing the behaviour of programs in this way, and I believe I demonstrate the efficacy of the policy.

Once we understand elimination rules, we may give proper attention to inductive datatypes. In particular, we may use chapter three's technology to derive from each 'conventional' eliminator a pair of alternative eliminators which usefully untangle the treatment of case analysis and recursion on structurally smaller terms. This gives effectively the same presentation as the `Case` and `Fix` constructs which are primitive notions in COQ. The equivalence was established by Eduardo Giménez [Gim94]—only minor adaptations are required to mechanise his construction. Chapter four is reassuringly unremarkable.

Case analysis on a restricted instance of an inductive family (henceforth a **subfamily**) inevitably involves equational reasoning. For example, we may define the family of lists *indexed by their length*—when analysing the instance constrained to contain only *nonempty* lists, we rule out the 'nil' constructor because the list it generates does not satisfy that constraint. More generally, for each constructor, we must represent at the object-level the constraint that its return type unifies with the subfamily under analysis. These constraints are similar to the unification problems which arise in 'unfolding' transformations for logic programs [TS83, GS91]. My MSc work involved a systematic solution for simply typed problems in constructor form, implemented in the form of a tactic [McB96].

Chapter five extends the treatment to dependent types. Of necessity, this requires us to compare sequences of terms where later elements may have propositionally equal but computationally distinct types, an area which has always proved troublesome for intensional type theory. I present a new, slightly more relaxed definition of equality which scales up to sequences without significant attendant clumsiness. It turns out to be equivalent to the more traditional inductive definition augmented by the axiom that identity proofs are unique. So equipped, we may easily prove for each datatype its 'no confusion' property—constructors are injective and disjoint—in the form of a single elimination rule. I also give a systematic proof that each datatype contains no cycles. It is these lemmas which justify the transitions of the unification tactic.

In [Coq92], Thierry Coquand characterises a class of 'pattern matching' programs over dependent types which ensure that patterns cover all possibilities (deterministically) and that recursion is structural. This is the class of programs made available (with unrestricted recursion) in the ALF system [Mag94]. Chapter six contains the principal metatheoretic result of this thesis, confirming that the same class of programs can be constructed from traditional datatype elimination rules, given uniqueness of identity proofs. The meta-level unification in Coquand's presentation is performed at the object level by the tactic developed in chapter five.

By way of illustration, if not celebration, the work of the thesis closes with two substantial examples of verified dependently typed programs. Both concern syntax:

- substitution for untyped λ-terms, shown to have the properties of a monad
- a *structurally recursive* first-order unification algorithm, shown to compute most general unifiers

It is well understood, at least in the type theory community, that we may only really make sense of terms *relative to a context* which explains their free variables. Both of these examples express that sense directly in their data structures, a gain which is reflected in the correctness proofs.

Neither example is new to the literature of program synthesis and verification. Substitution has been treated recently [BP99, AR99] via polymorphic recursion, and I include it simply to show that dependent types easily offer the same functionality, without recourse to counterfeiting index data at the type level.

The existing treatments of unification turn on the use of an externally imposed termination ordering. The novelty here is that by indexing terms with the number of variables which may occur in them, we gain access to computation over that index—this is enough to capture the program for structural recursion. Witness the benefit from a program which captures much more precisely the sense of the algorithm it implements.

Both developments adopt the methodology of characterising the behaviour of their subprograms by means of elimination rules. Establishing program correctness becomes sufficiently easy that in presenting the proofs, I cut corners only where to do otherwise would be monotonous in the extreme.

I would like to apologise for the length and linearity of this thesis. I hope it is not nearly as much trouble to read as it was to write.

1.2 this thesis in context

> *'I do press you still to give me an account of yourself, for you certainly did not spring from a tree or a rock . . . '* (Penelope. *Odyssey*, Homer)

I sprang from a little-known Belfast pattern-matcher in 1973. I have spent my whole life surrounded by pattern matching, I have implemented pattern matching almost every year since 1988, and now I am doing a PhD about pattern matching with Rod Burstall. Fortunately, my mother was not a computer scientist. Enough about me.

Martin-Löf's type theory [M-L71a] is a well established and convenient arena in which computational Christians are regularly fed to logical lions—until relatively recently, much more emphasis has been placed on type theory as a basis for constructive logic than for programming. Comparatively boring programs have been written; comparatively interesting theorems have been proven. This is a pity, as the expressiveness of type theory promises much benefit for both. But things have changed.

Induction on the natural numbers was presented explicitly in different guises by Pascal and Fermat in the seventeenth century, although it has been used implicitly for a lot longer. Frege and Dedekind independently gave inductive definitions an explanation in impredicative set theory. 'Structural induction' had been widely used in mathematical logic [CF58, MP67] by the time Burstall introduced the notion of inductive datatypes to programming, with elements built from constructor functions and taken apart by case analysis [Bur69].

Inductive datatypes have escaped from programming languages [McB70, BMS80][3] and arrived in type theory [M-L84, CPM90]. Since then, they have become more

[3] My father's LISP-with-pattern-matching was a programming language which escaped from an inductive datatype.

expressive, with the indexing power of dependent type theory giving a natural home to inductive families of types [Dyb91]. For example, as hinted above, the polymorphic datatype $\mathsf{list}\,A$, with constructors

$$\frac{}{\mathsf{nil}\,A : \mathsf{list}\,A} \qquad \frac{h : A \quad t : \mathsf{list}\,A}{\mathsf{cons}\,h\,t : \mathsf{list}\,A}$$

can be presented in a usefully indexed way as vectors—lists *of a given length*:

$$\frac{}{\mathsf{vnil}_{\mathrm{A}} : \mathsf{vect}_{\mathrm{A}}\,\mathsf{0}} \qquad \frac{h : \mathrm{A} \quad t : \mathsf{vect}_{\mathrm{A}}\,n}{\mathsf{vcons}\,h\,t : \mathsf{vect}_{\mathrm{A}}\,\mathsf{s}n}$$

Typing is strong enough to tell when a vector is empty, so potentially disastrous destructive operations like ‘head’ and ‘tail’ can be safely defused.

However, there are significant ways in which dependent datatypes are more troublesome—the question is ‘what datatypes shall we have and how shall we compute with them?’. The datatypes and families proposed by Thierry Coquand, Christine Paulin-Mohring and Peter Dybjer have been integrated with type theory in a number of variations.

Zhaohui Luo’s UTT [Luo94] is closest to the traditional presentation, equipping families based on safe ‘strictly positive’ schemata with elimination constants doubling as induction principles and recursion operators. This is a conservative treatment for which the appropriate forms of good behaviour were established by Healf Goguen [Gog94]. Unfortunately, recursion operators make somewhat unwieldy instruments for programming, as anyone who has ever added natural numbers in LEGO[Pol94] will tell you.

Thierry Coquand’s 1992 presentation of pattern matching for dependent types [Coq92], implemented in the ALF system by Lena Magnusson [Mag94], was shown to be non-conservative over conventional type theory by Hofmann and Streicher, since it implies uniqueness of identity proofs [HoS94]. Pattern matching for the full language of inductive families is contingent on unification, which is needed to check whether a given constructor can manufacture an element of a given family instance. Unification, once it escapes from simple first-order syntaxes, becomes frightening to anyone with a well developed instinct for survival, although some do survive.

Subsequent systems in the ALF family, such as Agda [Hal99], have been much more cautious about what datatypes they will allow, in order to be much more generous with facilities for working with them. In particular, the question of unification is avoided by forbidding datatype constructors to restrict their return types to a portion of the family (eg, empty vectors, nonempty vectors). Families are declared in a similar manner to datatypes in functional programming languages:

$$\mathsf{data}\ \mathsf{family}\ x_1 \dots x_n \ = \ \mathsf{con}\ T_1 \dots T_k \ \mid \ \dots$$

The indices $x_1 \dots x_n$ are distinct variables, indicating that each constructor con, whatever its domain types $T_1 \dots T_k$, its range is over the entire family Pattern matching over an instantiated subfamily just instantiates the x's in the types of the constructors, rather than generating an arbitrarily complex unification problem.

This is a sensible restriction with a sound motivation. It is also a serious one, forbidding, for example, the formulation of the identity type—the reflexivity constructor restricts its return type to the subfamily where the indices are equal. As we decompose elements of these datatypes, their indices can only become more instantiated—Agda datatype indices only 'go up'.

Some of the power lost in this way is recaptured by computing types from data. For example, the type of vectors, although not a datatype in Agda can be *computed* from the length index:

$$\begin{array}{lll} \mathsf{vect}_{\mathrm{A}}\ \ \mathsf{0} & = & \mathbb{1} \\ \mathsf{vect}_{\mathrm{A}}\ \ \mathsf{s}n & = & A \times (\mathsf{vect}_{\mathrm{A}}\ n) \end{array}$$

This kind of computed type is good for data which are in some way measured by the indices—elements are finite, not because they contain only finitely many constructor symbols per se, but because, as we decompose them, their indices recursively 'go down' some well-founded ordering. There is no place, in this setting, for inductive families whose indices, like those of the stock exchange, can go down as well as up.

The practical limitations of this system require further exploration. Certainly, the removal of unification from the pattern matching process makes it considerably more straightforward to implement attractively and to grasp. It has been even been implemented outside the protective environment of the interactive proof assistant—in Lennart Augustsson's dependently typed programming language, Cayenne [Aug98]. Cayenne allows general recursion, hence its typechecker requires a boredom threshold to prevent embarrassing nontermination. Of course, the programs which make sense do typecheck, and some interesting examples are beginning to appear [AC99].

On the other hand, two recent examples—both implementations of first-order unification, as it happens—cannot be expressed as they stand in this restricted system. Ana

Bove's treatment [Bove99] shows that a standard Haskell implementation of the algorithm can be imported almost systematically into type theory. However the general recursion of the original is replaced by 'petrol-powered recursion'[4] over an inductively defined accessibility predicate [Nor88] which can be expressed in ALF, but not its successors.

My implementation, in chapter seven of this thesis, is dependently typed, and exploits the power of 'constraining constructors' to represent substitutions as association lists in a way which captures the idea that each assignment gets rid of a variable. Variables are from finite sets indexed by size, $\mathsf{fin}\ n$, and terms are trees over a number of variables, $\mathsf{tree}\ n$. Association lists, $\mathsf{alist}\ m\ n$, represent substitutions from m variables to terms over n:

$$\frac{}{\mathsf{anil}_n : \mathsf{alist}\ n\ n} \qquad \frac{x : \mathsf{fin}\ \mathsf{s}m \quad t : \mathsf{tree}\ m \quad g : \mathsf{alist}\ m\ n}{\mathsf{acons}\ x\ t\ g : \mathsf{alist}\ \mathsf{s}m\ n}$$

Having said that, I am quite sure that 'gravity-powered' unification can be implemented in Agda using the restricted type system. I only use association lists because my application of substitutions is delayed and incremental. If you are happy to apply substitutions straight away, a functional representation suffices. Nonetheless, the alist type stands as a useful data structure—a 'context extension'—which one might reasonably hope to represent as a datatype.

The COQ system [Coq97] has inductive families of types with strictly positive schemata [P-M92, P-M96]. However, they have moved away from the traditional 'one-step' elimination operator, following a suggestion from Thierry Coquand: they now divide elimination into a `Case` analysis operator and a constructor-guarded `Fix`-point operator. Eduardo Giménez's conservativity argument [Gim94] is bolstered by a strong normalisation proof for the case of lists [Gim96]—he has recently proved strong normalisation for the general case [Gim98]. Bruno Barras has formalised much of the metathory for this system [Bar99], including the decidability of typechecking.

This `Case`-`Fix` separation is a sensible one, and it makes practical the technology in this thesis—working in what is effectively Luo's UTT [Luo94], I start from the traditional 'one-step' rule, but I have mechanised the derivation of `Case` and `Fix` for each datatype. Everything which then follows applies as much to COQ as to LEGO.

There is, though, a noticeable gap between programming by `Case` with `Fix` in COQ and programming by pattern matching in ALF or Cayenne. This gap has been addressed by the work of Cristina Cornes [Cor97]. She identifies a decidable class of

[4] my phrase

second-order unification problems which captures many pattern matching programs viewed as collections of functional equations. Solving these problems mechanically, she has extended COQ with substantial facilities for translating such programs in terms of `Case` and `Fix`.

This takes the form of a macro `Cases` which allows pattern matching style decomposition of multiple terms from *unconstrained* inductive families (eg $\mathsf{vect}\ n$—vectors of arbitrary length) and combines with `Fix` to yield recursive function definition in the style of ML. Although the full gamut of dependent families can be defined, she has adopted an Agda-like solution to the problem of computing with them.

The task of implementing pattern matching for *constrained* instances of inductive families (or 'subfamilies', eg $\mathsf{vect}\ \mathsf{s}n$—nonempty vectors) she leaves to the future. Where she leaves is where I arrive. I have not attempted to duplicate her machinery for the translation of equational programs. Rather, I have concentrated on the problem of case analysis for subfamilies, the last gap between her work and dependent pattern matching in ALF.

As I have already mentioned, we have known for some time that dependent pattern matching is not conservative—it implies the uniqueness of identity proofs, which does not hold in Hofmann's groupoid model of type theory [HoS94, Hof95]:

$$\begin{array}{llllll} \mathsf{IdUnique} & : \forall A:\mathit{Type}. & \forall a:A. & \forall e:a{=}a. & & e = \mathsf{refl}\ a \\ \mathsf{IdUnique} & \quad A & \quad a & \quad (\mathsf{refl}\ a) & & = \mathsf{refl}\ (\mathsf{refl}\ a) \end{array}$$

This points to a very real connection between pattern matching and the power of equality in type theory. Case analysis on inductive subfamilies (also known as 'inversion') necessarily involves equational reasoning—for each constructor, we must check that its return type unifies with the subfamily we are analysing. These unification problems resemble those which arise in unfold/fold program transformation [BD77, TS83, GS91]. They are treated at the meta-level in ALF [Coq92, Mag94].

Cristina Cornes made some progress in this area with her tactics for inverting inductively defined relations over simply typed data in COQ[CT95]. My MSc project was to import this technology for LEGO. I made explicit the separation between, on the one hand, the splitting of the family into its constructors, with the subfamily constraints becoming object-level equations, and on the other hand, the simplification of those constraints. I implemented a complete first-order unification algorithm for object-level equations over constructor forms in simple types [McB96].

The uniqueness of identity proofs contributes directly to the extension of this first-order unification algorithm to dependent types, yielding explicit object-level solutions to the same class of unification problems which ALF handles implicitly.[5] The last gap between programming with datatypes in LEGO or COQ and pattern matching in ALF has now been bridged.

Building that bridge has involved many engineering problems and the development of some, I feel, fascinating technology. In particular, the tactic which I built for deploying the elimination rules of inductive datatypes has a potential far beyond that purpose. I have begun to explore the use of rules in that style for specifying and proving properties of programs: this thesis contains several examples.

1.3 implementation

I have implemented a prototype version of the technology described in this thesis as an extension to LEGO. It contributed to, rather than benefiting from the full analysis set out here. Nonetheless, let me emphasise at this stage that although the prototype could work better, it does work.

Enough technology has been implemented to support all the example programs and proofs in this thesis. They have all been built with OLEG's assistance and checked with LEGO—core OLEG is a subset of LEGO's type theory for which Randy Pollack's typechecker runs unchanged. Sometimes I have had to hand-crank techniques I have subsequently shown how to mechanise, but the developments described in the thesis are an honest account of real machine proofs.

[5] In fact, ALF rejects cyclic equations as unification problems which are 'too hard', while I disprove them, so four years' work has been good for something.

Chapter 2

OLEG, a type theory with holes

Although you have just started reading this chapter, I have nearly finished writing it. When I started, a long time ago, I intended it to be an unremarkable summary of a familiar type theory, present largely out of the need to present the notational conventions used in this thesis. However, despite my best intentions, this chapter does contain original work—it describes a type theory, OLEG, which gives an account of incomplete constructions quite different from those in existing use.

Let me say from the outset that I did not set out to invent such a thing. For some years I have been writing programs which construct LEGO proofs of standard datatype equipment—constructor injectivity and so forth—together with tactics to deploy them. I began, in my MSc work, with direct synthesis of proof terms in the abstract syntax—this was, frankly, rather painful. However, as time went on, the tools I was building myself looked more and more like a theorem-prover. Eventually, the penny dropped—synthetic programming and proof is only for clever people with nothing better to do; busy people and stupid machines need an analytic framework with a sound treatment of refinement. What had previously been 'voodoo', an ad hoc assortment of syntactic trickery, became OLEG, a type theory for machines as well as people.

OLEG was thus manifest in code long before it was rationalised in this chapter. I put it together with the help of many spare parts from Randy Pollack's LEGO code. LEGO's treatment of 'metavariables' is remarkable in what it allows—too remarkable, in fact. Scope is not quite managed properly, so that the reliability of LEGO still lies in the final typecheck of the completed term. I did not consider it my business to repair this problem—I was looking for a more convenient way to represent proofs-with-holes for mechanical manipulation. I hit upon the idea of *binding* holes in the context because it required very little alteration of the term syntax, and because it made operations like refinement's 'turn the unknown premises into subgoals' just a matter of turning $\forall$s into ?s. This treatment of holes resonates strongly with Dale Miller's explicit binding of existential variables in the 'mixed' quantifier prefix of unification problems [Mil91, Mil92].

At the time, explicit substitution was not even an issue. If I had wanted to maintain the scope of holes via such technology (as is found in the ALF family [Mag94] and Typelab [vHLS98]), I should have had to re-engineer the whole LEGO syntax, reduction mechanism and typechecker. As it turned out, my adaptations were minimal. There is a profound reason for this—where explicit substitution relies on ingenuity, OLEG relies on cowardice. Instead of repairing the troublesome interactions between holes and computation by propagating bits of stack through the term structure, OLEG simply forbids them.

However, let me keep you in suspense no longer. OLEG consists of a computational core—Luo's ECC [Luo94] with local definition [SP94] but without Σ-types—wrapped in a development calculus, in much the same way that Extended ML [KST94] wraps core Standard ML [MTH90]. My reason for the separation is precisely the aforementioned cowardice with respect to holes and computation.

Extended ML's treatment of holes profits from the fact that, in simple type systems, it is always safe to replace a term (eg, the placeholder '?') with another of the same type. Although there is no way that core terms in a dependent type theory could ever hope to have such a replacement property (for a counterexample see section 2.2), it does hold for the terms (or 'partial constructions') of OLEG's development layer. This single metatheorem does most of the work in OLEG's successful reconstruction of refinement proof as we know it.

2.1 the OLEG core

DEFINITION: **universes, identifiers, bindings, terms**

universes $\mathcal{U} ::= \mathcal{P}rop \mid \mathcal{T}ype_j$
where j is a natural number

identifiers $I ::= x \mid y \mid \ldots$
Let us allow ourselves countably many identifiers.

I define families of bindings and terms indexed by the finite set of variables $\mathcal{V} \subset I$ permitted to appear free in them. My motivation is to ensure that identifiers are only used where they are meaningful.[1]

For any set $\mathcal{V}$ of variables, and any x not in $\mathcal{V}$ the sets $\mathcal{B}^x_{\mathcal{V}}$ of **bindings of** x **extending** $\mathcal{V}$ and $\mathcal{T}_{\mathcal{V}}$ of **terms over** $\mathcal{V}$ are defined inductively[2] as follows:

$$\frac{S \in \mathcal{T}_{\mathcal{V}}}{\forall x : S \in \mathcal{B}^x_{\mathcal{V}}} \qquad \frac{S \in \mathcal{T}_{\mathcal{V}}}{\lambda x : S \in \mathcal{B}^x_{\mathcal{V}}} \qquad \frac{s, S \in \mathcal{T}_{\mathcal{V}}}{!x = s : S \in \mathcal{B}^x_{\mathcal{V}}}$$

$$\frac{y \in \mathcal{V}}{y \in \mathcal{T}_{\mathcal{V}}} \qquad \frac{U \in \mathcal{U}}{U \in \mathcal{T}_{\mathcal{V}}} \qquad \frac{f, s \in \mathcal{T}_{\mathcal{V}}}{f\, s \in \mathcal{T}_{\mathcal{V}}} \qquad \frac{B \in \mathcal{B}^x_{\mathcal{V}} \quad t \in \mathcal{T}_{\mathcal{V} \cup \{x\}}}{B.t \in \mathcal{T}_{\mathcal{V}}}$$

Binding is the means of attaching to an identifier properties such as 'type', 'value' and any other behavioural attributes in which we may be interested. A structural linguist, following Saussure [Sau16], might point out that identifiers, like words, have no intrinsic significance. *Variables*, on the other hand, are signs. Binding creates a sign, linking signifier and signified.

Syntactically, a **binding** is a binding operator, followed by an identifier, followed by a sequence of properties each introduced by a special piece of punctuation, eg ':' for 'type' or '=' for 'value'. The binding operator determines the computational role of the variable. I would encourage you to think of bindings as important syntactic entities in their own right, and the '.' as a combinator which attaches a binding to its scope which, by convention, extends rightwards as far as possible.

OLEG's core binding operators comprise the usual $\forall$ (often written Π) for universal quantification and λ for functional abstraction, together with ! (pronounced 'let') representing local definition. I describe those bindings where the bound variable occurs nowhere in its scope as **fatuous**.

As usual, application is indicated by juxtaposition and associates leftwards. I shall denote by $x \in T$ that variable x occurs free in term T. When $x \notin T$, I shall freely abbreviate $\forall x : S.\ T$ by $S \to T$. Further, otherwise identical consecutive bindings of distinct variables may be abbreviated with commas, $\lambda x : S.\ \lambda y : S.\ T$, for example, becoming $\lambda x, y : S.\ T$.

α-convertible terms are identified, with $\equiv$ representing the consequent notion of **syntactic identity**. Let $[S/x]T$ denote the result of substituting S for free occurrences of x in T. Formally, we might prefer to live in a de Bruijn-indexed world [deB72]; informally, let us grant ourselves the luxury of names and the associated luxury of ignoring the issue of variable capture.

[1] My ulterior motive is to prepare the ground for the application of dependently typed functional programming to syntax in chapter seven.

[2] Natural deduction is the best style I have found for presenting indexed inductive families.

Having introduced all this syntax, let us abuse it wherever it suits us. We are not machines, and we can suppress inferrable information which machines might demand. To be sure, the machines are catching up, with work on implicit syntax from [Pol90] and beyond. I do not propose to give any mechanistic account of the arguments I shall omit, the parentheses I shall drop and the ad hoc notations I shall introduce—the purpose is purely presentational.

DEFINITION: **contexts and judgments**

The set $\mathcal{Ctxt}$ of **contexts** is defined inductively:

$$\frac{}{\langle\rangle \in \mathcal{Ctxt}}$$

$$\frac{\Gamma \in \mathcal{Ctxt} \quad S \in \mathcal{T}_\Gamma \quad x \notin \Gamma}{\Gamma; x : S \in \mathcal{Ctxt}} \qquad \frac{\Gamma \in \mathcal{Ctxt} \quad s, S \in \mathcal{T}_\Gamma \quad x \notin \Gamma}{\Gamma; x = s : S \in \mathcal{Ctxt}}$$

Note that we may treat a context 'forgetfully' as a set of variables, hence $\mathcal{T}_\Gamma$ is the set of terms over Γ.

If $\Gamma \in \mathcal{Ctxt}$, then $\mathcal{J}_\Gamma$ is the set of Γ-**judgments**. If $J \in \mathcal{J}_\Gamma$, we may assert that J holds by writing

$$\Gamma \vdash J$$

$\mathcal{J}_\Gamma$ contains context validity and typing judgments:

$$\frac{}{\textbf{valid} \in \mathcal{J}_\Gamma} \qquad \frac{t, T \in \mathcal{T}_\Gamma}{t : T \in \mathcal{J}_\Gamma}$$

In many presentations, a context is an assignment of types to identifiers. Here, value assignments are also permitted. Let us also indulge in a slight abuse of notation and write whole bindings in the context, effectively annotating entries with binding operators and perhaps additional properties.[3] Seen as a data structure, for example in the implementation of LEGO, a context is a stack of bindings. We can recover the 'formal' contexts defined above simply by forgetting the extra annotations. We shall often need to check whether a given variable has a particular property, whether or not it may have others. Let us, for example, write $\Gamma; x\!:\!T; \Gamma'$ for a context where x has the property that its type is T, regardless of other annotations.

As we explore a term, each variable we encounter is given its meaning by the stack of bindings under which we have passed. A variable is not a name; it is a reference to a

[3] It is often useful to know what colour a variable is.

binding. Names arise as a social phenomenon—just as in the story of Rumpelstiltskin, naming things gives us power over them.

Let us now define computation with respect to a context. We should feel no apprehension at this. Quite the reverse, syntax only makes sense relative to the context which explains its signs. Goguen's typed operational semantics for type theory [Gog94] necessarily and naturally involves the context, significantly reducing the cost of metatheory. Although the contextual information he requires is active in typing and passive in computation, this is more an accident of ECC than an inevitable restriction. Compagnoni and Goguen's more recent typed operational semantics for higher-order subtyping [CG99] exploits the potential to the full.

MANTRA:

> Γ is with me, wherever I go.

I, too, feel strongly provoked to exploit the potential he reveals by increasing the activity of the context in computation. Real programming language implementations keep values in stacks.

We may still employ the usual technique of supplying a number of **contraction schemes** which indicate the actual computation steps, together with a notion of **compatible closure** which allows computation to occur anywhere within a term.

DEFINITION: **contraction schemes**

> OLEG's **contraction schemes** are shown in table 2.1.

Subterms susceptible to the β, δ or ! contraction schemes are, respectively, β**-**, δ**-** and **!-redexes**. Note that the property of being a δ-redex is implicitly context-dependent. A term is in **normal form** if it contains no redexes.

It is perhaps helpful to think of $\leadsto^{\beta}$ and $\leadsto^{\delta}$ as 'work', while $\leadsto^{!}$ is 'waste disposal'. As a fan of Fritz Lang's 1926 classic silent film, 'Metropolis', I like to imagine what computation sounds like: β-reduction sounds like paper-shuffling; δ-reduction sounds like filing cabinets and photocopiers; ! pops like sudden suction.

DEFINITION: **compatible closure**

> If $\leadsto^{\cdot}$ is a contraction scheme, its compatible closure, $\leadsto_{\cdot}$, is given by 2.2

$$\beta \qquad \frac{}{\Gamma \vdash (\lambda x\!:\!S.\, t)\, s \leadsto^{\beta} !x = s\!:\!S.\, t}$$

$$\delta \qquad \frac{}{\Gamma; x = s\!:\!S; \Gamma' \vdash x \leadsto^{\delta} s}$$

$$! \qquad \frac{}{\Gamma \vdash !x = s\!:\!S.t \leadsto^{!} t}\ x \notin t$$

Table 2.1: contraction schemes

$$\frac{\Gamma \vdash s \leadsto^{\cdot} s'}{\Gamma \vdash s \leadsto_{.} s'}$$

$$\frac{\Gamma \vdash s \leadsto_{.} s'}{\Gamma \vdash s\, t \leadsto_{.} s'\, t} \qquad \frac{\Gamma \vdash t \leadsto_{.} t'}{\Gamma \vdash s\, t \leadsto_{.} s\, t'}$$

$$\frac{\Gamma \vdash S \leadsto_{.} S'}{\Gamma \vdash \forall x\!:\!S.\, T \leadsto_{.} \forall x\!:\!S'.\, T} \qquad \frac{\Gamma \vdash S \leadsto_{.} S'}{\Gamma \vdash \lambda x\!:\!S.\, t \leadsto_{.} \lambda x\!:\!S'.\, t}$$

$$\frac{\Gamma \vdash s \leadsto_{.} s'}{\Gamma \vdash !x = s\!:\!S.t \leadsto_{.} !x = s'\!:\!S.\, t} \qquad \frac{\Gamma \vdash S \leadsto_{.} S'}{\Gamma \vdash !x = s\!:\!S.t \leadsto_{.} !x = s\!:\!S'.\, t}$$

$$\frac{\Gamma; B^x_\Gamma \vdash t \leadsto_{.} t'}{\Gamma \vdash B^x_\Gamma.t \leadsto_{.} B^x_\Gamma.t'}$$

Table 2.2: compatible closure

$$\frac{\Gamma \vdash S \cong T}{\Gamma \vdash S \preceq T}$$

$$\frac{\Gamma \vdash R \preceq S \quad \Gamma \vdash S \preceq T}{\Gamma \vdash R \preceq T}$$

$$\frac{}{\Gamma \vdash \mathit{Prop} \preceq \mathit{Type}_j} \qquad \frac{j < k}{\Gamma \vdash \mathit{Type}_j \preceq \mathit{Type}_k}$$

$$\frac{\Gamma \vdash S \cong S' \quad \Gamma; \forall x : S' \vdash T \preceq T'}{\Gamma \vdash \forall x\!:\!S.\, T \preceq \forall x\!:\!S'.\, T'}$$

Table 2.3: cumlativity

empty	$\dfrac{}{\langle\rangle \vdash \textbf{valid}}$
declare	$\dfrac{\Gamma \vdash S : \mathit{Type}_j}{\Gamma; Bx{:}S \vdash \textbf{valid}} \quad B \in \{\forall, \lambda\}$
define	$\dfrac{\Gamma \vdash s : S}{\Gamma; !x = s{:}S \vdash \textbf{valid}}$
prop	$\dfrac{\Gamma \vdash \textbf{valid}}{\Gamma \vdash \mathit{Prop} : \mathit{Type}_0}$
type	$\dfrac{\Gamma \vdash \textbf{valid}}{\Gamma \vdash \mathit{Type}_j : \mathit{Type}_{j+1}}$
var	$\dfrac{\Gamma; x{:}S; \Gamma' \vdash \textbf{valid}}{\Gamma; x{:}S; \Gamma' \vdash x : S}$
imp	$\dfrac{\Gamma; \forall x : S \vdash P : \mathit{Prop}}{\Gamma \vdash \forall x{:}S.\, P : \mathit{Prop}}$
all	$\dfrac{\Gamma \vdash S : \mathit{Type}_j \quad \Gamma; \forall x : S \vdash T : \mathit{Type}_j}{\Gamma \vdash \forall x{:}S.\, T : \mathit{Type}_j}$
abs	$\dfrac{\Gamma; \lambda x : S \vdash t : T}{\Gamma \vdash \lambda x{:}S.\, t : \forall x{:}S.\, T}$
app	$\dfrac{\Gamma \vdash f : \forall x{:}S.\, T \quad \Gamma \vdash s : S}{\Gamma \vdash f s \ : !x = s{:}S.\, T}$
let	$\dfrac{\Gamma; !x = s{:}S \vdash t : T}{\Gamma \vdash !x = s{:}S.\, t \ : !x = s{:}S.\, T}$
cuml	$\dfrac{\Gamma \vdash t : S}{\Gamma \vdash t : T} \quad \Gamma \vdash S \preceq T$

Table 2.4: OLEG core inference rules

METATHEOREM: **Church-Rosser**

$$\Gamma \vdash s \cong t$$

implies existence of a **common reduct** r such that

$$\Gamma \vdash s \rhd r \quad \Gamma \vdash t \rhd r$$

METATHEOREM: **strengthening**

$$\Gamma; \mathcal{B}^x_\Gamma; \Gamma' \vdash t : T \quad x \notin \Gamma', t, T$$

implies

$$\Gamma; \Gamma' \vdash t : T$$

METATHEOREM: **subject reduction**

$$\Gamma \vdash s : T \quad \Gamma \vdash s \rhd t$$

implies

$$\Gamma \vdash t : T$$

METATHEOREM: **strong normalisation**

$$\Gamma \vdash t : T$$

ensures that t is strongly normalising.

METATHEOREM: **cut**

$$\Gamma; !x = s : S; \Gamma' \vdash t : T$$

implies

$$\Gamma; [s/x]\Gamma' \vdash [s/x]t : [s/x]T$$

Table 2.5: metatheoretic properties

Henceforth, I shall elide the context in casual discussion. I write $\rightsquigarrow$ for the union of the labelled compatible closures, and $\triangleright$ for its finite transitive closure. A term is **strongly normalising** if admits only finite sequences of reductions. The smallest equivalence relation closed under $\triangleright$ is called **conversion** and denoted $\cong$.

Observe that !-binding allows us to avoid meta-level substitution in describing computation. Explanations of identifiers are activated by putting them into the context, not by propagating them through terms. The traditional β-contraction

$$(\lambda x\!:\!S.\ t)\ s \quad \rightsquigarrow \quad [s/x]t$$

becomes a 'noisier' reduction sequence

$$(\lambda x\!:\!S.\ t)\ s \quad \rightsquigarrow^{\beta} \quad !x = s\!:\!S.\ t \quad \rightsquigarrow_{\delta} \cdots \quad !x = s\!:\!S.\ [s/x]t \quad \rightsquigarrow^{!} \quad [s/x]t$$

Following Luo, I combine the notions of conversion and universe inclusion in a type cumulativity preorder with respect to $\cong$:

DEFINITION: **cumulativity**

> The **cumulativity relation**, $\preceq$, is defined inductively in table 2.3.

In [Luo94], Luo shows that $\preceq$ is antisymmetric and hence a partial order with respect to $\cong$. In fact, every well-typed term t (under Γ) has a principal type T in the sense that

$$\Gamma \vdash t : T' \quad \Longleftrightarrow \quad \Gamma \vdash T \preceq T'.$$

Consequently it will be my habit to omit the index in $\mathcal{Type}$ where uncontroversial—this phenomenon is known as **typical ambiguity** [HP91]. In practice, the cumulativity constraints required to ensure consistency of any development can be stored as a finite directed graph and checked for offending cycles.

The system of inference rules for the validity of contexts and typing judgments in the OLEG core calculus is given in table 2.4. The formulation is slightly unusual in that it involves no meta-level substitution in types—the same job is done by the computational behaviour of local definition, as performed by the **cuml** rule.

All the usual metatheoretic properties (see table 2.5) hold as we might expect them to. They contribute no insight unavailable from the Luo's treatment of ECC in [Luo94]. Severi and Poll have shown how to extend dependent type systems with local definitions [SP94]. The Church-Rosser property follows by the 'parallel reduction' argument of Tait and Martin-Löf, as modernised by Takahashi[Tak95]. Subject reduction, strengthening and cut follow by induction on typing derivations, differing only in minor details from the proofs for ECC. I have omitted weakening from the list because it is a special case of the **monotonicity** property which I shall prove in section 2.2.2.

Strong normalisation for the OLEG core is a direct consequence of strong normalisation for ECC. In the style of Severi and Poll, a type-preserving translation maps OLEG terms to ECC terms adding apparently pointless β-redexes[4], so that each step in an OLEG reduction sequence can then be simulated by a step in an ECC reduction of its translation. Consequently, an infinite reduction sequence for a well-typed OLEG term becomes an infinite reduction sequence for a well-typed ECC term, and we know that no such thing exists.

The interesting aspect of OLEG is its development superstructure. Let us now give our attention to that.

2.2 the OLEG development calculus

> '*You can't put a hole where a hole don't belong.*' (Bernard Cribbins)

Holes stand for the parts of constructions we have not yet invented. Every hole should tell us two things about the candidates which may fill it:

- their **type**, T
- the **context**, Γ, of variables they may employ

We may ascribe these properties to the hole itself, by way of convenient abbreviation. The point is that it must be safe to fill such a hole with any t such that $\Gamma \vdash t : T$. Solutions must be **locally checkable** if working with holes is to be practicable.

As I have already mentioned, the treatment of holes for simple type systems is greatly helped by the fact that terms do not 'leak' into types. Consequently, any subterm may safely be replaced by another of the same type, without affecting the type of the containing term—local checkability of hole solutions is just one special case. Holes may safely be represented by unlabelled ? symbols, as typing places no dependency between them.

[4] It may help to think of the translation to ECC as tying old tin cans onto terms to make the ECC reduction as noisy as the OLEG reduction.

However, the application of a dependently typed function smuggles the argument term into the result type—this is why the replacement property fails. Consider the following example, in a context defining an equality symbol for natural numbers:

$$\begin{array}{ll}\lambda{=_N} & : \mathbb{N} \to \mathbb{N} \to \mathcal{P}rop \\ \lambda\mathrm{refl}_N & : \forall n : \mathbb{N}.\ n{=_N}n \\ \lambda\mathrm{sym}_N & : \forall m, n : \mathbb{N}.\ m{=_N}n \ \to\ n{=_N}m\end{array}$$

It may be reasonable to infer that

$$\mathrm{sym}_N\ ?\ ?\ (\mathrm{refl}_N\ ?)\ : ?{=_N}?$$

but we may not instantiate any of the ?s and retain this typing unless we instantiate them all—sym_N's first two arguments appear in the required type of $(\mathrm{refl}_N\ ?)$.

Somehow we must represent the information that the three ?s in $\mathrm{sym}_N\ ?\ ?\ (\mathrm{refl}_N\ ?)$ signify the same, and what more natural way could we choose than to give them a single sign?

Hence, let us invent a new binding operator '?' (pronounced 'hole') to introduce variables standing for holes in a proof which must be instantiated by a common candidate of an appropriate type. We may now add to the context

$$?x : \mathbb{N}$$

and infer

$$\mathrm{sym}_N\ x\ x\ (\mathrm{refl}_N\ x)\ :\ x{=_N}x$$

When we think of a suitable candidate, we may 'solve the hole' by changing the ?-binding to, say, $!x = 0 : \mathbb{N}$, and the typing will stand.

However, the danger has not gone away. We could quite reasonably infer

$$\mathrm{sym}_N\ (?n : \mathbb{N}.\ n)\ (?n : \mathbb{N}.\ n)\ (\mathrm{refl}_N\ ?n : \mathbb{N}.\ n)\ : ?n : \mathbb{N}.\ n =_N\ ?n : \mathbb{N}.\ n$$

since one binding $?n : \mathbb{N}.\ n$ is syntactically identical to another, even if we regard the bound variables as distinct (indeed, not in the same scope). We are not free to solve one without the others, so we must avoid this situation.

The point is that although there is nothing wrong with holes leaking into types, disaster strikes if we permit ?-*bindings* to do so. The OLEG development calculus ensures that ?-bindings are always in safe places—ie, that they may always be solved independently. Indeed, this arises as a corollary of a more general replacement property, just like in the simply typed case.

By introducing an explicit binding operator for holes, OLEG follows Dale Miller's lead [Mil91, Mil92] in representing the state of the system as a judgment whose context or 'mixed prefix' explains the variously quantified variables involved.

The OLEG development calculus represents the store of a theorem prover directly at the judgment level. Theorem provers tend to contain four kinds of information:

- assumptions
- proved theorems
- unproved claims
- partial proofs of claims

These four **components** are each represented by a form of binding—respectively, the four given in the definition below. A **state** is a context of such bindings. The terms of the development calculus are called **partial constructions**.

DEFINITION: **states, components and partial constructions**

states $\mathcal{State}$

$$\frac{}{\langle\rangle \in \mathcal{State}} \qquad \frac{\Delta \in \mathcal{State} \quad C \in \mathcal{C}^x_{\mathcal{V}}}{\Delta; C \in \mathcal{State}}$$

components $\mathcal{C}^x_{\mathcal{V}}$ for $x \in \mathcal{I} - \mathcal{V}$

$$\frac{S \in \mathcal{T}_{\mathcal{V}}}{\lambda x : S \in \mathcal{C}^x_{\mathcal{V}}} \qquad \frac{s, S \in \mathcal{T}_{\mathcal{V}}}{!x = s : S \in \mathcal{C}^x_{\mathcal{V}}}$$

$$\frac{S \in \mathcal{T}_{\mathcal{V}}}{?x : S \in \mathcal{C}^x_{\mathcal{V}}} \qquad \frac{q \in \mathcal{P}_{\mathcal{V}} \quad S \in \mathcal{T}_{\mathcal{V}}}{?x \approx q : S \in \mathcal{C}^x_{\mathcal{V}}}$$

partial constructions $\mathcal{P}_{\mathcal{V}}$

$$\frac{t \in \mathcal{T}_{\mathcal{V}}}{t \in \mathcal{P}_{\mathcal{V}}} \qquad \frac{C \in \mathcal{C}^x_{\mathcal{V}} \quad p \in \mathcal{P}_{\mathcal{V} \cup \{x\}}}{C.\, p \in \mathcal{P}_{\mathcal{V}}}$$

Observe that partial proofs, or 'guesses' are attached to holes with a '$\approx$' symbol, indicating that they have not the computational force of !-bound values attached with '='. Indeed, we may view any state Δ as a core context by forgetting all but the ':' and '=' properties of each variable. This, in particular, means that guesses are invisible to the core.

States Δ in the development calculus are equipped with Δ-judgment forms $\mathcal{DevJ}_\Delta$ corresponding to those of the core. If $J \in \mathcal{DevJ}_\Delta$, we may assert that J holds by writing $\Delta \Vdash J$.

$\mathcal{DevJ}_\Delta$ is given by

$$\frac{}{\textbf{valid} \in \mathcal{DevJ}_\Delta} \qquad \frac{p \in \mathcal{P}_\Delta \quad T \in \mathcal{T}_\Delta}{p : \mathcal{T}_\Delta \in \mathcal{DevJ}_\Delta}$$

Even the abstract form of the typing judgment contains an important piece of information—the development calculus does not extend the type system, only the language of terms. Holes are never bound to the right of the ':'. This only serves to emphasise the analytic view that types come before terms—we do not explain terms with types, we search for terms inside types.

Table 2.6 shows the new inference rules. Note that core judgments $\Delta \vdash J$ only validate Δ viewed forgetfully as a core context—any guesses in Δ will not be checked. This accounts for the extra $\Delta \Vdash$ **valid** premises appearing in some rules.

The analogous metatheoretic properties from table 2.5 continue to hold for this extended system. The parallel reduction treatment for Church-Rosser and the derivation inductions for subject reduction, strengthening and cut can easily be adapted. Strong normalisation for the development calculus reduces to strong normalisation for the core by a translation argument which adds the assumption $\mathit{Imagine} : \forall A : \mathcal{Type}.\, A$ at the root of the context and turning ?-bindings into !-bindings: every hole without a guess, $?x : S$, becomes $?x \approx \mathit{Imagine}\, S : S$, then the translated guesses become !-bound values.

The core terms are embedded in the partial constructions. The forgetful interpretation of states as core contexts allows variables to appear in partial constructions via the **term** rule. The effect of the core/development separation is thus to restrict where holes may be *bound*. A partial construction containing no ?-bindings is said to be **pure**, that is, expressible as a core term. Pure terms may, of course, refer to variables ?-bound in their context.

In particular, ?-bindings cannot occur inside applications or !-bound values. This is enough to ensure that they have no interaction whatever with computation. ?-bindings state validity

empty $$\frac{}{\langle\rangle \Vdash \mathbf{valid}}$$

declare $$\frac{\Delta \Vdash \mathbf{valid} \quad \Delta \vdash S : \mathit{Type}_j}{\Delta; Cx\!:\!S \Vdash \mathbf{valid}} \quad C \in \{\lambda, ?\}$$

define $$\frac{\Delta \Vdash \mathbf{valid} \quad \Delta \vdash s : S}{\Delta; !x = s\!:\!S \Vdash \mathbf{valid}}$$

construct $$\frac{\Delta \Vdash p : S}{\Delta; ?x \approx p\!:\!S \Vdash \mathbf{valid}}$$

typing partial constructions

term $$\frac{\Delta \Vdash \mathbf{valid} \quad \Delta \vdash t : T}{\Delta \Vdash t : T}$$

abs $$\frac{\Delta; \lambda x : S \Vdash p : T}{\Delta \Vdash \lambda x\!:\!S.\, p : \forall x\!:\!S.\, T}$$

let $$\frac{\Delta; !x = s\!:\!S \Vdash p : T}{\Delta \Vdash !x = s\!:\!S.p : !x = s\!:\!S.T}$$

hole $$\frac{\Delta; ?x : S \Vdash p : T}{\Delta \Vdash ?x\!:\!S.\, p : T} \quad x \notin T$$

guess $$\frac{\Delta; ?x \approx q\!:\!S \Vdash p : T}{\Delta \Vdash ?x \approx q\!:\!S.\, p : T} \quad x \notin T$$

cuml $$\frac{\Delta \Vdash p : S}{\Delta \Vdash p : T} \quad \Delta \vdash S \preceq T$$

Table 2.6: development calculus inference rules

are allowed to appear in 'guesses'—partial constructions attached to holes as potential solutions. However, unlike !-bindings, guesses are merely typechecked annotations—they have no computational behaviour and hence no effect on subsequent typing.

In fact, the only contraction scheme at the partial construction level is !-contraction removing spent value bindings.

$$\frac{}{\Delta \Vdash !x = s\!:\!S.\,p \leadsto^{!} p} \quad x \notin p$$

All the other computations occur within embedded terms via the closure rules given below. Of course δ-reduction in terms can exploit !-components in the context.

$$\frac{\Delta \vdash t \leadsto. t'}{\Delta \Vdash t \leadsto. t'}$$

$$\frac{\Delta \vdash S \leadsto. S'}{\Delta \Vdash \lambda x\!:\!S.\,p \leadsto. \lambda x\!:\!S'.\,p} \qquad \frac{\Delta \vdash S \leadsto. S'}{\Delta \Vdash ?x\!:\!S.\,p \leadsto. ?x\!:\!S'.\,p}$$

$$\frac{\Delta \vdash s \leadsto. s'}{\Delta \Vdash !x = s\!:\!S.\,p \leadsto. !x = s'\!:\!S.\,p} \qquad \frac{\Delta \vdash S \leadsto. S'}{\Delta \Vdash !x = s\!:\!S.\,p \leadsto. !x = s\!:\!S'.\,p}$$

$$\frac{\Delta \Vdash q \leadsto. q'}{\Delta \Vdash ?x \approx q\!:\!S.\,p \leadsto. ?x \approx q'\!:\!S.\,p} \qquad \frac{\Delta \vdash S \leadsto. S'}{\Delta \Vdash ?x \approx q\!:\!S.\,p \leadsto. ?x \approx q\!:\!S'.\,p}$$

$$\frac{\Delta; C^x_\Delta \Vdash p \leadsto. p'}{\Delta \Vdash C^x_\Delta.\,p \leadsto. C^x_\Delta.\,p'}$$

A crucial role is played by the side-conditions in the **hole** and **guess** rules. These insist that although the term under a ?-binding may exploit the bound variable, its type may not. Without them, we should have to allow ?-bindings in types. Furthermore, these restrictions simply reflect the natural ways in which holes arise—in a refinement style proof, we only make claims motivated by the need to construct an inhabitant of a type which we already know.

The inability to leak into types is something partial constructions share with terms in simply typed systems. We should expect the **replacement** property to follow easily, and in exactly the same manner—replacing the typing subderivation for the replaced term.

2.2.1 positions and replacement

The **positions** $\mathcal{P}os_{\mathcal{V}}$ are formed from the partial constructions $\mathcal{P}_{\mathcal{V}}$ by deleting one sub-partial construction. A position $P \in \mathcal{P}os_\Delta$ 'forgetfully' induces a context extension

$\bar{P}$ which collects the components under which the deletion point lies. If $p \in \mathcal{P}_{\Delta;\bar{P}}$, then $P[p]$ denotes the partial construction obtained by inserting p at P's deletion point. Positions are defined inductively as follows

DEFINITION: **positions**

For $x \in \mathcal{I} - \mathcal{V}$

$$\frac{}{\circ \in \mathcal{P}os_{\mathcal{V}}} \qquad \frac{P \in \mathcal{P}os_{\mathcal{V}\cup\{x\}}}{\mathcal{C}^{x}_{\mathcal{V}}.P \in \mathcal{P}os_{\mathcal{V}}} \qquad \frac{P \in \mathcal{P}os_{\mathcal{V}} \quad S \in \mathcal{T}_{\mathcal{V}} \quad q \in \mathcal{P}_{\mathcal{V}\cup\{x\}}}{?x \approx P : S.\, q \in \mathcal{P}os_{\mathcal{V}}}$$

$$\begin{array}{rcl} \circ[p] & = & p \\ (\mathcal{C}^{x}_{\Delta}.\, P)[p] & = & \mathcal{C}^{x}_{\Delta}.\, P[p] \\ (?x \approx P : S.\, q)[p] & = & ?x \approx P[p] : S.\, q \end{array} \qquad \begin{array}{rcl} \Delta; \bar{\circ} & = & \Delta \\ \Delta; \overline{\mathcal{C}^{x}_{\Delta}.\, P} & = & \Delta; \mathcal{C}^{x}_{\Delta}; \bar{P} \\ \Delta; \overline{?x \approx P : S.\, q} & = & \Delta; \bar{P} \end{array}$$

Crucially, any derivation of $\Delta \Vdash P[p] : T$ follows from a subderivation of some $\Delta; \bar{P} \Vdash p : S$.

We may compose positions, writing $P; P'$ for the position obtained by replacing the $\circ$ in P by P'. Clearly $(P; P')[p] = P[P'[p]]$ and $\overline{P; P'} = \bar{P}; \bar{P'}$.

METATHEOREM: **replacement**

If	$\Delta \Vdash P[p] : T$
follows from	$\Delta; \bar{P} \Vdash p : R$
and	$\Delta; \bar{P} \Vdash p' : R$
then	$\Delta \Vdash P[p'] : T$

PROOF

The proof is by induction over the derivation of $\Delta \Vdash P[p] : T$, then case analysis on the position.

In all cases where the position is $\circ$, ie $P[p] = p$, the typing derivations for $P[p]$ and p must be the same, yielding that R is T—the conclusion is exactly the typing of p'.

We only need the strength of the induction when the position is nontrivial.

- **term**—$\circ$ is the only position
- **abs**

We have	$\Delta \Vdash \lambda x : S.\, P[p] : \forall x : S.\, T$
so we must have	$\Delta; \lambda x : S \Vdash P[p] : T$
following from	$\Delta; \lambda x : S; \bar{P} \Vdash p : R$
Suppose	$\Delta; \lambda x : S; \bar{P} \Vdash p' : R$
Inductively	$\Delta; \lambda x : S \Vdash P[p'] : T$
Hence	$\Delta \Vdash \lambda x : S.\, P[p'] : \forall x : S.\, T$

- **let**, **hole**

  We go under the component in the same way as for **abs**.

- **guess**

  If the position goes under the component, the above argument applies. If the position goes into the guess:

We have	$\Delta \Vdash ?x \approx P[p]\!:\!S.\, q \;:\; T$
so we must have	$\Delta \Vdash P[p] \;:\; S$
following from	$\Delta; \bar{P} \Vdash p \;:\; R$
Suppose	$\Delta; \bar{P} \Vdash p' \;:\; R$
Inductively	$\Delta \Vdash P[p'] \;:\; S$
Hence	$\Delta \Vdash ?x \approx P[p']\!:\!S.\, q \;:\; T$

  Note that the typing of q and the side-condition $x \notin T$ are not affected by the change of guess.

$\square$

To claim that the simplicity of this theorem belies its utility is to misunderstand the pragmatics of theory. There is no utility without simplicity.

2.2.2 the state information order

We shall need a little more metatheoretical apparatus before we are ready to reconstruct theorem proving in OLEG. In particular, we shall need a notion of 'progress' between OLEG states. The idea is that a state Δ' 'improves' Δ if it contains at least as much information—Δ' must simulate the behaviour of every variable in Δ.

DEFINITION: **state information order**

For valid states Δ and Δ', we say $\Delta \sqsubseteq \Delta'$, if

- For each $x \in \Delta$, if $\Delta \vdash x \;:\; T$ then $\Delta' \vdash x \;:\; T$.
- For each $x \in \Delta$, if $\Delta \vdash x \leadsto^{\delta} s$ then $\Delta' \vdash x \cong s$.

$\sqsubseteq$ is clearly a preorder.

Notice that inserting new components into Δ moves it up the order. So does replacing a type-only binding or ?-with-guess by a !-binding of an appropriately typed value. Furthermore, guesses may be added to holes, removed or modified at will, so long as their intended type is respected—the replacement property helps us to check modifications. On the other hand, once a variable has a computational behaviour, we may not take it away.

If $\Delta \sqsubseteq \Delta'$ then, viewed as variable sets $\Delta \subset \Delta'$, so $\mathcal{DevJ}_{\Delta} \subset \mathcal{DevJ}_{\Delta'}$. As the ordering preserves all the observable behaviour of a state, we should expect to find the following holds

METATHEOREM: **monotonicity**

If $\Delta \sqsubseteq \Delta'$, then for all $J \in \mathcal{DevJ}_{\Delta}$, $\Delta \Vdash J$ implies $\Delta' \Vdash J$

PROOF

We must first generalise a little:

If	$\Delta_1 \sqsubseteq \Delta'_1$		
then	$\Delta_1, \Delta_2 \Vdash J$	implies	$\Delta'_1, \Delta_2 \Vdash J$
and	$\Delta_1; \Gamma \vdash J$	implies	$\Delta'_1; \Gamma \vdash J$
provided	Δ'_1 captures no variables from Δ_2 or Γ		

This allows us an easy induction on derivations. From the definition of $\sqsubseteq$, we shall acquire exactly the components we need to replace those subderivations which look up types from Δ_1, perform δ-reductions from Δ_1 or simply validate Δ_1. That is, the interesting cases are

- the **var** rule

  If the variable being typed lies in Δ_1, the definition of $\sqsubseteq$ tells us how to derive the same type from Δ'_1. Otherwise, the result follows from the inductive hypothesis, which replaces the prefix in the premise, and the **var** rule, which recovers the type from the unchanged suffix.

- the validity rules

  If the context being validated is Δ_1, replace the entire derivation with that of $\Delta'_1 \vdash$ **valid**. Otherwise, the context strictly contains Δ_1, so the premise context contains Δ_1, hence the inductive hypothesis applies.

- the **cuml** rule

  The inductive hypothesis supplies the modified premise. As for the computational side condition, the definition of $\sqsubseteq$ enables us to replace every δ-reduction for variables in Δ_1 with an equivalent conversion valid in Δ'_1.

□

2.3 life of a hole

I now present four basic replacement operations which act as a basis for working with holes.

claim (birth) $$\frac{\Delta \Vdash p : T \quad \Delta \vdash S : \mathit{Type}}{\Delta \Vdash ?x{:}S.\, p : T}$$

try (marriage) $$\frac{\Delta \Vdash ?x{:}S.\, p : T \quad \Delta \Vdash q : S}{\Delta \Vdash ?x \approx q{:}S.\, p : T}$$

regret (divorce) $$\frac{\Delta \Vdash ?x \approx q{:}S.\, p : T}{\Delta \Vdash ?x{:}S.\, p : T}$$

solve (death) $$\frac{\Delta \Vdash ?x \approx q{:}S.\, p : T}{\Delta \Vdash !x = q{:}S.\, p : T} \quad q \text{ pure}$$

It is clear that each of these rules is admissible. We may read them as justifying the replacement at any position of the construction in the premise by the construction in the conclusion. Monotonicity justifies the corresponding steps which insert and modify new ?-components in the state.

Effectively, **claim** allows us to insert a new hole at any position. Holes are naturally born this way—we claim that S holds in order to develop our proof of T. The side-condition on the **hole** typing rule holds as a matter of course.

The **try** and **regret** steps allow us to attach and discard guesses repeatedly—hopefully our judgment improves as we go round the cycle. Once the guess contains no ?-components, it has become a core term—death is not the end of the journey, but the transition by which a hole is **solve**d, becoming a local definition.

These four rules allow us to extend the notion of 'information order' to *positions*. This gives us the means to relate operations which are focused at a particular position to the amount of information available at that position.

DEFINITION: **position information order**

For valid states Δ and $P, P' \in \mathcal{P}os_\Delta$, the **position information order** $\Delta \Vdash P \sqsubseteq P'$ is given inductively by the following rules:

refl $$\frac{}{\Delta \Vdash P \sqsubseteq P}$$

trans $$\frac{\Delta \Vdash P \sqsubseteq P' \quad \Delta \Vdash P' \sqsubseteq P''}{\Delta \Vdash P \sqsubseteq P''}$$

claim $$\frac{\Delta; \bar{P} \vdash S : \mathit{Type}}{\Delta \Vdash P; P' \sqsubseteq P; ?x\!:\!S.\, P'}$$

try $$\frac{\Delta; \bar{P} \Vdash q : S}{\Delta \Vdash P; ?x\!:\!S.\, P' \sqsubseteq P; ?x \approx q\!:\!S.\, P'}$$

regret $$\frac{}{\Delta \Vdash P; ?x \approx q\!:\!S.\, P' \sqsubseteq P; ?x\!:\!S.\, P'}$$

solve $$\frac{}{\Delta \Vdash P; ?x \approx q\!:\!S.\, P' \sqsubseteq P; !x = q\!:\!S.\, P'} \quad q \text{ pure}$$

The admissibility of the four basic replacement operations given above ensures that if $\Delta \Vdash P \sqsubseteq P'$, then

- if $\Delta; \bar{P} \Vdash$ **valid** then $\Delta; \bar{P'} \Vdash$ **valid** and $\Delta; \bar{P} \sqsubseteq \Delta; \bar{P'}$
- if $\Delta \Vdash P[p] : T$ then $\Delta \Vdash P'[p] : T$

We may now reconstruct the familiar tools of refinement proof as operations which manipulate OLEG states, preserving their validity. Moreover, we can make assurance double sure at any stage by rederiving the state's validity judgment. This direct correspondence between judgments of the type theory and states of the machine, and thus between admissible rules and tactics, is quite a solid basis on which to build a proof assistant.

2.4 displaying an OLEG state

The prototype implementation of OLEG was written for use by other programs, rather than by people. However, this thesis is full of OLEG proofs, so we shall need some way to see what we are doing. Let us think how we might display an OLEG state.

I propose to list the components of a state vertically, so that the more local bindings are literally as well as metaphorically under the more global ones. For each binding, we should give the binding operator, the identifier, then a table showing the property indicators (':' or '=') with the associated terms (types or values). It will serve the cause of brevity if we sometimes relax vertical alignment, combining bindings when more than one identifier is being given the same treatment.

Now, any term can be viewed as a subterm under a context of binders—let us allow ourselves to format these contexts in the same way as the 'main' state and write the subterm directly underneath. For example, we might have the state shown on the right.

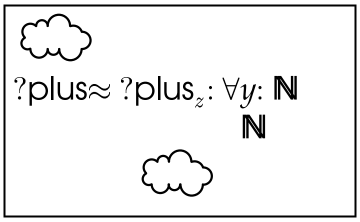
The initial four assumptions introduce a type $\mathbb{N}$ of natural numbers, its two constructors and a primitive recursion operator. Following this, I have shown a partial development of the addition function. Observe the completed successor case introduced by a ! binding, while the zero case is still an unknown bound with a ?. The partial proof is bound with a ?, indicating that it still contains ?-bindings which must not be duplicated. Note also that !-binding enables us to inspect terms such as plus_s which would otherwise be stuck beneath the application of rec.

$$
\boxed{\begin{array}{l}
\lambda\mathbb{N} \quad : \mathit{Type}\\
\lambda 0 \quad\; : \mathbb{N}\\
\lambda \mathsf{s} \quad\; : \mathbb{N}\to\mathbb{N}\\
\lambda\mathsf{rec} : \forall\Phi:\mathit{Type}\\
\qquad\quad\;\; \forall\phi_{\mathrm{z}}:\Phi\\
\qquad\quad\;\; \forall\phi_{\mathrm{s}}:\Phi\to\Phi\\
\qquad\quad\;\; \forall n\;:\mathbb{N}\\
\qquad\qquad\; \Phi\\
?\mathsf{plus}\approx\ ?\mathsf{plus}_z : \;\mathbb{N}\to\mathbb{N}\\
\qquad\quad\;\; !\mathsf{plus}_s = \lambda\mathsf{plus}_x : \mathbb{N}\to\mathbb{N}\\
\qquad\qquad\qquad\qquad \lambda y \qquad : \mathbb{N}\\
\qquad\qquad\qquad\qquad\; \mathsf{s}(\mathsf{plus}_x\, y)\\
\qquad\qquad\qquad\quad\;\; : \forall\phi : \mathbb{N}\to\mathbb{N}\\
\qquad\qquad\qquad\qquad\; \forall y : \mathbb{N}\\
\qquad\qquad\qquad\qquad\quad \mathbb{N}\\
\qquad\quad\;\; \mathsf{rec}\;(\mathbb{N}\to\mathbb{N})\;\mathsf{plus}_z\;\mathsf{plus}_s\\
\qquad : \mathbb{N}\to\mathbb{N}\to\mathbb{N}
\end{array}}
$$

In general, then, a state is displayed as a tree whose forking nodes are bindings. From each binding, one edge points 'underneath' to its scope, and others point 'sideways' into the terms attached by the property indicators ':', '=' and '≈'. The sequence of components which make up a state form a spine of the tree, vertically aligned at the left hand side, starting at the root and following the 'underneath' edges until the last component is reached. In the above example, the spine consists of the bindings for $\mathbb{N}$, 0, s, rec and plus. The subtrees reached by going 'sideways' from this spine (representing, for example, the type of rec, or the incomplete development of plus) have terms or partial constructions at the leaves. As we have seen, wherever we find a partial construction, we may replace it by another of the same type.

It is unlikely that we should always want to see the whole tree of a large state fully expanded as above. You can, perhaps, imagine using a mouse to draw clouds round uninteresting parts of proofs, introducing a cloud symbol in the state display. Perhaps we can double-click on the cloud to restore the expanded tree.

Let us expand in detail only where we are interested, keeping connected subtrees of uninteresting proof obscured by clouds. For instance, if we are simply interested in unsolved goals, the above example can be reduced to this picture.



I like this interface for obscuring irrelevant details because I can focus on a subtree without displaying the full path back to the root. Further, if we allow subtrees containing clouds to be obscured by bigger clouds, we can structure our lack of interest—when we expand the larger cloud to return to that part of the development, the bits marked as dull remain hidden.

Now that we can visualise the state, let us visualise tactics as direct manipulations of the displayed image. Each symbol is given a tangible presence by its binding. Operations which affect a symbol should be addressed, by mouse or whatever, to its binding. We shall soon find ourselves dragging bindings about the place, and so forth. It is, perhaps, an advantage of making ?-bindings explicit that they afford such visual metaphors.

2.5 basic component manipulations

I shall present tactics as qualified state transitions. The validity of the final state must follow from that of the initial state, given the side-conditions.

$$\boxed{\textit{before}} \Rightarrow \boxed{\textit{after}}$$

side-conditions

Table 2.7 shows some basic tactics for manipulating components at the outer level of the OLEG state. These tactics are justified by monotonicity, except for **cut** and **abandon** which are standard metatheoretic properties.

We may also represent replacement as a tactic, instantiating it to acquire tactics which work at any position in the development. Consequently, we may apply **claim**, **abandon**, **try**, **regret**, **solve** and **cut** within guesses. However, we are not free to create and destroy λs, as these would not preserve the type of the partial construction.

$$\boxed{\begin{array}{l}\Delta\\ ?x \approx P[p] : S\\ \Delta'\end{array}} \Rightarrow \boxed{\begin{array}{l}\Delta\\ ?x \approx P[p'] : S\\ \Delta'\end{array}}$$

$$\begin{array}{l}\Delta; \bar{P} \Vdash p \ : \ R\\ \Delta; \bar{P} \Vdash p' \ : \ R\end{array}$$

Of course, we will want to operate with a little more sophistication than to edit partial constructions directly with **try** and **regret**. What we have established is the machinery

assume $\boxed{\begin{array}{l}\Delta\\ \Delta'\end{array}} \Rightarrow \boxed{\begin{array}{l}\Delta\\ \lambda x : S\\ \Delta'\end{array}}$ $\Delta \vdash S : \mathcal{Type}$	**justify** $\boxed{\begin{array}{l}\Delta\\ \lambda x : S\\ \Delta'\end{array}} \Rightarrow \boxed{\begin{array}{l}\Delta\\ ?x : S\\ \Delta'\end{array}}$	**claim** $\boxed{\begin{array}{l}\Delta\\ \Delta'\end{array}} \Rightarrow \boxed{\begin{array}{l}\Delta\\ ?x : S\\ \Delta'\end{array}}$ $\Delta \vdash S : \mathcal{Type}$
try $\boxed{\begin{array}{l}\Delta\\ ?x : S\\ \Delta'\end{array}} \Rightarrow \boxed{\begin{array}{l}\Delta\\ ?x \approx p : S\\ \Delta'\end{array}}$ $\Delta \Vdash p : S$	**regret** $\boxed{\begin{array}{l}\Delta\\ ?x \approx p : S\\ \Delta'\end{array}} \Rightarrow \boxed{\begin{array}{l}\Delta\\ ?x : S\\ \Delta'\end{array}}$	**solve** $\boxed{\begin{array}{l}\Delta\\ ?x \approx p : S\\ \Delta'\end{array}} \Rightarrow \boxed{\begin{array}{l}\Delta\\ !x = p : S\\ \Delta'\end{array}}$ p pure
cut $\boxed{\begin{array}{l}\Delta\\ !x = s : S\\ \Delta'\end{array}} \Rightarrow \boxed{\begin{array}{l}\Delta\\ [s/x]\Delta'\end{array}}$	**abandon** $\boxed{\begin{array}{l}\Delta\\ ?x : S\\ \Delta'\end{array}} \Rightarrow \boxed{\begin{array}{l}\Delta\\ \Delta'\end{array}}$ $x \notin \Delta'$	**postpone** $\boxed{\begin{array}{l}\Delta\\ ?x : S\\ \Delta'\end{array}} \Rightarrow \boxed{\begin{array}{l}\Delta\\ \lambda x : S\\ \Delta'\end{array}}$

Table 2.7: basic component manipulations

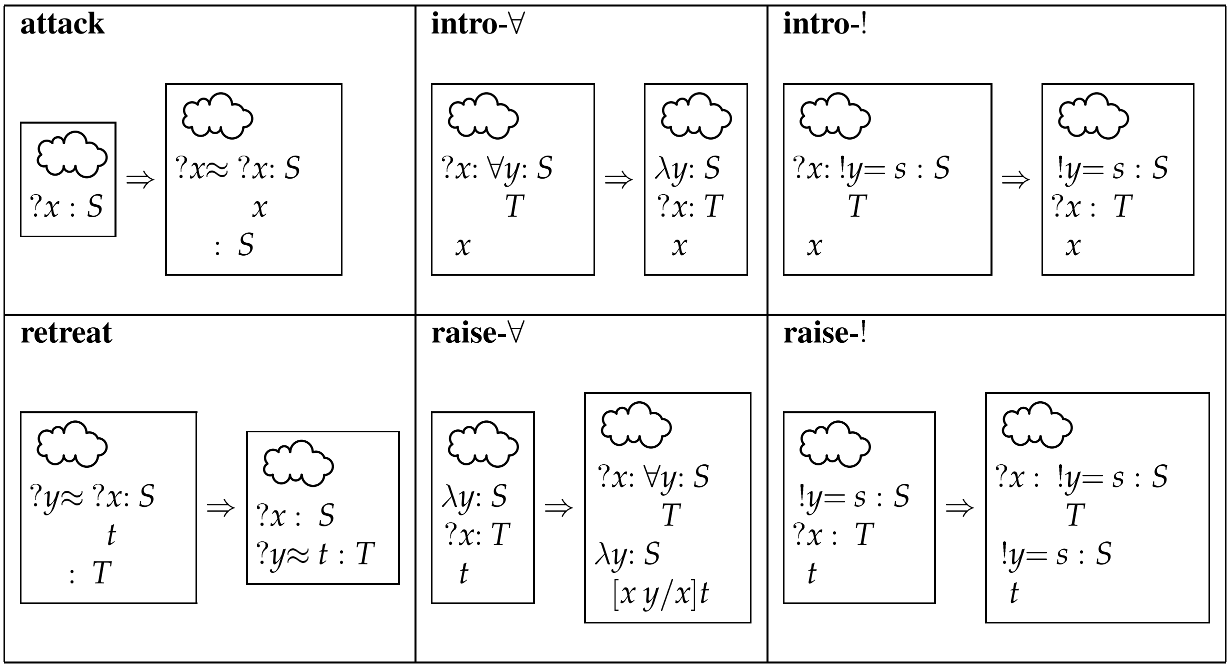


Table 2.8: moving holes through their types

for making holes appear and disappear in a given place. Just like the cinema, if we work this machinery fast enough, we create the illusion of movement.

2.6 moving holes

Traditionally, we may 'introduce' a term of functional type by filling a hole with a λ-binding whose body is a new hole—the context of the new hole contains the argument of the function. We may 'animate' this manoeuvre by pretending that the ?-binding has moved under the argument and shortened its type.

Another familiar manoeuvre undoes the effect of introduction, generalising a hole functionally over the assumptions from which it was to be proven. The ?-binding moves outwards through the binding of the assumption, and its type gets longer. Miller calls this 'raising' [Mil91]. We may also shuffle holes in and out through !-bindings appearing in their types.

These moves are collected in table 2.8.

Notice that the two introduction tactics only replace constructions of the form $?x{:}S.\,x$. If the body was some arbitrary t, the introductions would affect the x's in t, rather than the whole expression. Fortunately, any hole not of this form can be made ready for introduction by the **attack** tactic.

The raising tactics, however, have no such restriction. They allow us to move holes out through assumptions and definitions, becoming more functional as they go, until they are outermost in a guess. The **retreat** tactic may then be used to extract them from the development. A partial construction can always be made pure by raising and retreating its remaining ?-bindings.

2.7 refinement and unification

The **intros**-$\forall$ tactic makes progress by filling a hole with a λ-term. This section builds tactics to fill holes with applications.

Let us begin with a very simple motivating example, say, developing the double function for natural numbers in terms of plus.

$?\mathsf{double}\colon \forall n\colon \mathbb{N}$
$\quad \mathbb{N}$

We may introduce the argument by **attack**, then **intros**-λ.

$?\mathsf{double} \approx \lambda n\colon \mathbb{N}$
$\quad ?d\colon \mathbb{N}$
$\quad d$

At this point, we might decide to solve for d by adding two numbers together. We can do this, even if we have not yet decided which two numbers, by inserting holes for the numbers. That is, we first **claim** . . .

$?\mathsf{double} \approx \lambda n \;\colon \mathbb{N}$
$\quad ?x, y\colon \mathbb{N}$
$\quad ?d \;\colon \mathbb{N}$
$\quad d$

. . . then **solve** d with plus x y and **cut**.

We have filled one old hole, d, with plus applied to two new holes—that is, we have 'refined d by plus'.

$?\mathsf{double} \approx \lambda n \;\colon \mathbb{N}$
$\quad ?x, y\colon \mathbb{N}$
$\quad \mathsf{plus}\; x\; y$

To complete the development, refine each of x and y by n.

$!\mathsf{double} = \lambda \mathrm{n}\colon \mathbb{N}$
$\quad \mathsf{plus}\; \mathrm{n}\; \mathrm{n}$

The tactic **naïve-refine** solves one hole by a given function applied to unknowns represented by new holes. It is a combination of **claim**, **try** and **solve**.

Note that I have not explained how the length of the argument sequence $\vec{x}$ is to be chosen. We could leave it to the user. However, since convertability is decidable, we can simply try the successively longer sequences afforded by the ∀s in the type of *f* until either one works or we run out of arguments.

naïve-refine

$$\boxed{?y : T} \Rightarrow \boxed{\begin{array}{l} ?\vec{x} : \vec{S} \\ !y = f\,\vec{x} \\ \quad : T \end{array}}$$

$f : \forall \vec{x} : \vec{S}.\ T$

Hence, in the above example, we effectively solved *d* by **naïve-refine** with plus, then each of *x* and *y* by **naïve-refine** with *n*.

While **naïve-refine** is good for simple types, as one might expect, it is not sufficiently powerful to be of real use in the dependently typed setting. Consider this (admittedly somewhat artificial) problem.

$$\begin{array}{ll} \lambda n & : \mathbb{N} \\ \lambda nsym & : \forall x : \mathbb{N} \\ & \quad \forall nex : n =_N x \\ & \quad\quad x =_N n \\ \lambda m & : \mathbb{N} \\ \lambda nem & : n =_N m \\ ?y & : \mathbb{N} \\ ?mey & : m =_N y \end{array}$$

We have to find a y such that $m =_N y$. Careful examination shows that n will do, with a derivation via *nsym*. However, we cannot simply do **naïve-refine** on *mey* with *nsym*, because *nsym* proves $x =_N n$, not $m =_N y$.

We can, however, start by building an application of *nsym* in a !-binding.

$$\begin{array}{ll} ?x & : \mathbb{N} \\ ?nex & : n =_N x \\ !xen = nsym\ x\ nex & \\ & : x =_N n \\ ?mey & : m =_N y \end{array}$$

If we could solve x and y so that the types of *xen* and *mey* became convertible, we could complete the refinement. That is, we need to **unify** the two types. Note that it is unification we need, not just one-sided matching—we need to infer values for holes in the goal, not just unknown arguments.

This thesis is not the place for a discussion of unification for proof search—there is much work on this in the literature [Pym90]. For my purposes, something similar in power to first-order unification on normalised terms will prove adequate. OLEG's explicit bindings of holes and assumptions, and its support for various operations which permute them, suggest that Miller's technology for unification 'under a mixed prefix' [Mil92] could be imported very easily.

Let us therefore imagine 'buying in' a pre-existing unification tool and use it to drive the tactic shown below.

unify

$$\boxed{\begin{array}{l}\Delta\\ ?y\approx P\begin{bmatrix}!x'=s:S\\ ?x\ :\ T\\ p\end{bmatrix}\\ \\ \end{array}} \Rightarrow \boxed{\begin{array}{l}\Delta'\\ ?y\approx P'\begin{bmatrix}!x=s:T\\ [x/x']p\end{bmatrix}\\ \\ \end{array}}$$

$$\begin{array}{l}\Delta\ \sqsubseteq\ \Delta'\\ \Delta'\ \Vdash\ P\ \sqsubseteq\ P'\\ \Delta';\bar{P}'\ \vdash\ S \preceq T\end{array}$$

The idea is that **unify** solves holes until enough new δ-reductions have been added to give $S \preceq T$. At that point, the desired s, temporarily stored in a !-binding, can be filled in as the value for x.

Note that there is nothing to stop **unify** creating new holes, although this is unusual for first-order algorithms. Nor do I require the unification process to terminate, although this sometimes helps.

We may now build a two-phase tactic which incorporates unification in the refinement process.

unify-refine

1. $$\boxed{\begin{array}{l}\Delta\\ ?y\approx P\begin{bmatrix}?x\!:T\\ p\end{bmatrix}\\ \\ \end{array}} \Rightarrow \boxed{\begin{array}{l}\Delta\\ ?y\approx P\begin{bmatrix}?\vec{z}\ :\ \vec{R}\\ !x'=f\,\vec{z}:S\\ ?x\ :\ T\\ p\end{bmatrix}\\ \\ \end{array}}$$

   $f\ :\ \forall\vec{z}\!:\!\vec{R}.\ S$

2. **unify** x' and x (at position $P; ?\vec{z} : \vec{R}$)

As with **naïve-refine**, it is not necessary to specify how many arguments f should have in advance. Provided we are willing to wait for the unification attempts, we may simply start with none and keep trying successively until either the unification succeeds or we run out of ∀s. This is exactly the behaviour of LEGO's notorious '`Refine`' tactic.[5]

An alternative to this search behaviour is a more precise 'drag-and-drop' technique. We can imagine a mouse action picking up a hypothesis by a suffix of the functional part of its type and dropping it on a hole. The arguments of the hypothesis before the point we selected would be the ones made holes by **unify-refine**.

Perhaps you are familiar with the children's toy which consists of a postbox with various holes of different shapes in the top. The toy comes with a number of blocks, and the object of the exercise is to post each block through the correct hole. In order to do this, the child is given a refinement tactic which takes the form of a blue plastic hammer. There is an initial phase where the connection between the shape of the block and the shape of the hole has not yet been made—a phase characterised by violent hammering and tantrums. Everyone who has ever learned LEGO has undergone a similar experience.

2.8 discharge and other permutations

Let me complete this reconstruction of basic theorem-proving in OLEG with some more technology for shuffling components around. It is fairly clear that we may permute components in the state in any way which preserves the dependencies between them.

Where dependency does arise, we may still reorder the components, but we have to account for it by introducing appropriate functional behaviour. In particular, this allows us to **discharge** an assumption by making everything which follows from it functional over it. LEGO implements this transformation by its '`Discharge`' tactic. We may reconstruct it piecewise by the four manipulations given in table 2.9.

Although we may read each of the 'four discharges' as pulling the binding of x through the binding of y, this is just a cinematic illusion. They are, of course, proven by creating an earlier binding for y, then expressing the later one in terms of it. By monotonicity, we may make the same permutations for ?-components as we can for λ-components.

We may also make permutations and deletions in the argument types of functional holes, so long as we do not break any dependencies. See table 2.10. If we bracket such moves with raising and introduction, we can make similar permutations and deletions in arguments which have already been introduced.

[5] LEGO tries quite hard to keep going, applying weak head-normalisation at each step, in an attempt to reveal a fresh ∀-binding.

λ-through-λ $\boxed{\begin{array}{l}\Delta \\ \lambda x : S \\ \lambda y : T \\ \Delta'\end{array}} \Rightarrow \boxed{\begin{array}{l}\Delta \\ \lambda y : \forall x\!:\!S.\ T \\ \lambda x : S \\ [y\ x/y]\Delta'\end{array}}$	**λ-through-!** $\boxed{\begin{array}{l}\Delta \\ \lambda x : S \\ !y = t\!:\!T \\ \Delta'\end{array}} \Rightarrow \boxed{\begin{array}{l}\Delta \\ \ !y = \lambda x\!:\!S.\ t \\ \quad :\ \forall x\!:\!S.\ T \\ \lambda x :\ S \\ [y\ x/y]\Delta'\end{array}}$
!-through-λ $\boxed{\begin{array}{l}\Delta \\ !x = s\!:\!S \\ \lambda y : T \\ \Delta'\end{array}} \Rightarrow \boxed{\begin{array}{l}\Delta \\ \lambda y :!x = s\!:\!S.\ T \\ !x = s\!:\!S \\ \Delta'\end{array}}$	**!-through-!** $\boxed{\begin{array}{l}\Delta \\ !x = s\!:\!S \\ !y = t\!:\!T \\ \Delta'\end{array}} \Rightarrow \boxed{\begin{array}{l}\Delta \\ !y = !x = s\!:\!S.\ t \\ \quad :\ !x = s\!:\!S.\ T \\ !x = s : S \\ \Delta'\end{array}}$

Table 2.9: the four discharges

swap-independent $\boxed{\begin{array}{l} ?f\!: \forall \vec{x}\!: \vec{S} \\ \quad \forall y\!: Y \\ \quad \forall z\!: Z \\ \quad\ T\end{array}} \Rightarrow \boxed{\begin{array}{l} ?f' :\ \forall \vec{x}\!: \vec{S} \\ \qquad \forall z\!: Z \\ \qquad \forall y\!: Y \\ \qquad\ T \\ !f = \lambda \vec{x}, y, z.\ f'\ \vec{x}\ z\ y\end{array}}$ $y \notin Z$	**delete-unused** $\boxed{\begin{array}{l} ?f\!: \forall \vec{x}\!: \vec{S} \\ \quad \forall y\!: Y \\ \quad\ T\end{array}} \Rightarrow \boxed{\begin{array}{l} ?f' :\ \forall \vec{x}\!: \vec{S} \\ \qquad T \\ !f = \lambda \vec{x}, y.\ f'\ \vec{x}\end{array}}$ $y \notin T$

Table 2.10: permuting and deleting arguments

2.9 systems with explicit substitution

Now seems a good time to compare OLEG's treatment of holes with that of other systems.

The key issue is how to cope with holes leaking out of the scope of their explanation. LEGO ignores this issue and reaps the consequent frightful harvest—although instantiations are typechecked, they may involve out-of-scope values which are only detected once the completed 'proof' is being verified. OLEG deals with the problem by forbidding it—a hole may not escape its scope, but its scope may be widened by raising, keeping the dependency information explicit and intact.

The real comparison lies with systems which treat this problem via explicit substitution, such as TypeLab [vHLS98] and ALF [Mag94]. Holes appear in the calculi underlying both systems without explicit binding. Instead, the context and type of a hole are recorded in an external ledger. By good design, this context coincides with the collection of bound variables under which the hole makes its initial appearance, but computation may destroy this coincidence, so explicit substitution is required to fix it up.

[vHLS98] illustrates this with a simple example. Suppose $?$ is defined to have type T in context $x : T$. That is, its ledger entry is $x : T \vdash ? : T$. Now consider the term

$$(\lambda x : T.?)\, t$$

We are told that the λ-abstracted x is 'the same object' as the x in the ledger. On the one hand, we may instantiate $?$ with x and β-reduce to get t. On the other, we may β-reduce to get $?$ which we can then instantiate with x. The two do not commute, as they show in this diagram:

$$\begin{tikzcd}
(\lambda x : T.?)\, t \arrow[rr, "\{? := x\}"] \arrow[d, "\beta"'] & & (\lambda x : T.x)\, t \arrow[d, "\beta"] \\
? \arrow[r, "\{? := x\}"'] & x & t
\end{tikzcd}$$

The trouble is that performing the β-reduction first introduces a discrepancy—the term no longer contains a binding occurrence of x corresponding to the x in the ledger, hence the subsequent instantiation is a touch anachronistic. In fact, the substitution implicit in the β-reduction has passed through $?$ without stopping to consider the fact that some x's might appear when it is instantiated, hence the solution is to delay explicitly the application of the substitution to $?$. When $?$ is instantiated, the substitution may proceed. That is, we repair the leak in scope by attaching an explicit substitution to the hole:

$$\begin{tikzcd}
(\lambda x : T.?)\, t \arrow[r, "\{? := x\}"] \arrow[d, "\beta"'] & (\lambda x : T.x)\, t \arrow[d, "\beta"] \\
?[x := t] \arrow[r, "\{? := x\}"'] & t
\end{tikzcd}$$

The extra $[x := t]$ is really a kind of binding which maintains consistency with the ledger, so that $?$ remains a 'function' of x. That is, the problem remains 'think of a $T \to T$ function', and the value remains 'whatever-it-is applied to t'.

The OLEG approach to this problem is total cowardice—since such situations cause trouble, they are forbidden. In particular, we may not bind holes inside an application, so there is no relationship with β-reduction to untangle.

We can, of course, have the state shown on the right. However, the guess for f has no computational force. We cannot reduce $f\,t$ unless we widen f's scope by raising and retreating, undoing the introduction of the λ and leaving us with an explicitly functional hole.

$\lambda t :\ T$
$?f \approx \lambda x \colon T$
$\quad ?y \colon T$
$\quad\ y$
$\ f\,t$

Is this an unbearable restriction? I can assure you that it will give us no trouble in the course of this thesis. The point is that OLEG offers a genuine compromise between the ingenuity of explicit substitution and the pain of representing holes as, say, skolem functions over the entire context—holes need only be kept functional as far as they are used computationally.

2.10 sequences, telescopes, families, triangles

Finally, for this chapter, let me digress for a moment to introduce an important notational convenience which will serve both to abbreviate and clarify what follows. We will frequently encounter sequences of terms, often as arguments to functions or indices of type families. I wish to avoid the traditional $t_1 \ldots t_n$ for a number of reasons:

- it's too wide
- it introduces a subscript which is frequently irrelevant
- our binding syntax involves significant dots—throwing more dots around (in threes, no less) can only cause confusion

Forswear therefore the pointillist sequence in favour of de Bruijn's **telescope** notation [deB91]. A sequence $\vec{t}$, indicates a finite, perhaps empty sequence of terms, and, following that primitive monoidal urge, we have composition operator ; and empty sequence ε.

de Bruijn explains how to give such a sequence a 'type'. In a simply typed setting, we could just write $\vec{T}$, but things are a little more complicated for us: in our dependently typed world, the values of earlier terms in a sequence can affect the types of later terms. We cannot afford to lose this dependency information, hence we must incorporate some kind of placeholder into the type sequence notation.

DEFINITION: **telescope**

> If $\mathcal{V}$ is a set of variables not containing $x_1 \ldots x_n$, and $T_i \in \mathcal{T}_{\mathcal{V} \cup \{x_1 \ldots x_{i-1}\}}$ for $1 \leq i \leq n$, then $\vec{T}$ is an **$\vec{x}$-telescope** (where $\vec{x}$ abbreviates $x_1; \ldots x_n$ and $\vec{T}$ abbreviates $T_1; \ldots T_n$).
>
> That is, we define a sequence of types relative to a sequence of identifiers which become bound in turn and stand as placeholders for earlier values in later types.

For example, the $\vec{x}$-telescope

$$\overbrace{\mathit{Type}}^{x_1} ; \overbrace{x_1 \to \mathit{Prop}}^{x_2} ; \overbrace{\forall y : x_1 .\ x_2\ y}^{x_3}$$

represents a triple of, respectively, a type x_1, a predicate x_2 over x_1, and a proof x_3 that all elements of x_1 satisfy x_2.

We may now exploit telescopes in all sorts of circumstances. For example, if $\vec{T}$ is an $\vec{x}$-telescope, the judgment $\Gamma \vdash \vec{t} : \vec{T}$ abbreviates the conjunction of the judgments

$$
\begin{array}{l}
\Gamma \vdash t_1 \;:\; T_1 \\
\Gamma \vdash t_2 \;:\; [t_1/x_1]T_2 \\
\vdots \\
\Gamma \vdash t_n \;:\; [t_{n-1}/x_{n-1}]\ldots[t_1/x_1]T_n
\end{array}
$$

Further, the binding $\lambda\vec{y} : \vec{T}$ abbreviates the sequence of n λ-bindings giving y_i the type T_i with y's for the x's, while $!\vec{y} = \vec{t}\!:\!\vec{T}$ abbreviates the corresponding n !-bindings, and similarly for the other binding operators.

We may thus speak of a sequence of bound variables as having a telescope where we would speak of a single bound variable as having a type. I shall glibly omit a telescope's placeholding variables, unless they are necessary to avoid ambiguity. When essential for clarity, I shall attach the placeholder variables to the types in situ, rather than naming them beforehand, making the above example

$$x_1 : \mathit{Type}; x_2 : x_1 \to \mathit{Type}; x_3 : \forall y\!:\!x_1.\, x_2\, y$$

The term 'telescope' comes from its notation-shrinking power, inspired by the kind of collapsible telescope that Horatio Nelson once famously put to his blind eye. It is a more appropriate metaphor for abbreviating a dependent type sequence than other collapsible structures such as accordions or opera hats because each of the concentric cylinders which makes up the telescope has a lip which constrains the next (and hence all the following cylinders).

The optical behaviour of telescopes is helpful also. Broadly speaking, the longer an optical telescope, the smaller the field of view and the greater the magnification. Similarly, as you extend a type telescope, each new type acts as a new constraint, so the collection of inhabiting sequences 'visible through the telescope' becomes smaller but more informative.[6]

There is another sense in which type telescopes are collapsible—if we instantiate the first placeholder, we acquire a more specific telescope shorter by one.

DEFINITION: **telescope application**

> If $\vec{T}$ is the $x_{1\ldots n}$-telescope $T_1; T_2; \ldots T_n$ and $t : T_1$, then the **application**
>
> $$\vec{T}\, t$$
>
> is the $x_{2\ldots n}$-telescope

[6] de Bruijn talks of sequences 'fitting into' telescopes, but I prefer to avoid the mixed metaphor.

$[t/x_1]T_2; \ldots [t/x_1]T_n$

Observe that if $t; \vec{t} : \vec{T}$ then $\vec{t} : \vec{T}\; t$.

This notion of application for telescopes may be iterated over a term sequence in the same way as function application, shortening a telescope by instantiating any prefix. That is, if $\vec{S}$ is an $\vec{x}$-telescope and $\vec{s} : \vec{S}$ then $(\vec{S}; \vec{T})\vec{s}$ is just $[\vec{s}/\vec{x}]\vec{T}$.

Note that the use semicolon for sequential composition leaves the comma free for its usual role, indicating multiple inhabitation of the same type or telescope. That is, $\vec{x}, \vec{y} : \vec{T}$ means that each of $\vec{x}$ and $\vec{y}$ inhabits $\vec{T}$, where $\vec{x}; \vec{y} : \vec{T}$ means that the concatenation of $\vec{x}$ and $\vec{y}$ inhabits $\vec{T}$.

Let us also introduce a notation for making multiple copies of a telescope.

DEFINITION: **iterated sequence or telescope**

If $\vec{t}$ is a sequence of terms or a telescope, then

$$\left\{\vec{t}\right\}^n$$

is the sequential composition of n copies of $\vec{t}$.

If $\vec{t}_i$ is a sequence of terms or telescope containing a free subscript i, then

$$\left\{\vec{t}_i\right\}_i^n$$

is the sequential composition

$$\vec{t}_1; \ldots \vec{t}_n$$

The empty sequence or telescope is thus $\{\}^0$.

Hence we may say that plus has type $\{\mathbb{N}\}^2 \to \mathbb{N}$ and still intend the curried form of the function.

Observe that if $\vec{t}_1, \ldots \vec{t}_n : \vec{T}$ then $\left\{\vec{t}_i\right\}_i^n : \left\{\vec{T}\right\}^n$.

Similarly $\left\{f\,\vec{t}_i\right\}_i^n$ abbreviates $(f\,\vec{t}_1); \ldots (f\,\vec{t}_n)$.

Now that we have the telescope notation for expressing types of indices, we may define the notion of an indexed family.

DEFINITION: **indexed family**

> If $\vec{S}$ is a telescope and T is a type, then an **$\vec{S}$-indexed T-family** is an inhabitant of $\forall \vec{x} : \vec{S}.\ T$.

For example, if for all $n : \mathbb{N}$ we define $\mathsf{fin}\, n$ to be the finite datatype with n elements, we may say that fin is $\mathbb{N}$-indexed $\mathcal{Type}$-family. Or, perhaps perversely, we may describe the function which decides equality on the natural numbers as a $\{\mathbb{N}\}^2$-indexed $\mathbf{2}$-family.

For any $\mathcal{Type}$-family (henceforth 'type family'), we may define the following telescope:

DEFINITION: **free telescope for a type family**

> If $\vec{T}$ is an $\vec{x}$-telescope and A is a $\vec{T}$-indexed type family then, $\overline{A}$, the **free telescope** for A, is
>
> $$\vec{T}; (A\ \vec{x})$$

For example, $\overline{\mathsf{fin}}$ is just $n : \mathbb{N}; x : \mathsf{fin}\ n$.

What is visible through this telescope? Every member of the family A, of course! That is, if $a : A\ \vec{t}$, then $\vec{t}; a : \overline{A}$. Note also that $\overline{A}\ \vec{t}$ is the same as one element telescope $A\ \vec{t}$!

Finally, let us consider how to abstract over arbitrary telescopes. Simply taking

$$\forall \vec{T} : \left\{\mathcal{Type}\right\}^n.\ \ldots$$

does not capture the potential for type dependency within the telescope: T_2 may depend on a value of type T_1 and so on. We may represent this by taking $T_2 : T_1 \to \mathcal{Type}$. We then have not a telescope of types, but a telescope of type families:

$$\begin{array}{l}
T_1 : \mathcal{Type}; \\
T_2 : T_1 \to \mathcal{Type}; \\
T_3 : \forall t_1 : T_1.\ (T_2\ t_1) \to \mathcal{Type}; \\
\vdots \\
T_n : \forall \vec{t} : \left\{T_i\ \{t_j\}_j^i\right\}_i^{n-1}.\ \mathcal{Type}
\end{array}$$

This is a very special $\vec{T}$-telescope which I call $\triangle^n \mathcal{Type}$, and any sequence which inhabits it is a **triangle** of length n. That is, a triangle is a sequence which represents a telescope.

It is not hard to convert an $\vec{x}$-telescope $\vec{T}$ into a triangle: we simply turn the abstractions implicit in the telescope notation into λ-bindings which capture the earlier x's in later T's. The resulting triangle is

$$\begin{array}{l} T_1; \\ \lambda x_1 \!:\! T_1.\ T_2; \\ \lambda x_1 \!:\! T_1.\ \lambda x_2 \!:\! T_2.\ T_3; \\ \vdots \\ \lambda \vec{x} \!:\! \{T_i\}_i^{n-1}.\ T_n \end{array}$$

Correspondingly, if $\vec{S}$ is a triangle, the $\vec{x}$-telescope it represents is

$$\begin{array}{l} x_1 : S_1; \\ x_2 : S_2\ x_1; \\ x_3 : S_3\ x_1\ x_2; \\ \vdots \\ x_n : S_n\ \{x_i\}_i^{n-1} \end{array}$$

There is no ambiguity between triangles and the telescopes they represent. You can easily spot which is which by which side of the colon they appear. I shall happily write

$\forall \vec{T} \!:\! \triangle^n \mathit{Type}.$	here $\vec{T}$ is a triangle
$\forall \vec{t} \!:\! \vec{T}.$	here $\vec{T}$ is the represented telescope

What could $\vec{T}$ mean as a triangle in a type position? Its elements are type families and, apart from the first unindexed one, these type families are not types.

Observe that if $T; \vec{S}$ is the $n+1$ length triangle representing telescope $T; \vec{T}$ and $t : T$, then the triangle representing $(T; \vec{T})\ t$ is $\{S_i\ t\}_i^n$. Telescope application is thus represented in the triangle coding by function applications.

These notational forms give us the syntactic power to manipulate dependent type families and their inhabitants cleanly and with hardly any more effort than for simple types. Since dependent type families feature strongly in this thesis, we are sure to be glad of the convenience.

Chapter 3

Elimination Rules for Refinement Proof

Introduction rules tell us how to establish new information. Elimination rules tell us how to exploit what we know. This chapter identifies a particularly useful class of elimination rule and develops a tactic to deploy them in refinement proof.

My first encounter with things described as 'elimination rules' was when I was being taught natural deduction as an undergraduate mathematician. In particular, I learned elimination rules for the propositional connectives $\wedge$ and $\vee$:

$$\frac{\mathrm{P} \wedge \mathrm{Q}}{\mathrm{P}} \qquad \frac{\mathrm{P} \wedge \mathrm{Q}}{\mathrm{Q}} \qquad \frac{\mathrm{P} \vee \mathrm{Q} \quad \begin{array}{c}[\mathrm{P}] \\ \vdots \\ \Phi\end{array} \quad \begin{array}{c}[\mathrm{Q}] \\ \vdots \\ \Phi\end{array}}{\Phi}$$

I recall thinking the two $\wedge$-elim rules uncontroversial, whilst being somewhat confused by the convoluted behaviour of $\vee$-elim. It was only when I caught my supervisor building a proof from the bottom of the blackboard upwards that I began to see the point. $\vee$-elim tells us how to exploit a disjunctive hypothesis to gain leverage on whatever Φ it is we are trying to prove. The $\wedge$-elim rules seem somewhat undermotivated by comparison—they project out one or other conjunct, so we have to arrange to want the conjuncts.

We can reformulate the $\wedge$-elim rules as a single rule in the style of $\vee$-elim:

$$\frac{P \wedge Q \qquad \begin{array}{c}[P]\\ [Q]\\ \vdots\\ \Phi\end{array}}{\Phi}$$

This rule (often called ‘uncurrying’) makes explicit the ‘see if you can prove it from the conjuncts’ technique which the original pair of rules tacitly require the reasoner to apply. And that is the key point. Elimination rules should supply a proof technique which analyses the hypothesis in question to give leverage on whatever the objective may be. The ‘projective’ rules only manage to be both applicable and motivated if we are lucky enough to be trying to prove one or other of the projections.

MANTRA:

> The end motivates the means.

It is only because $\wedge$ is a pretty boring connective—there is no choice about how to prove $P \wedge Q$—that we can get away with projective elimination rules. A disjunctive hypothesis yields no definite conclusion, so forward synthesis is blocked—we have to work analytically, reasoning by cases.

$\vee$-elim helps us prove Φ from $P \vee Q$ by splitting the task into two subtasks, decomposing the hypothesis. However, this is not the only way a well-designed elimination rule can make analytical progress. We can also decompose the objective (or ‘goal’) into more specific cases, our favourite example being the ‘principle of mathematical induction’:

$$\frac{\Phi\ 0 \qquad \begin{array}{c}[\Phi\ n]\\ \vdots\\ \Phi\ \mathsf{s}n\end{array}}{\forall n\!:\!\mathbb{N}.\ \Phi\ n}$$

This rule explains how to prove an arbitrary goal Φ indexed by a natural number n: we must show that proofs of Φ are made the same way that numbers are. The subgoals instantiate the index with more specific natural numbers. This instantiation may provide us with the concrete data we need to perform some computation or simplification, and this is, by and large, how inductive proofs work.

Henceforth, I shall intend by ‘elimination rule’ only this kind of rule whose conclusion is an arbitrary goal, possibly abstracted over indices. This characterisation is very broad, including rules where there is nothing being eliminated. This may seem odd, but it is sometimes useful to characterise what progress we can make towards an arbitrary goal without exploiting any *further* information. Good examples to bear in mind are the impredicative encodings of the true proposition and the absurd proposition, respectively $\forall \Phi : \mathcal{P}rop.\ \Phi \rightarrow \Phi$ and $\forall \Phi : \mathcal{P}rop.\ \Phi$. The former exploits no information in the cause of proving its arbitrary Φ, and consequently exerts no leverage, leaving Φ as a subgoal. The latter is only derivable in the context of a contradiction, and it indicates that we already have all we need to establish whatever Φ we want.

In order to exploit elimination rules whose conclusion is abstracted over indices, we need to make the corresponding abstractions from the goal we are trying to prove. It is, of course, obvious how to do this when the goal already looks like $\forall n : \mathbb{N}.\ \Phi\ n$. This chapter is largely devoted to explaining how to make the abstractions under less obvious circumstances.

3.1 propositional equality (definition deferred)

One of the tools we shall shortly require is a *propositional* notion of equality. The conventional formulations become awkward once type dependency enters the picture. The trouble is that two instances of a type family with indices which are not convertible, just propositionally equal, are not the same type. The familiar definitions permit only equations within one type—they forbid us even from stating the equality of elements drawn from the two instances of the family.

Huet and Saïbi encounter a similar problem in their formalisation of category theory [SH95]—they need to state the equality of arrows whose domains are not necesssarily computationally equal. Their solution is to relax the formulation rule for equations on arrows whilst still supplying only the reflexive constructor. With care, this approach may be extended to the commonplace propositional equality, and that is what I propose to do.

Rather than presenting my definition at this stage, with slender motivation and less context, I shall defer the treatment until we have more idea of what its properties should be, and more language with which to describe them.

Since I do not use a familiar equality, I shall not presume to use the familiar '$=$' symbol. Instead I shall write '$\simeq$'. Experienced readers who dislike suspense will find its definition in chapter 5. Otherwise, read on here—let us look out for the required behaviour of $\simeq$ as we go.

3.2 anatomy of an elimination rule

Let us first establish notation for elimination rules and give names to their components. Presenting elimination rules as raw types, or even in the conventional natural deduction style is relatively uninformative, as I have found in the past to my cost. In this section, I shall motivate what I hope is a clearer presentation (arising from a blackboard conversation with Rod Burstall). It is important that we come to some systematic understanding of these rules, for we shall need to teach machines to use them.

In order to make sense of any elimination rule, we need to know

- what it eliminates—its **target**
- what family of arbitrary goals it proves—its **scheme**

For example, mathematical induction (right) eliminates a natural number, n, and proves goals of the form $\Phi\ n$, where Φ is a family of propositions (ie, a predicate) over $\mathbb{N}$.

$\mathbb{N}\mathsf{Induction}$

$\Phi : \forall n : \mathbb{N}.\ \mathcal{P}rop$

$$\frac{\Phi\ 0 \qquad \begin{array}{c}\Phi\ n\\ \cdots\cdots \\ \Phi\ \mathsf{s}n\end{array}}{\forall \boxed{n : \mathbb{N}}.\ \Phi\ n}$$

I mark the target with a box. We can tell Φ is the scheme because it stands at the head of the rule's return type.

If we want to apply this rule, the target marker tells us that we must select a natural number to eliminate, which will stand in the place of n. Having done so, we will need to abstract it from the goal to make an appropriate scheme for Φ. It is important to type the scheme prominently. The index types are not always so obvious as here. Further, we may need to be precise about which type universe the goal must inhabit: the '$\mathcal{P}rop$' in the above rule makes it suitable only for propositional goals—this rule cannot be used for programming.

Schemes always have types of form $\forall \vec{\imath} : \vec{I}.\ \mathcal{U}$. I call the $\vec{\imath}$ the rule's **indices**, and the indexed telescope $\vec{I}$ the rule's **aperture**. Later we shall see elimination rules for the same thing, but with different apertures. We shall also see how to change the aperture of a rule. In conventional proofs by mathematical induction, the scheme is often called the 'induction predicate'. However, we shall have need of schemes which are not predicates and rules which are not inductive.

Wherever Φ is applied, its arguments are called **patterns**. The universally quantified variables appearing in patterns are **pattern variables**. Target selection must instantiate all the pattern variables in the conclusion of the goal—otherwise we will not know what to abstract to build the scheme. For mathematical induction, the only pattern variable involved is the target itself, so this requirement is clearly fulfilled.

Above the solid line are the rule's **cases**, each of which proves Φ applied to some **case patterns** (such as $\mathsf{0}$ and $\mathsf{s}n$ above). Any subgoal-specific assumptions appear above a dotted line, the horizontal cousin of natural deduction's vertical ellipsis. Those which do not involve Φ are **case data**. Those which do are described as **inductive hypotheses** or **recursive calls**.

The visual aspect of this presentation is intended to convey the idea that the cases of an elimination rule are the ghosts of the corresponding introduction rules. Prawitz's 'inversion principle' captures this relationship between the introduction and elimination rules of natural deduction [Pra65]—he attributes the idea to Gentzen who in [Gen35] expresses the property as follows:

> In eliminating a symbol, we may use the formula with whose terminal symbol we are dealing only 'in the sense afforded it by the introduction of that symbol'.

In essence, elimination rules show us how to mimic the structure of the hypotheses on which they act. Mathematical induction shows us to how to make $\Phi\ n$ imitate $n : \mathbb{N}$. I will freely suppress implicit assumptions (such as the $n : \mathbb{N}$ in the 'successor' subgoal above) in order to strengthen this resemblance.

MANTRA:

> Decomposition is the exposition of construction.

Before I describe how to work with elimination rules in more detail, let me place the discussion in context by exhibiting a number of variations on the theme.

3.3 examples of elimination rules

Parameterised data structures like lists have parameterised elimination rules.

In particular, we say that an elimination rule's **parameters** are those hypotheses on which the scheme's and cases' types depend. They may, where interesting, be listed at the top of the rule.

listElim

$A : \mathit{Type}$
$\Phi : (\mathsf{list}\, A) \to \mathit{Type}$

$$\frac{\Phi\ (\mathsf{nil}\, A) \qquad \begin{array}{c} h : A \quad \Phi\ t \\ \cdots\cdots\cdots \\ \Phi\ (\mathsf{cons}\, h\ t) \end{array}}{\forall \boxed{l : \mathsf{list}\, A}.\ \Phi\ l}$$

Note that I supply the case datum $h : A$ explicitly, despite its appearance in the cons case pattern, in order to emphasise the imitation of the constructor.

A class of elimination rule which we will construct and use over and over again in this thesis is the **case analysis** or **inversion** principle. For any notion given by introduction rules, the corresponding inversion principle asserts that those introduction rules are *exhaustive*. There is one case for each introduction rule, and there are no inductive hypotheses.[1]

Consider, by way of example, $\leq$ for $\mathbb{N}$, presented here in its 'suffix' variant.

$$\frac{}{m{\leq}m} \qquad \frac{m{\leq}n}{m{\leq}\mathsf{s}n}$$

The traditional 'Clark completion' [Cla78] presentation represents the choice of derivations as a disjunction of existentially quantified equations.

$$\begin{array}{l} \forall m, n.\ m{\leq}n \to \\ \qquad \bigvee \begin{array}{l} \exists m'.\quad\ \ m{\simeq}m' \wedge n{\simeq}m' \\ \exists m', n'.\ m{\simeq}m' \wedge n{\simeq}\mathsf{s}n' \wedge m'{\leq}n' \end{array} \end{array}$$

There is one disjunct for each introduction rule—the schematic variables become existentially quantified over equations demanding that the conclusion proves the inverted hypothesis and that the premises hold. This construction is somewhat mechanical, in that it explicitly constrains each argument of the hypothesis even if the constraint is redundant, like the $\exists m' \ldots m{\simeq}m'$ in each case.

In [McB96], I gave a standardised 'elimination rule' presentation of inversion, essentially currying the Clark completion. For example, the generic class of hypothesis $m{\leq}n$ would be inverted thus:

[1] In fact, it is good to think of induction as inversion augmented with recursive information.

$$\begin{array}{l}
\leq\mathsf{ClarkInv} \\[1ex]
\Phi : \mathit{Prop} \\[2ex]
\dfrac{\begin{array}{c} m{\simeq}m' \quad n{\simeq}m' \\ \vdots \\ \Phi \end{array} \qquad \begin{array}{c} m{\simeq}m' \quad n{\simeq}\mathsf{s}n' \quad m'{\leq}n' \\ \vdots \\ \Phi \end{array}}{\boxed{m{\leq}n} \to \Phi}
\end{array}$$

m and n are parametric to the whole rule. Once they have been instantiated, the equations in the subgoals may be simplified automatically. This approach is somewhat clumsy, but it is very easy to apply, as the scheme Φ may be any proposition—no abstraction is necessary. We shall shortly develop the abstraction technology required to exploit a more streamlined version, with an indexed scheme removing the need for equational constraints on the parameters:

$$\begin{array}{l}
\leq\mathsf{Inv} \\[1ex]
\Phi : \forall m, n : \mathbb{N}.\ \mathit{Prop} \\[2ex]
\dfrac{\Phi\ m\ m \qquad \begin{array}{c} m{\leq}n \\ \vdots \\ \Phi\ m\ \mathsf{s}n \end{array}}{\forall m, n.\ \boxed{m{\leq}n} \to \Phi\ m\ n}
\end{array}$$

This inversion principle differs from the Clark rule only in its aperture. They are, of course, interderivable, suggesting that there might be a systematic way to change the aperture of an elimination rule. In fact, that is the essence of the tactic this chapter develops.

The process which simplifies the constraints arising from inversion makes critical use of the fact that constructors are injective and disjoint (the 'no confusion' property). For natural numbers, we might plausibly choose to derive two of Peano's postulates:

- $\forall m, n : \mathbb{N}.\ \mathsf{s}m{\simeq}\mathsf{s}n \to m{\simeq}n$
- $\forall n : \mathbb{N}.\ 0 {\not\simeq} \mathsf{s}n$

The above formulation of injectivity is essentially projective after the fashion of the awkward $\wedge$-elim rules—directly useful only if it is $m{\simeq}n$ we are trying to prove. For non-unary constructors, cons for example, the problem gets worse—we either have separate head and tail injectivity theorems, or a single result which yields a tuple of equations which we then eliminate.

Consequently, I present injectivity as an inversion rule for an equation of successors. This is really just the 'tuple' version in curried form—the 'predecessor' equations are the hypotheses of the rule's only case.

$$\begin{array}{l} \mathsf{sInjective} \\ \\ \Phi : \mathcal{P}rop \\ \\ \dfrac{\begin{array}{c} \mathsf{m} \simeq \mathsf{n} \\ \vdots \\ \Phi \end{array}}{\boxed{\mathsf{sm} \simeq \mathsf{sn}} \to \Phi} \end{array}$$

Turning to the 'constructors disjoint' result, if we think of 'not' as 'implies false' and 'false' as the absurd proposition 'anything is true', we discover that we had an elimination rule all along, with a fortunate number of cases.

$$\begin{array}{l} \mathsf{0\_not\_s} \\ \\ \Phi : \mathcal{P}rop \\ \\ \dfrac{}{\boxed{\mathsf{0} \simeq \mathsf{sn}} \to \Phi} \end{array}$$

I shall show how to prove rules like these in chapter 5.

We should not think of elimination rules as solely belonging to datatypes and relations. They also provide neat tools for reasoning about functions. After all, what is the extension of a function, but a relation on which a total and deterministic computational mode has been imposed.

An equational presentation of a function corresponds to a set of introduction rules, with recursive calls becoming inductive premises. It makes sense to reason about the behaviour of the function by the corresponding elimination rule.

Consider $\mathbb{N}\mathsf{Eq}$—the function which decides the equality of two natural numbers. Later, we shall see how to define it by recursive pattern matching equations as shown.

$$\begin{array}{lll} \mathbb{N}\mathsf{Eq}\ \mathsf{0}\ \ \mathsf{0} & = & \mathsf{true} \\ \mathbb{N}\mathsf{Eq}\ \mathsf{s}m\ \mathsf{0} & = & \mathsf{false} \\ \mathbb{N}\mathsf{Eq}\ \mathsf{0}\ \ \mathsf{s}n & = & \mathsf{false} \\ \mathbb{N}\mathsf{Eq}\ \mathsf{s}m\ \mathsf{s}n & = & \mathbb{N}\mathsf{Eq}\ m\ m \end{array}$$

The corresponding elimination rule allows us to do what John McCarthy calls **recursion induction** [McC67], effectively packaging up the recursive structure of $\mathbb{N}\mathsf{Eq}$ as a single induction principle.

$$\begin{array}{l} \mathbb{N}\mathsf{EqRecI} \\ \\ \Phi : \forall m, n : \mathbb{N}.\ \forall \boxed{b} : \mathbf{2}.\ \mathcal{P}rop \\ \\ \dfrac{\Phi\ \mathsf{0}\ \mathsf{0}\ \mathsf{true} \quad \Phi\ \mathsf{0}\ \mathsf{s}n\ \mathsf{false} \quad \Phi\ \mathsf{s}m\ \mathsf{0}\ \mathsf{false} \quad \begin{array}{c} \Phi\ m\ n\ b \\ \vdots \\ \Phi\ \mathsf{s}m\ \mathsf{s}n\ b \end{array}}{\forall m, n.\ \Phi\ m\ n\ \boxed{\mathbb{N}\mathsf{Eq}\ m\ n}} \end{array}$$

Many proofs about functions operate by choosing the right combination of inductions and case analyses on the arguments to make the computation unfold. Recursion induction on functions does away with the apparent cunning of this choice by wrapping up 'the right combination' in a derived rule which targets *applications* of the function directly. The proof of a recursion induction principle follows the construction of the function it describes, step by step.

In order to make proper use of such a recursion induction principle, or any other rule eliminating a function application, we must choose a scheme Φ which abstracts that application from the goal. Each subgoal thus replaces the application by the appropriate value.

Such abstractions are usually unnecessary when eliminating datatypes or relations. However, exactly when and where this abstraction behaviour is required seems to vary from rule to rule, and even from problem to problem—I cannot see how to infer it reliably from the structure of the rule or its target.

The user must be free to indicate which arguments are to be abstracted in any given case—I put a box in the type of the scheme around any index for which abstraction is to be attempted. When b is boxed in $\mathbb{N}\mathsf{EqRecI}$, it indicates that we would like to abstract occurrences of $(\mathbb{N}\mathsf{Eq}\; m\; n)$ as b.

For many functions, typically of a 'searching' or 'testing' character, recursion induction is still too close to the implementation to be really useful. For example, regardless of how the test works, we should like to know that $\mathbb{N}\mathsf{Eq}$ returns true for equal and false for unequal arguments. We can represent these requirements as 'extensional' introduction rules, via the propositional equality:

$$\frac{}{\mathbb{N}\mathsf{Eq}\; x\; x \simeq \mathsf{true}} \qquad \frac{x \not\simeq y}{\mathbb{N}\mathsf{Eq}\; x\; y \simeq \mathsf{false}}$$

These equations may not be computational, but we can still use them for conditional rewriting, should we be lucky enough to encounter applications of $\mathbb{N}\mathsf{Eq}$ which look like the left hand sides.

We are often less lucky. Imagine we are trying to prove a property of a program

$$\forall x, y.\; P\; (\text{if } \boxed{\mathbb{N}\mathsf{Eq}\; x\; y} \text{ then } S \text{ else } T)$$

The computation is blocked at the box, because the 'if' will only reduce given a boolean value, and inside the box because x and y are not numerals. Neither rewrite rule applies, because we do not know whether or not x and y are equal. We can remove the blockage if we split the problem into the two cases where the $\mathbb{N}\mathsf{Eq}$ call returns true and false respectively.

This is exactly the behaviour of the inversion principle corresponding to the rewrite rules.

Inverting a $\mathbb{N}\mathsf{Eq}$ call yields two cases: one where the arguments are the same and the result is true, the other where the arguments differ and false is returned.

$\mathbb{N}\mathsf{EqInv}$

$$\Phi : \forall m, n : \mathbb{N}.\ \forall \boxed{b} : \mathbf{2}.\ \mathit{Prop}$$

$$\dfrac{\Phi\ n\ n\ \mathsf{true} \qquad \begin{array}{c} m \not\simeq n \\ \vdots \\ \Phi\ m\ n\ \mathsf{false} \end{array}}{\forall m, n.\ \Phi\ m\ n\ \boxed{\mathbb{N}\mathsf{Eq}\ m\ n}}$$

Again, boxing $\boxed{b}$ indicates that $(\mathbb{N}\mathsf{Eq}\, m\, n)$ is to be abstracted from the scheme. Consequently, it is replaced in one subgoal by true and in the other by false. In both cases, the 'if' reduces—further, in the true case, x and y are coalesced:

- $\forall x.\ [x/y](P\ S)$
- $\forall x, y.\ x \not\simeq y\ \rightarrow\ P\ T$

Inversion requires much less effort than extracting the same information from 'characterisation theorems' like the following (from the LEGO library):

$$\forall m, n : \mathbb{N}.\ m \simeq n \leftrightarrow \mathbb{N}\mathsf{Eq}\ m\ n \simeq \mathsf{true}$$

To achieve the effect of the inversion, you need to combine this lemma with projection from the '$\leftrightarrow$', boolean case analysis and a rewriting mechanism.

MANTRA:

> Invert the blocking computation.

The point is simple. Introduction rules construct information. Elimination rules exploit information. It is a serious weakness to confuse these purposes. In my view, an equational specification is the wrong tool to exploit the properties of one program in a proof about another. By construction, elimination rules, especially those which invert blocked computations, are much better tools for that purpose. Over the course of this thesis, you will see this point reinforced in example after example.

3.4 legitimate targets

In order to refine a goal by an elimination rule, we must do two things:

- select a target of the kind the rule eliminates
- construct a suitable scheme from the goal

I shall discuss the latter in the next section, but the first issue requires comment now, because it impacts on how we should present elimination rules in the first place.

The point is that, in order to be able to select a target, we must know what kind of target the rule eliminates. We must define what it means to be a 'legitimate target' of a rule, so that when we tell the machine which rule we want to use, it can tell us what we may use it on.

As we have just seen, there are many different kinds of elimination rule, eliminating many different kinds of target. The elimination rule for a datatype eliminates an arbitrary element of that type, abstracted in the rule and appearing in the concluding pattern:

$$\forall \boxed{n : \mathbb{N}}. \Phi\ n$$

Inverting an inductively defined relation like $\leq$ eliminates hypothetical inhabitants of the relation, but the pattern ($\Phi\ m\ n$) only involves the relation's indices (m and n), not the target (the proof of $m{\leq}n$) itself:

$$\forall m, n. \boxed{m{\leq}n} \to \Phi\ m\ n$$

An elimination rule for a function specifically eliminates applications of that function, rather than arbitrary elements of the result type, so the target appears only in the patterns.

$$\forall m, n. \Phi\ m\ n\ \boxed{\mathbb{N}\mathsf{Eq}\ m\ n}$$

More diverse variations include 'double induction', where we must provide two targets for a nested analysis. There is no way we can expect a machine to cope with this diversity, looking only at a type and trying to second-guess the intention behind it.

Let us place the burden of specifying what an elimination rule targets where it belongs—with the manufacturer of the rule. In the Northern Irish tradition, a legitimate target is whatever we say it is.

Consequently, the boxes around targets become more than a notational courtesy between you and me—they are annotations which the machine can also see. One way to represent such annotations is to store the boxed term and type in a fatuous !-binding with a special identifier, '$\Box$', below:

- $\forall n\!:\!\mathbb{N}.\ !\Box = n\!:\!\mathbb{N}.\ \Phi\, n$
- $\forall m, n.\ \forall H\!:\!m{\le}n.\ !\Box = H\!:\!m{\le}n.\ \Phi\, m\, n$
- $\forall m, n.\ !\Box = \mathbb{N}\mathsf{Eq}\, m\, n\!:\!\mathbf{2}.\ \Phi\, m\, n\, (\mathbb{N}\mathsf{Eq}\, m\, n)$

Given the 'manufacturer's instructions', the machine can ask us for legitimate targets in the order that the annotations appear in the type. When we indicate what to eliminate, a process known in the business as **fingering**, the machine can match it against the target annotation, inferring the universally quantified variables therein.

This opens the interesting possibility that the type of an elimination rule might be computed from its targets. After all, we cannot compute the elimination scheme until we know what it is we intend to eliminate. We will see an example of this technique later—the 'injectivity' and 'conflict' rules for a given datatype will be combined into a single rule which computes the inversion appropriate to the equation being eliminated once targetting has instantiated the two sides with constructor expressions. This is not a caprice on my part—it really is the easy way to prove the Peano-style properties of dependent datatypes.

3.5 scheming with constraints

> '*You can have any color you like, as long as it's black.*' (Henry Ford)

Undergraduates should count themselves fortunate that the exercises in inductive proof with which they are traditionally presented involve goals of form:

> 'For all $n \in \mathbb{N}$, rhubarb rhubarb n.'

The formulation of the 'base' and 'step' cases then involves mindless copying of the 'rhubarb' bit, with appropriate values substituted for the 'n'. Even if they cannot complete the question, they can still manufacture the proof template (once any tendency to write 'suppose $n = k$, show $n = k + 1$' has been beaten out of them, that is) and thus collect some credit.[2]

[2]For such purposes 'rhubarb rhubarb' makes as worthy a predicate as any.

When reasoning about even modestly complex notions, such as $\leq$ for $\mathbb{N}$, we are less likely to be favoured by goals bearing so close a resemblence to an elimination rule conclusion, such as that of $\leq\mathsf{Inv}$:

$$\leq\mathsf{Inv} : \ldots \forall m, n.\ \boxed{m\leq n} \ \rightarrow\ \Phi\ m\ n$$

$\leq\mathsf{Inv}$'s scheme abstracts over arbitrary pairs of natural numbers, but how are we to deal with less arbitrary pairs? How can we cope with particular restrictions of relations, datatypes and so forth? How might we apply a generic rule like $\leq\mathsf{Inv}$ to a more restricted instance of $\leq$? Consider the boxed hypothesis in

$$?\mathsf{0}\mathit{least} : \forall x.\ \boxed{x\leq\mathsf{0}} \ \rightarrow\ x\simeq\mathsf{0}$$

We need to construct a scheme which is constrained according to the problem in hand, but still abstracted over the entire aperture of the rule. The constraint we need can be expressed by means of propositional equality, taking

$$\Phi \ =\ \lambda x, n.\ n\simeq\mathsf{0} \ \rightarrow\ x\simeq\mathsf{0}$$

As it were, 'you can have any $n : \mathbb{N}$ you like, as long as it's $\mathsf{0}$'.

Plugging in this scheme, the conclusion of $\leq\mathsf{Inv}$ becomes

$$\forall m, n.\ \boxed{m\leq n} \ \rightarrow\ n\simeq\mathsf{0} \ \rightarrow\ m\simeq\mathsf{0}$$

Now, if we fill in the details of our selected target, $x\leq\mathsf{0}$, this is further instantiated to

$$\mathsf{0}\simeq\mathsf{0} \ \rightarrow\ x\simeq\mathsf{0}$$

and we can surely prove $\mathsf{0}\simeq\mathsf{0}$—let us presume there is some

$$\mathsf{refl} : \forall A : \mathcal{T}\!ype.\ \forall a : A.\ a\simeq a$$

More generally, suppose we have an elimination rule proving some scheme Φ for patterns $\vec{p}[\vec{y}]$, as shown to the right. The notation $\vec{p}[\vec{y}]$ represents the sequence of patterns with the pattern variables abstracted: more generally, $\vec{p}[\vec{t}]$ means 'the patterns with t's substituted for the y's'.

$$\boxed{\begin{array}{l} \Phi : \forall\vec{\imath}.\ \mathcal{U} \\ \\ \dfrac{\mathit{rule\ subgoals}}{\forall\vec{y}.\ \Phi\ \vec{p}[\vec{y}]} \end{array}}$$

We may apply this rule to a more specific goal—let us presume that targetting has produced a matching σ giving the rule's pattern variables in terms of the goal's hypotheses.

That is, consider a goal which looks like

$$\forall \vec{x}.\ \Psi[\vec{p}[\sigma\vec{y}]]$$

We may choose a scheme Φ with explicit equational constraints:

$$\lambda\vec{\imath}.\ \forall \vec{x}.\ \vec{\imath}{\simeq}\vec{p}[\sigma\vec{y}] \to \Psi[\vec{p}[\sigma\vec{y}]]$$

What is $\vec{\imath}{\simeq}\vec{p}[\sigma\vec{y}]$? It is a **telescopic equation**: in general, if $\vec{s}$ and $\vec{t}$ are sequences of length n, then the telescopic equation $\vec{s}{\simeq}\vec{t}$ abbreviates the telescope of equations:

$$\{s_j{\simeq}t_j\}_j^n$$

Observe, though, that we must be able to express these constraints even in the presence of type dependency. For example, if we were building constraints on an aperture

$$n : \mathbb{N}; v : \mathsf{vect}\,n$$

we might need something like

$$n_1{\simeq}n_2; v_1{\simeq}v_2$$

even though $v_1 : \mathsf{vect}\,n_1$ and $v_2 : \mathsf{vect}\,n_2$. That is, we need a notion of equality which scales to telescopes—exactly what $\simeq$ will provide.

Let us instantiate the rule's conclusion, filling in the pattern variables according to σ and Φ with the scheme we have constructed:

$$\forall \vec{x}.\ \vec{p}[\sigma\vec{y}]{\simeq}\vec{p}[\sigma\vec{y}] \to \Psi[\vec{p}[\sigma\vec{y}]]$$

If we can solve the equations, we will recover the target goal. Fortunately, they are reflexive.

The point is this: in much the same way that Henry Ford's customers could ask for any colour of Model T, but would only receive satisfaction if they happened to choose black, the above scheme is indeed abstracted over the entire aperture, but the patterns to which it applies are subject to equational constraints which recover their specificity.

Notice that the formulation of this scheme requires no abstraction. The $\Psi[\vec{p}[\sigma\vec{y}]]$ remains untouched. It is targetting which identifies the $\vec{p}[\sigma\vec{y}]$—they need not even occur in the goal, although the exercise is perhaps a little pointless if they do not.

We have established the basic technique for constructing schemes when our goal is more specific than the conclusion of the elimination rule. It is broadly effective, but it sometimes generates redundant information. For example, constraints are unnecessary wherever the goal really is as general as the rule—there is no point in saying 'you can have any color you like, as long as it's a color'. We should try to avoid equations where abstraction will do.

The next three subsections describe techniques to make the basic scheme less clumsy, in accordance with the following three observations:

- wherever a fresh variable is constrained to equal an index, we can coalesce the two and remove the constraint
- we can avoid abstracting the scheme over redundant information
- if an index is constrained to equal a complex pattern (for example, when we apply an elimination rule characterising a function) we may sometimes simplify the scheme by replacing copies of the pattern with the index

3.5.1 simplification by coalescence

The simpler the example, the more unnecessary constraints there are likely to be: if we wanted to prove

$$\forall n : \mathbb{N}.\ \text{rhubarb rhubarb } n$$

the generic constrained scheme would be

$$\lambda m : \mathbb{N}.\ \forall n : \mathbb{N}.\ m \simeq n \ \rightarrow \text{rhubarb rhubarb } n$$

This is not the scheme which I want my students to write down, so it had better not be the scheme which the machine computes. Wherever a scheme constrains a λ-bound index to a equal fresh $\forall$-bound variable of the same type, we may coalesce the two. Our example becomes

$$\lambda n : \mathbb{N}.\ \text{rhubarb rhubarb } n$$

as we might hope for.

When we coalesce two variables, we have a choice of which name to keep—it is polite to preserve the name from the goal. Note that if the same $\forall$-bound variable is constrained to equal more than one index, that effectively forces those indices to be the same—we may only make one coalescence, otherwise we lose this 'diagonalisation'.

3.5.2 what to fix, what to abstract

Which of the goal's premises do we really want the scheme to abstract? Which should remain fixed over the whole scope of the elimination? Unfortunately, these can be quite subtle questions. Imagine, for example, that we are building the map function for polymorphic lists:

$$\mathsf{map} \ : \ \forall S, T\!:\!\mathit{Type}.\ \forall f\!:\!S \to T.\ \forall x\!:\!\mathsf{list}\ S.\ \mathsf{list}\ T$$

In order to do recursion on x we must certainly fix S—the element type is parametric to the elimination rule for list. We may fix T and f or not as we please.

On the other hand, when we are constructing functions which require nested recursion, we may not be so free to fix arguments. Consider, for example, Ackermann's function:

$$\begin{array}{l}
\mathsf{ack} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
\mathsf{ack}\ 0\ n \ = \ \mathsf{s}n \\
\mathsf{ack}\ \mathsf{s}m\ 0 \ = \ \mathsf{ack}\ m\ \mathsf{s}0 \\
\mathsf{ack}\ \mathsf{s}m\ \mathsf{s}n \ = \ \mathsf{ack}\ m\ (\mathsf{ack}\ \mathsf{s}m\ n)
\end{array}$$

When we apply the outer recursion on the first argument, we must not fix the second argument—as you can see, the recursive calls which decrease the former also vary the latter.

Abstracting wherever we are not forced to fix sounds like a promising policy—it does not hurt us to have too much flexibility, only too little.

However, sometimes abstraction is definitely redundant. Recall our earlier example, proving

$$\forall x.\ \boxed{x \le 0} \to x \simeq 0$$

perhaps by $\le\mathsf{Inv}$ (shown to the right).

$\le\mathsf{Inv}$

$\Phi : \forall m, n\!:\!\mathbb{N}.\ \mathit{Prop}$

$$\dfrac{\Phi\ m\ m \qquad \begin{array}{c} m \le n \\ \vdots \\ \Phi\ m\ \mathsf{s}n \end{array}}{\forall m, n.\ \boxed{m \le n} \to \Phi\ m\ n}$$

As things stand, the basic scheme abstracts all the premises

$$\Phi \;=\; \lambda m, n.\, \forall x.\, x{\leq}\mathsf{0} \;\rightarrow\; m{\simeq}x \;\rightarrow\; n{\simeq}\mathsf{0} \;\rightarrow\; x{\simeq}\mathsf{0}$$

Coalescence removes $\forall x$ (and renames m):

$$\Phi \;=\; \lambda x, n.\, x{\leq}\mathsf{0} \;\rightarrow\; n{\simeq}\mathsf{0} \;\rightarrow\; x{\simeq}\mathsf{0}$$

Plugging this into the conclusion of the rule, we find we have a proof that

$$\forall x, n.\, x{\leq}n \;\rightarrow\; x{\leq}\mathsf{0} \;\rightarrow\; n{\simeq}\mathsf{0} \;\rightarrow\; x{\simeq}\mathsf{0}$$

The extra inequality, $x{\leq}\mathsf{0}$, is redundant. It is present because we have abstracted over what we were eliminating, but it is not in any way useful because the scheme is not indexed over the proof of the inequality.

Typically, once targetting has filled in what is being eliminated, the application of an elimination rule looks like

$$\mathit{rule}\, \Phi\, \vec{\phi}\, \sigma\vec{y} \;:\; \Phi\, \vec{p}[\sigma\vec{y}]$$

The premises occurring in the inferred arguments $\sigma\vec{y}$ are the ones being eliminated. However, some y's may not appear in the patterns, so some eliminated premises may not appear in the instantiated patterns $\vec{p}[\sigma\vec{y}]$. The elimination thus tells us nothing about them, so we may omit them from the scheme provided type dependency permits.

That is, we may omit a premise x on grounds of redundancy provided

- x occurs in the arguments of the elimination rule inferred by targetting
- x does not occur in the instantiated patterns $\vec{p}[\sigma\vec{y}]$
- the remainder of the goal does not depend on x

Inductive relations like $\leq$ are usually formulated in exactly this 'proof irrelevant' way. In our example, the eliminated hypothesis $x{\leq}\mathsf{0}$ satisfies the three conditions. We omit it, leaving

$$\Phi \;=\; \lambda x, n.\, n{\simeq}\mathsf{0} \;\rightarrow\; x{\simeq}\mathsf{0}$$

This is the scheme we want.

3.5.3 abstracting patterns from the goal

Rules with indices marked for abstraction oblige us to carry out further simplification on the scheme, in order that they have the intended 'rewriting' effect.

Recall $\mathbb{N}\mathsf{EqInv}$ from section 3.3—we might use this rule to rewrite an application of $\mathbb{N}\mathsf{Eq}$ in a goal like the following:

$\mathbb{N}\mathsf{EqInv}$

$$\Phi : \forall m, n : \mathbb{N}.\ \forall \boxed{b} : \mathbf{2}.\ \mathit{Prop}$$

$$\frac{\Phi\ n\ n\ \mathsf{true} \qquad \begin{array}{c} m \not\simeq n \\ \vdots \\ \Phi\ m\ n\ \mathsf{false} \end{array}}{\forall m, n.\ \Phi\ m\ n\ \boxed{\mathbb{N}\mathsf{Eq}\ m\ n}}$$

$$\forall x, y.\ \Psi[\text{if } \boxed{\mathbb{N}\mathsf{Eq}\ x\ y} \text{ then } s \text{ else } t]$$

Targetting infers $[x/m][y/n]$. The coalesced scheme is thus

$$\Phi \ = \ \lambda x, y, \boxed{b}.\ b \simeq \mathbb{N}\mathsf{Eq}\ x\ y \ \rightarrow \ \Psi[\text{if } \boxed{\mathbb{N}\mathsf{Eq}\ x\ y} \text{ then } s \text{ else } t]$$

The boxed $\boxed{b}$ tells us that we should abstract away occurrences of $(\mathbb{N}\mathsf{Eq}\ x\ y)$ from the goal. Once we have done this, we can throw the constraint away.

$$\Phi \ = \ \lambda x, y, b.\ \Psi[\text{if } b \text{ then } s \text{ else } t]$$

Abstracting arbitrary terms in dependent type theory is a sensitive business—we are not always free to replace a given subterm by a variable of the same type, because the typing of the whole term may depend on the particular intensional properties of the subterm being replaced. However, it is worth a try—if unsuccessful, we may leave the constraint as it is and continue.

This rewriting technique is very powerful. The trouble caused by the intensionality of the type theory is a real pity. Perhaps a part of the problem could be avoided with appropriate facilities for reconstructing broken typings from propositional equalities, as proposed by Hofmann [Hof95].

3.5.4 constraints in inductive proofs

Let us see how constrained schemes affect inductive proofs. We will acquire constraints on the inductive hypotheses, as well as those on the conclusions of the subgoals.

$$\Phi : \forall m, n{:}\mathbb{N}.\ \mathcal{P}rop$$

$$\frac{\Phi\ m\ m \qquad \dfrac{\Phi\ m\ n}{\Phi\ m\ \mathsf{s}n}}{\forall m, n.\ \boxed{m{\le}n} \to \Phi\ m\ n}$$

Consider applying the weak induction principle[3] for $\le$ (see right) in a proof of

$$?strict \colon \forall x, y.\ \boxed{\mathsf{s}x{\le}y} \to x{<}y$$

Targetting gives $\sigma\ =\ [\mathsf{s}x/m][y/n]$, so we infer the scheme (coalescing y and n):

$$\Phi\ =\ \lambda m, y.\ \forall x.\ m{\simeq}\mathsf{s}x \to x{<}y$$

The corresponding subgoals are shown to the right. The constraints which appear as *hypotheses* in the subgoals, e_b and e_s, are 'friendly'—they restrict the m's and x's we have to deal with. The constraint in the inductive hypothesis, e_s, is 'unfriendly'—it restricts our choice of x'.

$$\begin{array}{l}
?base \colon \forall m, x \colon \mathbb{N} \\
\quad \forall e_b \ \ \colon m{\simeq}\mathsf{s}x \\
\quad\quad x{<}m \\
?step \colon \forall m, n \colon \mathbb{N} \\
\quad \forall hyp \colon \forall x' \colon \mathbb{N} \\
\quad\quad\quad \forall e_h \colon m{\simeq}\mathsf{s}x' \\
\quad\quad\quad\quad x'{<}n \\
\quad \forall x \ \ \colon \mathbb{N} \\
\quad \forall e_s \ \ \colon m{\simeq}\mathsf{s}x \\
\quad\quad x{<}\mathsf{s}n
\end{array}$$

A closer examination of these constraints reveals a more subtle but crucial distinction.

The variables appearing in these constraints come from two sources:

- the pattern variables for each case of the elimination rule, m and n above—these become premises of the subgoals, and appear on the *left*-hand side of constraints

- the variables universally quantified in the scheme, x and x' above—these become premises of the subgoals and also parameters of the inductive hypothesis: they appear on the *right*-hand side of constraints

The 'friendly' constraints tell us useful information about the variables which occur as subgoal premises, whether they come from the scheme or the patterns. In chapter

[3] An inductively defined relation like $\le$ also has a strong induction principle—the distinction is explained in section 4.1.5.

5, we will see how to simplify them, solving for variables appearing on *either* side—'friendly' constraints constitute *unification* problems.

In our example, let us imagine we can perform this simplification on e_b and e_s, instantiating the m's to leave the subgoals shown.

$$
\begin{array}{l}
?base' \colon \forall x \colon \mathbb{N} \\
\quad x < \mathsf{s}x \\
?step' \colon \forall x, n \colon \mathbb{N} \\
\quad \forall hyp \colon \forall x' \colon \mathbb{N} \\
\quad\quad \forall e_h \colon \mathsf{s}x \simeq \mathsf{s}x' \\
\quad\quad\quad x' < n \\
\quad x < \mathsf{s}n
\end{array}
$$

The 'unfriendly' constraints cannot tell us anything about the variables which occur as subgoal premises—e_h does not allow us to infer x. Rather, they narrow our choices for the copies of the scheme variables (like x') which parameterise inductive hypotheses. That is, 'unfriendly' constraints can only determine variables appearing on the *right*-hand side—they are *matching* problems.

Look back before we simplified the 'friendly' constraints: we cannot find an x' to solve the matching problem $m \simeq \mathsf{s}x'$. However, now that we have done the unification, a solution has become available. Inferring x for x' we can obtain the subgoals shown on the right.

$$
\begin{array}{l}
?base'' \colon \forall x \colon \mathbb{N} \\
\quad x < \mathsf{s}x \\
?step'' \colon \forall x, n \colon \mathbb{N} \\
\quad \forall hyp \colon x < n \\
\quad x < \mathsf{s}n
\end{array}
$$

Something interesting has happened, and we will see what it is if we present these subgoals in natural deduction style:

$$
base'' \quad \frac{}{x < \mathsf{s}x} \qquad\qquad step'' \quad \frac{x < n}{x < \mathsf{s}n}
$$

This looks like a plausible recursive specification of $<$! In fact, what we have done is apply the standard unfold/fold technique for logic programs [TS83, GS91] to transform our goal, viewed as a specification of $<$ in terms of $\leq$, into subgoals which give $<$ recursively. The unification problems in the conclusions are those which arise in unfolding; the matching problems in the inductive hypotheses are those involved in folding.

3.6 an elimination tactic

In this section, I shall present a tactic, **eliminate**, which refines a given goal by a given elimination rule—the user is required to finger the targets, then the tactic constructs an appropriate scheme and solves the goal, generating a subgoal for each case.

eliminate operates in five stages:

- preparing a proforma application of the elimination rule to arguments initially unknown
- fingering the targets and inferring the pattern variables
- constructing a constrained scheme
- proving the goal
- tidying up

I have implemented a prototype of this tactic with much of the functionality described here as a key component in my extension of LEGO. Of course, if I had known then what I know now, it would have all the functionality. This section is the blueprint for the revised version.

I shall present each stage as a little tactic. The $\leq$ induction we have just seen in the previous section makes a useful running example. The tactic should reproduce exactly the effect we manufactured by hand.

The rule we shall use and the goal we shall prove are shown in OLEG notation on the right. The boxed premise in the rule is the inequality it eliminates: the boxed inequality in the goal is the one we shall target.

$$
\begin{array}{lll}
\lambda{\leq}\mathsf{Elim}: & \forall\Phi & : \forall m, n : \mathbb{N} \\
 & & \quad \mathit{Prop} \\
 & \forall\phi_{\mathrm{m}} & : \forall m : \mathbb{N} \\
 & & \quad \Phi\ m\ m \\
 & \forall\phi_{\mathrm{s}} & : \forall m, n : \mathbb{N} \\
 & & \quad \forall\phi_{\mathrm{mn}} : \Phi\ m\ n \\
 & & \quad \Phi\ m\ \mathsf{s}n \\
 & \forall m, n : \mathbb{N} & \\
 & \forall \boxed{L\ :\ m \leq n} & \\
 & \Phi\ m\ n & \\
 & & \\
?\mathit{leGoal}: & \forall x, y : \mathbb{N} & \\
 & \forall \boxed{H\ :\ \mathsf{s}x \leq y} & \\
 & x < y &
\end{array}
$$

3.6.1 preparing the application

The preparation step could be carried out for any goal to be solved by any lemma. It is just an administrative manoeuvre, getting everything in the right place for the real work which follows.

TACTIC: **eliminate-prepare**

$$
\boxed{\begin{array}{l}
\mathit{elim}\colon \forall\vec{s}\colon\vec{S}.\ R[\vec{s}] \\
\\
?\mathit{goal}\colon \forall\vec{x}\colon \vec{X} \\
\qquad\qquad Y[\vec{x}]
\end{array}}
\Rightarrow
\boxed{\begin{array}{l}
\mathit{elim}:\ \forall\vec{s}\colon\vec{S}.\ R[\vec{s}] \\
\\
?\mathit{goal} \approx \lambda\vec{x} \quad : \vec{X} \\
\qquad\qquad ?\vec{s} \quad : \vec{S} \\
\qquad\qquad !\mathit{app} = \mathit{elim}\ \vec{s} \\
\qquad\qquad\qquad : R[\vec{s}] \\
\qquad\qquad ?\mathit{conc} : Y[\vec{x}] \\
\qquad\qquad\ \ \mathit{conc}
\end{array}}
$$

The goal's hypotheses are introduced; the lemma's hypotheses are inserted as unknowns. A 'proforma' application of the lemma is then manufactured and stored as a !-binding. Ultimately, this application will be used to solve *conc*. We must first fill in some of the $\vec{s}$.

The prepared application for our example is shown on the right.

$$
\boxed{\begin{array}{l}
?\mathit{leGoal} \approx \lambda x, y \ : \mathbb{N} \\
\qquad\qquad \lambda \boxed{H \ : \ \mathsf{s}x \le y} \\
\qquad\qquad ?\Phi \quad : \ \ldots \\
\qquad\qquad ?\phi_{\mathrm{m}} \quad : \ \ldots \\
\qquad\qquad ?\phi_{\mathrm{s}} \quad : \ \ldots \\
\qquad\qquad ?m, n : \mathbb{N} \\
\qquad\qquad ?\boxed{L \ : \ m \le n} \\
\qquad\qquad !\mathit{app} = \le\!\mathsf{Elim}\ \Phi \ldots L \\
\qquad\qquad\qquad : \Phi\ m\ n \\
\qquad\qquad ?\mathit{conc} : x < y \\
\qquad\qquad\ \ \mathit{conc}
\end{array}}
$$

3.6.2 fingering targets

Having installed an application of the rule in the proof of the goal, the next step is to infer some of its arguments by targetting. We may presume that the rule has a sequence of targets marked by its manufacturer. The user must now finger a sequence of matching expressions to be eliminated.

We may make use of the **unify** tactic to do our matching, although this may be a slight overkill. Something like the following happens:

$$
\boxed{\begin{array}{l}
?\mathit{goal} \approx \lambda\vec{x} \quad : \vec{X} \\
\qquad\qquad ?\vec{s}_1;\vec{s}_2 : \vec{S}_1;\vec{S}_2 \\
\qquad\qquad !\mathit{app} = \mathit{elim}\ \vec{s}_1;\vec{s}_2 \\
\qquad\qquad\qquad : R[\vec{s}_1;\vec{s}_2] \\
\qquad\qquad ?\mathit{conc} : Y[\vec{x}] \\
\qquad\qquad\ \ \mathit{conc} \\
\\
t[\vec{s}_2] : T[\vec{s}_2] \text{ target} \\
e[\vec{x}] : E[\vec{x}] \text{ to be eliminated}
\end{array}}
\Rightarrow
\boxed{\begin{array}{l}
?\mathit{goal} \approx \lambda\vec{x} \quad : \vec{X} \\
\qquad\qquad ?\vec{s}_1 \quad : \vec{S}_1 \\
\qquad\qquad !\vec{s}_2 \quad = \vec{r}[\vec{x}] \\
\qquad\qquad !\mathit{app} = \mathit{elim}\ \vec{s}_1;\vec{r}[\vec{x}] \\
\qquad\qquad\qquad : R[\vec{s}_1;\vec{r}[\vec{x}]] \\
\qquad\qquad ?\mathit{conc} : Y[\vec{x}] \\
\qquad\qquad\ \ \mathit{conc} \\
\\
T[\vec{r}[\vec{x}]] \ \cong \ E[\vec{x}] \\
t[\vec{r}[\vec{x}]] \ \cong \ e[\vec{x}]
\end{array}}
$$

That is, targetting tries to match terms and types. If successful, some of the rule's arguments $\vec{s}_2$ will be inferred as $\vec{r}[\vec{x}]$. Others, $\vec{s}_1$, will not be inferred. The two kinds do not have to be bound in separate clumps—it is just easier to write down that way.

If a rule has more than one target, we will have to repeat this step for each.

In our example, we successfully match H to L. Matching the types also infers m and n.

$$\begin{array}{l}
!m \;\; = \mathsf{s}\, x \\
!n \;\; = y \\
!L \;\; = H \\
!app = \leq\!\mathsf{Elim}\ \Phi \ldots H \\
\qquad : \ \Phi\ \mathsf{s}\, x\ y
\end{array}$$

Now, if an elimination rule is particularly complicated, its later structure may be computed from earlier arguments inferred by targetting. The instantiated type of *app* may reduce, revealing more premises to be inferred. The tactic should create holes for these and add them to the application. Computation may also reveal more targets. Incorporating this possibility, the real behaviour of the targetting step is as follows

TACTIC: **eliminate-target**

$$\boxed{\begin{array}{l}
?goal \approx \lambda\vec{x} \quad : \ \vec{X} \\
\qquad ?\vec{s}_1;\vec{s}_2 : \ \vec{S}_1;\vec{S}_2 \\
\qquad !app \ = elim\ \vec{s}_1;\vec{s}_2 \\
\qquad\qquad : \ R[\vec{s}_1;\vec{s}_2] \\
\qquad ?conc : \ Y[\vec{x}] \\
\qquad\ \ conc \\
\\
t[\vec{s}_2] : T[\vec{s}_2] \text{ target} \\
e[\vec{x}] : E[\vec{x}] \text{ to be eliminated}
\end{array}}
\Rightarrow
\boxed{\begin{array}{l}
?goal \approx \lambda\vec{x} \quad : \ \vec{X} \\
\qquad ?\vec{s}_1 \quad : \ \vec{S}_1 \\
\qquad !\vec{s}_2 \quad = \vec{r}[\vec{x}] \\
\qquad ?\vec{s}_3 \quad : \ \vec{S}_3 \\
\qquad !app' = elim\ \vec{s}_1;\vec{r}[\vec{x}];\vec{s}_3 \\
\qquad\qquad : \ R'[\vec{x};\vec{s}_1;\vec{s}_3] \\
\qquad ?conc : \ Y[\vec{x}] \\
\qquad\ \ conc \\
\\
T[\vec{r}[\vec{x}]] \ \cong \ E[\vec{x}] \\
t[\vec{r}[\vec{x}]] \ \cong \ e[\vec{x}] \\
R[\vec{s}_1;\vec{r}[\vec{x}]] \ \rhd \ \forall\vec{s}_3 : \vec{S}_3.\ R'[\vec{x};\vec{s}_1;\vec{s}_3]
\end{array}}$$

Observe that not only have the $\vec{s}_2$ been inferred and turned into !-bindings, but some $\vec{s}_3$ have appeared as a result of computation. The proforma application is extended accordingly.

3.6.3 constructing the scheme

$$
\begin{array}{lll}
?goal \approx \lambda\vec{a} & : & \vec{A} \\
\quad \lambda\vec{x} & : & \vec{X} \\
\quad ?\Phi & : & \forall\vec{\imath} : \vec{I}[\vec{a}] \\
 & & \quad \mathcal{U} \\
 & & \\
\quad !app & = & elim\ \vec{r} \\
 & : & \Phi\ \vec{p}[\vec{x}] \\
\quad ?conc & : & Y[\vec{x}] \\
\quad conc & &
\end{array}
$$

If the targetting phase has left the state as shown, the tactic may proceed to construct the elimination scheme. The scheme variable, Φ, has been uncovered and the patterns, $\vec{p}[\vec{x}]$, have been inferred. The task is now to compute Φ. We must put the analysis of section 3.5 into practice.

Recall that the basic scheme is manufactured by abstracting all the premises and constraining the indices to equal the instantiated patterns.

$$
\begin{array}{l}
?\Phi \approx \lambda\vec{\imath} : \vec{I}[\vec{a}] \\
\quad \forall\vec{x}' : \vec{X} \\
\quad \forall\vec{e} : \vec{\imath} \simeq \vec{p}[\vec{x}'] \\
\quad\quad Y[\vec{x}']
\end{array}
$$

Correspondingly, the tactic begins by building a basic scheme, copying the non-parametric premises $\vec{x}$ from the goal and constraining all the indices. A premise a is considered parametric exactly when it occurs in the type of Φ. The tactic may fail at this point if the goal being addressed is too 'big' for the universe over which the rule eliminates.

The remainder of this phase prunes the basic scheme down to something less clumsy, wherever this is possible. Of course, in a real implementation, we would try to save work by approaching the desired scheme more directly, but I suspect that 'pruning the basic scheme' gives a clearer exposition. There are two passes:

- For decreasing[4] j, remove $\forall x'_j$ from the scheme if it is redundant, ie

if	$x_j \in \vec{r}$	(x_j has been targetted ...)
and	$x_j \notin \vec{p}[\vec{x}]$	(... but is not being 'inspected' in the patterns ...)
and	$x_j \notin \vec{X}, Y[\vec{x}]$	(... or depended on by the rest of the goal)

- For increasing[5] k, try to simplify constraint $\forall e_k : i_k \simeq p_k[\vec{x}]$

  There are two simplifications to check for: in order,

  – **coalescence**

if	p_k is some x'_j	(index constrained to equal fresh variable ...)
and	$I_k \cong X_j$	(... of same type)
then	replace x'_j by i_k, remove $\forall x'_j$ from scheme	

  Strictly, we should then rename i_k to x'_j, keeping the name from the goal, but that would make this presentation more complex than it already is.

[4]Later redundant premises must not be used as excuses to retain earlier redundant premises.
[5]Simplifying earlier constraints may unify the types of later constraints.

– **abstraction for rewriting**

When i_k is marked for abstraction with $\forall \boxed{i_k}$ in the type of Φ, try replacing all occurrences of p_k in the scheme by i_k. If the result is well-typed, discard e_k, otherwise leave the scheme alone.

Once simplification is complete, the pruned scheme is made accessible by changing the $?\Phi$ to $!\Phi$. The type of *app* can then reduce.

In our example, the basic scheme is more complex than it needs to be. Reflecting the 'proof irrelevant' nature of inductive relations, the H' is redundant. Furthermore, we may remove e_n by coalescence.

$$\begin{array}{lll} ?\Phi \approx \lambda m, n : \mathbb{N} & & \\ \quad \forall x', y' : \mathbb{N} & & \\ \quad \forall H' & : \mathsf{s}x' \le y' & \\ \quad \forall e_m & : m \simeq \mathsf{s}x' & \\ \quad \forall e_n & : n \simeq y' & \\ \quad\; x' < y' & & \end{array}$$

The pruned scheme is exactly the one we came up with when we did this example by hand. The type of *app* reduces accordingly.

$$\begin{array}{l} !\Phi \;= \lambda m, y' : \mathbb{N} \\ \quad \forall x' \;\; : \mathbb{N} \\ \quad \forall e_m \; : m \simeq \mathsf{s}x' \\ \quad\; x' < y' \\[1ex] !app = \le\mathsf{Elim}\ \Phi \ldots H \\ \quad : \forall x' : \mathbb{N} \\ \quad\;\; \forall e_m : \mathsf{s}x \simeq \mathsf{s}x' \\ \quad\;\;\; x' < y \end{array}$$

I summarise the behaviour of this phase as a tactic step:

TACTIC: **eliminate-scheme**

$$\begin{array}{l} ?\Phi \;\; : \forall \vec{\imath} : \vec{I}[\vec{a}] \\ \qquad\quad \mathcal{U} \\[1ex] !app = elim\ \ldots \\ \qquad : \Phi\ \vec{p}[\vec{x}] \end{array} \quad \Rightarrow \quad \begin{array}{l} !\Phi \;= \lambda \vec{\imath} \;: \vec{I}[\vec{a}] \\ \quad \forall \vec{x}'_p : \vec{X}'_p \\ \quad \forall \vec{e}_p : \vec{\imath}_p \simeq \vec{p}_p[\vec{x}'_p] \\ \quad\;\; Y'[\vec{\imath}; \vec{x}'_p] \\[1ex] !app = elim\ \ldots \\ \quad : \forall \vec{x}'_p : \vec{X}'_p \\ \quad\;\; \forall \vec{e}_p : \vec{p}_p[\vec{x}_p] \simeq \vec{p}_p[\vec{x}'_p] \\ \quad\;\;\; Y'[\vec{p}[\vec{x}]; \vec{x}'_p] \end{array}$$

The $\vec{x}'_p$ are what remain of the $\vec{x}'$ after pruning—$\vec{x}_p$ is the corresponding selection from $\vec{x}$.

The $\vec{e}_p$ are what remain of the $\vec{e}$ after pruning, equating a pruned sequence of indices $\vec{\imath}_p$ to a pruned sequence of patterns $\vec{p}_p[\vec{x}'_p]$.

Recall that the conclusion we are trying to prove is $Y[\vec{x}]$: by construction, $Y'[\vec{p}[\vec{x}]; \vec{x}_p] \;\equiv\; Y[\vec{x}]$

3.6.4 proving the goal

TACTIC: **eliminate-goal**

$$\begin{array}{|l|}\hline
\begin{array}{lll}
!app & = & elim\ \vec{r} \\
 & : & \forall \vec{x}'_p \colon \vec{X}'_p \\
 & & \quad \forall \vec{e}_p : \vec{p}_p[\vec{x}_p] \simeq \vec{p}_p[\vec{x}'_p] \\
 & & \quad\quad Y'[\vec{p}[\vec{x}]; \vec{x}'_p] \\
?conc & : & Y[\vec{x}]
\end{array} \\ \hline
\end{array}
\Rightarrow
\begin{array}{|l|}\hline
\begin{array}{lll}
!conc & = & app\ \vec{x}_p\ (\mathsf{refl}\ \vec{p}_p[\vec{x}_p]) \\
 & : & Y[\vec{x}]
\end{array} \\ \hline
\end{array}$$

This phase proves *conc* from *app* by instantiating the premises abstracted in the scheme with their ‘originals’, making the constraints reflexive and the return type the desired $Y[\vec{x}]$.

The effect on our example is shown on the right.

$$\begin{array}{|l|}\hline
\begin{array}{lll}
!app & = & \leq\mathsf{Elim}\ \Phi \ldots H \\
 & : & \forall x' : \mathbb{N} \\
 & & \quad \forall e_m \colon \mathsf{s}x \simeq \mathsf{s}x' \\
 & & \quad\quad x' < y \\
!conc & = & app\ x\ (\mathsf{refl}\ \mathsf{s}x) \\
 & : & x < y
\end{array} \\ \hline
\end{array}$$

3.6.5 tidying up

TACTIC: **eliminate-tidy**

$$\begin{array}{|l|}\hline
\begin{array}{llll}
?goal \approx & \lambda \vec{x} & : & \vec{X} \\
 & !\Phi & = & \ldots \\
 & ?\vec{sub} & : & \vec{S} \\
 & !\vec{s} & = & \vec{r}[\vec{x}] \\
 & !app & = & elim\ \Phi\ \vec{sub}\ \vec{r}[\vec{x}] \\
 & !conc & = & app\ \vec{x}_p\ (\mathsf{refl}\ \vec{p}_p[\vec{x}_p]) \\
 & conc & &
\end{array} \\ \hline
\end{array}$$

$$\Rightarrow$$

$$\begin{array}{|l|}\hline
\begin{array}{lll}
?\vec{sub}' & : & \forall \vec{x}_d \colon \vec{X}_d .\ \vec{S} \\
!goal & = & \lambda \vec{x} \colon \vec{X} \\
 & & \quad elim \ldots (\vec{sub}'\ \vec{x}_d) \ldots
\end{array} \\ \hline
\end{array}$$

Each $sub_i{}'$ proves S_i generalised over the $\vec{x}_d$ it depends on.

Firstly this phase **cut**s the !-bindings for inferred arguments $\vec{s}$, and also Φ, *app* and *conc*.

The task is then to shuffle the subgoals—the rule arguments not inferred by targetting—outside the proof of *goal*. This is done by discharging the λs through them, so that they are generalised over only what their types depend on (as opposed to raising the ?s, which would generalise over everything regardless). Typically, this will re-abstract the fixed parameters.

Once the ?s are outside the λs, the **retreat** tactic moves them outside the binding of *goal*. At this point, smart implementations try to η-reduce the proof of *goal*. Finally, *goal* is **solve**d, becoming a !-binding.

In our example, the subgoals do not depend on any of the premises, so no generalisation is necessary. The final subgoals and proof term are as follows:

$$
\begin{array}{lll}
?sub_1 & : & \forall m, x'\colon \mathbb{N} \\
& & \forall e \quad : m \simeq \mathsf{s}x' \\
& & \quad x' < m \\
?sub_2 & : & \forall m, n\colon \mathbb{N} \\
& & \forall hyp : \forall x'\colon \mathbb{N} \\
& & \qquad\qquad \forall e : m \simeq \mathsf{s}x' \\
& & \qquad\qquad\quad x' < n \\
& & \forall x' \quad : \mathbb{N} \\
& & \forall e \quad : m \simeq \mathsf{s}x' \\
& & \quad x' < \mathsf{s}n \\
!leGoal = & & \lambda x, y\colon \mathbb{N} \\
& & \lambda H \;\; : \mathsf{s}x \leq y \\
& & \quad \leq\!\mathsf{Elim}\ (\lambda m, y'.\ \forall x'.\ m \simeq \mathsf{s}x' \to x' < y') \\
& & \qquad sub_1\ sub_2\ \mathsf{s}x\ y\ H \\
& & \qquad x\ (\mathsf{refl}\ \mathsf{s}x) \\
& : & \forall x, y\colon \mathbb{N} \\
& & \forall H \;\; : \mathsf{s}x \leq y \\
& & \quad x < y
\end{array}
$$

3.7 an example—$\mathbb{N}$Eq

We have built our hammer—let us bang in a few nails. I propose to synthesise the $\mathbb{N}\mathsf{Eq}$ function described earlier in the chapter, and to prove some useful theorems about it. We will make use of the **eliminate** tactic for both programming and proof.

$\mathbb{N}\mathsf{Eq}$ is a recursive function on $\mathbb{N}$, so the starting point for the development will be $\mathbb{N}$'s elimination rule, $\mathbb{N}\mathsf{Elim}$, which doubles as the traditional induction principle and its primitive recursion operator.

$\mathbb{N}\mathsf{Elim}$

$\Phi : \forall n\!:\!\mathbb{N}.\ \mathit{Type}$

$$\frac{\Phi\ \mathsf{0} \qquad \begin{array}{c}\Phi\ n\\ \vdots\\ \Phi\ (\mathsf{s}n)\end{array}}{\forall \boxed{n : \mathbb{N}}.\ \Phi\ n}$$

The sequence of work is then as follows:

- Use $\mathbb{N}\mathsf{Elim}$ to build an implementation of $\mathbb{N}\mathsf{Eq}$ corresponding to the obvious functional program.

$$\begin{array}{l}
\mathbb{N}\mathsf{Eq}\ \ \mathsf{0}\ \ \ \mathsf{0}\ = \mathsf{true}\\
\mathbb{N}\mathsf{Eq}\ \mathsf{s}m\ \ \mathsf{0}\ = \mathsf{false}\\
\mathbb{N}\mathsf{Eq}\ \ \mathsf{0}\ \ \mathsf{s}n = \mathsf{false}\\
\mathbb{N}\mathsf{Eq}\ \mathsf{s}m\ \mathsf{s}n = \mathbb{N}\mathsf{Eq}\ m\ m
\end{array}$$

- Use $\mathbb{N}\mathsf{Elim}$ again to prove $\mathbb{N}\mathsf{Eq}$'s recursion induction principle:

$\mathbb{N}\mathsf{EqRecI}$

$\Phi : \forall m, n\!:\!\mathbb{N}.\ \forall \boxed{b}\!:\!\mathbf{2}.\ \mathit{Prop}$

$$\frac{\Phi\ \mathsf{0}\ \mathsf{0}\ \mathsf{true} \qquad \Phi\ \mathsf{0}\ \mathsf{s}n\ \mathsf{false} \qquad \Phi\ \mathsf{s}m\ \mathsf{0}\ \mathsf{false} \qquad \begin{array}{c}\Phi\ m\ n\ b\\ \vdots\\ \Phi\ \mathsf{s}m\ \mathsf{s}n\ b\end{array}}{\forall m, n.\ \Phi\ m\ n\ \boxed{\mathbb{N}\mathsf{Eq}\ m\ n}}$$

- Use $\mathbb{N}\mathsf{EqRecI}$ to prove a more convenient elimination rule for $\mathbb{N}\mathsf{Eq}$—the inversion principle suggested earlier in the chapter.

$\mathbb{N}\mathsf{EqInv}$

$\Phi : \forall m, n\!:\!\mathbb{N}.\ \forall \boxed{b}\!:\!\mathbf{2}.\ \mathit{Prop}$

$$\frac{\Phi\ n\ n\ \mathsf{true} \qquad \begin{array}{c}m \not\simeq n\\ \vdots\\ \Phi\ m\ n\ \mathsf{false}\end{array}}{\forall m, n.\ \Phi\ m\ n\ \boxed{\mathbb{N}\mathsf{Eq}\ m\ n}}$$

- Use $\mathbb{N}\mathsf{EqInv}$ to show that $\mathbb{N}\mathsf{Eq}$ satisfies its equational specification, given here as 'introduction rules'.

$$\frac{}{\mathbb{N}\mathsf{Eq}\ x\ x \simeq \mathsf{true}} \qquad \frac{x \not\simeq y}{\mathbb{N}\mathsf{Eq}\ x\ y \simeq \mathsf{false}}$$

3.7.1 constructing $\mathbb{N}\mathsf{Eq}$

Let us implement $\mathbb{N}\mathsf{Eq}$ by a nested recursion, on the first argument and then the second.

PROGRAM: $\mathbb{N}\mathsf{Eq}$

$$
\begin{array}{l}
!\mathbb{N}\mathsf{EQ} = \lambda m, n : \mathbb{N}.\ \mathbf{2} \\
?\mathbb{N}\mathsf{Eq}\ :\ \forall m, n : \mathbb{N}.\ \mathbb{N}\mathsf{EQ}\ \boxed{m}\ n \\
\text{satisfying} \\
\mathbb{N}\mathsf{Eq}\ \ 0\ \ \ 0 = \mathsf{true} \\
\mathbb{N}\mathsf{Eq}\ \mathsf{s}m\ \ 0 = \mathsf{false} \\
\mathbb{N}\mathsf{Eq}\ \ 0\ \ \mathsf{s}n = \mathsf{false} \\
\mathbb{N}\mathsf{Eq}\ \mathsf{s}m\ \mathsf{s}n = \mathbb{N}\mathsf{Eq}\ m\ m
\end{array}
$$

DEVELOPMENT

The above goal is shown with a box around our first target. Note the !-binding which replaces the return type of $\mathbb{N}\mathsf{Eq}$ with a more informative alias.

See how the return type of $\mathbb{N}\mathsf{Eq}$ looks a bit like the left-hand side of a pattern matching definition? We can find our target m there. **eliminate** it with $\mathbb{N}\mathsf{Elim}$!

$$
\boxed{
\begin{array}{lll}
?\mathbb{N}\mathsf{Eq}_0 & : & \forall n : \mathbb{N} \\
 & & \quad \mathbb{N}\mathsf{EQ}\ 0\ \boxed{n} \\
?\mathbb{N}\mathsf{Eq}_s & : & \forall m : \mathbb{N} \\
 & & \forall rec : \forall n : \mathbb{N} \\
 & & \qquad\quad \mathbb{N}\mathsf{EQ}\ m\ n \\
 & & \forall n : \mathbb{N} \\
 & & \quad \mathbb{N}\mathsf{EQ}\ \mathsf{s}m\ n \\
!\mathbb{N}\mathsf{Eq} & = & \mathbb{N}\mathsf{Elim}\ (\lambda m.\ \forall n.\ \mathbb{N}\mathsf{EQ}\ m\ n) \\
 & & \qquad \mathbb{N}\mathsf{Eq}_0\ \mathbb{N}\mathsf{Eq}_s
\end{array}
}
$$

We now have a base case and a step case. Note the way the return types have picked up the patterns corresponding to the case analysis.

In the base case, we are ready to **eliminate** the second argument, n, again with $\mathbb{N}\mathsf{Elim}$.

$$
\boxed{
\begin{array}{lll}
?\mathbb{N}\mathsf{Eq}_{00} & : & \mathbb{N}\mathsf{EQ}\ 0\ 0 \\
?\mathbb{N}\mathsf{Eq}_{0s} & : & \forall n : \mathbb{N} \\
 & & \forall rec : \mathbb{N}\mathsf{EQ}\ 0\ n \\
 & & \quad \mathbb{N}\mathsf{EQ}\ 0\ \mathsf{s}n \\
!\mathbb{N}\mathsf{Eq}_0 & = & \mathbb{N}\mathsf{Elim}\ (\lambda n.\ \mathbb{N}\mathsf{EQ}\ 0\ n) \\
 & & \qquad \mathbb{N}\mathsf{Eq}_{00}\ \mathbb{N}\mathsf{Eq}_{0s}
\end{array}
}
$$

We can now 'fill in the right-hand sides' by introducing the premises, then refining by true for $\mathbb{N}\mathsf{Eq}_{00}$ and false for $\mathbb{N}\mathsf{Eq}_{0s}$.

$$
\boxed{
\begin{array}{l}
?\mathbb{N}\mathsf{Eq}_s \approx \lambda m \quad : \mathbb{N} \\
\qquad\qquad \lambda rec \ : \forall n : \mathbb{N} \\
\qquad\qquad\qquad\qquad \mathbb{N}\mathsf{EQ}\ m\ n \\
\qquad\qquad ?\mathbb{N}\mathsf{Eq}_s : \forall n : \mathbb{N} \\
\qquad\qquad\qquad\qquad \mathbb{N}\mathsf{EQ}\ \mathsf{s}m\ \boxed{n} \\
\qquad\quad \mathbb{N}\mathsf{Eq}_s
\end{array}
}
$$

The step case is kept neat by introducing m and its associated recursive call before eliminating n with $\mathbb{N}\mathsf{Elim}$. Note that the type of the recursive call tells us which argument patterns it is good for.

We solve $\mathbb{N}\mathsf{Eq}_{s0}$ with false. For $\mathbb{N}\mathsf{Eq}_{ss}$, we introduce the premises and refine by the recursive call $rec\ n$.

$$
\begin{array}{l}
?\mathbb{N}\mathsf{Eq}_{s0} : \ \mathbb{N}\mathsf{EQ}\ \mathsf{s}m\ 0 \\
?\mathbb{N}\mathsf{Eq}_{ss} : \ \forall n \quad : \mathbb{N} \\
\qquad\qquad \forall rec' : \mathbb{N}\mathsf{EQ}\ \mathsf{s}m\ n \\
\qquad\qquad\ \ \mathbb{N}\mathsf{EQ}\ \mathsf{s}m\ \mathsf{s}n \\
!\mathbb{N}\mathsf{Eq}_{s} = \mathbb{N}\mathsf{Elim}\ (\lambda n.\ \mathbb{N}\mathsf{EQ}\ \mathsf{s}m\ n) \\
\qquad\qquad\quad \mathbb{N}\mathsf{Eq}_{s0}\ \mathbb{N}\mathsf{Eq}_{ss}
\end{array}
$$

□

We have built our first function with **eliminate**!

3.7.2 proving $\mathbb{N}$EqRecI

There is a standard technique for proving the recursion induction principle for a function. We fix an arbitrary scheme Φ indexed by the function's arguments and result type. We also assume that Φ is preserved by each 'introduction rule', ie recursive equation. We then prove that Φ holds for any arguments and the corresponding result—this proof has exactly the same recursive structure as the function itself. Discharging the fixed assumptions will give us the general rule.

THEOREM: **$\mathbb{N}$EqRecI**

$$
\begin{array}{ll}
\lambda\Phi & : \forall m, n : \mathbb{N}.\ \forall b : \mathbf{2}.\ \mathit{Type} \\
\lambda\phi_{00} & : \Phi\ 0\ 0\ \mathsf{true} \\
\lambda\phi_{0s} & : \forall n : \mathbb{N}.\ \Phi\ 0\ \mathsf{s}n\ \mathsf{false} \\
\lambda\phi_{s0} & : \forall m : \mathbb{N}.\ \Phi\ \mathsf{s}m\ 0\ \mathsf{false} \\
\lambda\phi_{ss} & : \forall m, n : \mathbb{N} \\
 & \ \ \forall b \quad : \mathbf{2} \\
 & \ \ \forall hyp : \Phi\ m\ n\ b \\
 & \quad \Phi\ \mathsf{s}m\ \mathsf{s}n\ b \\
?\mathbb{N}\mathsf{EqRecI} & : \forall m, n : \mathbb{N} \\
 & \quad \Phi\ m\ n\ (\mathbb{N}\mathsf{Eq}\ \boxed{m}\ n)
\end{array}
$$

PROOF

For our $\mathbb{N}\mathsf{Eq}$ example, we fix Φ and assume it is preserved by each of the four equations.

We are left proving $\Phi\ m\ n\ (\mathbb{N}\mathsf{Eq}\ m\ n)$ for any m and n, where before we computed $\mathbb{N}\mathsf{EQ}\ m\ n$. We eliminate with $\mathbb{N}\mathsf{Elim}$ in exactly the same places.

$$
\begin{array}{ll}
\lambda\Phi & : \forall m,n:\mathbb{N}.\ \forall b:\mathbf{2}.\ \mathit{Type} \\
\lambda\phi_{00} & : \Phi\ 0\ 0\ \mathsf{true} \\
\lambda\phi_{0s} & : \forall n:\mathbb{N}.\ \Phi\ 0\ \mathsf{s}n\ \mathsf{false} \\
\lambda\phi_{s0} & : \forall m:\mathbb{N}.\ \Phi\ \mathsf{s}m\ 0\ \mathsf{false} \\
\lambda\phi_{ss} & : \forall m,n:\mathbb{N} \\
 & \quad \forall b\ \ : \mathbf{2} \\
 & \quad \forall \mathit{hyp} : \Phi\ m\ n\ b \\
 & \qquad \Phi\ \mathsf{s}m\ \mathsf{s}n\ b \\
?\mathbb{N}\mathsf{EqRecI} & : \forall m,n:\mathbb{N} \\
 & \qquad \Phi\ m\ n\ (\mathbb{N}\mathsf{Eq}\ \boxed{m}\ n)
\end{array}
$$

I will show one base case and the step case.

Once elimination has instantiated the arguments of $\mathbb{N}\mathsf{Eq}$ appropriately, it reduces in each subgoal, making them vulnerable to the assumptions constructed with exactly that purpose. The base cases follow directly.

$$
\begin{array}{ll}
?\mathbb{N}\mathsf{EqRecI}_{00} & : \Phi\ 0\ 0\ (\mathbb{N}\mathsf{Eq}\ 0\ 0) \\
 & \qquad \rhd\ \Phi\ 0\ 0\ \mathsf{true}
\end{array}
$$

Similarly, the conclusion of the step case reduces to the conclusion of the relevant assumption, ϕ_{ss}, with b suitably instantiated.

rec n computed the recursive call in the construction of the function. Here, *rec n* fills in the premise of ϕ_{ss} to complete the proof.

$$
\begin{array}{ll}
\lambda \mathit{rec} & : \forall n:\mathbb{N} \\
 & \quad \Phi\ m\ n\ (\mathbb{N}\mathsf{Eq}\ m\ n) \\
?\mathbb{N}\mathsf{EqRecI}_{ss} & : \forall n\ \ :\mathbb{N} \\
 & \quad \forall \mathit{rec}' : \Phi\ \mathsf{s}m\ n\ (\mathbb{N}\mathsf{Eq}\ \mathsf{s}m\ n) \\
 & \qquad \Phi\ \mathsf{s}m\ \mathsf{s}n\ (\mathbb{N}\mathsf{Eq}\ \mathsf{s}m\ \mathsf{s}n) \\
 & \qquad \rhd\ \Phi\ \mathsf{s}m\ \mathsf{s}n\ (\mathbb{N}\mathsf{Eq}\ m\ n)
\end{array}
$$

Discharging the subgoals proves the general rule we want. □

Let us mark $\mathbb{N}\mathsf{EqRecI}$ as targetting $(\mathbb{N}\mathsf{Eq}\ m\ n)$, and by default abstracting it.

This proof method gives a recursion induction principle for many of the functions we can build in OLEG—it mimics exactly their construction. In effect, it packages up the sequence of eliminations which made the function, so that they can be used at one stroke in proofs of its properties.

3.7.3 proving $\mathbb{N}$EqInv

The proof of $\mathbb{N}$EqInv by $\mathbb{N}$EqRecI is a good example of deriving an inversion principle from a recursion induction principle. It illustrates a technique which I shall use relentlessly in similar circumstances for the rest of this thesis.

The proof of recursion induction principles is relatively simple. They directly describe the computational behaviour of the function in question, so we should not be surprised to find that the computational mechanism of the underlying calculus does all the hard work. Recall that in each subgoal of the inductive proof, the conclusion *reduces* to exactly what is proven by the corresponding premise.

Contrarily, inversion principles often cut against the computational grain, characterising the extensional properties of functions, rather than the mechanism by which they operate. The key to proving them is not to fix their schemes as λ-bindings outside the induction, but rather to let them vary *inside* the induction. This means that the inductive hypotheses are themselves inversion principles—we use inversion, not computation, to simplify the inductive steps.

THEOREM: **$\mathbb{N}$EqInv**

$$
\begin{array}{lll}
?\mathbb{N}\mathsf{EqInv}\colon & \forall\Phi & : \forall m, n : \mathbb{N}.\ \forall b : \mathbf{2}.\ \mathit{Type} \\
& \forall\phi_t & : \forall m : \mathbb{N}.\ \Phi\ m\ m\ \mathsf{true} \\
& \forall\phi_f & : \forall m, n : \mathbb{N} \\
& & \quad \forall uneq\colon m \not\simeq n \\
& & \qquad \Phi\ m\ n\ \mathsf{false} \\
& \multicolumn{2}{l}{\forall m, n\colon \mathbb{N}} \\
& \multicolumn{2}{l}{\quad \Phi\ m\ n\ \boxed{\mathbb{N}\mathsf{Eq}\ m\ n}}
\end{array}
$$

PROOF

We fix nothing in the context and eliminate by $\mathbb{N}$EqRecI, abstracting ($\mathbb{N}$Eq m n).

The scheme generated by **eliminate** is abstracted over the scheme of the rule we are trying to prove.

Observe that the original ($\mathbb{N}$Eq m n) in the conclusion has been replaced by b.

$$
\boxed{
\begin{array}{ll}
\multicolumn{2}{l}{\lambda m, n\colon \mathbb{N}} \\
\lambda b & : \mathbf{2} \\
\forall\Phi & : \forall m, n : \mathbb{N}.\ \forall b : \mathbf{2}.\ \mathit{Type} \\
\forall\phi_t & : \forall m : \mathbb{N}.\ \Phi\ m\ m\ \mathsf{true} \\
\forall\phi_f & : \forall m, n : \mathbb{N} \\
& \quad \forall uneq\colon m \not\simeq n \\
& \qquad \Phi\ m\ n\ \mathsf{false} \\
\multicolumn{2}{l}{\quad \Phi\ m\ n\ b}
\end{array}
}
$$

The recursion induction gives us directly the three base cases and the step case. Again, one base case is sufficently representative.

In this off-diagonal case, the recursion induction has already filled in the answer false. Hence, introducing the premises and refining by ϕ_f, we are left proving $0 \not\simeq \mathsf{s}n$. This is not difficult, as we shall see in chapter five.

$$
\begin{array}{l}
?\mathbb{N}\mathsf{EqInv}_{0s} : \forall n : \mathbb{N} \\
\quad \forall \Phi : \forall m, n : \mathbb{N}.\ \forall b : \mathbf{2}.\ \mathit{Type} \\
\quad \forall \phi_t : \forall m : \mathbb{N}.\ \Phi\ m\ m\ \mathsf{true} \\
\quad \forall \phi_f : \forall m, n : \mathbb{N} \\
\qquad \forall \mathit{uneq} : m \not\simeq n \\
\qquad\ \ \Phi\ m\ n\ \mathsf{false} \\
\quad \Phi\ 0\ \mathsf{s}n\ \mathsf{false}
\end{array}
$$

The step case is more entertaining. We do not know whether to use ϕ_t or ϕ_f, because we do not yet know what b is. However, the inductive hypothesis is an elimination rule telling us about m, n and b. I have called the scheme Ψ to reduce confusion.

$$
\begin{array}{lll}
?\mathbb{N}\mathsf{EqInv}_{ss} \approx & \lambda m, n & : \mathbb{N} \\
& \lambda b & : \mathbf{2} \\
& \lambda \mathit{hyp} & : \forall \Psi : \forall \boxed{m}, \boxed{n} : \mathbb{N}.\ \forall \boxed{b} : \mathbf{2}.\ \mathit{Type} \\
& & \quad \forall \psi_t : \forall m : \mathbb{N}.\ \Psi\ m\ m\ \mathsf{true} \\
& & \quad \forall \psi_f : \forall m, n : \mathbb{N} \\
& & \qquad \forall \mathit{uneq} : m \not\simeq n \\
& & \qquad\ \ \Psi\ m\ n\ \mathsf{false} \\
& & \quad \Psi\ m\ n\ b \\
& \lambda \Phi & : \forall m, n : \mathbb{N}.\ \forall \boxed{b} : \mathbf{2}.\ \mathit{Type} \\
& \lambda \phi_t & : \forall m : \mathbb{N}.\ \Phi\ m\ m\ \mathsf{true} \\
& \lambda \phi_f & : \forall m, n : \mathbb{N} \\
& & \quad \forall \mathit{uneq} : m \not\simeq n \\
& & \quad \Phi\ m\ n\ \mathsf{false} \\
& ?\mathbb{N}\mathsf{EqInv}_{ss} : \Phi\ \mathsf{s}m\ \mathsf{s}n\ b & \\
& \quad \mathbb{N}\mathsf{EqInv}_{ss} &
\end{array}
$$

Introducing everything, we may now **eliminate** the conclusion with *hyp*, abstracting all the indices. No targetting is necessary as the patterns are fully instantiated. The generated scheme abstracts m, n and b:

$$\Psi \ = \ \lambda m, n, b.\ \Phi\ \mathsf{s}m\ \mathsf{s}n\ b$$

We are left with two subgoals, each with the equality decided:

$$
\begin{array}{l}
?\mathbb{N}\mathsf{EqInv}_{sst} : \forall m : \mathbb{N}.\ \Phi\ \mathsf{s}m\ \mathsf{s}m\ \mathsf{true} \\
?\mathbb{N}\mathsf{EqInv}_{ssf} : \forall m, n : \mathbb{N} \\
\qquad\qquad \forall \mathit{uneq} : m \not\simeq n \\
\qquad\qquad\ \ \Phi\ \mathsf{s}m\ \mathsf{s}n\ \mathsf{false}
\end{array}
$$

These follow respectively from ϕ_t and ϕ_f without much difficulty, completing the proof. □

As with $\mathbb{N}$EqRecI, mark $\mathbb{N}$EqInv as targetting ($\mathbb{N}$Eq $m\ n$) and by default abstracting it.

3.7.4 proving the 'introduction rules'

THEOREM: **$\mathbb{N}$Eqtrue**

$$
\begin{array}{ll}
?\mathbb{N}\mathsf{Eqtrue}\colon & \forall m\colon \mathbb{N} \\
 & \boxed{\mathbb{N}\mathsf{Eq}\ m\ m} \simeq \mathsf{true}
\end{array}
$$

PROOF

Eliminating with $\mathbb{N}$EqInv introduces a constraint because the target is diagonalised:

$$\Phi \ = \ \lambda m, n\colon\mathbb{N}.\ \lambda b\colon\mathbf{2}.\ m\simeq n \ \rightarrow b\simeq\mathsf{true}$$

Both subgoals are easy.

$$
\begin{array}{ll}
?\mathbb{N}\mathsf{Eqtrue}_t\colon & \forall m\colon \mathbb{N} \\
 & \forall e\ \colon m\simeq m \\
 & \ \mathsf{true} \simeq \mathsf{true} \\
?\mathbb{N}\mathsf{Eqtrue}_f\colon & \forall m, n\colon \mathbb{N} \\
 & \forall uneq\colon m \not\simeq n \\
 & \forall e\quad\ \colon m\simeq n \\
 & \ \mathsf{false} \simeq \mathsf{true}
\end{array}
$$

□

THEOREM: **$\mathbb{N}$Eqfalse**

$$
\begin{array}{ll}
?\mathbb{N}\mathsf{Eqfalse}\colon & \forall m, n\colon \mathbb{N} \\
 & \forall uneq\colon m \not\simeq n \\
 & \boxed{\mathbb{N}\mathsf{Eq}\ m\ n} \simeq \mathsf{false}
\end{array}
$$

PROOF

Eliminating with $\mathbb{N}$EqInv, both subgoals are even easier.

$$
\begin{array}{ll}
?\mathbb{N}\mathsf{Eqfalse}_t : & \forall m \quad : \mathbb{N} \\
& \forall uneq : m \not\simeq m \\
& \quad \mathsf{true} \simeq \mathsf{false} \\
?\mathbb{N}\mathsf{Eqfalse}_f : & \forall m, n \ : \mathbb{N} \\
& \forall uneq : m \not\simeq n \\
& \forall uneq' : m \not\simeq n \\
& \quad \mathsf{false} \simeq \mathsf{false}
\end{array}
$$

$\square$

Chapter 4

Inductive Datatypes

This chapter gives a formal definition of the class of inductive datatypes and families with which we shall work in OLEG. I shall broadly follow Luo's choice of which definitions to admit, and show how their elimination and computation rules are generated [Luo94]. Goguen has checked that the usual metatheoretic properties such as strong normalisation continue to hold for ECC extended with this notion of datatype [Gog94].

Basically, we shall have the datatypes and families arising from strictly positive schemata, as proposed by Coquand, Paulin-Mohring and Dybjer [CPM90, Dyb91]. These are the datatypes of COQ, LEGO and ALF. Induction and recursion over them will be provided by means of the traditional elimination rules, which do exactly one step of case analysis, attaching an inductive hypothesis to each recursive subterm so exposed. Each type is equipped with an 'elimination constant' whose type codes up the elimination rule—computation is then added by associating the appropriate contraction schemes (or ι-reductions) with these constants. Elimination rules for inductively defined relations were first formulated by Martin-Löf in [M-L71b].

This is the exactly the presentation described in Luo's book [Luo94] and implemented in LEGO[Pol94] by Claire Jones. COQ has basically the same datatypes, but separates the 'inversion' and 'recursion' aspects of elimination by providing a `Case` construct for the former and a `Fix` construct for the latter. `Fix` is carefully checked to ensure that recursive calls are made only on terms which are **guarded** by constructors and hence strictly smaller than the term being decomposed.

The `Case`/`Fix` presentation is much the neater one, for two reasons:

- Even if there is a particular argument on which I wish my function to do recursion, that is no reason to suppose it is the first argument on which it should do case analysis. Sometimes I want to look at another argument first, and then, perhaps in not all of the cases arising, to decompose the recursive argument. The conventional eliminator ties the two notions together inappropriately.

- The conventional eliminator only facilitates recursion after exactly one constructor has been stripped away. The `Fix` operator allows recursion on any subterm exposed by `Case`. This serves a more useful purpose than merely to admit inefficient definitions of the Fibonacci function. Working interactively, we do not need to predict so precisely in advance the inductive structure you require.

Eduardo Giménez showed the conservativity and confluence of `Case` and `Fix` in [Gim94]. He showed strong normalisation for the Calculus of Constructions extended with lists in this style [Gim96], and there seems no reason to suppose this does not extend to other types. Intuitively, ι-reductions make a sound like the clanking of a giant metal cog in a ratchet. However deeply under skyscraping storeys of β- and δ-'administration' the real work may be buried, we can still hear the great machines going clank—we know that the hands of the clock will go forward and that the bell will ring for midnight.

This is a rather prosaic chapter in which I show how to mechanise Giménez's argument in OLEG. The summary, for those who would rather skip the detail, is that I equip each datatype with two alternative elimination rules, in the sense of the previous chapter. It is, of course, the **eliminate** tactic which provides the means of their construction.

At this point, I should remark that I have omitted some classes of datatype found in LEGO and COQ. Both these systems permit *mutually defined* types: for example, even and odd numbers given by a 'zero' constructor (which makes an 'even') and two 'successor' constructors (taking 'even' to 'odd' and 'odd' to 'even'). I omit them, not because they are awkward in principle, but because discussing them in general terms is a notational nightmare: I have no examples in this thesis which require them. However, all of the technology developed here for solitary inductive definitions extends to the mutual case without any difficulty—indeed the implemented system does handle mutual definitions. In any case, a mutual definition can always be represented as a single inductive family of datatypes indexed by a finite type whose elements label the branches—we might define a family $\mathsf{Parity} : \mathbf{2} \to \mathit{Type}$ with $\mathsf{Parity}\ \mathsf{true}$ containing the even numbers and $\mathsf{Parity}\ \mathsf{false}$ the odd numbers.

COQ also allows *embedded* datatypes, where an existing datatype is used as an auxiliary to a new datatype—for example, defining the finitely branching trees by a single 'node' constructor which takes a *list* of subtrees. This facility is both neat and labour-saving, but it adds no extra power. As Paulin-Mohring observes in [P-M96], embedded datatypes can be turned into mutual dataypes with extra branches duplicating the behaviour of the auxiliary types—we may define 'finitely-branching-tree' mutually with 'list-of-finitely-branching-trees'.

4.1 construction of inductive datatypes

Rather than plunging at the deep end and drowning in subscripts, let us establish a simply-typed theme, then examine variations: parameterised (or, when the parameters are themselves types, polymorphic) types, types with higher-order constructors and dependent inductive families, then degenerate types like relations and records.

The components of any inductive datatype definition are as follows

- The **type former** is the new constant which names the type or type family, eg $\mathbb{N}$, list, vect.
- The **constructors** (or 'introduction rules') are the means of forming the canonical elements of the datatype, eg $\mathsf{0}$ and s for $\mathbb{N}$.
- The **elimination rule** (or 'induction principle') provides the mechanism for decomposing elements of the datatype in the cause of constructing something else, be it a proof 'by induction' or some recursively computed value. This rule must be marked with a target so that **eliminate** can use it.
- The **ι-reductions** animate this mechanism, defining the computational behaviour of the elimination rule for each canonical element.

4.1.1 simple inductive datatypes like $\mathbb{N}$

Componentwise

- The **type former** is a constant which inhabits some universe

  $$\overline{\mathit{Ind} : \mathcal{Type}}$$

  $\mathbb{N}$ is an example of such an Ind.

- The **constructors** are function symbols $\mathsf{Con}_1 \ldots \mathsf{Con}_c$, where for each j in $1 \ldots c$

$$\mathsf{Con}_j : \forall \vec{a} : \vec{A}_j.\ \forall \vec{x} : \{\mathit{Ind}\}^{r_j}.\ \mathit{Ind}$$

The $\vec{A}_j$ are called the **non-recursive** arguments because they may not refer to Ind. Neither may they involve any universe as large as that which Ind inhabits, in order to avoid the paradoxical embedding of a larger universe inside a smaller one—we may usually rely on Harper and Pollack's universal policeman [HP91] and use the unlabelled $\mathcal{Type}$ regardless.

We say that Con_j has r_j **recursive** arguments. Think of elements of Ind as tree structures made from nodes of different kinds given by the constructors, a Con_j node having r_j out-edges and a label of telescope $\vec{A}_j$. Actually, there is no need for the recursive arguments to come after the non-recursive ones, but it makes the presentation simpler if we pretend they always do—since non-recursive arguments cannot have types involving Ind, they may certainly always be permuted to the front.

We may also think of constructors as introduction rules for Ind:

$$\frac{\vec{a} : \vec{A}_j \quad x_1 : \mathit{Ind} \quad \ldots \quad x_{r_j} : \mathit{Ind}}{\mathsf{Con}_j\ \vec{a}\ \vec{x} : \mathit{Ind}}$$

The derivation trees composed from such rules correspond exactly to the tree notion of inductive data structures mentioned above.

$\mathbb{N}$ has two constructors:

$$\frac{}{\mathsf{0} : \mathbb{N}} \qquad \frac{n : \mathbb{N}}{\mathsf{s}n : \mathbb{N}}$$

Observe also that, if Ind is to be inhabited, it will need at least one constructor with no recursive arguments.

- Let us examine $\mathit{IndElim}$, the constant whose type gives the **elimination rule** for Ind, in accordance with the general analysis of elimination rules presented earlier.

The pattern which $\mathit{IndElim}$ eliminates is the **free pattern** on Ind, which matches any element of Ind. Hence $\mathit{IndElim}$ has a scheme indexed by Ind, ie $\Phi : \forall x : \mathit{Ind}.\ \mathcal{Type}$ and a rule goal targetting the element to be eliminated. $\forall \boxed{x : \mathit{Ind}}.\ \Phi\ x$. The outline of the rule is as shown.

$\mathit{IndElim}$

$$\Phi : \forall x : \mathit{Ind}.\ \mathcal{Type}$$

$$\frac{?}{\forall \boxed{x : \mathit{Ind}}.\ \Phi\ x}$$

In order to build a proof of $\Phi\ x$ for an arbitrary x, we need a method for each constructor, showing how Φ for its conclusion follows from Φ for its recursive arguments—more succinctly, that each Con_j **preserves** Φ. We may think of Φ as a property which must hold wherever its argument is an *Ind*, hence it must have 'introduction rules'—the rule subgoals of *IndElim*—analogous to those of *Ind*. Thus we manufacture the rule subgoals of *IndElim* from the introduction rules of *Ind* by writing $\Phi\ p$ in the former wherever the latter has $p : \mathit{Ind}$:

$$\frac{\vec{a} : \vec{A}_j \quad \Phi\ x_1 \quad \ldots \quad \Phi\ x_{r_j}}{\Phi\ (\mathsf{Con}_j\ \vec{a}\ \vec{x})}$$

Note that the recursive arguments $\vec{x} : \{\mathit{Ind}\}^{r_j}$ have not disappeared entirely. The types of the recursion hypotheses depend on them, hence we may infer that they are themselves present as case hypotheses, and suppress them from the written rule accordingly. Functional programmers may be more familiar with 'fold operators'—the cut down version, where Φ is a constant and the recursive arguments are supplanted by the recursion hypotheses.

We now have all the pieces we need to complete the *IndElim* rule:

$\Phi : \forall x{:}\mathit{Ind}.\ \mathit{Type}$

$$\frac{\dfrac{\vec{a} : \vec{A}_1 \quad \Phi\ x_1 \quad \ldots \quad \Phi\ x_{r_1}}{\Phi\ (\mathsf{Con}_1\ \vec{a}\ \vec{x})} \qquad \cdots \qquad \dfrac{\vec{a} : \vec{A}_c \quad \Phi\ x_1 \quad \ldots \quad \Phi\ x_{r_c}}{\Phi\ (\mathsf{Con}_c\ \vec{a}\ \vec{x})}}{\forall \boxed{x : \mathit{Ind}}.\ \Phi\ x}$$

Or, more inscrutably,

$$\begin{aligned}
\mathit{IndElim} : \forall \Phi : \mathit{Ind} \to \mathit{Type}.& \\
& (\forall \vec{a}{:}\vec{A}_1.\ \forall \vec{x}{:}\{\mathit{Ind}\}^{r_1}.\ \{\Phi\ x_i\}_i^{r_1} \to \Phi\ (\mathsf{Con}_1\ \vec{a}\ \vec{x})) \to \\
& \ \vdots \\
& (\forall \vec{a}{:}\vec{A}_c.\ \forall \vec{x}{:}\{\mathit{Ind}\}^{r_1}.\ \{\Phi\ x_i\}_i^{r_1} \to \Phi\ (\mathsf{Con}_c\ \vec{a}\ \vec{x})) \to \\
& \forall \boxed{x : \mathit{Ind}}.\ \Phi\ x
\end{aligned}$$

For the natural numbers, then, we get

$$\begin{aligned}
\mathbb{N}\mathsf{Elim} : \forall \Phi : \forall n{:}\mathbb{N}.\ \mathit{Type}.& \\
& (\Phi\ 0) \to \\
& (\forall n{:}\mathbb{N}.\ (\Phi\ n) \to \Phi\ \mathsf{s}n) \to \\
& \forall \boxed{n : \mathbb{N}}.\ \Phi\ n
\end{aligned}$$

$\mathbb{N}\mathsf{Elim}$

$\Phi : \forall n{:}\mathbb{N}.\ \mathit{Type}$

$$\frac{\Phi\ 0 \qquad \dfrac{\Phi\ n}{\Phi\ \mathsf{s}n}}{\forall \boxed{n : \mathbb{N}}.\ \Phi\ n}$$

- Those of us given to a skeptical disposition would be unlikely to accept the validity of *IndElim* if we did not see how to plug the proofs of its rule subgoals together to build an inhabitant of Φ for any particular x which we might make from *Ind*'s constructors. This process is represented in our type theory by the ι-reductions associated with *Ind*. By this means we imbue *IndElim* with a computational meaning, allowing us to evaluate recursive functions over *Ind*.

  We add an ι-reduction for the effect of *IndElim* on each constructor:

  $$\mathit{IndElim}\, \Phi\, \vec{\phi}\, (\mathsf{Con}_j\, \vec{a}\, \vec{x}) \leadsto_\iota \phi_j\, \vec{a}\, \vec{x} \left\{ \mathit{IndElim}\, \Phi\, \vec{\phi}\, x_i \right\}_i^{r_j}$$

  For the natural numbers, we get two such rules:

  $$\begin{array}{l} \mathbb{N}\mathsf{Elim}\, \Phi\, \phi_z\, \phi_s\ \ \mathsf{0} \leadsto_\iota \phi_z \\ \mathbb{N}\mathsf{Elim}\, \Phi\, \phi_z\, \phi_s\, \mathsf{s}n \leadsto_\iota \phi_s\, n\, (\mathbb{N}\mathsf{Elim}\, \Phi\, \phi_z\, \phi_s\, n) \end{array}$$

Given the type former and constructors for a simple inductive datatype, the elimination rule and ι-reductions can be computed in a straightforward way.

4.1.2 parameterised datatypes like list

It is not hard to represent datatypes such as lists of natural numbers via the above mechanism:

$$\frac{}{\mathbb{N}\mathsf{list} : \mathit{Type}} \qquad \frac{}{\mathbb{N}\mathsf{nil} : \mathbb{N}\mathsf{list}} \qquad \frac{n : \mathbb{N} \quad t : \mathbb{N}\mathsf{list}}{\mathbb{N}\mathsf{cons}\, n\, t : \mathbb{N}\mathsf{list}}$$

However, it seems much preferable to define lists once, *polymorphically* and instantiate that definition for each type of element we encounter than to define a new list type for every element type. That is, we should be able to define lists in a way which is parameterised by the choice of element type, allowing us to write the functions which operate on arbitrarily-typed lists once and for all. For each $A : \mathit{Type}$, $\mathsf{list}\, A$ should be the simple inductive datatype of lists of A elements. Such entities are sometimes called 'families of inductive datatypes', because each element of the family is an inductive datatype.

This kind of parameterisation is very simple—once the parameters have been instantiated, they are fixed for the entire inductive definition—constructors, elimination rule, the lot. For a given parameter telescope $\vec{p} : \vec{P}$, then, we need merely bind it parametrically to each of the defined constants and rewrite rules, correspondingly replacing each C by $\mathsf{C}\, \vec{p}$ wherever they are applied.

Hence

- type former

$$\frac{}{\mathit{Ind} : \forall \vec{p}{:}\vec{P}.\ \mathcal{Type}}$$

- constructors

$$\mathsf{Con}_j : \forall \vec{p}{:}\vec{P}.\ \forall \vec{a}{:}\vec{A}_j.\ \forall \vec{x}{:}\{\mathit{Ind}\,\vec{p}\}^{r_j}.\ \mathit{Ind}\,\vec{p}$$

(or as an introduction rule)

$$\frac{\vec{a} : \vec{A}_j \quad x_1 : \mathit{Ind}\,\vec{p} \quad \ldots \quad x_{r_j} : \mathit{Ind}\,\vec{p}}{\mathsf{Con}_j\,\vec{a}\,\vec{x} : \mathit{Ind}\,\vec{p}}$$

- elimination rule

$$\boxed{\begin{array}{l} \Phi : (\mathit{Ind}\,\vec{p}) \to \mathcal{Type} \\ \dfrac{\begin{array}{c}\vec{a} : \vec{A}_1 \quad \Phi\,x_1 \quad \ldots \quad \Phi\,x_{r_1} \\ \vdots \\ \Phi\,(\mathsf{Con}_1\,\vec{p}\,\vec{a}\,\vec{x})\end{array} \qquad \cdots \qquad \begin{array}{c}\vec{a} : \vec{A}_c \quad \Phi\,x_1 \quad \ldots \quad \Phi\,x_{r_c} \\ \vdots \\ \Phi\,(\mathsf{Con}_c\,\vec{p}\,\vec{a}\,\vec{x})\end{array}}{\forall \boxed{x : \mathit{Ind}\,\vec{p}}.\ \Phi\,x} \end{array}}$$

(or as a type)

$$\begin{array}{l} \mathit{IndElim} : \forall \vec{p}{:}\vec{P}.\ \forall \Phi{:}(\mathit{Ind}\,\vec{p}) \to \mathcal{Type}. \\ \qquad (\forall \vec{a}{:}\vec{A}_1.\ \forall \vec{x}{:}\{\mathit{Ind}\,\vec{p}\}^{r_1}.\ \{\Phi\,x_i\}_i^{r_1} \to \Phi\,(\mathsf{Con}_1\,\vec{p}\,\vec{a}\,\vec{x})) \to \\ \qquad \vdots \\ \qquad (\forall \vec{a}{:}\vec{A}_c.\ \forall \vec{x}{:}\{\mathit{Ind}\,\vec{p}\}^{r_c}.\ \{\Phi\,x_i\}_i^{r_c} \to \Phi\,(\mathsf{Con}_c\,\vec{p}\,\vec{a}\,\vec{x})) \to \\ \qquad \forall \boxed{x : \mathit{Ind}\,\vec{p}}.\ \Phi\,x \end{array}$$

- ι-reductions

$$\mathit{IndElim}\,\vec{p}\,\Phi\,\vec{\phi}\,(\mathsf{Con}_j\,\vec{p}\,\vec{a}\,\vec{x}) \rightsquigarrow_\iota \phi_j\,\vec{a}\,\vec{x}\,\left\{\mathit{IndElim}\,\vec{p}\,\Phi\,\vec{\phi}\,x_i\right\}_i^{r_j}$$

The family of datatypes, list, is thus given by

$$\frac{}{\mathsf{list}\,A : \mathcal{Type}} \qquad \frac{}{\mathsf{nil}\,A : \mathsf{list}\,A} \qquad \frac{h : A \quad t : \mathsf{list}\,A}{\mathsf{cons}\,h\,t : \mathsf{list}\,A}$$

$$
\begin{array}{|l|}
\hline
\mathsf{listElim} \\[1em]
\Phi : (\mathsf{list}\, A) \to \mathit{Type} \\[2em]
\qquad \dfrac{\Phi\ (\mathsf{nil}\, A) \qquad \begin{array}{c} h : A \quad \Phi\ t \\ \vdots \\ \Phi\ (\mathsf{cons}\, h\ t) \end{array}}{\forall \boxed{l : \mathsf{list}\, A}.\ \Phi\ l} \\[1em]
\hline
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{listElim}\, A\ \Phi\ \phi_n\ \phi_c\ \ (\mathsf{nil}\, A) & \leadsto_\iota & \phi_n \\
\mathsf{listElim}\, A\ \Phi\ \phi_n\ \phi_c\ (\mathsf{cons}\, h\ t) & \leadsto_\iota & \phi_c\ h\ t\ (\mathsf{listElim}\, A\ \Phi\ \phi_n\ \phi_n\ t)
\end{array}
$$

Note that I suppress the parameter A when writing $\mathsf{cons}\, h\, t$, because it can be inferred from the type of h, conversely leaving it visible in $\mathsf{list}\, A$ and $\mathsf{nil}\, A$. In general, I shall avoid mentioning parameters wherever convenient.

4.1.3 datatypes with higher-order recursive arguments, like ord

So far, each of the datatype constructors we have seen has a fixed number of recursive arguments—in the tree metaphor, a fixed number of out-edges to smaller subtrees. One might choose to see these as a family of out-edges indexed by a finite set, and proceed to wonder whether any other types might be acceptable for indexing recursive arguments. And yes, any small enough type (telescope) can be used to index a recursive argument, as long as it does not involve the type being defined[1], giving us the increased power of **higher-order recursive arguments** addressing infinite families of subterms.

Higher-order recursive arguments are thus functions returning elements of the inductive datatype. The elimination thus rule has higher-order recursion hypotheses—functions returning proofs of Φ.

For example, we may construct a type of ordinal numbers which supplements the ‘zero’ and ‘successor’ constructors with the ‘supremum’ of a possibly infinite family of smaller ordinals:

$$
\frac{}{\mathsf{zero} : \mathsf{ord}} \qquad \frac{x : \mathsf{ord}}{\mathsf{suc}\, x : \mathsf{ord}} \qquad \frac{f : \mathbb{N} \to \mathsf{ord}}{\mathsf{sup}\, f : \mathsf{ord}}
$$

The sup constructor takes a family of ordinals indexed by $\mathbb{N}$, admitting a notionally transfinite structure.[2] The corresponding subgoal in the elimination rule gives access

[1] a restriction known as **strict positivity**

[2] Of course, $\mathbb{N} \to \mathsf{ord}$ has only countably many inhabitants.

to a family of recursion hypotheses:

$$\begin{array}{|l|}\hline \mathsf{ordElim} \\ \Phi : \forall x\!:\!\mathsf{ord}.\ \mathcal{T}\!ype \\ \dfrac{\Phi\ \mathsf{zero}\qquad \begin{array}{c}\Phi\ x\\ \cdots\cdots\cdots\\ \Phi\ (\mathsf{suc}\ x)\end{array}\qquad \begin{array}{c}\forall n\!:\!\mathbb{N}.\ \Phi\ (f\,n)\\ \cdots\cdots\cdots\\ \Phi\ (\mathsf{sup}\,f)\end{array}}{\forall \boxed{x : \mathsf{ord}}.\ \Phi\ x} \\ \hline\end{array}$$

How can we compute over such a type? The sup branch expects a family of proofs of Φ for the image of its functional argument—we may manufacture such a family by λ-abstracting over the recursive call:

$$\mathsf{ordElim}\,\Phi\,\phi_z\,\phi_s\,\phi_{sup}\,(\mathsf{sup}f) \leadsto_\iota \phi_{sup}f\,(\lambda n\!:\!\mathbb{N}.\ \mathsf{ordElim}\,\Phi\,\phi_z\,\phi_s\,\phi_{sup}\,(f\,n))$$

In the same way, we can allow constructors of an arbitrary inductive datatype to have families of recursive arguments, with the elimination rule acquiring families of recursion hypotheses:

- type former

$$\overline{\mathit{Ind} : \mathcal{T}\!ype}$$

- constructors

$$\mathsf{Con}_j : \forall \vec{a}\!:\!\vec{A}_j.\ \forall \vec{f}\!:\!\left\{\forall \vec{h}_i\!:\!\vec{H}_i.\ \mathit{Ind}\right\}_i^{r_j}.\ \mathit{Ind}$$

(or as an introduction rule)

$$\frac{\vec{a} : \vec{A}_j \quad f_1 : \forall \vec{h}_1\!:\!\vec{H}_1.\ \mathit{Ind} \quad \ldots \quad f_{r_j} : \forall \vec{h}_{r_j}\!:\!\vec{H}_{r_j}.\ \mathit{Ind}}{\mathsf{Con}_j\,\vec{a}\,\vec{f} : \mathit{Ind}}$$

- elimination rule

$$\begin{array}{|l|}\hline \Phi : \mathit{Ind} \to \mathcal{T}\!ype \\ \dfrac{\ldots\qquad \begin{array}{c}\vec{a} : \vec{A}_j \quad \forall \vec{h}_1\!:\!\vec{H}_1.\ \Phi\ (f_1\,\vec{h}_1) \quad \ldots \quad \forall \vec{h}_{r_j}\!:\!\vec{H}_{r_j}.\ \Phi\ (f_{r_j}\,\vec{h}_{r_j})\\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\\ \Phi\ (\mathsf{Con}_j\,\vec{a}\,\vec{f})\end{array}\qquad \ldots}{\forall \boxed{x : \mathit{Ind}}.\ \Phi\ x} \\ \hline\end{array}$$

(or as a type)

$$
\begin{array}{l}
\mathsf{IndElim} : \forall \Phi : \mathsf{Ind} \to \mathit{Type}. \\
\quad (\forall \vec{a} : \vec{A}_1.\ \forall \vec{f} : \left\{ \forall \vec{h}_i : \vec{H}_i.\ \mathsf{Ind} \right\}_i^{r_1}. \\
\qquad \left\{ \forall \vec{h}_i : \vec{H}_i.\ \Phi\ (f_i\ \vec{h}_i) \right\}_i^{r_1} \to \Phi\ (\mathsf{Con}_1\ \vec{a}\,\vec{f})) \to \\
\quad \vdots \\
\quad (\forall \vec{a} : \vec{A}_1.\ \forall \vec{f} : \left\{ \forall \vec{h}_i : \vec{H}_i.\ \mathsf{Ind} \right\}_i^{r_c}. \\
\qquad \left\{ \forall \vec{h}_i : \vec{H}_i.\ \Phi\ (f_i\ \vec{h}_i) \right\}_i^{r_c} \to \Phi\ (\mathsf{Con}_c\ \vec{a}\,\vec{f})) \to \\
\quad \forall \boxed{x : \mathsf{Ind}}.\ \Phi\ x
\end{array}
$$

- ι-reductions

$$\mathsf{IndElim}\ \Phi\ \vec{\phi}\ (\mathsf{Con}_j\ \vec{a}\,\vec{f}) \leadsto_\iota \phi_j\ \vec{a}\,\vec{f} \left\{ \lambda \vec{h}_i : \vec{H}_i.\ \mathsf{IndElim}\ \Phi\ \vec{\phi}\ (f_i\ \vec{h}_i) \right\}_i^{r_j}$$

4.1.4 dependent inductive families like the fins

Let us now extend the notion of inductive datatypes to include inductively defined indexed families of types as in [Dyb91].

For example, consider the finite sets. For any n, it is not hard to define a simple type with n elements. Types such as $\mathbb{0}$, $\mathbb{1}$ and $\mathbb{2}$ are commonplace. However, our choice of n is at the meta-level, and we must define each type separately. How much more useful if we could define $\mathsf{fin} : \mathbb{N} \to \mathit{Type}$, enabling us to reason at the object level about arbitary finite sets. Of course, $\mathsf{fin}\ 0$ had better be empty, and we can make $\mathsf{fin}\ \mathsf{s}n$ by inventing a 'new' element, then embedding all the 'old' elements of $\mathsf{fin}\ n$. That is, fin is a mutually defined family of datatypes with constructors:

$$\frac{}{\mathsf{fz}\,n : \mathsf{fin}\ \mathsf{s}n} \qquad \frac{x : \mathsf{fin}\ n}{\mathsf{fs}\,x : \mathsf{fin}\ \mathsf{s}n}$$

By convention, I choose to think of these sets growing in a 'push-down' fashion. The new element introduced by fz is 'zero', while the old elements are embedded by a 'successor' function. By a deBruijn influenced predisposition, I see the newest as the closest and lowest in number. Note that we may leave n as an implicit argument to fs.

fin has a family of elimination rules with a family of schemes

$$\Phi : \forall n : \mathbb{N}.\ (\mathsf{fin}\ n) \to \mathit{Type}$$

We form the rule subgoals by demanding that $\Phi\ n$ holds wherever $\mathsf{fin}\ n$ is inhabited—that is, we select the scheme corresponding to the relevant branch of the mutual definition. Hence, $\mathsf{finElim}$

$$
\boxed{
\begin{array}{l}
\mathsf{finElim} \\[1em]
\Phi : \forall n : \mathbb{N}.\ (\mathsf{fin}\ n) \to \mathit{Type} \\[2em]
\dfrac{\Phi\ \mathsf{s}n(\mathsf{fz}\ n) \qquad \begin{array}{c}\Phi\ n\ x \\ \cdots\cdots\cdots\cdots \\ \Phi\ \mathsf{s}n\ (\mathsf{fs}\ x)\end{array}}{\forall n : \mathbb{N}.\ \forall \boxed{x : \mathsf{fin}\ n}.\ \Phi\ n\ x}
\end{array}
}
$$

with computational behaviour

$$
\begin{array}{l}
\mathsf{finElim}\ \Phi\ \phi_{fz}\ \phi_{fs}\ \mathsf{s}n\ (\mathsf{fz}\ n)\ \leadsto_\iota\ \phi_{fz}\ n \\
\mathsf{finElim}\ \Phi\ \phi_{fz}\ \phi_{fs}\ \mathsf{s}n\ (\mathsf{fs}\ x)\ \leadsto_\iota\ \phi_{fs}\ x\ (\mathsf{finElim}\ \Phi\ \phi_{fz}\ \phi_{fs}\ n\ x)
\end{array}
$$

fin is thus an inductively defined family of types—the instances of the family are not inductive datatypes taken in isolation; only collectively do they form a mutual inductive definition. Contrast this with a family of inductive datatypes such as list, where each member, eg list $\mathbb{N}$ is an inductive datatype in its own right.

In the light of this example, let us generalise to dependent inductive families,

$$
\mathit{Fam} : \forall \vec{\imath} : \vec{I}.\ \mathit{Type}
$$

The constructors now take recursive arguments from and return values in any instance of the type family being defined, that is any $\mathit{Fam}\ \vec{t}$ for terms $\vec{t} : \vec{I}$. Thus, in the 'introduction rule' style, we get

$$
\frac{\vec{a} : \vec{A} \quad x_1 : \mathit{Fam}\ \vec{t}_1 \quad \ldots \quad x_r : \mathit{Fam}\ \vec{t}_r}{\mathsf{Con}\ \vec{a}\ \vec{x} : \mathit{Fam}\ \vec{t}_{con}}
$$

The scheme of $\mathit{FamElim}$ must be indexed over the entirety of the types being defined, that is

$$
\Phi : \forall \vec{\imath} : \vec{I}.\ (\mathit{Fam}\ \vec{\imath}) \to \mathit{Type}
$$

Recall that the 'free telescope' notation abbreviates this to $\Phi : \overline{\mathit{Fam}} \to \mathit{Type}$.

The rule subgoals demand that for all $\vec{\imath}$, $\Phi\ \vec{\imath}$ holds wherever $\mathit{Fam}\ \vec{\imath}$ is inhabited; more succinctly that Φ holds wherever $\overline{\mathit{Fam}}$ is inhabited. Hence we get $\mathit{FamElim}$:

$$\begin{array}{|c|}\hline \Phi : \overline{\mathsf{Fam}} \to \mathit{Type} \\ \\ \begin{array}{ccc} & \vec{a} : \vec{A} \quad \Phi\, \vec{t}_1; x_1 \quad \ldots \quad \Phi\, \vec{t}_r; x_r & \\ & \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots & \\ \ldots & \Phi\, \vec{t}_{con}; (\mathsf{Con}\, \vec{a}\, \vec{x}) & \ldots \\ \hline \multicolumn{3}{c}{\forall \vec{\imath}; \boxed{x} : \overline{\mathsf{Fam}}.\ \Phi\, \vec{\imath}; x} \end{array} \\ \hline \end{array}$$

Observe that there is still only one targetter: unifying term and type gives enough information to infer an inhabitant of $\overline{\mathsf{Fam}}$.

The reduction rule for each constructor is thus:

$$\mathsf{FamElim}\, \Phi\, \vec{\phi}\, \vec{t}_{con}; (\mathsf{Con}\, \vec{a}\, \vec{x}) \leadsto_\iota \phi_{con}\, \vec{a}\, \vec{x} \left\{ \mathsf{FamElim}\, \Phi\, \vec{\phi}\, \vec{t}_j; x_j \right\}_j^r$$

4.1.5 inductively defined relations like $<$

Inductively defined relations bear a strong resemblance to dependent inductive families of datatypes. However, their presentation is differently motivated: inductive relations are families of *propositions* and their role is in reasoning rather than computation—they sit outside the domain of programs and data characterising aspects of it.

Propositions are types, and the terms which inhabit them constitute proofs. An inductive relation's inhabitants are built by constructor functions, just like a datatype—we may think of these constructors as *inference rules*—but their elimination rules do not inspect proofs explicitly in terms of their constructors.

Technically, the difference between inductive relations and datatypes is manifested in two ways:

- the type formers of an inductive relation range over the impredicative universe Prop, and correspondingly, the schemes of their elimination rules are also families of propositions
- inductive relations are *proof irrelevant*—the apertures of their elimination rules abstract the *indices* of the relation, but not the proofs themselves, hence the rule cases never identify the constructors to which they correspond

We shall need at least one relation which *can* interfere with computation, and that is $\simeq$. We use $\simeq$ to represent constraints in the elimination process for datatypes as well as relations, and hence we must allow it to eliminate over *Type* as well as *Prop*. Indeed, this is not the only way in which $\simeq$ does not fit the presentation of inductive relations given here. It is treated specially, and gets the next chapter to itself. For the moment, let us consider inductive relations for reasoning.

Many dependent datatypes have relational analogues. For example, the fin family corresponds to the 'less than' relation:

$$< \; : \forall m, n : \mathbb{N}.\ \mathit{Prop}$$

There are two introduction rules for $<$:

$$<\!\mathsf{new}\ \ \frac{}{n < \mathsf{s}n} \qquad\qquad <\!\mathsf{old}\ \ \frac{m < n}{m < \mathsf{s}n}$$

The names of the rules are really the constructor symbols, but taking them to the side emphasises the proof irrelevant nature of relations. This leaves us free to write propositions with no prefixed proofs in the introduction rules.

Compare $m < n$ with $\mathsf{fin}\ n$. $m < 0$ is clearly empty. For each $\mathsf{s}n$, $<\!\mathsf{new}$ proves that n is the 'new' thing only just smaller, whilst $<\!\mathsf{old}$ lifts the proofs for those m's already smaller than n: exactly as fz creates the 'new' element of each finite set and fs embeds the 'old' ones.

The elimination rule OLEG provides for $<$ is sometimes known as its **strong induction principle**, $<\!\mathsf{Elim}$:

$$\boxed{\begin{array}{l} <\!\mathsf{Elim} \\[1em] \Phi : \forall m, n : \mathbb{N}.\ \mathit{Prop} \\[2em] \dfrac{\Phi\ n\ \mathsf{s}n \qquad \begin{array}{c} m < n \quad \Phi\ m\ n \\ \vdots \\ \Phi\ m\ \mathsf{s}n \end{array}}{\forall m, n : \mathbb{N}.\ \forall \boxed{H : m{<}n}.\ \Phi\ m\ n} \end{array}}$$

Note that the scheme is indexed only over the two numbers, not the proof that the first is less than the second. Correspondingly, the targetted H does not occur in the goal patterns, nor do the constructor symbols $<\!\mathsf{new}$ and $<\!\mathsf{old}$ appear in the subgoals. Consequently, the step case hypothesis $m < n$ is no longer implicitly given by the inductive hypothesis, so we must write it explicitly if we mean it to be there. As a matter of fact, we can choose to omit it from the rule, obtaining the **weak induction principle**. The two are equivalent given an appropriate notion of conjunction, but the strong version is more useful in practice: it is generally preferable to discard unnecessary hypotheses than to reconstruct necessary ones.

It is not clear to me why inductively defined relations should be equipped with computational behaviour: computation belongs within the realm of datatypes, and any of these inductive relations over which computation is desired can easily be redefined as a dependent family. On the other hand, in the sense that computation explains induction, it should be possible to equip relations with reduction rules which are meaningful, if not desirable. For $<$, we get

$$\begin{array}{l} {<}\mathsf{Elim}\ \Phi\ \phi_{new}\ \phi_{old}\ \ n\ \mathsf{s}n\ ({<}\mathsf{new}\ n)\ \leadsto_\iota\ \phi_{new}\ n \\ {<}\mathsf{Elim}\ \Phi\ \phi_{new}\ \phi_{old}\ m\ \mathsf{s}n\ \ ({<}\mathsf{old}\ H)\ \leadsto_\iota\ \phi_{old}\ H\ ({<}\mathsf{Elim}\ \Phi\ \phi_{new}\ \phi_{old}\ m\ n\ H) \end{array}$$

With $<$ to guide us, here is the general treatment:

- proposition former

$$\mathit{Rel} : \forall\vec{\imath}{:}\vec{I}.\ \mathcal{P}rop$$

- inference rules (constructors)

$$\frac{\vec{a}:\vec{A}\quad x_1 : \mathit{Rel}\,\vec{t}_1\quad \ldots\quad x_r : \mathit{Rel}\,\vec{t}_r}{\mathit{Rule}\,\vec{a}\,\vec{x} : \mathit{Rel}\,\vec{t}_{rule}}$$

- elimination rule (strong induction principle)

$$\boxed{\begin{array}{l} \Phi : \forall\vec{\imath}{:}\vec{I}.\ \mathcal{P}rop \\ \\ \dfrac{\ldots\qquad \begin{array}{c} \vec{a}:\vec{A}\quad \mathit{Rel}\,\vec{t}_1\quad\ldots\quad \mathit{Rel}\,\vec{t}_r\quad \Phi\,\vec{t}_1\quad\ldots\quad\Phi\,\vec{t}_r \\ \vdots \\ \Phi\,\vec{t}_{rule}\end{array}\qquad\ldots}{\forall\vec{\imath};\boxed{H}:\overline{\mathit{Rel}}.\ \Phi\,\vec{\imath}} \end{array}}$$

- ι-reductions

$$\mathit{RelElim}\,\Phi\,\vec{\phi}\,\vec{t}_{rule}\,(\mathit{Rule}\,\vec{a}\,\vec{x})\ \leadsto_\iota\ \phi_{rule}\,\vec{a}\,\vec{x}\left\{\mathit{RelElim}\,\Phi\,\vec{\phi}\,\vec{p}_i\,x_i\right\}_i^r$$

4.1.6 record types

We can represent (dependent) record types as a degenerate case of inductive datatypes. A simple datatype Rec with one constructor rec which has no recursive arguments is just a tupling wrapper for the non-recursive arguments, or **fields**, as we might like to call them.

The typical type former and constructor are as follows:

- type former

$$\frac{}{\mathsf{Rec} : \mathcal{T}ype}$$

- constructor (singular)

$$\frac{\vec{\mathsf{field}} : \vec{A}}{\mathsf{rec}\ \vec{\mathsf{field}} : \mathsf{Rec}}$$

The 'official' field names $\vec{\mathsf{field}}$ are significant in that they allow us to adopt a more conventional named-tuple notation as syntactic sugar—I write $X \implies Y$ to indicate that X is a sugared notation for Y:

$$\left\langle \vec{\mathsf{field}} = \vec{t} \right\rangle \implies \mathsf{rec}\ \vec{t}$$

This presumes that the sequence of names $\vec{\mathsf{field}}$ determines which of the defined record types is intended. Underneath the layer of sugar, the names of fields are irrelevant.

Having established this syntax, the elimination and computation rules become

- elimination rule $\mathsf{RecElim}$

$$\boxed{\begin{array}{l} \Phi : \mathsf{Rec} \to \mathcal{T}ype \\ \dfrac{\begin{array}{c} \vec{t} : \vec{A} \\ \cdots\cdots\cdots\cdots\cdots \\ \Phi \left\langle \vec{\mathsf{field}} = \vec{t} \right\rangle \end{array}}{\forall \boxed{x : \mathsf{Rec}}.\ \Phi\ x} \end{array}}$$

- ι-reduction

$$\mathsf{RecElim}\ \Phi\ \phi\ \left\langle \vec{\mathsf{field}} = \vec{t} \right\rangle \leadsto_\iota \phi\ \vec{t}$$

These record types do not come ready-equipped with projections. Instead, their elimination rules require a function of the fields: introducing the arguments effectively extends the local context with λ-bindings for the fields. That is, $\mathsf{RecElim}$ has a similar behaviour to pattern-matching for named tuples, SML's 'open' for structures or Pascal's 'with ... do' construct. Underneath the λs, you are entitled to place any well-typed expression you choose, involving as many or as few fields as you like.

In an interactive, analytical setting, eliminating by $\mathsf{RecElim}$ is preferable to projection because it is more focused on the goal. Also, a single elimination exposes all of the fields together, where projection gives you but one at a time. To me it seems a rather more honest account, especially when there may be type dependency between fields. Understanding records by atomising them into fields in spite of the structure which weaves them together is a bit like understanding London in terms of discrete hinterlands for each tube station. Plenty of people (including me) navigate London on that basis, but they are not the Londoners.

Let us nonetheless define the projections with the conventional notation $(\cdot).\mathsf{field}_i$. Type dependency requires us to do so in order—earlier projections appear in the types of later ones.

Presuming we have defined $(\cdot).\mathsf{field}_1 \ldots (\cdot).\mathsf{field}_n$, $(\cdot).\mathsf{field}_{n+1}$ is

$$\begin{array}{rl} (\cdot).\mathsf{field}_{n+1} & = \mathsf{RecElim} \\ & \qquad (\lambda R\!:\!\mathsf{Rec}.\ \{[R.\mathsf{field}_i/\mathsf{field}_i]\}_i^n A_{n+1}) \\ & \qquad (\lambda \vec{a}\!:\!\vec{A}.\ a_{n+1}) \\ & : \forall R\!:\!\mathsf{Rec}.\ \{[R.\mathsf{field}_i/\mathsf{field}_i]\}_i^n A_{n+1} \end{array}$$

I refer to this use of '.' as 'spot' because I think of it as an ugly thing which I wish to distance from the 'dot' used for binding. 'dot' marks a scope which may contain any well-typed expression whose identifiers have been explained. 'spot' only allows the name of a field. Let us apply generous makeup to hide our acne. If $R : \mathsf{Rec}$, we may write

$$R[\vec{x}].t \implies !\vec{x} = R.\vec{\mathsf{field}}\!:\!\vec{A}.\ t$$

This syntactic sugar abbreviates a bunch of !-bindings which **open** the record with our chosen local names. The dot introduces the scope of the bindings—we may naturally have anything we like under it. Let us abbreviate further, in the case where the chosen names are the 'offical' ones:

$$R.t \implies R[\vec{\mathsf{field}}].t$$

If t happens to be a field name, we recover the effect of projection. However, if $R = \langle \mathsf{x} = 3; \mathsf{y} = 5; \mathsf{z} = 7 \rangle$, then $R.\mathsf{x} * \mathsf{y} * \mathsf{z} \; = \; 105$.

There is a superficial resemblance between this 'opening' notation and the 'explicit environments' of Sato, Sakurai and Burstall [SSB99]. However, their treatment propagates environments through the term structure in the manner of explicit substitutions, rather than giving them the 'action at a distance' effect of !-binding. I have implemented these 'first class local definition' records as an experimental extension to LISP [McB92].

4.2 a compendium of inductive datatypes

This section defines formally a number of familiar datatypes as used in this thesis and in everyday functional programming. Its purpose is partly to consolidate the material of the previous section, but mostly to confine to one contiguous portion of this thesis a lot of boring definitions.

Some finite types, see table 4.1, are standard equipment: $\mathbb{0}$ ('empty'), $\mathbb{1}$ ('unit') and $\mathbb{2}$ ('bool'). The constructor $\Diamond$ is pronounced 'void'.

Let us also have disjoint sums, $+$, and, specifically, maybe: table 4.2.

A dependent family, often to be found in the examples of this thesis are the vectors, vect: table 4.3. Note the suppression of inferrable arguments.

4.3 abolishing Σ-types and reinventing them

Luo supplies dependent pairs, or **Σ-types**, as basic features of ECC, equipped with first and second projections. However, with our facility for datatypes, it seems preferable to present pairing as a parameterised record type. Also, pairs might as well acquire the apparatus we shall shortly build for other datatypes.

- record former
  $$\frac{B : A \to \mathit{Type}}{\Sigma\, B : \mathit{Type}}$$
- fields

  $\mathsf{1} : A$
  $\mathsf{2} : B\ \mathsf{1}$

$$
\begin{array}{|c|c|c|}
\hline
\dfrac{}{\mathbb{0} : \mathit{Type}} & \dfrac{}{\mathbb{1} : \mathit{Type}} & \dfrac{}{\mathbb{2} : \mathit{Type}} \\[2ex]
 & \dfrac{}{\Diamond : \mathbb{1}} & \dfrac{}{\mathsf{true} : \mathbb{2}} \\[2ex]
 & & \dfrac{}{\mathsf{false} : \mathbb{2}} \\
\hline
\end{array}
$$

Table 4.1: standard finite types

$$
\begin{array}{|cc|cc|}
\hline
\dfrac{\mathrm{L}, \mathrm{R} : \mathit{Type}}{\mathrm{L}{+}\mathrm{R} : \mathit{Type}} & & \dfrac{\mathrm{X} : \mathit{Type}}{\mathsf{maybe}\ \mathrm{X} : \mathit{Type}} & \\[3ex]
\dfrac{l : \mathrm{L}}{\mathsf{inL}\, l : \mathrm{L}{+}\mathrm{R}} & \dfrac{r : \mathrm{R}}{\mathsf{inR}\, r : \mathrm{L}{+}\mathrm{R}} & \dfrac{x : \mathrm{X}}{\mathsf{yes}\, x : \mathsf{maybe}\ \mathrm{X}} & \dfrac{}{\mathsf{no} : \mathsf{maybe}\ \mathrm{X}} \\
\hline
\end{array}
$$

Table 4.2: + and maybe

$$
\begin{array}{|cc|}
\hline
\dfrac{\mathrm{A} : \mathit{Type} \quad n : \mathbb{N}}{\mathsf{vect}_{\mathrm{A}}\, n : \mathit{Type}} & \\[3ex]
\dfrac{}{\mathsf{vnil}_{\mathrm{A}} : \mathsf{vect}_{\mathrm{A}}\, \mathsf{0}} & \dfrac{h : \mathrm{A} \quad t : \mathsf{vect}_{\mathrm{A}}\, n}{\mathsf{vcons}\, h\, t : \mathsf{vect}_{\mathrm{A}}\, \mathsf{s}n} \\
\hline
\end{array}
$$

Table 4.3: vectors

intro-Σ			**raise**-Σ		
$?x \colon \Sigma y \colon S$ T x	$\Rightarrow$	$?y \colon S$ $?x \colon T$ $\langle y; x\rangle$	$?y \colon S$ $?x \colon T$ u	$\Rightarrow$	$?x \colon \Sigma y \colon S$ T $x.[1/y][2/x]u$

Table 4.4: tactics for Σ-types in goals

The only penalty we risk paying is slightly clumsy syntax, but it is in our power to sugar this problem away. Let us have lots.

$\Sigma x : S.\, T \implies \Sigma\ (\lambda x{:}S.\, T)$	Σ is a fake binding operator
$S{\times}T \implies \Sigma\_ : S.\, T$	the usual special case
$\langle s; t\rangle \implies \langle 1 = s; 2 = t\rangle$	unlabelled pairs
$\Sigma\{\}^0 \implies \mathbb{1}$	empty telescope gives unit type
$\Sigma\vec{S}; T \implies \Sigma x_1 : S_1.\ \ldots \Sigma x_n : S_n.\, T$	nonempty telescope gives Σ-type
$\langle\{\}^0\rangle \implies \Diamond$	empty sequence gives void
$\langle\vec{s}; t\rangle \implies \langle s_1; \langle \ldots \langle s_n; t\rangle \ldots\rangle\rangle$	nonempty sequence gives pair

There is no conflict between using Σ both for binding and as an operator which turns telescopes into the types of tuples, represented as pairs nested to the right. Also, we still have the dot-notation from record types as sugar for $\Sigma\mathsf{Elim}$. Our cunning choice of field names gives us the familiar $(\cdot).1$ and $(\cdot).2$ projections as a special case.

We should equip OLEG with the tactics for dragging ?-bindings through our fake Σ-bindings. See table 4.4. Both are replacements. Applied recursively, **intro**-$\forall$ and **intro**-Σ turn a goal full of $\forall$s and Σs into a partial proof full of λs and ?s. Correspondingly, **raise**-Σ combines with **raise**-$\forall$ to allow multiple subgoals to **retreat** from a partial contruction as a single outstanding proof obligation.

4.3.1 the blunderbuss tactic

intro-Σ caters for Σ-types in goals, allowing us to solve them piecewise. What about Σ-types in hypotheses? Although we usually try to curry them away wherever possible, we do still find Σ-types in inductive hypotheses, for example, when the original goal was to compute a pair.

It is awkward to exploit such hypotheses with tactics such as LEGO's `Refine` which are specifically geared to use functional information. I therefore propose the following 'blunderbuss' tactic,[3] which will search inside Σ as well as under $\forall$:. .

TACTIC: **blunderbuss**

$$\begin{array}{|l|}\hline !root = s : S \\ ?goal : G \\ \hline\end{array}$$

This tactic tries to use some s to solve a *goal* by a depth-first search strategy. The nodes of the search tree are given by !-bindings of proofs to try. Initially, the root node is set to s.
At each $!node = s : S$, starting with *root*, **blunderbuss** behaves as follows:

- try to **unify** *node* with *goal*—if successful, stop, otherwise . . .
- reduce S to weak head-normal form
- generate subnodes by the type-directed methods given in table 4.5 and try **blunderbuss** with each in turn—**blunder**-refl subnodes are tried before **blunder**-$\forall$ subnodes

We recover exactly LEGO's `Refine` tactic if we only have **blunder**-$\forall$. However, now we can just as well blunder under a Σ.

I have taken this opportunity to sneak in **blunder**-refl. Recall that when **eliminate** generates a constrained scheme, the equations generated appear as a matching problem in any inductive hypotheses which may arise. **blunder**-refl is intended to make it easier to exploit such hypotheses whenever the matching problem has an obvious solution.

Hence, whenever an equational premise is required, **blunderbuss** tries to unify the two sides in order to supply a refl proof. If this fails, then **blunder**-$\forall$ introduces the premise as normal. It would be very unusual if making a possible unification turned out to be an unfortunate choice.

The construction of the 'guarded fixpoint' operator in the next section uses a style of hypothesis for whose exploitation **blunderbuss** is exactly the right tactic.

[3] A blunderbuss is an old-fashioned kind of gun with a barrel which opens out like a horn. It fires almost anything at almost everyone in a wide spread. The phrase 'blunderbuss tactics' is used to describe the technique of throwing everything you have got at a problem in the hope that something will work.

blunder-refl

$$\begin{array}{l}\Delta\\ ?y\approx P\left[\begin{array}{lll}!node= s & & \\ & : & \forall x\colon t\simeq t'\\ & & \quad B\\ & \vdots & \\ ?goal & : & G\\ p & & \end{array}\right]\end{array} \Rightarrow \begin{array}{l}\Delta'\\ ?y\approx P'\left[\begin{array}{lll}!sub = s\ (\mathsf{refl}\ t) & & \\ & : & [\mathsf{refl}\ t/x]B\\ & \vdots & \\ ?goal & : & G\\ p & & \end{array}\right]\end{array}$$

t and t' **unify**, ie
$\Delta \ \sqsubseteq\ \Delta'$
$\Delta' \ \Vdash\ P \ \sqsubseteq\ P'$
$\Delta'; \bar{P}' \ \vdash\ t \ \cong\ t'$

blunder-$\forall$

$$\begin{array}{l}\Delta\\ ?y\approx P\left[\begin{array}{lll}!node= s & & \\ & : & \forall x\colon A\\ & & \quad B\\ & \vdots & \\ ?goal & : & G\\ p & & \end{array}\right]\end{array} \Rightarrow \begin{array}{l}\Delta\\ ?y\approx P\left[\begin{array}{lll}?x & : & A\\ !sub = s\ x & & \\ & : & B\\ & \vdots & \\ ?goal & : & G\\ p & & \end{array}\right]\end{array}$$

blunder-Σ

$$\begin{array}{l}\Delta\\ ?y\approx P\left[\begin{array}{lll}!node= s & & \\ & : & \Sigma x\colon A\\ & & \quad B\\ & \vdots & \\ ?goal & : & G\\ p & & \end{array}\right]\end{array} \Rightarrow \begin{array}{l}\Delta\\ ?y\approx P\left[\begin{array}{lll}!sub_1 = s.1 & : & A\\ !sub_2 = s.2 & & \\ & : & [sub_1/x]B\\ & \vdots & \\ ?goal & : & G\\ p & & \end{array}\right]\end{array}$$

Table 4.5: **blunderbuss** search methods

4.4 constructing `Case` and `Fix`

This section shows how to derive two alternative eliminators for each datatype, corresponding to the `Case` and `Fix` operators in COQ.

4.4.1 case analysis for datatypes and relations

From the elimination rule given for a datatype or relation, we may construct a 'sawn-off' version which embodies the notion that we may reason about an arbitrary inhabitant of the type by considering each of the possibilities for its outermost 'head' constructor, but without any recursive information. For $\mathbb{N}$, we get $\mathbb{N}\mathsf{Case}$.

$\mathbb{N}\mathsf{Case}$

$\Phi : \forall n{:}\mathbb{N}.\ \mathit{Type}$

$$\frac{\Phi\ 0 \qquad \begin{array}{c} n : \mathbb{N} \\ \cdots\cdots \\ \Phi\ \mathsf{s}n \end{array}}{\forall \boxed{n : \mathbb{N}}.\ \Phi\ n}$$

This construction builds theorem and proof together by a technique which I call **hubris**: we proudly attempt to prove a blatantly false claim and fail, turning the remaining subgoals into premises, just like a lecturer leaving the bits he has forgotten how to do as exercises for the students. The trick is to **postpone** the remaining ?-bindings at the outside level, turning them into λ-bindings, and then to **discharge** them.

CONSTRUCTION: **case analysis**

Suppose we have a inductive family

$\mathit{Fam} : \forall \vec{\imath}{:}\vec{I}.\ \mathcal{U}_i$

where $\vec{\imath}$ are the indices (as in dependent datatypes or relations)
$\mathcal{U}_i$ is the universe the family of types inhabits

We need only consider indices, fixing the parameters of families like list and vect for the whole construction.

This family will have an elimination rule $\mathit{FamElim}$

$\Phi : \forall \vec{z}.\ \mathcal{U}_e$

$$\frac{\mathit{subgoals}}{\forall \vec{y}; \boxed{y} : \overline{\mathit{Fam}}.\ \Phi\ \vec{p}}$$

where $\vec{z}$ inhabits a prefix of $\overline{\mathit{Fam}}$
$\vec{p}$ is the corresponding prefix of $\vec{y}; y$
$\mathcal{U}_e$ is the universe over which the family of types eliminates

For our inductive definitions, the subgoals $\vec{\phi}$ have conclusions which are applications of Φ.

Let us boldly fix a Φ and attempt to prove

This is patently untrue, but never mind, **eliminate** using *FamElim*.

$$\begin{array}{ll}
\Delta & \\
\lambda\Phi & : \forall\vec{z}.\ \mathcal{U}_e \\
?\mathit{FamCase} & : \forall\vec{y};\boxed{y}: \overline{\mathit{Fam}} \\
& \quad \Phi\ \vec{p}
\end{array}$$

Note that the holes may not appear so neatly ordered, but no matter.

The subgoals $\vec{\phi}$ correspond to the constructors of the datatype.

$$\begin{array}{ll}
?\vec{\phi} & : \ \ldots \to \Phi \ldots \\
?\vec{s} & : \ \vec{S} \\
!\mathit{FamCase} & = \mathit{FamElim}\ \Phi\ \ldots \\
& : \ \forall\vec{y};\boxed{y}: \overline{\mathit{Fam}} \\
& \quad \Phi\ \vec{p}
\end{array}$$

For each ϕ_i, we divide its hypotheses into case data $\vec{c}$ and inductive hypotheses $\vec{rec}_\Phi$.

$$\begin{array}{ll}
?\phi: \forall\vec{c} & : \vec{C} \\
\quad \forall\vec{rec}_\Phi : \ldots \to \Phi \ldots & \\
\quad \Phi\ \vec{q}[\vec{c}] &
\end{array}$$

For our inductive definitions, nothing is permitted to depend on the inductive hypotheses. Hence we may remove them with **delete-unused**.

Having modified the subgoals in this way, let us **postpone** them. The state is now as shown.

$$\begin{array}{ll}
\Delta & \\
\lambda\Phi & : \ \forall\vec{z}.\ \mathcal{U}_e \\
\lambda\vec{\phi} & : \ \forall\vec{c}.\ \Phi\ \vec{q}[\vec{c}] \\
\lambda\vec{s} & : \ \vec{S} \\
!\mathit{FamCase} & = \mathit{FamElim}\ \Phi\ \ldots \\
& : \ \forall\vec{y};\boxed{y}: \overline{\mathit{Fam}} \\
& \quad \Phi\ \vec{p}
\end{array}$$

Finally, we may discharge the assumptions, recovering the case analysis principle as we might expect it.

$$\begin{array}{lll}
!\mathit{FamCase} = & \lambda\Phi : \forall\vec{z}.\ \mathcal{U}_e & \\
& \lambda\vec{\phi} : \forall\vec{c}.\ \Phi\ \vec{q}[\vec{c}] & \\
& \lambda\vec{s} : \vec{S} & \\
& \quad \mathit{FamElim}\ \Phi\ \ldots & \\
: & \forall\Phi & : \forall\vec{z}.\ \mathcal{U}_e \\
& \forall\vec{\phi} & : \forall\vec{c}.\ \Phi\ \vec{q}[\vec{c}] \\
& \forall\vec{s} & : \vec{S} \\
& \forall\vec{y};\boxed{y}: \overline{\mathit{Fam}} & \\
& \quad \Phi\ \vec{p} &
\end{array}$$

It is not hard to see that the following reductions hold

$$\mathit{FamCase}\ \Phi\ \vec{\phi}\ \vec{\imath};\ (\mathsf{Con}_j\ \vec{x}) \ \rhd\ \phi_j\ \vec{x}$$

4.4.2 the guarded fixpoint principle

Before giving the construction of the elimination rule which performs the job of COQ's `Fix` construct, let us look at an example which motivates both the need for it and the manner in which it is done.

It is that famous old troublemaker: the Fibonacci function, which is used for counting rabbits, drawing attractive rectangles and making Euclid's algorithm go as slowly as possible:

$$
\begin{array}{lcl}
\mathsf{fib}\ \ 0 & = & \mathsf{s}0 \\
\mathsf{fib}\ \ \mathsf{s}0 & = & \mathsf{s}0 \\
\mathsf{fib}\ \mathsf{ss}n & = & \\
\quad \mathsf{plus}\ (\mathsf{fib}\ n)\ (\mathsf{fib}\ \mathsf{s}n) & &
\end{array}
$$

Let us see what goes wrong if we just blunder in with $\mathbb{N}$Elim, trying to mimic this definition. Here is the initial state, with the return type decorated by !-binding, so we can see what is happening.

$$
\begin{array}{l}
!\mathsf{Fib} = \lambda\_ : \mathbb{N}.\ \mathbb{N} \\
\qquad : \ \mathbb{N} \to \mathit{Type} \\
?\mathsf{fib}\ :\ \forall n : \mathbb{N} \\
\qquad\quad \mathsf{Fib}\ \boxed{n}
\end{array}
$$

Let us **eliminate** n.

Again, the !-binding tracks the arguments. We can certainly fill in fib_0. Now watch what happens when we **eliminate** again to split the successor case:

$$
\begin{array}{l}
?\mathsf{fib}_0 : \mathsf{Fib}\ 0 \\
?\mathsf{fib}_s : \forall n \quad : \mathbb{N} \\
\qquad \forall \mathit{fib}_n : \mathsf{Fib}\ n \\
\qquad\quad \mathsf{Fib}\ \mathsf{s}\boxed{n}
\end{array}
$$

The $\mathsf{Fib}\ \mathsf{s}0$ case is fine, but for double-successor, disaster has struck! We have our $\mathsf{Fib}\ \mathsf{s}n$ safely enough, but what has happened with $\mathsf{Fib}\ n$? It has appeared, all right, but in the wrong place—we have no hope of accessing it.

$$
\begin{array}{l}
?\mathsf{fib}_{s0} : \forall \mathit{fib}_0 : \mathsf{Fib}\ 0 \\
\qquad\quad \mathsf{Fib}\ \mathsf{s}0 \\
?\mathsf{fib}_{ss} : \forall n \quad : \mathbb{N} \\
\qquad \forall \mathrm{hyp} : \forall \mathit{fib}_n : \mathsf{Fib}\ n \\
\qquad\qquad\qquad \mathsf{Fib}\ \mathsf{s}n \\
\qquad \forall \mathit{fib}_{sn} : \mathsf{Fib}\ \mathsf{s}n \\
\qquad\quad \mathsf{Fib}\ \mathsf{ss}n
\end{array}
$$

Of course, the classic definition of the Fibonacci function is famous for its abominable run-time.[4] The traditional remedy is to write a linear recursion computing a pair of successive values. In [BD77], Burstall and Darlington transform the above definition into the following more efficient form:

$$
\begin{array}{rrcl}
\mathsf{fib} & 0 & = & \mathsf{s}0 \\
\mathsf{fib} & \mathsf{s}0 & = & \mathsf{s}0 \\
\mathsf{fib} & \mathsf{ss}n & = & (\mathsf{fibss}\ n)[\mathrm{u};\mathrm{v}].\mathsf{plus}\ \mathrm{u}\ \mathrm{v} \\
\mathsf{fibss} & 0 & = & \langle \mathsf{s}0, \mathsf{s}0 \rangle \\
\mathsf{fibss} & \mathsf{s}n & = & (\mathsf{fibss}\ n)[\mathrm{u};\mathrm{v}].\langle \mathrm{v}, \mathsf{plus}\ \mathrm{u}\ \mathrm{v} \rangle
\end{array}
$$

[4]Exercise: compute this.

By design, the auxiliary function fibss computes exactly the information required to complete the double-successor case, and it does so by a one-step recursion. The main function is thus reduced to a case analysis.

In [Gim94], Giménez effectively generalises this technique to an encoding of recursion on **guarded** arguments, and it this technique which I present below.

DEFINITION: **guarded**

- if Con is a datatype constructor with non-recursive arguments $\vec{a}$ and recursive arguments $\vec{r}$, then each r_i is **guarded**[5] (by Con) in $\mathsf{Con}\,\vec{a}\,\vec{r}$
- if r is guarded in s and s is guarded in t, then r is guarded in t

The idea is to introduce an intermediate data structure which stores for each input the recursive values we need to compute the output. We may code this up as an elimination rule:

$$\Phi : \forall n\!:\!\mathbb{N}.\ \mathit{Type}$$

$$\dfrac{\begin{array}{c} \mathit{Aux}_\Phi\ n \\ \vdots \\ \Phi\ n \end{array}}{\forall \boxed{n\!:\!\mathbb{N}}.\ \Phi\ n}$$

Once we have applied this rule, case analysis on n allows us to split the subgoal into cases for the separate patterns we wish to treat: for each pattern p, we must prove $\Phi\ p$ using the information supplied in $\mathit{Aux}_\Phi\ p$.

Of course, to prove this rule, we shall have to be able to show

$$\forall n\!:\!\mathbb{N}.\ \mathit{Aux}_\Phi\ n$$

This proof will go by recursion on n: we must generate the auxiliary information for $\mathsf{s}n$ from the corresponding information for n. Just as in the Fibonacci function, we may carry over any information we need to keep, together with computing the new value in exactly the same way as the 'main' function does.

What should Aux_Φ be? Different depths of recursion necessitate different amounts of auxiliary information. For Fibonacci, we may choose

$$\begin{array}{lcl} \mathsf{Aux}_{Fib}\ \ 0 & = & \mathbb{1} \\ \mathsf{Aux}_{Fib}\ \ \mathsf{s}0 & = & \mathbb{1} \\ \mathsf{Aux}_{Fib}\ \mathsf{ss}n & = & (\mathsf{Fib}\ n)\times(\mathsf{Fib}\ \mathsf{s}n) \end{array}$$

[5] more carefully, if r_i is a higher-order recursive argument of type $\forall\vec{h}:\vec{H}.$, then it is $r_i\ \vec{h}$ which is guarded, for any $\vec{h}$

Stylish users may choose to develop their auxiliary data structure as they develop their function, for each follows the case analysis of the other.

More generally, we may give a single auxiliary structure suitable for all occasions. Giménez defines it inductively for a parametric Φ:

$$\frac{}{\mathbb{N}\mathsf{AuxData}_{\Phi}\ 0} \qquad \frac{\Phi\ n \quad \mathbb{N}\mathsf{AuxData}_{\Phi}\ n}{\mathbb{N}\mathsf{AuxData}_{\Phi}\ \mathsf{s}n}$$

For each datatype, the auxiliary mimics the constructors and recursion pattern. Each recursive argument is decorated with a Φ proof, so that for each element of the original type, the auxiliary stores Φ for all its proper subterms. Proofs of

$$\forall n\!:\!\mathbb{N}.\ (\mathbb{N}\mathsf{AuxData}_{\Phi}\ \mathsf{s}n) \to \Phi\ n$$

then go by case analysis on $\mathbb{N}\mathsf{AuxData}$, at the same time splitting the $\mathbb{N}$-patterns and surfacing the recursions for the exposed subterms.

My treatment differs only pragmatically, in that I compute the auxiliary structure rather than defining it inductively.

$$\begin{array}{lcl} \mathbb{N}\mathsf{Aux}\,\Phi\ \ 0 & = & \mathbb{1} \\ \mathbb{N}\mathsf{Aux}\,\Phi\ \mathsf{s}n & = & (\Phi\ n)\times(\mathbb{N}\mathsf{Aux}\,\Phi\ n) \end{array}$$

As case analysis feeds $\mathbb{N}\mathsf{Aux}$ constructor expressions, it unfolds like one of those wallets for people with too many credit cards, revealing the proofs of Φ for the exposed subterms. The **blunderbuss** tactic can be used to extract the required hypothesis, provided it can be identified from its type.

Let us try to prove

$$\forall n\!:\!\mathbb{N}.\ \mathbb{N}\mathsf{Aux}\,\Phi\ n$$

by induction on n. The base case is trivial. The step case is

$$\forall n\!:\!\mathbb{N}.\ (\mathbb{N}\mathsf{Aux}\,\Phi\ n) \to \mathbb{N}\mathsf{Aux}\,\Phi\ \mathsf{s}n$$

which reduces to

$$\forall n\!:\!\mathbb{N}.\ (\mathbb{N}\mathsf{Aux}\,\Phi\ n) \to (\Phi\ n)\times(\mathbb{N}\mathsf{Aux}\,\Phi\ n)$$

We can clearly establish the second component of the pair. This leaves the requirement

$$\forall n\!:\!\mathbb{N}.\ (\mathbb{N}\mathsf{Aux}\,\Phi\,n) \to \Phi\,n$$

Again using the 'hubris' technique, we may **postpone** and **discharge** this subgoal, we have the auxiliary generation lemma $\mathbb{N}\mathsf{AuxGen}$:

$$\frac{\forall n\!:\!\mathbb{N}.\ (\mathbb{N}\mathsf{Aux}\,\Phi\,n) \to \Phi\,n}{\forall n\!:\!\mathbb{N}.\ \mathbb{N}\mathsf{Aux}\,\Phi\,n}$$

and hence the elimination rule $\mathbb{N}\mathsf{Fix}$.

$\mathbb{N}\mathsf{Fix}$

$$\Phi : \forall n\!:\!\mathbb{N}.\ \mathit{Type}$$

$$\frac{\begin{array}{c}\mathbb{N}\mathsf{Aux}\,\Phi\,n\\ \vdots\\ \Phi\,n\end{array}}{\forall\boxed{n\!:\!\mathbb{N}}.\ \Phi\,n}$$

I shall give the general construction for simple types, then discuss extensions.

CONSTRUCTION: **guarded fixpoint**

Consider an inductive family of datatypes with c constructors as shown right.

$$\overline{\mathit{Ind} : \mathit{Type}}$$

$$\frac{\vec{a} : \vec{A}_j \quad \vec{x} : \{\mathit{Ind}\}^{r_j}}{\mathsf{Con}_j\,\vec{a}\,\vec{x} : \mathit{Ind}}$$

The $\vec{a}$ are non-recursive and the $\vec{x}$ are r_j recursive arguments. Let $\mathit{IndElim}$ be its standard elimination rule.

Let us fix the components to be supplied by the user and make holes for the components to be supplied by machine. A !-binding IndAUX helps us track the development of IndAux.

$\lambda\Phi$	:	$\forall x\!:\!\mathit{Ind}.\ \mathit{Type}$
$!\mathit{IndAUX}$	=	$\lambda x\!:\!\mathit{Ind}.\ \mathit{Type}$
$?\mathit{IndAux}$	:	$\forall x\!:\mathit{Ind}$ $\mathit{IndAUX}\,x$
λbody	:	$\forall x\ \ :\mathit{Ind}$ $\forall aux\!:\mathit{IndAux}\,x$ $\Phi\,x$
$?\mathit{IndAuxGen}$	:	$\forall x\!:\mathit{Ind}$ $\mathit{IndAux}\,x$
$?\mathit{IndFix}$	:	$\forall\boxed{x : \mathit{Ind}}$ $\Phi\,x$

We may immediately prove IndFix with

$$\mathit{IndFix} = \lambda x\!:\!\mathit{Ind}.\ \text{body}\ x\ (\mathit{IndAuxGen}\,x)$$

Now let us eliminate the x in both the auxiliary and its generator, aquiring a subgoal for each constructor. One is enough to illustrate the point, and reduces the subscript terror.

$$
\begin{array}{lll}
?\mathit{IndAux}_{con} & : & \forall \vec{a} : \vec{A} \\
 & & \forall \vec{x} : \{\mathit{Ind}\}^{r} \\
 & & \forall \vec{T} : \{\mathit{IndAUX}\, x_i\}^{r}_{i} \\
 & & \quad \mathit{IndAUX}\, (\mathsf{Con}\, \vec{a}\, \vec{x}) \\
!\mathit{IndAux} & = & \mathit{IndElim}\ \mathit{IndAUX}\ \mathit{IndAux}_{con} \ldots
\end{array}
$$

$$
\begin{array}{lll}
?\mathit{IndAuxGen}_{con} & : & \forall \vec{a} : \vec{A} \\
 & & \forall \vec{x} : \{\mathit{Ind}\}^{r} \\
 & & \forall \vec{t} : \{\mathit{IndAux}\, x_i\}^{r}_{i} \\
 & & \quad \mathit{IndAux}_{con}\, \vec{a}\, \vec{x}\, \vec{t} \\
!\mathit{IndAuxGen} & = & \mathit{IndElim}\ \mathit{IndAux}\ \mathit{IndAuxGen}_{con} \ldots
\end{array}
$$

To build IndAux_{con}, we introduce the arguments and return the iterated Σ of pair-types collecting, for each recursive argument x_i, both $\Phi\ x_i$ and T_i, which the lovely let-binding reminds us is really $\mathit{IndAux}\, x_i$.

For $\mathit{IndAuxGen}_{con}$, we introduce the arguments and return the corresponding iterated tuple of pairs, passing on the accumulated proof t_i and adding the next layer, computed by body.

$$
\begin{array}{lll}
!\mathit{IndAux}_{con} & = & \lambda \vec{a} : \vec{A} \\
 & & \lambda \vec{x} : \{\mathit{Ind}\}^{r} \\
 & & \lambda \vec{T} : \{\mathit{IndAUX}\, x_i\}^{r}_{i} \\
 & & \quad \Sigma\{(\Phi\ x_i) \times T_i\}^{r}_{i}
\end{array}
$$

$$
\begin{array}{lll}
!\mathit{IndAuxGen}_{con} & = & \lambda \vec{a} : \vec{A} \\
 & & \lambda \vec{x} : \{\mathit{Ind}\}^{r} \\
 & & \lambda \vec{t} : \{\mathit{IndAux}\, x_i\}^{r}_{i} \\
 & & \quad \big\langle \{\langle \text{body}\ x_i\ t_i ; t_i \rangle\}^{r}_{i} \big\rangle \\
!\mathit{IndAuxGen} & = & \mathit{IndElim}\ \mathit{IndAux}\ \mathit{IndAuxGen}_{con} \ldots
\end{array}
$$

Cutting IndAUX and the proofs of the subgoals, then discharging the fixed hypotheses, we are left with

$$
\begin{array}{lll}
!\mathit{IndAux} & = & \ldots \\
& : & \forall\Phi\colon \forall x : \mathsf{Ind}.\ \mathcal{T}ype \\
& & \forall x : \mathsf{Ind} \\
& & \quad \mathcal{T}ype \\
!\mathit{IndAuxGen} & = & \ldots \\
& : & \forall\Phi \quad : \forall \mathsf{Ind} : x.\ \mathcal{T}ype \\
& & \forall \mathrm{body} : \forall x \quad : \mathsf{Ind} \\
& & \qquad\qquad \forall aux : \mathit{IndAux}\, x \\
& & \qquad\qquad\quad \Phi\ x \\
& & \forall x \qquad : \mathsf{Ind} \\
& & \quad \mathit{IndAux}\, \vec{x} \\
!\mathit{IndFix} & = & \ldots \\
& : & \forall\Phi \quad : \forall x : \mathsf{Ind}.\ \mathcal{T}ype \\
& & \forall \mathrm{body} : \forall x \quad : \mathsf{Ind} \\
& & \qquad\qquad \forall aux : \mathit{IndAux}\, x \\
& & \qquad\qquad\quad \Phi\ x \\
& & \forall \boxed{x \quad : \quad \mathsf{Ind}} \\
& & \quad \Phi\ x
\end{array}
$$

$$
\boxed{\begin{array}{c}
\Phi : \forall x : \mathsf{Ind}.\ \mathcal{T}ype \\
\\
\mathit{IndAux}\,\Phi\, x \\
\vdots \\
\Phi\ x \\
\hline
\forall \boxed{x : \mathsf{Ind}}.\ \Phi\ n
\end{array}}
$$

The following conversions hold:

$$
\begin{array}{l}
\mathit{IndAux}\,\Phi\ (\mathsf{Con}\,\vec{a}\,\vec{x}) \ \cong\ \Sigma\{(\Phi\ x_i)\times(\mathit{IndAux}\,\Phi\ x_i)\}_i^r \\
\mathit{IndAuxGen}\,\Phi\ f\,(\mathsf{Con}\,\vec{a}\,\vec{x}) \ \cong \\
\langle\{\langle \mathit{IndFix}\,\Phi\ f\,x_i;\ \mathit{IndAuxGen}\,\Phi\ f\,x_i\rangle\}_i^r\rangle \\
\mathit{IndFix}\,\Phi\ f\,x \ \cong\ f\,(\mathit{IndAuxGen}\,\Phi\ f\,x)\ x
\end{array}
$$

For dependent families Fam, we have exactly the same construction, replacing Ind by $\overline{\mathit{Fam}}$ or some $\mathit{Fam}\,\vec{\imath}$ as appropriate:

$$
\begin{array}{ll}
\lambda\Phi & : \forall\vec{x}\!:\!\overline{\mathsf{Fam}}.\ \mathcal{Type} \\
!\mathsf{FamAUX} & = \lambda\vec{x}\!:\!\overline{\mathsf{Fam}}.\ \mathcal{Type} \\
!\mathsf{FamAux}_{con} & = \lambda\vec{a}:\vec{A} \\
 & \quad \lambda\vec{x}\colon \{\mathsf{Fam}\,\vec{s}_i\}_i^r \\
 & \quad \lambda\vec{T}\colon \{\mathsf{FamAUX}\,\vec{s}_i;x_i\}_i^r \\
 & \quad\ \ \Sigma\{(\Phi\,\vec{s}_i;x_i)\times T_i\}_i^r \\
!\mathsf{FamAux} & = \mathsf{FamElim}\ \mathsf{FamAUX}\ \mathsf{FamAux}_{con}\ldots \\
\lambda\text{body} & : \forall\vec{x}\quad : \overline{\mathsf{Fam}} \\
 & \quad \forall aux\colon \mathsf{FamAux}\,\vec{x} \\
 & \quad\ \ \Phi\,\vec{x} \\
!\mathsf{FamAuxGen}_{con} & = \lambda\vec{a}\colon\vec{A} \\
 & \quad \lambda\vec{x}\colon \{\mathsf{Fam}\,\vec{s}_i\}_i^r \\
 & \quad \lambda\vec{t}\colon \{\mathsf{FamAux}\,\vec{s}_i;x_i\}_i^r \\
 & \quad\ \ \langle\{\langle(\text{body}\,\vec{s}_i;x_i\ t_i);t_i\rangle\}_i^r\rangle \\
!\mathsf{FamAuxGen} & = \mathsf{FamElim}\ \mathsf{FamAux}\ \mathsf{FamAuxGen}_{con}\ldots \\
!\mathsf{FamFix} & = \lambda\vec{x}\!:\!\overline{\mathsf{Fam}}.\ \text{body}\ \vec{x}\ (\mathsf{FamAuxGen}\,\vec{x})
\end{array}
$$

If we have higher-order recursive arguments, we must abstract the pairs over them:

$$
\begin{array}{ll}
!\mathsf{IndAux}_{ho} & = \ldots \\
 & \quad \lambda x\!:\forall\vec{h}\!:\!\vec{H}.\ \mathsf{Ind} \\
 & \quad \ldots \\
 & \quad \lambda T\colon \forall\vec{h}\!:\!\vec{H}.\ \mathsf{IndAUX}\,(x\,\vec{h}) \\
 & \quad\ \ \Sigma\ldots\forall\vec{h}\!:\!\vec{H}.\ (\Phi\ (x\,\vec{h}))\times(T\,\vec{h}) \\
!\mathsf{IndAuxGen}_{ho} & = \ldots \\
 & \quad \lambda x\colon \forall\vec{h}\!:\!\vec{H}.\ \mathsf{Ind} \\
 & \quad \ldots \\
 & \quad \lambda t: \forall\vec{h}\!:\!\vec{H}.\ \mathsf{IndAux}\,(x\,\vec{h}) \\
 & \quad\ \ \Big\langle\ldots\lambda\vec{h}\!:\!\vec{H}.\ \Big\langle(\text{body}\ (x\,\vec{h})\ (t\,\vec{h}));(t\,\vec{h})\Big\rangle\Big\rangle
\end{array}
$$

Now that we have built these useful elimination rules, let us move on to consider the technology we need to solve the constraints which arise when we use them for dependent subfamilies.

Chapter 5

Equality and Object-Level Unification

This chapter examines different notions of propositional equality in Type Theory, together with the forms of equational reasoning they support.

In particular, I shall give a formal treatment of the $\simeq$ predicate which I have been exploiting glibly until now: it is merely a convenient packaging of Martin-Löf's identity type together with the 'uniqueness of identity proofs' axiom proposed by Altenkirch and Streicher [Str93]. The reason for reformulating equality in this way is to improve the treatment of equality for *sequences* of terms in the presence of type dependency.

Once we have a definition of equality we can work with, the task is then to build a tactic, **simplify**, which solves first-order constructor form equations appearing as premises to goals. To achieve this, we will need to construct still more machinery for each inductive datatype:

- a proof that constructors are injective and disjoint
- a disproof of cyclic equations like $n \simeq \mathsf{s}\, n$

5.1 two nearly inductive definitions of equality

5.1.1 Martin-Löf's identity type

$$\frac{a, b : A}{a{=}b : \mathit{Prop}}$$

$$\frac{a : A}{\mathsf{refl}_{=}\; a : a{=}a}$$

$$\mathsf{idElim}\; A\; a\; \Phi\; \phi_{\mathit{refl}}\; a\; (\mathsf{refl}_{=}\; a) \;\leadsto_{\iota}\; \phi_{\mathit{refl}}$$

idElim

$\Phi : \forall b{:}A.\; (a{=}b) \to \mathit{Type}$

$$\frac{\Phi\; a\; (\mathsf{refl}_{=}\; a)}{\forall b{:}A.\; \forall \boxed{q : a{=}b}.\; \Phi\; b\; q}$$

idElim is known in the business as 'J', for historical reasons.

We may easily prove that this equality is substitutive in the usual sense.

$\mathsf{idSubst}$

$\Phi : A \to \mathit{Type}$

$$\frac{\Phi\; a}{\forall b{:}A.\; \forall \boxed{q : a{=}b}.\; \Phi\; b}$$

The proof fixes Φ and the proof of the single case, then applies **eliminate** with idElim. The generated scheme makes no use of the equation's proof—$\mathsf{idSubst}$ is 'proof irrelevant'.

It will prove convenient to have some sugar for applications of $\mathsf{idSubst}$:

- substitution

  $[q]_{=}^{\Phi} s \implies \mathsf{idSubst}\; A\; a\; \Phi\; s\; b\; q$

  $$\frac{q : a{=}b \quad s : \Phi\; a}{[q]_{=}^{\Phi} s : \Phi\; b}$$

- coercion

  $[q]_{=} s \implies [q]_{=}^{\lambda T{:}\mathit{Type}.\; T} s$

  $$\frac{q : S{=}T \quad s : S}{[q]_{=} s : T}$$

The computational behaviour of $\mathsf{idSubst}$ follows from that of idElim:

$$[\mathsf{refl}_{=}\; a]_{=}^{\Phi} t \;\triangleright\; t$$

5.1.2 uniqueness of identity proofs

Altenkirch and Streicher suggest that = should be equipped with the additional elimination rule shown, together with its computational behaviour.

$$\mathsf{idUnique}\ A\ a\ \Phi\ \phi_{\mathit{refl}}\ (\mathsf{refl}_{=}\ a) \rightsquigarrow_{\iota} \phi_{\mathit{refl}}$$

$$\boxed{\begin{array}{l}\mathsf{idUnique} \\ \\ \Phi : (a{=}a) \to \mathit{Type} \\ \\ \dfrac{\Phi\ (\mathsf{refl}_{=}\ a)}{\forall \boxed{q : a{=}a}.\ \Phi\ q}\end{array}}$$

This rule is sometimes known in the business as 'K', largely because it comes after 'J'.[1]

For a given element type, A, the aperture of idElim, ie the space of equations over which its scheme must range is two dimensional: $A \times A$. However, $\mathsf{idUnique}$'s scheme ranges only over the diagonal. Of course, it is only the diagonal which is inhabited.

Hofmann and Streicher have shown that $\mathsf{idUnique}$ is not derivable from idE-lim[HoS94]. On the other hand, Streicher adds that idElim is unnecessary if $\mathsf{idSubst}$ and $\mathsf{idUnique}$ are taken as axiomatic: we may first use $\mathsf{idSubst}$ to replace b by a, say, then $\mathsf{idUnique}$ to reduce the remaining arbitrary proof of $a{=}a$ to $(\mathsf{refl}_{=}\ a)$. Effectively, we divide the idElim process into two phases: the proof irrelevant phase ($\mathsf{idSubst}$) reduces the = family to its inhabited subfamily of reflexive equations, so the proof relevant phase ($\mathsf{idUnique}$) need only be concerned with that restricted case.

5.1.3 $\simeq$, or 'John Major' equality

It is now time to reveal the definition of $\simeq$, the 'John Major' equality relation.[2] John Major's 'classless society' widened people's *aspirations* to equality, but also the gap between rich and poor. After all, aspiring to be equal to others than oneself is the politics of envy. In much the same way, $\simeq$ forms equations between members of any type, but they cannot be treated as equals (ie substituted) unless they are of the same type. Just as before, each thing is only equal to itself.

[1] Aficionados of the trombone might fondly imagine that the two rules are named after legendary jazz duo J.J. Johnson and Kai Winding. I do not propose to pour cold water on this explanation.

[2] John Major was the last ever leader of the Conservative Party to be Prime Minister (1990 to 1997) of the United Kingdom, in case he has slipped your mind.

$$\frac{a : A \quad b : B}{a \simeq b : \mathcal{P}rop}$$

$$\frac{a : A}{\mathsf{refl}\, a : a \simeq a}$$

$$\mathsf{eqElim}\, A\, a\, \Phi\, \phi_{refl}\, a\, (\mathsf{refl}\, a) \rightsquigarrow_\iota \phi_{refl}$$

eqElim

$$\Phi : \forall a' \!:\! A.\ (a \simeq a') \to \mathcal{T}ype$$

$$\frac{\Phi\ a\ (\mathsf{refl}\, a)}{\forall a' \!:\! A.\ \forall \boxed{l : a \simeq a'}.\ \Phi\ a'\ l}$$

Observe that eqElim is not the elimination rule which one would expect if $\simeq$ was inductively defined.

The 'usual' rule eliminates over all the formable equations, and it is quite useless: it cannot be used to substitute two values of the same type because the scheme must be abstracted over an arbitrary type.

eqIndElim

$$\Phi : \forall B \!:\! \mathcal{T}ype.\ \forall b \!:\! B.\ (a \simeq b) \to \mathcal{T}ype$$

$$\frac{\Phi\ A\ a\ (\mathsf{refl}\, a)}{\forall B \!:\! \mathcal{T}ype.\ \forall b \!:\! B.\ \forall \boxed{e : a \simeq b}.\ \Phi\ B\ b\ e}$$

By contrast, eqElim eliminates only over the subfamily where the two types are the same, the 'type diagonal': of course, all the inhabitants lie in this subfamily.

5.1.4 equality for sequences

The reason for adopting $\simeq$ rather than = when working with dependent types can be seen clearly when we attempt to extend the notion of equality to cover not just two terms in a type but two sequences of terms in a telescope. Suppose we have $\vec{r}, \vec{s} : \vec{T}$ for some $\vec{x}$-telescope $\vec{T}$. We may not, in general, state the equality of sequences $\vec{r}$ and $\vec{s}$ as

$$r_1 = s_1; r_2 = s_2; \ldots \qquad (\times)$$

since $r_2 : T_2[r_1]$ while $s_2 : T_2[s_1]$, and these may be different.

There is, of course, nothing to stop us writing

$$r_1 \simeq s_1; r_2 \simeq s_2; \ldots$$

which will henceforth be abbreviated as the **telescopic equation** $\vec{r} \simeq \vec{s}$.

We may correspondingly abbreviate the sequence of reflexivity proofs

$$(\mathsf{refl}\, r_1); (\mathsf{refl}\, r_2); \ldots$$

by refl $\vec{r}$.

Let us not stop at that: in fact, we may prove substitutivity and uniqueness for telescopic equations.

CONSTRUCTION: **telescopic substitution**

For each natural number n, we may derive a substitution principle for telescopic equations of length n.

$$\begin{array}{l} \mathsf{eqSubst}_n \\ \Phi : \vec{T} \to \mathit{Type} \\ \dfrac{\Phi\ \vec{r}}{\forall \vec{s} : \vec{T}.\ \forall \boxed{\vec{e} : \vec{r} \simeq \vec{s}}.\ \Phi\ \vec{s}} \end{array}$$

The reduction behaviour will be as follows:

$$\mathsf{eqSubst}_n\ \vec{T}\ \vec{r}\ \Phi\ \phi_{\mathit{refl}}\ \vec{r}\ (\mathsf{refl}\ \vec{r}) \ \triangleright\ \phi_{\mathit{refl}}$$

The construction is by recursion on n, effectively iterating eqElim.

The zero case is proved by the polymorphic identity function. Clearly the reduction behaviour is correct.

$$\begin{array}{l} \mathsf{eqSubst}_0 \\ \Phi : \mathit{Type} \\ \dfrac{\Phi}{\Phi} \end{array}$$

Now, assuming we have already constructed $\mathsf{eqSubst}_n$, let us construct $\mathsf{eqSubst}_{n+1}$.

$$\begin{array}{l} \mathsf{eqSubst}_{n+1} \\ \Phi : T; \vec{T} \to \mathit{Type} \\ \dfrac{\Phi\ r; \vec{r}}{\forall s; \vec{s} : T; \vec{T}.\ \forall \boxed{e; \vec{e} : r; \vec{r} \simeq s; \vec{s}}.\ \Phi\ s; \vec{s}} \end{array}$$

Fixing $T; \vec{T}$, $r; \vec{r}$, Φ and the proof of $\Phi\ r; \vec{r}$, we have the goal shown.

$$\begin{array}{l} ?\text{goal} : \forall s; \vec{s}\ : T; \vec{T} \\ \qquad \forall \boxed{e}; \vec{e} : r; \vec{r} \simeq s; \vec{s} \\ \qquad \Phi\ s; \vec{s} \end{array}$$

Now, e is a proof that $r \simeq s$, where both have type T, hence we may eliminate e by eqElim. The generated scheme includes all the $\vec{s}$ and $\vec{e}$:

$$\lambda s : T.\ \lambda \_ : r \simeq s.\ \forall \vec{s} : \vec{T}.\ \forall \vec{e} : \vec{r} \simeq \vec{s}.\ \Phi\ s; \vec{s}$$

Note that, as nothing depends on e, the proof relevance of eqElim is not necessary for this construction, just as in the construction of idSubst from idElim.

The elimination leaves with the subgoal shown.

$$?\text{subgoal}: \forall \vec{s} \; : (T;\vec{T})\; r \quad \forall \boxed{\vec{e} \; : \; \vec{r} \simeq \vec{s}} \quad \Phi\; r;\vec{s}$$

Note that $(T;\vec{T})\; r$ is just $[r/x]\vec{T}$, which is exactly the telescope of $\vec{r}$.

Now that $\vec{r}$ and $\vec{s}$ have the same telescope, we may eliminate the remaining $\vec{e}$ by $\mathsf{eqSubst}_n$: the scheme is just $\Phi\; r$. This leaves us with the subgoal $\Phi\; r;\vec{r}$, a proof of which we fixed in the context.

From the ι-reduction associated with eqElim and then the inductive hypothesis, we may deduce that

$$\begin{array}{l}
\mathsf{eqSubst}_{n+1}\; T;\vec{T}\; r;\vec{r}\; \Phi\; \phi_r\; r;\vec{r}\; (\mathsf{refl}\; r);(\mathsf{refl}\; \vec{r}) \rhd \\
\mathsf{eqElim}\; T\; r\; (\ldots)\; (\mathsf{eqSubst}_n \ldots)\; r\; (\mathsf{refl}\; r) \ldots \rhd \\
\mathsf{eqSubst}_n\; ((T;\vec{T})\; r)\; \vec{r}\; (\Phi\; r)\; \phi_r\; \vec{r}\; (\mathsf{refl}\; \vec{r}) \rhd \\
\phi_r
\end{array}$$

Observe that the same proof structure also yields substitutivity in the other direction.

Although the roles of $\vec{r}$ and $\vec{s}$ are reversed, we may still fix the $\vec{r}$ and abstract over the $\vec{s}$ (the right hand sides) as required by eqElim.

$\mathsf{eqSubstLR}_n$

$$\Phi : \vec{T} \to \mathit{Type}$$

$$\frac{\Phi\; \vec{s}}{\forall \vec{r} : \vec{T}.\; \forall \boxed{\vec{e} : \vec{r} \simeq \vec{s}}.\; \Phi\; \vec{r}}$$

CONSTRUCTION: **telescopic uniqueness**

For each natural number n, we may derive a substitution principle for telescopic equations of length n.

The reduction behaviour will be as follows:

$\mathsf{eqUnique}_n$

$$\Phi : \vec{t} \simeq \vec{t} \to \mathit{Type}$$

$$\frac{\Phi\; (\mathsf{refl}\; \vec{t})}{\forall \boxed{\vec{e} : \vec{t} \simeq \vec{t}}.\; \Phi\; \vec{e}}$$

$$\mathsf{eqUnique}_n\; \vec{T}\; \vec{r}\; \Phi\; \phi_{\mathit{refl}}\; \vec{r}\; (\mathsf{refl}\; \vec{r}) \;\rhd\; \phi_{\mathit{refl}}$$

This construction also proceeds by recursion on n, again with polymorphic identity as the base case. The step case is slightly more subtle than for $\mathsf{eqSubst}$.

Suppose we have already constructed $\mathsf{eqUnique}_n$: let us construct $\mathsf{eqUnique}_{n+1}$.

This time we fix everything except the proofs of the equations.

?goal: $\forall \boxed{e}; \vec{e}\colon t; \vec{t} \simeq t; \vec{t}$
$\Phi\ e; \vec{e}$

We have little choice but to eliminate e with eqElim. Perforce, this introduces equational constraints in the scheme:

$$\lambda s\!:\!T.\ \lambda l'\!:\!t \simeq s.\ \forall e; \vec{e}\!:\!t; \vec{t} \simeq t; \vec{t}.\ (s \simeq t) \to (l' \simeq e) \to \Phi\ e; \vec{e}$$

Neither of these constraints is disposable, since e definitely occurs in the goal, and, in general, we may expect t to occur (implicitly) in the types of the $\vec{e}$.

Consequently, the subgoal we get is as shown.

We may discard e', then eliminate E with $\mathsf{eqSubst}_1$.

?subgoal: $\forall e; \vec{e}\colon t; \vec{t} \simeq t; \vec{t}$
$\forall e' \ : t \simeq t$
$\forall \boxed{E}\colon \mathsf{refl}\ t \simeq e$
$\Phi\ e; \vec{e}$

Now we are ready to appeal to $\mathsf{eqUnique}_n$, with scheme $\Phi\ (\mathsf{refl}\ t)$.

?subgoal: $\forall \boxed{\vec{e}}\colon \vec{t} \simeq \vec{t}$
$\Phi\ (\mathsf{refl}\ t); \vec{e}$

This turns the remaining $\vec{e}$ into $(\mathsf{refl}\ \vec{t})$, so that the fixed proof of $\Phi\ (\mathsf{refl}\ t)$ completes our obligations.

As far as the reduction behaviour is concerned, forgive me if I omit the detail. The construction successively applies elimination rules for equations which reduce to their single subgoals when those equations are instantiated with reflexivity. Consequently, each $\mathsf{eqUnique}_n$ inherits this behaviour.

It is not impossible to build a notion of telescopic equality with substitution using $=$, but it is considerably more cumbersome. The method forces each equation to typecheck, by explicit appeal to the substitution operator for the prefixed equations. That is, we need the first n operators in order to formulate a telescopic equation of length $n + 1$, let alone establish its own substitutivity. Furthermore, in order to make the step in the construction, it is not sufficient simply to substitute for the first equation with $\mathsf{idSubst}$, but rather we must eliminate it with idElim, not only substituting the terms, but also instantiating the proof with reflexivity, allowing the substitutions repairing the remainder of the equations to reduce. By adopting $\simeq$, we achieve at least this telescopic extension without acquiring proof relevant dirt under our fingernails.

5.1.5 the relationship between = and $\simeq$

Having argued for the practicality of using $\simeq$ instead of = when working with dependent types, I nonetheless feel obliged to point out that the two are equivalent—provided we mean = equipped with idUnique. Let me now give the mutual construction. First, the easy direction:

CONSTRUCTION: **= from $\simeq$**

This is so easy that I will just tell you the answers—by construction, = is just telescopic equation for telescopes of length 1.

$$
\begin{array}{ll}
!= & = \lambda A \;\; : \mathit{Type} \\
 & \quad \lambda a, \mathsf{b} \colon A \\
 & \qquad a{\simeq}b \\
 & : \forall A \;\; : \mathit{Type} \\
 & \quad \forall a, \mathsf{b} \colon A \\
 & \qquad \mathit{Prop} \\
!\mathsf{refl}_= & = \mathsf{refl} \\
 & : \forall A \colon \mathit{Type} \\
 & \quad \forall a : A \\
 & \qquad a{=}a \\
!\mathsf{idSubst} & = \mathsf{eqSubst}_1 \\
 & : \forall A \;\; : \mathit{Type} \\
 & \quad \forall a \;\; : A \\
 & \quad \forall \Phi \;\; : A \to \mathit{Type} \\
 & \quad \forall \phi \;\; : \Phi\, a \\
 & \quad \forall b \;\; : A \\
 & \quad \forall \boxed{e \;:\; a{=}b} \\
 & \qquad \Phi\, b \\
!\mathsf{idUnique} & = \mathsf{eqUnique}_1 \\
 & : \forall A \;\; : \mathit{Type} \\
 & \quad \forall a \;\; : A \\
 & \quad \forall \Phi \;\; : a{\simeq}a \to \mathit{Type} \\
 & \quad \forall \phi \;\; : \Phi\, (\mathsf{refl}_=\, a) \\
 & \quad \forall \boxed{e \;:\; a{=}a} \\
 & \qquad \Phi\, e
\end{array}
$$

Furthermore, the reduction behaviour for idSubst and idUnique is exactly that for $\mathsf{eqSubst}_1$ and $\mathsf{eqUnique}_1$.

The other direction is the interesting one.

CONSTRUCTION: **$\simeq$ from = with idUnique**

Let us assume we have = and construct:

$$
\begin{array}{ll}
?\simeq & : \forall A : \mathit{Type} \\
 & \;\; \forall a : A \\
 & \;\; \forall \mathsf{B} : \mathit{Type} \\
 & \;\; \forall \mathsf{b} : \mathsf{B} \\
 & \;\;\;\; \mathit{Prop} \\
?\mathsf{refl} & : \forall A : \mathit{Type} \\
 & \;\; \forall a : A \\
 & \;\;\;\; a \simeq a \\
?\mathsf{eqElim} & : \forall A \;\; : \mathit{Type} \\
 & \;\; \forall a \;\; : A \\
 & \;\; \forall \Phi \;\; : \forall a' : A.\; a \simeq a' \to \mathit{Type} \\
 & \;\; \forall \phi \;\; : \Phi\; a\; (\mathsf{refl}\; a) \\
 & \;\; \forall a' \;\; : A \\
 & \;\; \forall \boxed{e \;:\; a \simeq a'} \\
 & \;\;\;\; \Phi\; a'\; l
\end{array}
$$

Let us first make a little abbreviation:

$$
\mathsf{cell} \implies \Sigma A : \mathit{Type}.\; A
$$

cell packages up a typed term. The idea is that $\simeq$ is just $=$ for cells:

$$
\begin{array}{ll}
!\simeq & = \lambda A : \mathit{Type} \\
 & \;\;\; \lambda a : A \\
 & \;\;\; \lambda \mathsf{B} : \mathit{Type} \\
 & \;\;\; \lambda \mathsf{b} : \mathsf{B} \\
 & \;\;\;\;\; \langle A; a \rangle = \langle \mathsf{B}; \mathsf{b} \rangle \\
!\mathsf{refl} & = \lambda A : \mathit{Type} \\
 & \;\;\; \lambda a : A \\
 & \;\;\;\;\; \mathsf{refl}_{=}\; \langle A; a \rangle
\end{array}
$$

This makes the elimination rule

$$
\begin{array}{ll}
?\mathsf{eqElim} & : \forall A \;\; : \mathit{Type} \\
 & \;\; \forall a \;\; : A \\
 & \;\; \forall \Phi \;\; : \forall b : A \\
 & \;\;\;\;\;\;\;\;\;\; \forall e : \langle A; a \rangle = \langle A; b \rangle \\
 & \;\;\;\;\;\;\;\;\;\;\;\; \mathit{Type} \\
 & \;\; \forall \phi \;\; : \Phi\; a\; (\mathsf{refl}_{=}\; \langle A; a \rangle) \\
 & \;\; \forall b \;\; : A \\
 & \;\; \forall \boxed{e \;:\; \langle A; a \rangle = \langle A; b \rangle} \\
 & \;\;\;\; \Phi\; b\; e
\end{array}
$$

If we could only deduce $a{=}b$ from e, we would be most of the way there. For that, we need a proof that equal cells have equal second projections.

The equivalence of idUnique and equality of second projections from dependent pairs is folklore knowledge, but I shall do the work nonetheless.

It is even difficult to state the equality of the second projections, because they are not of convertible types—we must use the substitutivity of equality to make a type coercion.

The lemma we need is as shown. Let us **claim** it globally and work on the main goal.

$$
\begin{array}{ll}
?\mathsf{sproj}: & \forall Aa, Bb : \mathsf{cell} \\
& \forall e \quad : Aa{=}Bb \\
& Aa[A, a] \\
& Bb[B, b] \\
& \forall q \quad : A{=}B \\
& \quad ([q]_{=} a){=}b
\end{array}
$$

Observe that

$$\mathsf{sproj}\ e : \forall q : A{=}A.\ ([q]_{=} a){=}b$$

so that

$$\mathsf{sproj}\ e\ (\mathsf{refl}_{=}\ A) : a{=}b$$

Let us exploit this discovery. Introducing all the hypotheses, this is the goal we now must solve.

$$
\begin{array}{ll}
!\mathrm{ab} & = \mathsf{sproj}\ e\ (\mathsf{refl}_{=}\ A) \\
& :\ a{=}b \\
?\mathrm{goal}: & \Phi\ b\ e
\end{array}
$$

As the type of e contains b, it is wise to reabstract it:

We may now **eliminate** ab by idSubst.

$$
\begin{array}{ll}
!\boxed{ab} & = \mathsf{sproj}\ e\ (\mathsf{refl}_{=}\ A) \\
& \quad a{=}b \\
?\mathrm{goal}' : & \forall e' : \langle A; a\rangle = \langle A; b\rangle \\
& \quad \Phi\ b\ e' \\
!\mathrm{goal}\ : & \mathrm{goal}'\ e
\end{array}
$$

Now e' is a reflexive equation!

We may **eliminate** it by idUnique.

$$
\begin{array}{ll}
?\mathrm{subgoal}: & \forall e' : \langle A; a\rangle = \langle A; a\rangle \\
& \quad \Phi\ a\ e'
\end{array}
$$

The subgoal we acquire follows from ϕ.

$$?\mathrm{immediate}: \Phi\ a\ (\mathsf{refl}_{eq}\ \langle A; a\rangle)$$

All that remains is to prove sproj. Firstly, we **eliminate** the equation on the cells, e with idSubst.

Although the two pairs unpacked by the binding sugar are the same, we have two names for each projection. We can clear this up by eliminating Aa, reducing the projections and cutting the sugared !-bindings.

$$\begin{array}{l} ?\mathrm{same} \colon \forall \boxed{Aa} \colon \mathsf{cell} \\ \quad Aa[A,a] \\ \quad Aa[B,b] \\ \quad \forall q \quad : A{=}B \\ \quad\quad ([q]_{=} a){=}b \end{array}$$

Now we may use $\mathsf{idUnique}$ to remove the reflexive q.

$$\begin{array}{l} ?\mathrm{open} \colon \forall A \colon \mathit{Type} \\ \quad \forall a : A \\ \quad \forall q : A{=}A \\ \quad\quad ([q]_{=} a){=}a \end{array}$$

The remaining subgoal has exactly the type of $\mathsf{refl}_{=}$!

$$\begin{array}{l} ?\mathrm{refl} \colon \forall A \colon \mathit{Type} \\ \quad \forall A \colon a \\ \quad\quad a{=}a \end{array}$$

As far as reduction behaviour is concerned, first observe that

$$\mathsf{sproj}\ (\mathsf{refl}_{=}\ \langle A; a\rangle)\ (\mathsf{refl}_{=}\ A)\ \cong\ (\mathsf{refl}_{=}\ a)$$

This is because sproj eliminates in succession the first equation, the cell, then the second equation, and all three are in constructor form. Consequently, when eqElim is applied to $(\mathsf{refl}\ a)$, the computed equality proof ab turns out to be $(\mathsf{refl}_{=}\ a)$. Since both these equations are reflexive, both the $\mathsf{idSubst}$ and $\mathsf{idUnique}$ steps reduce as required.

5.2 first-order unification for constructor forms

A typical application of an elimination rule with scheme variable $\Phi : \forall \vec{\imath} \colon \vec{I}.\ \mathit{Type}$ will engender a scheme

$$\Phi\ =\ \lambda\vec{\imath}.\ \forall\vec{x}.\ \vec{\imath}{\simeq}\vec{t}[\vec{x}] \rightarrow \Psi$$

Correspondingly, cases of form

$$\begin{array}{c} \vec{y} : \vec{Y} \\ \cdots\cdots\cdot \\ \Phi\ \vec{s}[y] \end{array}$$

yield subgoals in the proof of form

$$\forall\vec{y}.\ \forall\vec{x}.\ \vec{s}[y]\simeq\vec{t}[\vec{x}] \to \Psi$$

The equational constraints constitute a unification problem: if there is no solution, then the goal follows vacuously; if there is a most general unifier, we may use it to instantiate the $\vec{y}$ and $\vec{x}$.

Suppose, for example, we wish to write the 'vector tail' function, whose type prevents application to a null vector:

$$\begin{array}{l} ?\mathsf{vtail}\colon \forall n\colon \mathbb{N} \\ \qquad\quad \forall \mathrm{v}\colon \mathsf{vect}\ (\mathsf{s}n) \\ \qquad\quad\ \ \mathsf{vect}\ n \end{array}$$

Note that I have fixed the element type A to avoid clutter.

Eliminating v with $\mathsf{vectCase}$ creates a constrained scheme

$$\lambda i\colon\mathbb{N}.\ \lambda x\colon\mathsf{vect}\ i.\ \forall n\colon\mathbb{N}.\ \forall v\colon\mathsf{vect}\ (\mathsf{s}n).\ i\simeq\mathsf{s}n\ \to\ x\simeq v\ \to\ \mathsf{vect}\ n$$

The corresponding subgoals are as shown.

The $\mathsf{vtail}_{\mathsf{vnil}}$ subgoal features the impossible premise that zero equals a successor, whilst in the $\mathsf{vtail}_{\mathsf{vcons}}$ case the equations conveniently constrain the type of the tail to be the return type of the function.

$$\begin{array}{ll} ?\mathsf{vtail}_{\mathsf{vnil}} & \colon \forall n : \mathbb{N} \\ & \ \ \forall v : \mathsf{vect}\ (\mathsf{s}n) \\ & \ \ \forall e_1\colon 0\simeq\mathsf{s}n \\ & \ \ \forall e_2\colon x\simeq v \\ & \ \ \ \ \mathsf{vect}\ n \\ ?\mathsf{vtail}_{\mathsf{vcons}} & \colon \forall m\colon \mathbb{N} \\ & \ \ \forall h : A \\ & \ \ \forall t\ : \mathsf{vect}\ m \\ & \ \ \forall n : \mathbb{N} \\ & \ \ \forall v : \mathsf{vect}\ (\mathsf{s}n) \\ & \ \ \forall e_1\colon \mathsf{s}m\simeq\mathsf{s}n \\ & \ \ \forall e_2\colon x\simeq v \\ & \ \ \ \ \mathsf{vect}\ n \end{array}$$

If we could solve these unification problems, we would be left with this goal.

$$\begin{array}{ll} ?\mathsf{vtail}_{\mathsf{vcons}} & \colon \forall n\colon \mathbb{N} \\ & \ \ \forall h\colon A \\ & \ \ \forall t : \mathsf{vect}\ n \\ & \ \ \ \ \mathsf{vect}\ n \end{array}$$

We would then introduce the arguments and return the tail.

The task does seem to hinge on solving the unification problems generated in the course of elimination. In [McB96], I presented a tactic ('Qnify') for solving such problems, provided the terms comprised constructor forms in simple datatypes. I shall largely follow that treatment, extending the same procedure to dependent datatypes.

5.2.1 transition rules for first-order unification

The 'Qnify' tactic operates by successively eliminating from the goal hypothetical equations between constructor forms:

$$\forall \vec{x}.\, s \simeq t \to \Phi$$

DEFINITION: **constructor form**

t is a **constructor form** over variable set V if either

- $t \in V$
- $t \equiv \mathsf{con}\, \vec{t}$
  where each t_i is a constructor form over V

In the above goal, suppose s and t have the same type and are constructor forms over the $\vec{x}$. We may distinguish six possibilities by the following decision table:[3]

$s \quad \simeq \quad t$	x	$\mathsf{cheese}\, \vec{t}$
x	**identity**	if $x \in \vec{t}$ then **cycle**
y	**coalescence**	else **substitution**
$\mathsf{chalk}\, \vec{s}$	apply	**conflict**
$\mathsf{cheese}\, \vec{s}$	symmetry	**injectivity**

For each of these six kinds of constructor equation, there is an elimination rule. They are shown in table 5.1

These six rules, once we have proven them, will constitute the transition rules of a unification algorithm which is complete for the following class of problem:

DEFINITION: **constructor form unification problem**

A **constructor form unification problem** is a goal of form:

$$\forall \vec{x}.\, \vec{s} \simeq \vec{t} \to \Phi[\vec{x}]$$

where the $\vec{s}$ and $\vec{t}$ are sequences of constructor forms over $\vec{x}$ inhabiting some telescope T

[3] chalk and cheese are constructors as different as chalk and cheese.

identity	$\Phi : \mathcal{T}\!ype$ $\dfrac{\Phi}{\boxed{x \simeq x} \;\to\; \Phi}$	
coalescence	$\Phi : T \to \mathcal{T}\!ype$ $\dfrac{\Phi\ x}{\forall y{:}T.\ \boxed{y \simeq x} \;\to\; \Phi\ y}$	
cycle	$\Phi : \mathcal{T}\!ype$ $\dfrac{}{\boxed{x \simeq \mathsf{cheese}\ \vec{t}} \;\to\; \Phi}$	$x \in \vec{t}$
substitution	$\Phi : T \to \mathcal{T}\!ype$ $\dfrac{\Phi\ \mathsf{cheese}\ \vec{t}}{\forall x{:}T.\ \boxed{x \simeq \mathsf{cheese}\ \vec{t}} \;\to\; \Phi\ x}$	$x \notin \vec{t}$
conflict	$\Phi : \mathcal{T}\!ype$ $\dfrac{}{\boxed{\mathsf{chalk}\ \vec{s} \simeq \mathsf{cheese}\ \vec{t}} \;\to\; \Phi}$	
injectivity	$\Phi : \mathcal{T}\!ype$ $\dfrac{\vec{s} \simeq \vec{t} \to \Phi}{\boxed{\mathsf{cheese}\ \vec{s} \simeq \mathsf{cheese}\ \vec{t}} \;\to\; \Phi}$	

Table 5.1: elimination rules for constructor form equations

Since $s_1, t_1 : T_1$, the leading equation has both sides the same type, so that exactly one of the above rules must apply (using symmetry if necessary). We must also check that each of these rules preserves this structure.

LEMMA: **transition rules preserve problem structure**

> Given a constructor form unification problem
>
> $$\forall\vec{x}.\ \forall e; \vec{e}: s; \vec{s} \simeq t; \vec{t}.\ \Phi[\vec{x}]$$
>
> eliminating e by the appropriate transition rule either solves the goal or leaves a subgoal which is also a constructor form unification problem.

PROOF

Let us check, rule by rule:

- **identity**

  Before, we have

  $$\begin{aligned} ?before\colon & \forall\vec{x}\colon \vec{X} \\ & \forall e\colon x_i \simeq x_i \\ & \forall\vec{e}\colon \vec{s} \simeq \vec{t} \\ & \quad \Phi \end{aligned}$$

  where $x_i; \vec{s}, x_i; \vec{t} : \vec{T}$. Afterwards, we have

  $$\begin{aligned} ?after\colon & \forall\vec{x}\colon \vec{X} \\ & \forall\vec{e}\colon \vec{s} \simeq \vec{t} \\ & \quad \Phi \end{aligned}$$

  where $\vec{s}, \vec{t} : \vec{T}\, x_i$. Since the variable set is unchanged, $\vec{s}$ and $\vec{t}$ are still constructor forms.

- **coalescence** and **substitution**

  Up to a permutation of the goal (performed by the elimination tactic) we start with

  $$\begin{aligned} ?before\colon & \forall\vec{x}\colon \vec{X} \\ & \forall x\colon T_1 \\ & \forall e\colon x \simeq t \\ & \forall\vec{y}\colon \vec{Y}[x] \\ & \forall\vec{e}\colon \vec{s}[x] \simeq \vec{t}[x] \\ & \quad \Phi[x] \end{aligned}$$

where $x \notin t$ and $x; \vec{s}, t; \vec{t} : \vec{T}$ After elimination, we have

$$\begin{array}{ll} ?\mathit{after}\colon & \forall \vec{x}\colon \vec{X} \\ & \forall \vec{y}\colon \vec{Y}[t] \\ & \forall \vec{e}\colon \vec{s}[t] \simeq \vec{t}[t] \\ & \quad \Phi[t] \end{array}$$

Although, x has vanished from the variable set, it has been replaced by constructor form t which does not contain x. As for the remaining problem, $\vec{s}[t], \vec{t}[t] : \vec{T}\, t$.

- **cycle** and **conflict**

  There are no subgoals.

- **injectivity**

  Before:

$$\begin{array}{ll} ?\mathit{before}\colon & \forall \vec{x}\colon \vec{X} \\ & \forall e\colon \mathsf{cheese}\, \vec{s}' \simeq \mathsf{cheese}\, \vec{t}' \\ & \forall \vec{e}\colon \vec{s} \simeq \vec{t} \\ & \quad \Phi \end{array}$$

  where $(\mathsf{cheese}\vec{s}'); \vec{s}, (\mathsf{cheese}\vec{t}'); \vec{t} : \vec{T}$. Now, the type of constructor cheese must be

$$\forall \vec{y} : \vec{Y}.\ T[\vec{y}]$$

  with $\vec{s}', \vec{t}' : \vec{Y}$. After elimination:

$$\begin{array}{ll} ?\mathit{after}\colon & \forall \vec{x} : \vec{X} \\ & \forall \vec{e}' : \vec{s}' \simeq \vec{t}' \\ & \forall \vec{e} : \vec{s} \simeq \vec{t} \\ & \quad \Phi \end{array}$$

  Certainly, the problem still consists of constructor forms over the $\vec{x}$. Furthermore, both $\vec{s}'; \vec{s}$ and $\vec{t}'; \vec{t}$ inhabit the telescope $(\vec{y} : \vec{Y}); (\vec{T}\,(\mathsf{cheese}\,\vec{y}))$.

$\square$

Now we have checked that each transition rule preserves the structure of constructor form unification problems, the next step is to put them together to make a unification algorithm.

5.2.2 an algorithm for constructor form unification problems

The algorithm is very straightforward: it consists of repeatedly applying the transition rule appropriate to the leading equation until either the goal is proved outright or no equations remain.

From the above lemma, it is clear that if one step leaves a subgoal, the next step can be made. However, we must still show that unification terminates and computes most general unifiers:

DEFINITION: **unifier, most general unifier**

> If $\vec{s}\simeq\vec{t}$ is a constructor form unification problem over $\vec{x}$ and σ is a substitution from the $\vec{x}$ to terms over some $\vec{x}'$, then σ is **unifer** of $\vec{s}\simeq\vec{t}$ if $\sigma\vec{s} \equiv \sigma\vec{t}$.
>
> In addition, σ is a **most general unifier** or **mgu** of $\vec{s}\simeq\vec{t}$ if any unifier of $\vec{s}\simeq\vec{t}$ can be factorised $\rho \cdot \sigma$, where ρ is a substitution on the $\vec{x}'$.

LEMMA: **unification terminates**

> For all constructor form unification problems, the sequence of transition rule applications determined at each stage by the leading equation is finite.

PROOF

I shall use the traditional proof: we may establish a well-founded ordering on unification problems, being the lexicographical ordering on the following three quantities:

- the number of variables $\vec{x}$
- the number of constructor symbols appearing in the problem
- the number of equations in the problem

We may then check case by case that each transition rule either terminates directly or reduces this measure.

- **cycle** and **conflict** terminate directly
- **coalescence** and **substitution** decrement the number of variables
- **injectivity** preserves the number of variables but reduces the number of constructor symbols

- **identity** preserves the number of variables and the number of constructor symbols, but reduces the number of equations

□

LEMMA: **unification correct**

For any initial goal which is constructor form unification problem

$$\forall \vec{x}.\ \vec{s} \simeq \vec{t} \to \Phi[\vec{x}]$$

either $\vec{s}$ and $\vec{t}$ have no unifier, in which case the algorithm proves the goal, or there is a subset $\vec{x}' \subseteq \vec{x}$ and a substitution σ from the $\vec{x}$ to constructor forms over the $\vec{x}'$ such that σ is a mgu of $\vec{s}$ with $\vec{t}$ and the algorithm yields subgoal

$$\forall \vec{x}'.\ \Phi[\sigma \vec{x}]$$

PROOF

It is enough to check that at each step of the problem, either

- the goal has been proven and there is no unifier, or
- the goal is of form

$$\forall \vec{x}'.\ \vec{s}' \simeq \vec{t}' \to \Phi[\sigma \vec{x}]$$

  such that a most general unifier ρ of **remainder** $\vec{s}' \simeq \vec{t}'$ induces a most general unifier $\rho \cdot \sigma$ of $\vec{s} \simeq \vec{t}$

This invariant holds initially, with **accumulator** σ the identity substitution. If it holds finally with no goal, there was no unifier. Otherwise it holds finally with the empty remainder whose mgu is the identity substitution, so the accumulator is the mgu of $\vec{s} \simeq \vec{t}$.

Case by case, then:

- **cycle** and **conflict** prove the goal in cases where there is no unifer
- **identity** and **injectivity** change neither the accumulator nor the unifiers of the remainder

- **coalescence** and **substitution**

	remainder	accumulator
before	$x; \vec{s}' \simeq t; \vec{t}'$	σ
after	$[t/x]\vec{s}' \simeq [t/x]\vec{t}'$	$[t/x] \cdot \sigma$

  Suppose ρ is a mgu of the remainder after the transition. It is enough to show that $\rho \cdot [t/x]$ is a mgu of the remainder beforehand, with the invariant forcing $\rho \cdot [t/x] \cdot \sigma$ to be a mgu of $\vec{s} \simeq \vec{t}$.

  Clearly $\rho \cdot [t/x]$ unifies $x; \vec{s}' \simeq t; \vec{t}'$.

  Now suppose τ also unifies $x; \vec{s}' \simeq t; \vec{t}'$. Then $\tau = \tau \cdot [t/x]$, because

  - $\tau \cdot [t/x]x = \tau t = \tau x$ by hypothesis
  - $\tau \cdot [t/x]y = \tau y$ when $y \neq x$

  Hence τ unifies $[t/x]\vec{s}' \simeq [t/x]\vec{t}'$ and can thus be factorised $\upsilon \cdot \rho$. But $\tau = \tau \cdot [t/x] = \upsilon \cdot \rho \cdot [t/x]$. Thus $\rho \cdot [t/x]$ is most general as required.

□

I feel I should make some comment on these proofs, not that there is anything unusual about them, quite the reverse. I have deliberately given a conventional 'measure' proof of termination, by way of comparison with the structurally recursive algorithm I shall exhibit later as an example of programming with dependent datatypes.

Now that we have an algorithm which exploits the transition rules, it remains only to construct proofs of them. **identity** is trivial. **coalescence** and **substitution** are just applications of $\mathsf{eqSubst}_1$. **conflict**, **injectivity** and **cycle** all require some work.

Before I give the constructions, I want to draw attention to the computational aspect of the proofs built by the unification algorithm: we shall need this technology to build programs as well as proofs. If the algorithm generates

$$\begin{array}{l} ?\mathrm{soFar} : \forall \vec{x}'.\, \vec{s}' \simeq \vec{t}' \to \Phi[\sigma\vec{x}] \\ !\mathsf{start} \;=\; \ldots \\ \qquad\quad : \forall \vec{x}.\, \vec{s} \simeq \vec{t} \to \Phi[\vec{x}] \end{array}$$

we shall need the computational behaviour (for arbitrary $\vec{x}'$)

$$\mathsf{start}\, \sigma\vec{x}\, (\mathsf{refl}\, \sigma\vec{s}) \;\cong\; \mathrm{soFar}\, \vec{x}'\, (\mathsf{refl}\, \vec{s}')$$

Recall that the elimination tactic supplies refl proofs for the constraints. When an elimination rule with associated reductions is applied to a constructor-headed target, it reduces to one of the subgoal proofs, like start, and the refls are passed for the subgoal's constraint arguments—this must allow the subgoal proof to reduce to its simplified version soFar, and ultimately to the value the user has supplied for that case.

Once again, we may check this property stepwise. **identity** is implemented by a λ-abstraction with the appropriate β-behaviour, while **coalescence** and **substitution** exploit the established reduction of $\mathsf{eqSubst}_1$. For **conflict** and **cycle** there is nothing to prove, but we must pay attention in the case of **injectivity**.

5.2.3 conflict and injectivity

Consider an inductive family of datatypes

$$\mathit{Fam} : \forall \vec{\imath} : \vec{I}.\ \mathcal{Type}$$

with n constructors

$$\mathsf{Con}_i : \forall \vec{z} : \vec{Z}_i.\ \mathit{Fam}\ \vec{t}_i[\vec{z}]$$

We have already seen how to compute the case analysis principle $\mathit{FamCase}$:

$$\boxed{\begin{array}{l} \Phi : \overline{\mathit{Fam}} \to \mathcal{Type} \\ \\ \dfrac{\cdots \qquad \begin{array}{c} \vec{z} : \vec{Z}_i \\ \cdots\cdots\cdots\cdots \\ \Phi\ \vec{z};\ (\mathsf{Con}_i\ \vec{z}) \end{array} \qquad \cdots}{\forall \boxed{\vec{t} : \overline{\mathit{Fam}}}.\ \Phi\ \vec{t}} \end{array}}$$

Let us now use $\mathit{FamCase}$ to prove conflict and injectivity theorems for this class of datatype.

The conventional way to prove injectivity for the constructors of simple datatypes is to define a suite of predecessor functions for each argument of each constructor and use the fact that equality respects function application. This is the presentation used in [CT95, McB96]. We cannot do this in general for dependent types, as it is not always possible to supply dummy values for predecessor functions applied to constructors for which they were not originally intended. It is my contention, in any case, that predecessor functions are immoral: the whole idea of pattern matching is to expose the 'predecessors' locally to each constructor case—we should never apply techniques appropriate for one constructor only to arbitrary elements of a type.

Fortunately, the computational power of dependent type theory comes to our rescue. Instead of proving n^2 Peano-style conflict or injectivity theorems, we may manufacture a single 'Peano concerto' which eliminates any constructor-headed equation, computing the appropriate rule by case analysis.

CONSTRUCTION: **Peano concerto**

We begin by establishing the structure of the development: we wish to compute the Peano theorem appropriate to a given pair of elements, then prove it for an equal pair of elements:

$$
\begin{array}{ll}
?\mathit{FamPEANO}: & \forall \vec{x}: \overline{\mathit{Fam}} \\
 & \forall \vec{y}: \overline{\mathit{Fam}} \\
 & \quad \mathit{Type} \\
?\mathit{FamPeano}: & \forall \vec{\imath} \;:\; \vec{I} \\
 & \forall x \;:\; \mathit{Fam}\,\vec{\imath} \\
 & \forall y \;:\; \mathit{Fam}\,\vec{\imath} \\
 & \forall \boxed{e \;:\; x \simeq y} \\
 & \quad \mathit{FamPEANO}\,\vec{\imath}; x\;\vec{\imath}; y
\end{array}
$$

Note that it is perfectly reasonable to prove the theorem only for x and y in the same instance of the family $\mathit{Fam}\,\vec{\imath}$, because this is exactly the situation in which the theorem will be used: eliminating the leading equation in a unification problem, where both terms have the same type.

Looking first to the 'statement' problem, FamPEANO, we may eliminate each of $\vec{x}$ and $\vec{y}$ by $\mathit{FamCase}$, giving n^2 subgoals, of two varieties.

In the first 'off-diagonal' kind, we are asked to compute the conflict theorems for unlike constructors Con_i and Con_j

$$
\begin{array}{ll}
?\mathit{FamPEANO}_{ij}: & \forall \vec{x}: \vec{Z}_i \\
 & \forall \vec{y}: \vec{Z}_j \\
 & \quad \mathit{Type}
\end{array}
$$

We simply introduce all the premises and supply the rather useful elimination rule

$$\boxed{\begin{array}{l} \Phi : \mathit{Type} \\ \\ \dfrac{}{\Phi} \end{array}}$$

After all, an equation with unlike constructors at the head is very unlikely to be true.

More interestingly, on the diagonal, we must compute the injectivity theorems for like constructors

$$\begin{array}{l} ?\mathit{FamPEANO}_{ii} \colon \forall \vec{x} \colon \vec{Z}_i \\ \qquad\qquad\qquad \forall \vec{y} \colon \vec{Z}_i \\ \qquad\qquad\qquad\;\; \mathit{Type} \end{array}$$

Fortunately, the case analysis has exposed the predecessors we need, so all we do is pair them off. Introducing the $\vec{x}$ and $\vec{y}$, we supply the rule

$$\boxed{\begin{array}{l} \Phi : \mathit{Type} \\ \\ \dfrac{\vec{x} \simeq \vec{y} \to \Phi}{\Phi} \end{array}}$$

Crucially, the reduction behaviour of $\mathit{FamCase}$ really means that

$$\mathit{FamPEANO}\, \vec{x}; (\mathsf{Con}_i\, \vec{x}')\, \vec{y}; (\mathsf{Con}_j\, \vec{y}') \;\cong\; \mathit{FamPEANO}_{ij}\, \vec{x}'\, \vec{y}'$$

Now let us show that the rule we have assigned to each kind of constructor-headed equation really holds if the equation does. Recall the goal

$$\begin{array}{l} ?\mathit{FamPeano} \colon \forall \vec{\imath} \;\; : \vec{I} \\ \qquad\qquad\quad\; \forall x \;\; : \mathit{Fam}\, \vec{\imath} \\ \qquad\qquad\quad\; \forall y \;\; : \mathit{Fam}\, \vec{\imath} \\ \qquad\qquad\quad\; \forall \boxed{e \;:\; x \simeq y} \\ \qquad\qquad\quad\;\; \mathit{FamPEANO}\, \vec{\imath}; x\, \vec{\imath}; y \end{array}$$

We have quite a choice of things to eliminate here, but by far the most useful is the equation e. Applying $\mathsf{eqSubst}_1$, we are left with

$$\begin{array}{l} ?\mathit{FamPeano}_{like} \colon \forall \vec{\imath} : \vec{I} \\ \qquad\qquad\qquad\quad \forall x \colon \mathit{Fam}\, \vec{\imath} \\ \qquad\qquad\qquad\quad\;\; \mathit{FamPEANO}\, \vec{\imath}; x\, \vec{\imath}; x \end{array}$$

By eliminating the equation, we have restricted our attention exclusively to the diagonal, sparing ourselves the trouble of considering the untrue equations, let alone deducing their untrue consequences.

Now we may eliminate x with $\mathit{FamCase}$, yielding n subgoals

$$
\begin{array}{l}
?\mathit{FamPeano}_i \colon \forall \vec{z} \colon \vec{Z}_i \\
\qquad\qquad \mathit{FamPEANO}\ \vec{t}_i[\vec{z}]; (\mathsf{Con}_i\ \vec{z})\ \vec{t}_i[\vec{z}]; (\mathsf{Con}_i\ \vec{z})
\end{array}
$$

Reducing FamPEANO, now that its case analyses have been fed constructor symbols, we obtain

$$
\begin{array}{lll}
?\mathit{FamPeano}_i \colon & \forall \vec{z} & : \vec{Z}_i \\
& \forall \Phi & : \mathit{Type} \\
& \forall \mathit{hyp} \colon & \forall \vec{e} \colon \vec{z} \simeq \vec{z} \\
& & \quad \Phi \\
& \ \Phi &
\end{array}
$$

From here, we simply introduce all the hypotheses and prove Φ with

$$\mathit{hyp}\ (\mathsf{refl}\ \vec{z}).$$

Checking the reduction behaviour, we find

$$
\begin{array}{l}
\mathit{FamPeano}\ \vec{\imath}; (\mathsf{Con}_j\ \vec{x})\ \vec{\imath}; (\mathsf{Con}_j\ \vec{x})\ (\mathsf{refl}\ (\mathsf{Con}_j\ \vec{x})) \ \cong \\
\mathit{FamPeano}_{like}\ \vec{\imath}; (\mathsf{Con}_j\ \vec{x}) \ \cong \\
\mathit{FamPeano}_j\ \vec{x} \ \cong \\
\lambda \Phi : \mathit{Type}.\ \lambda \mathit{hyp} : \vec{x} \simeq \vec{x} \to \Phi.\ \mathit{hyp}\ (\mathsf{refl}\ \vec{x})
\end{array}
$$

This ensures that the **identity** transition decomposes refl proofs as required for its use in programs.

Note the critical use of targetting in making this rule applicable. It is not obvious that

$$\forall \vec{\imath} : \vec{I}.\ \forall x, y : \mathit{Fam}\ \vec{\imath}.\ \boxed{x \simeq y} \to \mathit{FamPEANO}\ \vec{\imath}; x\ \vec{\imath}; y$$

is an elimination rule, but that does not stop us unifying the targetter with a candidate equation. If the equated terms have constructor heads, then the instantiated rule will reduce, revealing the scheme variable and subgoals we would normally expect.

Although the unification algorithm only requires us to prove the Peano theorems for two elements of a particular instance $\mathit{Fam}\ \vec{\imath}$, and that is the construction I have given above, it is nonetheless possible to prove the stronger theorem which operates on any two sequences in $\overline{\mathsf{Fam}}$:

$$\mathsf{FamStrongPeano} : \forall \vec{x}, \vec{y} : \overline{\mathsf{Fam}}.\ \boxed{\vec{x} \simeq \vec{y}} \to \mathsf{FamPEANO}\, \vec{x}\, \vec{y}$$

If we eliminate all but the last equation, we have reduced the problem to $\mathsf{FamPeano}$!

It is possible to use this theorem to eliminate a constructor-headed equation from anywhere in a telescopic problem, not just at the front. This can improve the efficiency of unification: if we can see a conflict later in the telescopic equation, we can solve the goal without first hacking through the earlier stages of the problem. Such measures are not necessary when everything is in constructor form, but increase our efficacy for the wider class of problems polluted by non-constructor terms. It is much more difficult to work with inductive families involving indices which are not in constructor form. Such problems are beyond the technology developed in this thesis—I shall discuss them briefly in section 5.2.5.

5.2.4 cycle

Showing that cycles do not occur in our inductive datatypes is quite a subtle business. Even proving

$$\begin{array}{l} ?nNotSn : \forall n : \mathbb{N} \\ \qquad\qquad n \simeq \mathsf{s}n \to \bot \end{array}$$

requires quite a lot of technology. Let us do it. Eliminating n,

$$\begin{array}{lll} ?0NotS0 & : \mathsf{0} \simeq \mathsf{s0} \to \bot & \\ ?SnNotSSn & : \forall n & : \mathbb{N} \\ & \ \ \forall hyp & : n \simeq \mathsf{s}n \to \bot \\ & \ \ \forall e & : \mathsf{s}n \simeq \mathsf{ss}n \\ & \ \ \ \bot & \end{array}$$

Applying unification (without cycle elimination) to both subgoals, we can at least simplify the two constructor-headed equations via the $\mathbb{N}\mathsf{Peano}$ theorem. This eliminates the $\mathsf{0}$ case by conflict, while injectivity leaves us with an immediate step:

$$\begin{array}{lll} ?easy : & \forall n & : \mathbb{N} \\ & \forall hyp & : n \simeq \mathsf{s}n \to \bot \\ & \forall e & : n \simeq \mathsf{s}n \\ & \ \bot & \end{array}$$

In fact, we can follow the same structure for any number of s's, but this is only because the natural numbers are deceptively symmetrical. Watch what happens if we throw in a spare successor constructor, t, (making type $\mathbb{N}'$) and try to prove

$$
\begin{array}{l}
?nNotSTn \colon \forall n \colon \mathbb{N}' \\
\qquad\qquad n \simeq \mathsf{st}n \to \bot
\end{array}
$$

Induction on n yields $\mathsf{0}$ and t cases which perish by conflict. The s case is as follows:

$$
\begin{array}{l}
?SnNotSTSn \colon \forall n \quad \colon \mathbb{N}' \\
\qquad\qquad \forall hyp \colon n \simeq \mathsf{st}n \to \bot \\
\qquad\qquad \forall e \quad \colon \mathsf{s}n \simeq \mathsf{sts}n \\
\qquad\qquad\quad \bot
\end{array}
$$

Injectivity yields

$$
\begin{array}{l}
?tricky \colon \forall n \quad \colon \mathbb{N}' \\
\qquad \forall hyp \colon n \simeq \mathsf{st}n \to \bot \\
\qquad \forall e \quad \colon n \simeq \mathsf{ts}n \\
\qquad\quad \bot
\end{array}
$$

Oh dear! We have the wrong inductive hypothesis! The extra s appeared at the very inside, rotating the cycle: it is only because one successor usually looks much like another that these theorems are so easy for $\mathbb{N}$.

In order to prove the result in this style, we must first strengthen it:

$$
\begin{array}{l}
?noCycleST \colon \forall n \colon \mathbb{N}' \\
\qquad\qquad (n \simeq \mathsf{st}n \to \bot) \times (n \simeq \mathsf{ts}n \to \bot)
\end{array}
$$

By including not just the st cycle, but also all its rotations, we will have more to do: there will be work in both successor cases, although one is enough to show what happens:

$$
\begin{array}{l}
?noCycleST_{s} \colon \forall n \quad \colon \mathbb{N}' \\
\qquad\qquad \forall hyp \colon (n \simeq \mathsf{st}n \to \bot) \times (n \simeq \mathsf{ts}n \to \bot) \\
\qquad\qquad\quad (\mathsf{s}n \simeq \mathsf{sts}n \to \bot) \times (\mathsf{s}n \simeq \mathsf{tss}n \to \bot)
\end{array}
$$

The right conjunct follows by conflict. The left reduces by injectivity:

$$
\begin{array}{l}
?repaired \colon \forall n \quad : \mathbb{N}' \\
\qquad\qquad \forall hyp \colon (n \simeq \mathsf{st}n \to \bot) \times (n \simeq \mathsf{ts}n \to \bot) \\
\qquad\qquad\; n \simeq \mathsf{ts}n \to \bot
\end{array}
$$

The rotated conclusion follows by projecting the appropriately rotated conjunct of the inductive hypothesis.

This technique can be generalised to arbitrary cycles in arbitrary datatypes. The drawback is that (up to rotation), we need a new theorem for every cycle pattern. This leaves us little choice but to generate them on the fly.

A slightly more cunning technique, arising from a conversation with Andrew Adams, is mentioned in [McB96]. It constructs for a given cycle pattern $x \simeq p[x]$ [4] in some type Ind a quotient function $\mathit{quot}_p : \mathit{Ind} \to \mathbb{N}$

$$
\begin{array}{lcl}
\mathit{quot}_p\, p[x] & = & \mathsf{s}(\mathit{quot}_p\, x) \\
\mathit{quot}_p\, \_ & = & 0
\end{array}
$$

Applying quot_p to both sides of the cycle, we get

$$
(\mathit{quot}_p\, x) \simeq \mathsf{s}(\mathit{quot}_p\, x)
$$

We have already seen a disproof of that!

While the guarded recursion principles we have constructed for each datatype make these functions relatively easy to manufacture—indeed I have implemented this technique—we have still not escaped from the burden substantial on-the-fly construction work, cycle by cycle.

Remember, though, that in cycle $x \simeq p[x]$, $p[x]$ is a constructor form, and hence we can compute by structural recursion on it: perhaps there is a way to compute the proof we want. Unfortunately, though, there is no way to test whether we have decomposed as far as a non-canonical symbol like x: our programs have no access to the decidable conversion relation of the type theory which describes them.

Nonetheless, we can adopt blunderbuss tactics: for any element x of a datatype, we can construct the property of 'not being a proper subterm of x' in such a way that when x has a constructor head, the property reduces to a product explicitly enforcing 'not

[4] Without loss of generality, assume $p[x]$ has fresh variables in argument positions off the 'cycle-path'.

being any of the exposed subterms'. The idea works in the same as the auxiliary data structure with which we earlier constructed guarded recursion. In fact, the predicate we need is just an instance of that structure.

We may define this property for $\mathbb{N}$, together with its non-strict counterpart as follows:

$$
\begin{array}{ll}
!\mathbb{N}\mathsf{Unequal} = & \lambda x, y \colon \mathbb{N} \\
& x \simeq y \to \bot \\
!\mathbb{N}\mathsf{NotPSub} = & \lambda x \colon \mathbb{N} \\
& \mathbb{N}\mathsf{Aux}\,(\mathbb{N}\mathsf{Unequal}\,x) \\
!\mathbb{N}\mathsf{NotSub} \;= & \lambda x, y \colon \mathbb{N} \\
& (\mathbb{N}\mathsf{Unequal}\,x\,y) \times (\mathbb{N}\mathsf{NotPSub}\,x\,y)
\end{array}
$$

with conversion behaviour

$$
\begin{array}{lcl}
\mathbb{N}\mathsf{NotPSub}\,x\,0 & \cong & \mathbb{1} \\
\mathbb{N}\mathsf{NotPSub}\,x\,\mathsf{s}n & \cong & \mathbb{N}\mathsf{NotSub}\,x\,n \\
& \cong & (\mathbb{N}\mathsf{Unequal}\,x\,n) \times (\mathbb{N}\mathsf{NotPSub}\,x\,n)
\end{array}
$$

$\mathbb{N}\mathsf{NotPSub}\ x\ y$ is thus inhabited exactly when x is not a proper subterm of y, whilst $\mathbb{N}\mathsf{NotSub}\ x\ y$ adds the requirement $x \not\simeq y$ to indicate that x is not any subterm of y. $\mathbb{N}\mathsf{NotPSub}\ x\ y$ unfolds computationally to reveal a proof that x is not equal to any *guarded* subterm of y. Observe, for example,

$$
\begin{array}{ll}
\mathbb{N}\mathsf{NotPSub}\,x\,\mathsf{ss}x \cong & (\mathbb{N}\mathsf{Unequal}\,x\,\mathsf{s}x) \times \\
& (\mathbb{N}\mathsf{Unequal}\,x\,x) \times (\mathbb{N}\mathsf{NotPSub}\,x\,x)
\end{array}
$$

Suppose we can prove

$$\forall x, t \colon \mathbb{N}.\ x \simeq t \to \mathbb{N}\mathsf{NotPSub}\,x\,t$$

Then for any hypothetical proof of $x \simeq p[x]$, we have a proof of $\mathbb{N}\mathsf{NotPSub}\ x\ p[x]$, which will expand to a product containing

$$\mathbb{N}\mathsf{Unequal}\,x\,x$$

from which contradiction the goal should surely follow.

The first step in the proof is to eliminate the equation, leaving the highly plausible

$$
\begin{array}{l}
?cycle \colon \forall x \colon \mathbb{N} \\
\qquad \mathbb{N}\mathsf{NotPSub}\,x\,x
\end{array}
$$

You could be forgiven for hoping that we might get a cheap proof by the $\mathbb{N}\mathsf{AuxGen}$ theorem we have already built, but sadly, that only proves $\mathbb{N}\mathsf{Aux}\Phi$ when Φ is a constant and all we have to do at each stage is pass on the accumulated information, adding just the new layer. Here, the scheme varies over the recursion, so we must be more cunning.

Our next move is unsurprising: induction on x.

$$
\begin{array}{l}
?cycle_0 : \mathbb{N}\mathsf{NotPSub}\ 0\ 0 \\
?cycle_s : \forall x \ \ : \mathbb{N} \\
\qquad\quad \forall xh : \mathbb{N}\mathsf{NotPSub}\ x\ x \\
\qquad\qquad \mathbb{N}\mathsf{NotPSub}\ \mathsf{s}x\ \mathsf{s}x
\end{array}
$$

The base case is trivial as its type reduces to $\mathbb{1}$. Unfortunately, the step is genuinely difficult: $\mathbb{N}\mathsf{NotPSub}$ fixes its first argument, so there is no way, as things stand, that we can reduce the conclusion to the inductive hypothesis. Some intelligent strengthening will be necessary. First reduce the conclusion to its non-strict expansion:

$$
\begin{array}{l}
?cycle_s : \forall x \ \ : \mathbb{N} \\
\qquad\quad \forall xh : \mathbb{N}\mathsf{NotPSub}\ x\ x \\
\qquad\qquad \mathbb{N}\mathsf{NotSub}\ \mathsf{s}x\ x
\end{array}
$$

We must prove that if x is not a proper subterm of itself, $\mathsf{s}x$ is certainly not a subterm. We can see that if $\mathsf{s}x$ were a subterm, x would be a proper subterm, nomatter what is on the right hand side. Let us make the corresponding generalisation. That is, we introduce the hypotheses, create a hole for the more general version of the goal, then use it to solve the original:

$$
\begin{array}{l}
?cycle_s \approx \lambda x \quad : \mathbb{N} \\
\qquad\qquad \lambda xh \ : \mathbb{N}\mathsf{NotPSub}\ x\ x \\
\qquad\qquad ?gen : \forall y \qquad : \mathbb{N} \\
\qquad\qquad\qquad \forall xNPy : \mathbb{N}\mathsf{NotPSub}\ x\ y \\
\qquad\qquad\qquad\quad \mathbb{N}\mathsf{NotSub}\ \mathsf{s}x\ y \\
\qquad\qquad\ \ gen\ x\ xh
\end{array}
$$

Why is this a good move? Well, we have fixed the first argument of the predicates, and we are now free to let the second vary in an induction on y which corresponds to the computational behaviour of $\mathbb{N}\mathsf{NotPSub}$.

$$
\begin{array}{l}
?gen_0 \colon \forall xNP0 \colon \mathbb{N}\mathsf{NotPSub}\ x\ 0 \\
\qquad\quad \mathbb{N}\mathsf{NotSub}\ \mathsf{s}x\ 0 \\
?gen_s \colon \forall y \qquad : \mathbb{N} \\
\qquad\quad \forall yh \quad : \forall xNPy \colon \mathbb{N}\mathsf{NotPSub}\ x\ y \\
\qquad\qquad\qquad\quad \mathbb{N}\mathsf{NotSub}\ \mathsf{s}x\ y \\
\qquad\quad \forall xNPsy \colon \mathbb{N}\mathsf{NotPSub}\ x\ \mathsf{s}y \\
\qquad\quad\ \ \mathbb{N}\mathsf{NotSub}\ \mathsf{s}x\ \mathsf{s}y
\end{array}
$$

Applying a little computation, the base case becomes

$$
\begin{array}{l}
?gen_0 \colon \forall xNP0 \colon \mathbb{1} \\
\qquad\quad (\mathbb{N}\mathsf{Unequal}\ \mathsf{s}x\ 0) \times \mathbb{1}
\end{array}
$$

This is easily proven, with a little help from $\mathbb{N}\mathsf{Peano}$.

Reducing the step case, we get

$$
\begin{array}{l}
?gen_s \colon \forall y \qquad : \mathbb{N} \\
\qquad\quad \forall yh \quad : \forall xNPy \colon \mathbb{N}\mathsf{NotPSub}\ x\ y \\
\qquad\qquad\qquad\quad \mathbb{N}\mathsf{NotSub}\ \mathsf{s}x\ y \\
\qquad\quad \forall xNPsy \colon (\mathbb{N}\mathsf{Unequal}\ x\ y) \times (\mathbb{N}\mathsf{NotPSub}\ x\ y) \\
\qquad\quad\ \ (\mathbb{N}\mathsf{Unequal}\ \mathsf{s}x\ \mathsf{s}y) \times (\mathbb{N}\mathsf{NotSub}\ \mathsf{s}x\ y)
\end{array}
$$

The implication between the two right conjuncts is exactly given by the inductive hypothesis. As for the left conjuncts, expanding the conclusion's $\mathbb{N}\mathsf{Unequal}$ gives us a proof of $\mathsf{s}x \simeq \mathsf{s}y$ from which we must prove $\perp$. $\mathbb{N}\mathsf{Peano}$ exposes a proof of $x \simeq y$, for which we have a disproof at the ready.

Having established this property for the natural numbers, there is always the nagging suspicion that we have exploited in some hidden way the symmetry of that datatype, just as we would be wary of generalising to all triangles a property which held in the equilateral case. When there is only one step constructor, with only one recursive argument, the issue of whether phenomena behave conjuctively or disjunctively can become blurred. However, in this case, everything fits together perfectly.

CONSTRUCTION: **cycle**

Consider type former

$$\overline{\mathit{Ind} : \mathcal{T}\!ype}$$

and c constructors

$$\frac{\vec{a} : \vec{A} \quad \vec{x} : \{\mathit{Ind}\}^r}{\mathsf{Con}_i\, \vec{a}\, \vec{x} : \mathit{Ind}}$$

Note that I really should write r_i, as the number of recursive arguments may vary from constructor to constructor. However, the proof will be even less readable if I start subscripting superscripts.

We may define the inequality property

$$\mathit{IndUnequal} \;=\; \lambda x, y : \mathit{Ind}.\; x \simeq y \to \bot$$

We can then add the proper subterm relation

$$\mathit{IndNotPSub} \;=\; \lambda x : \mathit{Ind}.\; \mathit{IndAux}\,(\mathit{IndUnequal}\, x)$$

and the non-strict subterm relation

$$\begin{aligned}\mathit{IndNotSub} \;=\; & \lambda x, y : \mathit{Ind}. \\ & (\mathit{IndUnequal}\, x\, y) \times (\mathit{IndNotPSub}\, x\, y)\end{aligned}$$

The computational behaviour of these definitions is as one would hope:

$$\mathit{IndNotPSub}\, x\, (\mathsf{Con}_j\, \vec{a}\, \vec{y}) \;\cong\; \Sigma\big\{\mathit{IndNotSub}\, x\, y_k\big\}_k^r$$

We may now prove the cycle theorem:

$$\begin{aligned} ?\mathit{IndCycle} : \; & \forall x, t : \mathit{Ind} \\ & \forall \boxed{\, e \; : \; x \simeq t \,} \\ & \quad \mathit{IndNotPSub}\, x\, t \end{aligned}$$

First, we eliminate the equation, leaving

$$\begin{aligned} ?\mathit{IndCycle}' : \; & \forall x : \mathit{Ind} \\ & \quad \mathit{IndNotPSub}\, x\, x \end{aligned}$$

Next, we **eliminate** the x.

$$\begin{aligned} ?\mathrm{case}_i : \; & \forall \vec{a} : \vec{A}_i \\ & \forall \vec{x} : \{\mathit{Ind}\}^r \\ & \forall \vec{h} : \{\mathit{IndNotPSub}\, x_k\, x_k\}_k^r \\ & \quad \mathit{IndNotPSub}\, (\mathsf{Con}_i\, \vec{a}\, \vec{x})\, (\mathsf{Con}_i\, \vec{a}\, \vec{x}) \end{aligned}$$

The conclusion expands, yielding a product

$$
\begin{array}{l}
?\mathrm{case}_i \colon \forall \vec{a} \colon \vec{A}_i \\
\qquad \forall \vec{x} \colon \{\mathit{Ind}\}^r \\
\qquad \forall \vec{h} \colon \{\mathit{IndNotPSub}\, x_k\ x_k\}_k^r \\
\qquad\ \ \Sigma\{\mathit{IndNotSub}\,(\mathsf{Con}_i\ \vec{a}\ \vec{x})\ x_k\}_k^r
\end{array}
$$

Now we come to the strengthening step. The conclusion we are trying to show is r-fold now. The trick is to prove each separately, abstracting away the right hand x_k in r separate lemmas:

$$
\begin{array}{l}
?\mathrm{case}_i \approx \lambda \vec{a} \quad \colon \vec{A}_i \\
\qquad\qquad \lambda \vec{x} \quad \colon \{\mathit{Ind}\}^r \\
\qquad\qquad \lambda \vec{h} \quad \colon \{\mathit{IndNotPSub}\, x_k\ x_k\}_k^r \\
\qquad\qquad ?\vec{\mathrm{lem}} \colon \left\{ \begin{array}{l} \\ \\ \forall y \qquad \colon \mathit{Ind} \\ \forall xNPy \colon \mathit{IndNotPSub}\, x_k\ y \\ \ \ \mathit{IndNotSub}\,(\mathsf{Con}_i\ \vec{a}\ \vec{x})\ y \end{array} \right\}_k^r \\
\qquad\qquad\ \ \langle \{\mathrm{lem}_k\ x_k\ h_k\}_k^r \rangle
\end{array}
$$

The proof of each lemma is again inductive. We apply *IndElim*, Thus for each of the r, lemmas, we acquire c constructor cases:

$$
\begin{array}{l}
?\mathrm{lem}_j \colon \forall \vec{a}' \qquad \colon \vec{A}_j \\
\qquad\quad \forall \vec{y} \qquad \colon \{\mathit{Ind}\}^r \\
\qquad\quad \forall \vec{h} \qquad \colon \left\{ \begin{array}{l} \forall xNPy_l \colon \mathit{IndNotPSub}\, x_k\ y_l \\ \ \ \mathit{IndNotSub}\,(\mathsf{Con}_i\ \vec{a}\ \vec{x})\ y_l \end{array} \right\}_l^r \\
\qquad\quad \forall xNPc \colon \mathit{IndNotPSub}\, x_k\ (\mathsf{Con}_j\ \vec{b}\ \vec{y}) \\
\qquad\quad\ \ \mathit{IndNotSub}\,(\mathsf{Con}_i\ \vec{a}\ \vec{x})\ (\mathsf{Con}_j\ \vec{b}\ \vec{y})
\end{array}
$$

Now, a little computation is in order:

$$
\begin{array}{l}
?\mathrm{lem}_j \colon \forall \vec{a}' \qquad \colon \vec{A}_j \\
\qquad\quad \forall \vec{y} \qquad \colon \{\mathit{Ind}\}^r \\
\qquad\quad \forall \vec{h} \qquad \colon \left\{ \begin{array}{l} \forall xNPy_l \colon \mathit{IndNotPSub}\, x_k\ y_l \\ \ \ \mathit{IndNotSub}\,(\mathsf{Con}_i\ \vec{a}\ \vec{x})\ y_l \end{array} \right\}_l^r \\
\qquad\quad \forall xNPc \colon \Sigma\big\{(\mathit{IndUnequal}\, x_k\ y_l) \times (\mathit{IndNotPSub}\, x_k\ y_l)\big\}_l^r \\
\qquad\quad \Sigma_{-} \qquad \colon \mathit{IndUnequal}\,(\mathsf{Con}_i\ \vec{a}\ \vec{x})\ (\mathsf{Con}_j\ \vec{b}\ \vec{y}) \\
\qquad\quad\ \ \Sigma\big\{\mathit{IndNotSub}\,(\mathsf{Con}_i\ \vec{a}\ \vec{x})\ y_l\big\}_l^r
\end{array}
$$

Firstly, each

$$\mathit{IndNotSub}\,(\mathsf{Con}_i\,\vec{a}\,\vec{x})\,y_l$$

follows by h_l applied to the proof of

$$\mathit{IndNotPSub}\,x_k\,y_l$$

projected from *xNPc*.

Secondly, we must establish

$$\mathit{IndUnequal}\,(\mathsf{Con}_i\,\vec{a}\,\vec{x})\,(\mathsf{Con}_j\,\vec{b}\,\vec{y})$$

That is, we must prove

$$(\mathsf{Con}_i\,\vec{a}\,\vec{x})\simeq(\mathsf{Con}_j\,\vec{b}\,\vec{y}) \to \bot$$

so we apply *IndPeano*. If the constructors are different ($i \neq j$), the goal is proved at once, otherwise $i = j$ and we must show

$$\vec{a}\simeq\vec{b} \to \vec{x}\simeq\vec{y} \to \bot$$

But look! *xPNc* contains proofs for each l of

$$\mathit{IndUnequal}\,x_k\,y_l$$

We may select the proof of $x_k\simeq y_k$ from the injected equations, and the proof of

$$\mathit{IndUnequal}\,x_k\,y_k$$

from *xPNc* establishing a contradiction and completing the construction.

Let me remark only briefly on the extension to dependent families. For

$$\mathit{Fam} : \forall\vec{\imath}\!:\!\vec{I}.\ \mathcal{T}\!\mathit{ype}$$

the appropriate notion of inequality is

$$\mathit{FamUnequal} \;=\; \lambda\vec{x},\vec{y}\!:\!\overline{\mathit{Fam}}.\ (\vec{x}\simeq\vec{y}) \to \bot$$

We can then construct *FamNotPSub* and *FamNotSub* as before:

$$\begin{aligned}\mathit{FamNotPSub} &= \lambda\vec{x}\!:\!\overline{\mathit{Fam}}.\ \mathit{FamAux}\,(\mathit{FamUnequal}\,\vec{x})\\ \mathit{FamNotSub} &= \lambda\vec{x},\vec{y}\!:\!\overline{\mathit{Fam}}.\ (\mathit{FamUnequal}\,\vec{x}\,\vec{y})\times(\mathit{FamNotPSub}\,\vec{x}\,\vec{y})\end{aligned}$$

Since all three of these take two sequences in $\overline{\mathsf{Fam}}$, rather than two elements in some $\mathsf{Fam}\,\vec{\imath}$, no problem arises in the strengthening step: we are free to abstract away the whole right hand sequence, ensuring the induction is on the entire family.

As for the equational reasoning, suppose we are trying to prove some inequality $\vec{x}; x \simeq \vec{y}; y \ \rightarrow \ \bot$ where both sequences inhabit $\overline{\mathsf{Fam}}$, with both x and y constructor-headed. Rather than trying to unify the $\vec{x}$ and $\vec{y}$, we may apply the 'strong' version of the Peano theorem directly to the telescopic equation, solving the goal in the case of different constructors, and exposing the equations of the predecessors if the constructors are the same.

In effect, then, the construction scales up without any difficulty from elements of simple types to sequences in some $\overline{\mathsf{Fam}}$.

This construction also generalises easily to datatypes which use higher-order constructors to represent infinitely-branching structures. When the higher-order arguments appear as hypotheses they may simply be fixed, so that they may be used as the appropriate witnesses for higher-order arguments in goal positions. However, it is not easy to exploit this proof automatically, as it is undecidable whether an infinitely-branching structure contains a cycle. Suppose we have a hypothetical ordinal x, together with a function $f : \mathbb{N} \rightarrow \mathsf{ord}$ which yields x for input 37. If we have a hypothesis

$$x \simeq \mathsf{sup}\, f$$

we acquire a proof of $\mathsf{ordNotPSub}\; x\; (\mathsf{sup}\, f)$, which expands to uncover a proof of $\forall n : \mathbb{N}.\ x \simeq f \rightarrow \bot$ but the machine has no reliable way of guessing that 37 is the right number to expose the contradiction. Of course, if *we* know which branches a cycle takes, we can still apply $\mathsf{ordCycle}$ by hand.

5.2.5 a brief look beyond constructor form problems

There is nothing which restricts our use of dependent families to indices in constructor form. More complex indices lead to more complex unification problems, and the general case is inevitably undecidable. There are two ways in which such problems can arise, and they are not mutually exclusive:

- Non-constructor-form indices may appear in the type of a constructor. For example, we might define *sized* binary trees, $\mathsf{stree} : \mathbb{N} \rightarrow \mathit{Type}$ as follows:

$$\frac{}{\mathsf{empty} : \mathsf{stree}\; 0} \qquad \frac{X : \mathsf{stree}\; x \quad Y : \mathsf{stree}\; y}{\mathsf{node}\; X\; Y : \mathsf{stree}\; \mathsf{s}(\mathsf{plus}\; x\; y)}$$

- Non-constructor-form indices may appear in the type of an argument over which case analysis is to be performed. For example, we might wish to write

$$\mathsf{vprefix} : \forall A\!:\!\mathit{Type}.\ \forall m, n\!:\!\mathbb{N}.\ \forall v\!:\!\mathsf{vect}_A\ (\mathsf{plus}\, m\, n).\ \mathsf{vect}_A\ m$$

The tractability of such problems, even by hand, depends on the *types* of the non-constructor-form expressions:

- Many problems involving the comparison of types or functions are simply beyond us. On the one hand, we do not have theorems such as **conflict** at the level of types—we cannot disprove $\mathbb{N}\simeq\mathbf{2}$. On the other hand, the intensionality of $\simeq$ prevents us from solving even such simple higher-order problems as

  $$\forall f\!:\!\mathbb{N} \to \mathbb{N}.\ (\forall x\!:\!\mathbb{N}.\, f\, x \simeq \mathsf{s}x)\ \to \ldots$$

  Even though the extensional behaviour of f is completely determined, there are many intensionally distinct terms which exhibit that behaviour.

- Equations within datatypes involving defined functions like plus are less troublesome, especially if we have equipped those functions with derived elimination rules which do constructor-based analysis of the return values.

Let us examine the example of $\mathsf{vprefix}$. Induction on v will leave subgoals containing unsolved equational problems, such as the vnil case:

$$\begin{array}{l} ?\mathsf{vprefix}_n : \forall A\!:\!\mathit{Type}.\ \forall m, n\!:\!\mathbb{N}.\ \forall v\!:\!\mathsf{vect}_A\ (\mathsf{plus}\, m\, n). \\ \qquad\qquad \mathsf{0}\simeq(\mathsf{plus}\, m\, n)\ \to\ \mathsf{vnil}\simeq v\ \to\ \mathsf{vect}_A\ m \end{array}$$

Case analysis on m will get us out of this predicament, but only because we know how plus works. A more cunning approach is to address the troublesome plus directly, constructing $\mathsf{vprefix}$ with plus's recursion induction principle, shown on the right. Note that the plus symbol is completely absent from the cases.

$\mathsf{plusRecI}$

$$\Phi : \forall x, y, \boxed{z}\!:\!\mathbb{N}.\ \mathit{Type}$$

$$\frac{\Phi\ \mathsf{0}\ y\ y \qquad \dfrac{\Phi\ x\ y\ z}{\Phi\ \mathsf{s}x\ y\ \mathsf{s}z}}{\forall x, y\!:\!\mathbb{N}.\ \Phi\ x\ y\ \boxed{\mathsf{plus}\, x\, y}}$$

Targetting the $(\mathsf{plus}\, m\, n)$ in goal $\mathsf{vprefix}$ yields subgoals:

$$
\begin{aligned}
?\mathsf{vprefix}_z &: \forall A\!:\!\mathit{Type}.\ \forall n\!:\!\mathbb{N}.\ \forall v\!:\!\mathsf{vect}_A\ n.\ \mathsf{vect}_A\ 0 \\
?\mathsf{vprefix}_s &: \forall A\!:\!\mathit{Type}.\ \forall m, n, z\!:\!\mathbb{N}. \\
&\qquad (\forall v'\!:\!\mathsf{vect}_A\ z.\ \mathsf{vect}_A\ m) \ \to \\
&\qquad \forall v\!:\!\mathsf{vect}_A\ \mathsf{s}z.\ \mathsf{vect}_A\ \mathsf{s}m
\end{aligned}
$$

The remaining indices are in constructor form!

I draw two conclusions from this discussion. Firstly, dependently typed programming with non-constructor-form indices is difficult—a principled machine treatment is a long way off. Secondly, for hand treatments of such problems, derived elimination rules describing the behaviour of non-constructor functions are of considerable benefit.

Chapter 6

Pattern Matching for Dependent Types

We are now in a position to build tools for programming with dependent datatypes. In this chapter, I shall first discuss the interactive development of programs. However, I believe it also important to consider the translation of functional programs from the conventional equational style into real OLEG terms based on the elimination rules primitive for each datatype.

Why should we be interested in these programs? Some people like to write programs,[1] and raw type theory is hard to write, especially as it must record explicitly the unfication attendant to the elimination of dependent datatypes. That is why we get machines to do it.

I am an enthusiastic advocate of the analytic style of programming afforded by proof editors. For me, the key point is that the search for programs is carried out in a structured space of partial objects constrained to make sense: the machine performs most of the bookkeeping and checks for type errors locally and incrementally.

Synthesising programs in the conventional way involves unconstrained search amongst arbitrary sequences of potential gibberish for completed objects which a compiler either accepts or rejects. The incremental programming afforded by interactive declare-before-use environments common in the ML community is almost entirely useless because it is incremental from the bottom up: it requires the details to be presented before the outline and thus supports only the kind of lonely obsessiveness that gives programming a bad name. The module system offers some compensation, at a coarse granularity.

The trouble with raw type theory is not that it is hard to write, but that it is hard to read. Even if a program is generated with machine help, it is still a good thing if

[1] Others are merely paid to do it.

we can represent it in a way which is comprehensible to humans. Sequences of tactic applications are not especially informative and, in any case, run counter to the demands of a good user interface.

I hope, therefore, you will agree that it is good to have a high-level representation for synthesised proofs and programs which nonetheless exposes the analysis both by which it operates and by which it can be constructed. Pattern matching notation has been with us for three decades in theory and in practice [Bur69, McB70]. Perhaps it is because I have been brought up in these old ways that I am so slow to change, but I still prefer equational presentations of programs to this newfangled 'pointer dereferencing' or whatever it is the young people do these days. One side effect of a concise and readable notation is that we can still write programs on the backs of quite small envelopes.

What do these programs look like? Let us simplify matters for the time being, and consider only solitary functions:

$$\begin{array}{lllll} f & : & \forall \vec{x}\!:\!\vec{S}. & & T \\ f & & \vec{s}_1 & = & t_1 \\ \vdots & & \vdots & & \vdots \\ f & & \vec{s}_n & = & t_n \end{array}$$

Each $\vec{s}_i$ will contain some 'free' variables $\vec{y} : \vec{Y}_i$ which are really universally quantified. f may not appear in any of the $\vec{s}_i$. Both f and the $\vec{y}$ may appear in t_i. It is, of course, impossible to guess the $\vec{Y}_i$ for arbitrary $\vec{S}$ and $\vec{s}_i$, although it is not hard to imagine classes of problem for which it is routine. Let us assume they are also supplied by the programmer, but nonetheless omit them informally when unremarkable.

What do we mean by such a program? I suggest that we mean to determine the type and the *intensional* behaviour of the defined symbol f. It is not enough that the program should determine for each closed input $\vec{s} : \vec{S}$ a unique output t: that is merely to describe the extension of a function—to give equations which must cover all the cases and be true. The programs must also reflect a deterministic and terminating computation mechanism, even on open terms, and taking canonical inputs to canonical outputs. That is, the equations must have computational, not just propositional force. The programs must decode internally into combinations of abstractions, applications, case analysis and terminating recursion. This requirement is reflected to a considerable extent in the task of translating such programs in terms of the effective computational behaviour primitive to OLEG datatypes.

A common notion of pattern matching from functional programming with simple types requires the patterns (the $\vec{s}_i$ above) to be in constructor form, nonlinear, exhaustive and disjoint. This is not sufficient to guarantee the intensional behaviour required here. The classic counterexample (due, as far as I know, to Berry) is the three juror $\mathsf{majority}$ function:

$$
\begin{array}{lccccl}
\mathsf{majority} & : & \mathsf{verdict} \to & \mathsf{verdict} \to & \mathsf{verdict} \to & \mathsf{verdict} \\
\mathsf{majority} & & \mathsf{innocent} & \mathsf{innocent} & \mathsf{innocent} & = \mathsf{innocent} \\
\mathsf{majority} & & \mathsf{guilty} & \mathsf{innocent} & z & = z \\
\mathsf{majority} & & \mathsf{innocent} & y & \mathsf{guilty} & = y \\
\mathsf{majority} & & x & \mathsf{guilty} & \mathsf{innocent} & = x \\
\mathsf{majority} & & \mathsf{guilty} & \mathsf{guilty} & \mathsf{guilty} & = \mathsf{guilty}
\end{array}
$$

Now, imagine you are in a low-budget remake of the Henry Fonda film, 'Twelve Angry Men', entitled 'Three Mildly Peeved Men', and your task is to find out what the majority verdict is. The three jurors do not each know what the others think, so the only way you can gain any information is to ask them individually for their verdicts: you cannot ask 'should you have the casting vote'. Represent what you know by a pattern: initially, you know nothing, so the pattern is

$$x\, y\, z$$

When you ask a question, of the first juror, say, your state of knowledge divides in two possibilities

$$
\begin{array}{l}
\mathsf{innocent}\; y\, z \\
\mathsf{guilty}\; y\, z
\end{array}
$$

Based on this choice, you can adopt different strategies of questioning, ultimately giving you a set of possibilities from each of which you draw a conclusion. Does Berry's collection of patterns represent a set of such possibilities, arising from a conditional questioning strategy? No: each juror appears undeclared in at least one pattern, and at least two jurors must declare in order to determine the answer.

The following shorter and intensionally realisable function has the same extensional behaviour:[2]

$$
\begin{array}{lccccl}
\mathsf{majority} & : & \mathsf{verdict} \to & \mathsf{verdict} \to & \mathsf{verdict} \to & \mathsf{verdict} \\
\mathsf{majority} & & \mathsf{innocent} & \mathsf{innocent} & z & = \mathsf{innocent} \\
\mathsf{majority} & & \mathsf{innocent} & \mathsf{guilty} & z & = z \\
\mathsf{majority} & & \mathsf{guilty} & \mathsf{innocent} & z & = z \\
\mathsf{majority} & & \mathsf{guilty} & \mathsf{guilty} & z & = \mathsf{guilty}
\end{array}
$$

[2]It also has an advantage in some cases if you are the third juror and prone to moments of angst.

Extensional presentations of functions are not useless: they are merely non-computational. It is highly desirable, at times, to give such extensional properties in specifications of functions. The question is then whether they can be transformed into intensional programs, preserving the extensional requirements.

The fact that intensionally realisable patterns arise from such questioning strategies militates strongly in favour of the analytic view of programming: generating patterns by case splitting not only guarantees their computational meaningfulness, but also gives some guidance to the way we think about problems in the first place.[3]

Generating **coverings** of patterns by splitting is central to Thierry Coquand's characterisation of pattern matching for dependent types [Coq92], as implemented in ALF [Mag94]. It is worth taking the time to review this now, not only to place the work of this chapter in its wider context, but also because it is in his meta-level footsteps that I have followed with my object-level treatment.

6.1 pattern matching in ALF

Coquand proposes to admit functions defined in pattern matching style directly to the type theory as constants with reduction rules given by the equations provided they satisfy certain safety conditions, more stringent than necessary, but nonetheless allowing considerable freedom of expression. For

$$\begin{array}{rcccc} & f & : \forall \vec{x}:\vec{S}. & & T \\ \forall \vec{y}:\vec{Y}_1. & f & \vec{s}_1 & = & t_1 \\ & \vdots & \vdots & & \vdots \\ \forall \vec{y}:\vec{Y}_n. & f & \vec{s}_n & = & t_n \end{array}$$

he demands

no nesting : for each $f\,\vec{r}$ in any t_i, f does not occur in any r_j

guarded recursion : for some j and every i, every recursive $f\,\vec{r}$ in t_i has r_j guarded in s_{ij}

covering : the $\vec{s}_i$ form a **covering** of $\vec{S}$, in the sense to be defined below

[3] When teaching students ML, I have so frequently found myself asking 'What do you do with the empty list? What do you do with h cons t?' that it has become something of a mantra, for me, if not for them.

The definition of covering captures the notion of successive case-splitting. We shall first need a definition of such a split or **elementary covering**—this we iterate to yield covering.

DEFINITION: **elementary covering**

> The $\vec{s}_i$ form an **elementary covering** of $\vec{Y}$ if there is an argument position j such that
>
> - s_{ij} is constructor-headed for each i
> - for any argument sequence $\vec{r}$ with r_j constructor-headed, there is exactly one i and instantiation of the $\vec{y} : \vec{Y}_i$ which makes $\vec{s}_i \cong \vec{r}$

Note that, in particular, sequences with different constructors heading the jth argument must be covered by different patterns, and that all possible constructors must be covered: we have just asked the jth argument to reveal which constructor is at its head.

DEFINITION: **covering**

- the **free pattern**[4] $\vec{y} : \vec{S}$ is a covering of $\vec{S}$
- if $\vec{s}_i$ (over $\vec{y} : \vec{Y}_i$) is an elementary covering of $\vec{S}$ and $\vec{r}_{ij}$ (over $\vec{z} : \vec{Z}_{ij}$) are coverings of the $\vec{Y}_i$, then the $[\vec{r}_{ij}/\vec{y}]\vec{s}_i$ also form a covering of $\vec{S}$

Which coverings we can build interactively depends on which elementary coverings we can recognise as such—this is where unification comes in. Let us suppose that we have a family of types Fam, and that we wish to form an elementary covering of some telescope

$$\vec{y} : \vec{Y}; y : \mathsf{Fam}\,\vec{s}; \vec{y}' : \vec{Y}'$$

by case-splitting on y. Fam has constructors

$$\frac{\vec{x} : \vec{X}_i}{\mathsf{Con}_i\,\vec{x} : \mathsf{Fam}\,\vec{t}_i}$$

so the possible cases are those where the $\vec{s}$ unify with the $\vec{t}_i$, the flexible variables being $\vec{x} : \vec{X}_i; \vec{y} : \vec{Y}$.

We apply an appropriate unification algorithm, such as the constructor unification from last chapter, getting one of three responses

[4] my term

- a most general unifier σ_i from variables $\vec{x} : \vec{X}_i; \vec{y} : \vec{Y}$ to terms over some $\vec{z} : \vec{Z}$
- indication that there is no unifier
- failure due to ambiguity or getting stuck

If the unification is conclusive for each constructor case, then our elementary covering has one pattern for each mgu σ_i, given by

$$\sigma_i \vec{y}; (\mathsf{Con}_i\ \sigma_i \vec{x}); \vec{y}' \qquad (\text{over } \vec{z} : \vec{Z}; \vec{y}' : \sigma_i \vec{Y}')$$

We can now build coverings by starting with the free pattern and repeatedly applying case-splitting, as allowed by the unification. Note that unification is a meta-level notion here: it must be sound with respect to the computational equality. Apart from that, we can make it as clever or as stupid as we like. Constructor unification is already quite generous—this is essentially what the implementation of ALF provides.

Programming then proceeds in a type-directed way, building a covering for the argument telescope of a function, then filling in the right-hand sides by refinement, allowing recursive calls, provided the appropriate termination check is satisfied.[5]

It is not hard to see that all the ι-reductions so far presented in this thesis fall into this class of definable function (provided we make the appropriate straightforward extension for mutually defined functions on mutually defined datatypes): the elimination rules for datatypes have been constructed to yield elementary coverings of them, with one-step guarded recursion. In fact, we do not even need the unification algorithm to handle conflict, injectivity or cycles: coalescence and substitution are enough for the datatype ι-reductions, and we must add identity if we wish to support eqElim.

What about the other way around? If we fix on constructor form unification as that which informs the case-splitting process, then we may follow this treatment at the object-level.

6.2 interactive pattern matching in OLEG

This section contains the main metatheoretic result of this thesis: it proves that functions which can be manufactured interactively in ALF can be manufactured interactively in OLEG. Furthermore, the simulation is at an intensional level—the functions

[5] In the original ALF implementation, this was left as a moral obligation, but Coquand's criterion above is not hard to enforce.

we manufacture from OLEG elimination rules have the same computational behaviour as those defined directly in ALF.

Before we can progress to the theorem, we must examine computation with elimination rules in more detail.

6.2.1 computational aspects of elimination

Suppose a function f can be given in terms of another g as follows:

$$\mathsf{f} = \lambda\vec{x}:\vec{X}.\ \mathsf{g}\,\vec{s}$$

What can we infer about the computational behaviour of f from that of g?

This is a very common situation. If g is an elimination rule and we construct f by eliminating some of its arguments with g, this is exactly the structure which f will take. If g has a reduction behaviour given by ι-reductions or a pattern matching function in the Coquand style, we may be able to infer the corresponding behaviour for f. For example, we have already seen how to construct $\mathbb{N}\mathsf{Case}$ from $\mathbb{N}\mathsf{Elim}$ in this way: how does $\mathbb{N}\mathsf{Case}$ reduce when it is fed constructor-headed numbers? It is not hard to check that it inherits the appropriate behaviour from $\mathbb{N}\mathsf{Elim}$:

$$\begin{aligned}\mathbb{N}\mathsf{Case}\,\Phi\,\phi_0\,\phi_s\,\mathsf{0} &\cong \phi_0\\ \mathbb{N}\mathsf{Case}\,\Phi\,\phi_0\,\phi_s\,\mathsf{s}n &\cong \phi_s\,n\end{aligned}$$

Similarly, if we want to implement the pattern matching version of plus by means of $\mathbb{N}\mathsf{Elim}$, we need to be sure that the defining equations are intensionally recoverable. In particular, we need to show that any recursive calls to $\mathbb{N}\mathsf{Elim}$ in the implementation can be replaced by recursive calls to plus convertible to them. We can achieve this by a process of unfold/fold transformation on functional programs which respects the computational equality of OLEG.

Let us consider unfolding first.

Suppose g is given by a pattern matching program

$$\begin{aligned}&\mathsf{g} : \forall\vec{y}:\vec{S}.\ T\\ &\mathsf{g}\,\vec{s}_i = t_i \qquad \text{(over pattern variables } \vec{y} : \vec{Y}_i)\end{aligned}$$

From the definition of f, we can infer the lengthened equation

$$f\vec{x} \cong g\vec{s} \quad \text{(any } \vec{x} : \vec{X}\text{)}$$

For each $\vec{s}_i$, there are two possibilities

- $\vec{s}$ is at least as long as $\vec{s}_i$
- $\vec{s}$ is shorter than $\vec{s}_i$

In the former case, we may split $\vec{s}$ as $\vec{r};\vec{r}'$, so that $\vec{r},\vec{s}_i : \vec{S}$. If σ is a substitution from $\vec{x};\vec{y}$ to terms over $\vec{z} : \vec{Z}$ which unifies $\vec{s}_i$ and $\vec{r}$, then we have

$$f\sigma\vec{x} \cong g\sigma(\vec{r};\vec{r}') \equiv g\sigma\vec{s}_i;\sigma\vec{r}' \cong (\sigma t_i)\,\sigma\vec{r}' \quad \text{(over } \vec{z} : \vec{Z}\text{)}$$

In the latter case, it is $\vec{s}_i$ which we split as $\vec{r};\vec{r}'$ so that $\vec{s},\vec{r} : \vec{R}$, where $\vec{R}$ is a prefix of $\vec{S}$. If σ is a unifying substitution, then we have

$$f\sigma\vec{x} \cong g\sigma\vec{s} \equiv g\sigma\vec{r}$$

and therefore

$$f\sigma\vec{x};\sigma\vec{r}' \cong g\sigma(\vec{r};\vec{r}') \equiv g\sigma\vec{s}_i \cong \sigma t_i$$

Note that we may not, in general, pad out the application of f before the unification, because $f\vec{x}$ may not have functional type until the $\vec{x}$ have been instantiated.

Folding is more straightforward. If we know that

$$f\vec{x} \cong r \quad \text{where } \vec{x} \text{ is the free pattern, and}$$
$$f\vec{s} \cong t[\sigma r] \quad \text{(over } \vec{y} : \vec{Y}\text{)}$$

then

$$f\vec{s} \cong t[f\sigma\vec{x}] \quad \text{(over } \vec{y} : \vec{Y}\text{)}$$

I have not explained where these substitutions σ come from,[6] but I do not have to: unfolding and folding are a pair of techniques by which we can derive new intensional equations from old ones. I do not propose to use them to construct pattern matching programs, but rather to confirm their intensional status. For example, the program plus may be written in pattern matching notation

[6]Perhaps you can guess.

$$
\begin{array}{llcll}
\mathsf{plus} & \mathsf{0} & y & = & y \\
\mathsf{plus} & \mathsf{s}x & y & = & \mathsf{s}(\mathsf{plus}\, x\, y)
\end{array}
$$

This quite clearly falls within Coquand's class of definable functions. We have already seen plus defined somewhat less perspicuously in OLEG:

$$
\begin{array}{lcll}
\mathsf{plus} & = & \mathbb{N}\mathsf{Elim} & (\lambda x\!:\!\mathbb{N}.\ \mathbb{N} \to \mathbb{N}) \\
 & & & (\lambda y\!:\!\mathbb{N}.\ y) \\
 & & & (\lambda x\!:\!\mathbb{N}.\ \lambda \mathsf{plus}_n : \mathbb{N} \to \mathbb{N}.\ \lambda y\!:\!\mathbb{N}.\ \mathsf{s}(\mathsf{plus}_n\, y))
\end{array}
$$

We can check that the pattern matching equations hold intensionally for the OLEG definition. First unfolding with respect to each ι-reduction of $\mathbb{N}\mathsf{Elim}$:

$$
\begin{array}{llcl}
\mathsf{plus} & \mathsf{0} & = & \lambda y\!:\!\mathbb{N}.\ y \\
\mathsf{plus} & \mathsf{s}x & = & \lambda y\!:\!\mathbb{N}.\ \mathsf{s}(\mathbb{N}\mathsf{Elim} \ldots x\, y)
\end{array}
$$

Folding with respect to the OLEG definition:

$$
\begin{array}{llcl}
\mathsf{plus} & \mathsf{0} & = & \lambda y\!:\!\mathbb{N}.\ y \\
\mathsf{plus} & \mathsf{s}x & = & \lambda y\!:\!\mathbb{N}.\ \mathsf{s}(\mathsf{plus}\, x\, y)
\end{array}
$$

Lengthening:

$$
\begin{array}{llcll}
\mathsf{plus} & \mathsf{0} & y & = & y \\
\mathsf{plus} & \mathsf{s}x & y & = & \mathsf{s}(\mathsf{plus}\, x\, y)
\end{array}
$$

We have checked our implementation of the pattern matching program!

In fact, we can use lengthening, unfolding and folding to check all the dervived computation laws in this thesis, and we shall use them in particular to ensure the intensional validity of the pattern matching programs we shall shortly construct.

Of particular interest is the computational effect of case analysis followed by unification.

Suppose we face the goal

$$
?f\!:\ \forall \vec{x}\!:\!\vec{S}.\ T
$$

where S_i is some $\mathit{Fam}\,\vec{p}$, with the $\vec{p}$ in constructor form. Let Fam have constructors

$$\frac{\vec{z} : \vec{Z}_j}{\mathsf{Con}_j\, \vec{z} : \mathit{Fam}\, \vec{p}_j}$$

Eliminating x_i by *FamCase* yields, in general:

$$\begin{array}{l}
!\Phi = \lambda \vec{y} \colon \overline{\mathit{Fam}} \\
\quad\;\; \forall \vec{x} \colon \vec{S} \\
\quad\;\; \forall \vec{e} \colon \vec{y} \simeq \vec{p}; x_i \\
\quad\;\;\;\; T \\
\vdots \\
?f_j : \forall \vec{z} \colon \vec{Z}_j \\
\quad\;\; \forall \vec{x} \colon \vec{S} \\
\quad\;\; \forall \vec{e} \colon \vec{p}_j; (\mathsf{Con}_j\, \vec{z}) \simeq \vec{p}; x_i \\
\quad\;\;\;\; T \\
\vdots \\
!f = \lambda \vec{x} \colon \vec{S} \\
\quad\;\;\;\; \mathit{FamCase}\, \Phi \;\ldots f_j \ldots\; \vec{p}; x_i\; (\mathsf{refl}\, \vec{p}); (\mathsf{refl}\, x_i)
\end{array}$$

Now let us apply the unification algorithm to f_j. Either there is no unifier, in which case we have no need of a computational explanation, or there is a most general unifier σ_j. In this case, the new subgoal looks like

$$\begin{array}{l}
?f'_j \colon \forall \vec{y} \colon \vec{Y}_j \\
\quad\;\; \sigma_j T
\end{array}$$

Furthermore, having found σ_j, we may also unfold the definition of f with respect to *FamCase*, discovering that for all $\vec{y}$

$$\begin{array}{rl}
f\, \sigma_j \vec{x} & \cong f_j\; \sigma_j \vec{z}; \sigma_j \vec{x}; (\mathsf{refl}\, \sigma_j (\vec{p}; x_i)) \\
 & \cong f'_j\; \vec{y}
\end{array}$$

The latter conversion holds by the computational properties of the proof term generated by the unification algorithm established in the previous chapter.

This shows us that case analysis with constructor form unification really does correspond intensionally to Coquand's case-splitting step. We are now in a position to prove a crucial metatheorem.

6.2.2 conservativity of pattern-matching over OLEG

THEOREM: **conservativity of pattern-matching over OLEG**

> Suppose
>
> $$\begin{array}{lllll} & f & : & \forall\vec{x}\!:\!\vec{S}. & & T \\ \forall\vec{y}\!:\!\vec{Y}_1. & f & & \vec{s}_1 & = & t_1 \\ & \vdots & & \vdots & & \vdots \\ \forall\vec{y}\!:\!\vec{Y}_n. & f & & \vec{s}_n & = & t_n \end{array}$$
>
> is an admissible program according to the characterisation of the previous section, with
>
> - the $\vec{s}_i$ (over $\vec{y} : \vec{Y}_i$) a covering of $\vec{S}$ built interactively by case-splitting with constructor form unification
> - recursive calls structurally smaller on the rth argument
>
> Then there is an OLEG term $f : \forall\vec{x}\!:\!\vec{S}.\,T$ satisfying for each i, for any $\vec{y} : \vec{Y}_i$,
>
> $$f\,\vec{s}_i \;\cong\; t_i$$

PROOF

Let us present the main problem as one of theorem proving. We must prove goal

$$?f\!:\forall\vec{x}\!:\!\vec{S}.\,T$$

However, we must check that however we implement f, it satisfies the computational laws intended by the pattern matching equations.

One of the key aspects of this construction is justifying the recursive calls. We can help ourselves in this regard if we give them highly distinctive types. As they stand, they just have whatever type it is the function returns for the given arguments, which might be something dull. We can introduce a much more informative type as follows

$$\begin{array}{ll} ?G & : \forall\vec{x}\!:\!\vec{S}.\ \mathit{Type} \\ ?\mathit{call} & : \forall\vec{x}\!:\!\vec{S}.\ (G\,\vec{x}) \to T \\ ?\mathit{return} & : \forall\vec{x}\!:\!\vec{S}.\ T \to G\,\vec{x} \\ ?g & : \forall\vec{x}\!:\!\vec{S}.\ G\,\vec{x} \\ !f & = \lambda\vec{x}\!:\!\vec{S}.\ \mathit{call}\,(g\,\vec{x}) \end{array}$$

What has happened? I have defined f in terms of g, a function which returns elements of an as yet unknown $\vec{S}$-indexed $\mathcal{Type}$-family, G. Of course, G is going to turn out to be $\lambda\vec{x} : \vec{S}.\ T$, in the style of the decorative !-bindings from previous chapters, but for now, it remains obscure: we transfer values between T and $\mathsf{G}\,\vec{x}$ by means of a pair of unknowns call and return, both of which will turn out to be the identity function. As things stand, though, the type of a call to g identifies precisely its arguments—when we wish to make a recursive call, we, and also the **blunderbuss** tactic, shall be able to find the hypothesis we need just by looking at its type!

The next step is to eliminate the rth argument of g with the appropriate guarded recursion principle. Suppose S_r is $\mathit{Fam}\,\vec{p}$, where Fam is a dependent family of datatypes.[7] The guarded recursion principle we need is thus FamFix. Eliminating, we obtain the scheme

$$\Phi \;=\; \lambda\vec{z}\!:\!\overline{\mathit{Fam}}.\ \forall\vec{x}\!:\!\vec{S}.\ \vec{z}\!\simeq\!\vec{p};x_r \;\rightarrow\; \mathsf{G}\,\vec{x}$$

In fact, this scheme will have had its constraints optimised in the usual way—there will be none at all if Fam is a simple type. Let us nonetheless consider the general case. The immediate subgoal is

$$\begin{array}{l}
?\mathsf{g}_{guarded}\!: \begin{array}[t]{ll} \forall\vec{z} & : \overline{\mathit{Fam}} \\ \forall \mathit{recs} & : \mathit{FamAux}\,\Phi\,\vec{z} \\ \forall\vec{x} & : \vec{S} \\ \forall\vec{e} & : \vec{z} \simeq \vec{p}; x_r \\ \quad \mathsf{G}\,\vec{x} & \end{array}
\end{array}$$

Intensionally speaking, unfolding the definition of g with respect to FamFix tells us that

$$\begin{array}{rl}
\mathsf{g}\,\vec{x} & \cong \mathit{FamFix}\,\Phi\;\mathsf{g}_{guarded}\,\vec{p};x_r\;\vec{x}\;(\mathsf{refl}\,\vec{p};x_r) \\
 & \cong \mathsf{g}_{guarded}\,\vec{p};x_r\;(\mathit{FamAuxGen}\,\Phi\;\mathsf{g}_{guarded}\,\vec{p};x_r)\;\vec{x}\;(\mathsf{refl}\,\vec{p};x_r)
\end{array}$$

The subgoal constraints require exactly that the $\vec{x}$ are well typed arguments of g, so they reduce by unification to

$$\begin{array}{l}
?\mathsf{g}_{free}\!: \begin{array}[t]{ll} \forall\vec{x} & : \vec{S} \\ \forall \mathit{recs} & : \mathit{FamAux}\,\Phi\,\vec{p};x_r \\ \quad \mathsf{G}\,\vec{x} & \end{array}
\end{array}$$

[7] We may consider any parameters fixed.

That is, we have the same goal as before, but with the addition of the auxiliary premise which is ready to unfold revealing the available recursive calls as we split x_r into cases—it may not be the last argument, as shown here, but its position is immaterial.

Note also that the computational behaviour of terms generated by unification gives

$$\mathsf{g}\,\vec{x} \;\cong\; \mathsf{g}_{free}\,\vec{x}\,(\mathsf{FamAuxGen}\,\Phi\;\mathsf{g}_{guarded}\,\vec{p};x_r)$$

Now we replay the interactive case-splitting process which justified the covering $\vec{s}_i$. Splitting an argument means eliminating it by the case analysis rule for its datatype, then applying the unification tactic to the subgoals. Because the unification tactic implements the same unification algorithm as that which justifies the elementary covering induced by the split, we know we will achieve exactly the same effect.

We are left with subgoals corresponding to the covering

$$\begin{array}{l}
?\mathsf{g}_i : \forall \vec{y} \quad : \vec{Y}_i \\
\qquad \forall recs : \mathsf{FamAux}\,\Phi\,\vec{p}_i; s_{ir} \\
\qquad\quad \mathsf{G}\,\vec{s}_i
\end{array}$$

What is more, we know that case analysis with unification has the right intensional effect, so that

$$\mathsf{g}\,\vec{s}_i \;\cong\; \mathsf{g}_i\,\vec{y}\,(\mathsf{FamAuxGen}\,\Phi\;\mathsf{g}_{guarded}\,\vec{p}_i;s_{ir})$$

It is time to fill in the right-hand side. Let us introduce the premises and refine by *return*:

$$\begin{array}{l}
?\mathsf{g}_i \approx \lambda \vec{y} \quad : \vec{Y}_i \\
\qquad \lambda recs : \mathsf{FamAux}\,\Phi\,\vec{p}_i; s_{ir} \\
\qquad ?r_i \quad : [\vec{s}_i/\vec{x}]T \\
\qquad\quad \mathsf{return}\,\vec{s}_i\;r_i
\end{array}$$

t_i is the expression we want to supply for r_i, but it may contain some recursive $f\,\vec{z}_j$, so we cannot just refine by it. We must replace those applications by *call*s to fresh holes of type $\mathsf{G}\,\vec{z}_j$ first. Since there is no nesting, we may write them in any order, although if nesting was permitted, we would still be able to choose an order. I shall only write one of them in.

$$
\begin{array}{l}
?g_i \approx \lambda\vec{y} \quad : \vec{Y}_i \\
\qquad \lambda \mathit{recs} \colon \mathit{FamAux}\,\Phi\,\vec{p}_i; s_{ir} \\
\qquad ?g_{ij} : G\,\vec{z}_j \\
\qquad\; \mathit{return}\,\vec{s}_i\; t_i[\ldots\,\mathit{call}\,g_{ij}\,\ldots]
\end{array}
$$

Where are we to find these elements of $G\,\vec{z}_j$? From *recs*, of course! Since z_{jr} is, by assumption, structurally smaller than s_{ir}, and must have some type $\mathit{Fam}\,\vec{p}_{ij}$, the type $\mathit{FamAux}\,\Phi\,\vec{p}_i; s_r$ expands to a product containing $\Phi\,\vec{p}_{ij}; z_{jr}$, ie

$$\forall \vec{x} : \vec{S}.\; \vec{p}_{ij}; z_{jr} \simeq \vec{p}; x_r \;\rightarrow\; G\,\vec{x}$$

Let us project this out and call it r. Because $g\,\vec{z}_j$ is well typed, we can find a matching substitution which solves the constraints. Hence we may form

$$r\,\vec{z}_j\;(\mathsf{refl}\,\vec{p}_{ij}; z_{jr}) : G\,\vec{z}_j$$

and thus instantiate g_{ij}.

In point of fact, **blunderbuss** with *recs* is enough to solve g_{ij}, because it solves reflexive equations and searches through Σ-types. Since $G\,\vec{z}_j$ is uniquely the type of the recursive call on those arguments, there is no way the search can come back with the wrong value.

Let us check that this type-directed folding really finds the recursive call. The point is that

$$
\begin{array}{l}
\mathit{FamAuxGen}\,\Phi\,g_{guarded}\,\vec{p}_i; s_{ir} \;\cong \\
\Big\langle \ldots \Big\langle \mathit{FamFix}\,\Phi\,g_{guarded}\,\vec{p}_{ij}; z_{jr}; \ldots \Big\rangle \ldots \Big\rangle
\end{array}
$$

Projecting this out and applying it as shown above gives

$$\mathit{FamFix}\,\Phi\,g_{guarded}\,\vec{p}_{ij}; z_{jr}\;\vec{z}_j\;(\mathsf{refl}\,\vec{p}_{ij}; z_{jr})$$

Compare this with the definition of g above: it folds (by the matching substitution) to

$$g\,\vec{z}_j$$

Hence we know that for all $\vec{y}$

$$\mathsf{g}\,\vec{s}_i \;\cong\; \mathit{return}\, t_i[\ldots \mathit{call}\,(\mathsf{g}\,\vec{z}_j)\ldots]$$

All that remains is to solve and cut G, call and return as suggested earlier. We find that $\mathsf{f}\,\vec{x}$ is exactly $\mathsf{g}\,\vec{x}$. Hence (trivially unfolding and folding), the calls and returns disappear and the g's turn into f's. As required, for each i and all $\vec{y} : \vec{Y}_i$

$$\mathsf{f}\,\vec{s}_i \;\cong\; t_i[\ldots \mathsf{f}\,\vec{z}_j \ldots]$$

□

6.2.3 constructing programs

A man bought a full size replica of Michelangelo's 'David'. He put it in his back garden and invited his friends round to see.

'It's just a big block of white marble.' said they.

His reply: 'I haven't unwrapped it yet.'

The above theorem makes use of the guarded recursion, case analysis and unification technology from the previous two chapters to 'replay' the justification of a pattern matching function known to lie within Coquand's class of admissible definitions. We had the advantage of knowing the equations in advance, and indeed the derivation of the covering—we merely had to check that we could build a term with the right type and computational behaviour. As we shall shortly see, this is only a slight advantage—we can use essentially the same technique and construct the pattern equations as we go.

I propose to supply a collection of tactics for programming. As well as performing theorem-proving actions on the OLEG state, these tactics will create and manipulate associated pattern-matching programs in such a way that they are always justifiable by Coquand's criteria and, once the holes are filled, intensionally correct.

By way of a running example, I propose to construct vlast, the function which extracts the last element of a nonempty vector. Let us fix and suppress the element type A.

$$\mathsf{vlast} : \forall n\!:\!\mathbb{N}.\ \forall x\!:\!\mathsf{vect}\ \mathsf{s}n.\ A$$

I shall not tell you what the pattern matching program is, for the point is to unwrap it.

We begin with a goal

$$?goal\colon \forall n\!:\!\mathbb{N}.\ \forall x\!:\!\mathsf{vect}\ \mathsf{s}n.\ A$$

Here is a tactic which indicates that a given goal should be regarded as a programming problem.

TACTIC: **program**

$$\boxed{\begin{array}{l} ?goal\colon \forall x_1 : S_1 \\ \qquad \vdots \\ \qquad \forall x_r : \mathit{Fam}\,\vec{p} \\ \qquad \vdots \\ \qquad \forall x_n : S_n \\ \qquad\ T[\vec{x}] \end{array}} \Rightarrow \boxed{\begin{array}{lll} ?F & : & \forall\vec{x}\!:\!\vec{S}.\ \mathit{Type} \\ ?\mathit{call} & : & \forall\vec{x}\!:\!\vec{S}.\ (F\,\vec{x}) \to T[\vec{x}] \\ ?\mathit{return} & : & \forall\vec{x}\!:\!\vec{S}.\ T[\vec{x}] \to F\,\vec{x} \\ !\Phi & = & \lambda\vec{z}\!:\overline{\mathit{Fam}} \\ & & \quad \forall\vec{x}\!:\vec{S} \\ & & \quad \forall\vec{e}\!:\vec{z} \simeq \vec{p}; x_r \\ & & \qquad F\,\vec{x} \\ ?f_0 & : & \forall\vec{x}\ \ :\vec{S} \\ & & \quad \forall \mathit{recs}\colon \mathit{FamAux}\,\Phi\,\vec{p}; x_r \\ & & \qquad F\,\vec{x} \\ !f & = & \lambda\vec{x}\!:\!\vec{S}.\ \mathit{FamFix}\,\Phi\ f_0\ \vec{x}\,(\mathsf{refl}\,\vec{p}; x_r) \\ !goal & = & \lambda\vec{x}\!:\!\vec{S}.\ \mathit{call}\,(f\,\vec{x}) \end{array}}$$

$$[f_0]\ f\,\vec{x}\ = ? \qquad (\vec{x} : \vec{S})$$

PROOF

The tactic

program $n\ x_r$

turns *goal*, which must have at least n premises, into a programming problem. *goal* is solved by appeal to a function f of arity n, recursive on its rth argument.

As before, the more informative type family F is introduced, together with *call* and *return*, then x_r is eliminated with the relevant guarded recursion theorem. This leaves us filling in the body of the function f_0. Associated with f_0 is a pattern matching equation labelled $[f_0]$, with its pattern variables listed in parentheses, which describes the aspects of f's behaviour for which f_0 accounts. The left-hand side of this equation is $f\vec{x}$, indicating that f_0 describes the effect of f for any arguments matching the free pattern $\vec{x}$—that is, any arguments at all. The right-hand side is a placeholder $?$, indicating that we have not yet decided what f returns for arguments matching $\vec{x}$.

As we split the goal f_0 to yield subgoals for specific constructor cases, so we shall split the equation $[f_0]$ into the corresponding equations with more instantiated patterns. These equations constitute the pattern matching program we are building, and we shall maintain the invariant that their patterns constitute a **covering** in accordance with Coquand's definition. This is trivially the case for $[f_0]$. □

By the way, if the program is not recursive, let us allow the omission of the x_r from the tactic call. Correspondingly, we do not need to apply the guarded recursion theorem. The type of f_0 is then the same as that of f. The rest of the techique is unaffected.

Now, vlast is structurally recursive on either of its arguments, so it is immaterial which we choose. I shall pick the vector. Let us see the effect of the tactic:

$$
\begin{array}{lll}
?\mathsf{Vlast} & : & \forall n{:}\mathbb{N}.\ \forall x{:}\mathsf{vect}\ \mathsf{s}n.\ \mathit{Type} \\
?\mathit{call} & : & \forall n{:}\mathbb{N}.\ \forall x{:}\mathsf{vect}\ \mathsf{s}n.\ (\mathsf{Vlast}_n\ x) \to A \\
?\mathit{return} & : & \forall n{:}\mathbb{N}.\ \forall x{:}\mathsf{vect}\ \mathsf{s}n.\ A \to \mathsf{Vlast}_n\ x \\
!\Phi & = & \lambda m{:}\ \mathbb{N} \\
 & & \lambda y : \mathsf{vect}\ m \\
 & & \forall n : \mathbb{N} \\
 & & \forall x : \mathsf{vect}\ \mathsf{s}n \\
 & & \forall e_1{:}\ m \simeq \mathsf{s}n \\
 & & \forall e_2{:}\ y \simeq x \\
 & & \quad \mathsf{Vlast}_n\ x \\
?f_0 & : & \forall n \quad : \mathbb{N} \\
 & & \forall x \quad : \mathsf{vect}\ \mathsf{s}n \\
 & & \forall \mathit{recs}{:}\ \mathsf{vectAux}\ \Phi\ n\ x \\
 & & \quad \mathsf{Vlast}_n\ x \\
!\mathsf{vlast} & = & \lambda n{:}\mathbb{N}.\ \lambda x{:}\mathsf{vect}\ \mathsf{s}n.\ \mathsf{vectFix}\ \Phi\ f_0\ \mathsf{s}n\ x\ (\mathsf{refl}\ \mathsf{s}n)\ (\mathsf{refl}\ x) \\
!\mathit{goal} & = & \lambda n{:}\mathbb{N}.\ \lambda x{:}\mathsf{vect}\ \mathsf{s}n.\ \mathit{call}\ (\mathsf{vlast}_n\ x)
\end{array}
$$

$$[f_0]\ \mathsf{vlast}_n\ x\ = ? \qquad (n : \mathbb{N}; x : \mathsf{vect}\ \mathsf{s}n)$$

Now that we know how to start the process, we must figure out how to build coverings.

TACTIC: **split**

$$\boxed{\begin{array}{l} ?f_i\colon \forall \vec{y} \quad : \vec{Y}_i \\ \quad \forall recs\colon \mathit{FamAux}\,\Phi\ \vec{p}_i; s_{ir} \\ \qquad F\vec{s}_i \end{array}} \Rightarrow \boxed{\begin{array}{l} ?f_j\colon \forall \vec{z} \quad : \vec{Z}_j \\ \quad \forall recs\colon \mathit{FamAux}\,\Phi\ \sigma_j(\vec{p}_i; s_{ir}) \\ \qquad F\,\sigma_j\vec{s}_i \\ \vdots \end{array}}$$

$$[f_i]\ f\vec{s}_i\ = ? \quad (\vec{y} : \vec{Y}_i) \qquad\qquad [f_j]\ f\sigma_j\vec{s}_i\ = ? \quad (\vec{z} : \vec{Z}_j)$$
$$\vdots$$

where $y_k : \mathit{Fam}'\ \vec{r}$
and $\mathsf{Con}_j : \forall \vec{x} : \vec{X}.\ \mathit{Fam}'\ \vec{t}$
and σ_j is a most general unifier (from $\vec{y}; \vec{x}$ to terms over $\vec{z}$) of $\vec{r}; y_k$ and $\vec{t}; (\mathsf{Con}_j\ \vec{x})$, both in $\overline{\mathit{Fam}'}$

PROOF

The tactic

split y_k

performs a case split on y_k in subgoal f_i, yielding a bunch of subgoals f_j. The equation for f_i is split correspondingly into equations for the f_j.

As above, we eliminate y_k in subgoal f_i via the case analysis principle for Fam' and then apply unification. The tactic succeeds provided unification in each case either shows that there is no unifier or finds a mgu σ_j. The resulting collection of mgus justifies the new covering, and it also justifies the unfoldings which show that $f_i\ \vec{s}_i$ reduces in each case to $f_j\ \sigma_j\vec{s}_i$.

The invariant that the patterns of the equations associated with the subgoals form a covering is maintained. $\square$

In our example, we shall certainly need to split on the vector x. There is no way to make a nonempty vector with vnil, so only the vcons case survives:

$$
\begin{array}{l}
?f_c\colon \forall n \;:\mathbb{N}\\
\quad \forall h \;: A\\
\quad \forall t \;: \mathsf{vect}\; n\\
\quad \forall recs\colon \Sigma t_{rec}\colon \forall m\colon \mathbb{N}\\
\qquad\qquad\qquad \forall x : \mathsf{vect}\; \mathsf{s}m\\
\qquad\qquad\qquad \forall e_1\colon n \simeq \mathsf{s}m\\
\qquad\qquad\qquad \forall e_2\colon t \simeq x\\
\qquad\qquad\qquad\quad \mathsf{Vlast}_m\; x\\
\qquad\qquad\quad \mathsf{vectAux}\; \Phi\; n\; t\\
\qquad \mathsf{Vlast}\; n\; (\mathsf{vcons}\; h\; t)\\
[f_c]\; \mathsf{vlast}_n\; (\mathsf{vcons}\; h\; t) \;\; = ? \quad (n : \mathbb{N}; h : A; t : \mathsf{vect}\; n)
\end{array}
$$

Note that the wallet of recursions has unfolded by one step, showing us the recursive call we could make for t, but for the fact that it is not known to be nonempty. The effect of our informative retyping has been to make the conclusion of f_c tell us the patterns in the corresponding equation.

We must make one more split before we can finish the job. If this were simply typed programming, we would split t to see whether h is the last element or not. However, we do not need to destructure t—splitting n will tell us all we need.

$$
\begin{array}{l}
?f_{cz}\colon \forall h \;: A\\
\quad \forall t \;: \mathsf{vect}\; 0\\
\quad \forall recs\colon \Sigma t_{rec}\colon \forall m\colon \mathbb{N}\\
\qquad\qquad\qquad \forall x : \mathsf{vect}\; \mathsf{s}m\\
\qquad\qquad\qquad \forall e_1\colon 0 \simeq \mathsf{s}m\\
\qquad\qquad\qquad \forall e_2\colon t \simeq x\\
\qquad\qquad\qquad\quad \mathsf{Vlast}_m\; x\\
\qquad\qquad\quad \mathsf{vectAux}\; \Phi\; 0\; t\\
\qquad \mathsf{Vlast}\; 0\; (\mathsf{vcons}\; h\; t)\\
?f_{cs}\colon \forall n \;:\mathbb{N}\\
\quad \forall h \;: A\\
\quad \forall t \;: \mathsf{vect}\; \mathsf{s}n\\
\quad \forall recs\colon \Sigma t_{rec}\colon \forall m\colon \mathbb{N}\\
\qquad\qquad\qquad \forall x : \mathsf{vect}\; \mathsf{s}m\\
\qquad\qquad\qquad \forall e_1\colon \mathsf{s}n \simeq \mathsf{s}m\\
\qquad\qquad\qquad \forall e_2\colon t \simeq x\\
\qquad\qquad\qquad\quad \mathsf{Vlast}_m\; x\\
\qquad\qquad\quad \mathsf{vectAux}\; \Phi\; \mathsf{s}n\; t\\
\qquad \mathsf{Vlast}\; \mathsf{s}n\; (\mathsf{vcons}\; h\; t)\\
[f_{cz}]\; \mathsf{vlast}_0\; (\mathsf{vcons}\; h\; t) \;\; = ? \quad (h : A; t : \mathsf{vect}\; 0)\\
[f_{cs}]\; \mathsf{vlast}_{\mathsf{s}n}\; (\mathsf{vcons}\; h\; t) \;\; = ? \quad (n : \mathbb{N}; h : A; t : \mathsf{vect}\; \mathsf{s}n)
\end{array}
$$

Observe that in the former case, there is no way the matching problem can ever be solved to allow access to a recursive call, whilst in the latter, the way is clear.

Having split as far as is necessary, we should like to fill in the right-hand sides.

TACTIC: **return**

$$\boxed{\begin{array}{l} ?f_i\colon \forall \vec{y} \quad : \vec{Y}_i \\ \quad \forall recs\colon \mathsf{FamAux}\,\Phi\,\vec{p}_i; s_{ir} \\ \qquad F\,\vec{s}_i \end{array}} \Rightarrow \boxed{\begin{array}{l} !f_i = \lambda \vec{y} \quad : \vec{Y}_i \\ \quad \lambda recs\colon \mathsf{FamAux}\,\Phi\,\vec{p}_i; s_{ir} \\ \qquad \mathit{return}\,t'_i \end{array}}$$

$$[f_i]\ f\,\vec{s}_i \ = ? \quad (\vec{y} : \vec{Y}_i) \qquad\qquad [f_i]\ f\,\vec{s}_i \ = \ \mathit{return}\,t'_i \quad (\vec{y} : \vec{Y}_i)$$

PROOF

Given

return t_i

we may, as before, form t'_i by replacing the recursive calls $f\,\vec{z}$ by *call*s to holes of type $F\,\vec{z}$. If these calls are structurally smaller at argument r, we will once again be able to solve these holes by appeal to appropriate projections from *recs*.

The structural condition ensures that the new equation is acceptable, and the same argument as that in the above theorem shows that it holds intensionally. □

Our example has two cases. For the singleton, the value should just be the head. **return** h gives us

$$\begin{array}{l} !f_{cz}\ \lambda h \quad : A \\ \quad \lambda t \quad : \mathsf{vect}\,0 \\ \quad \lambda recs\colon \Sigma t_{rec}\colon \forall m\colon \mathbb{N} \\ \qquad\qquad\qquad \forall x : \mathsf{vect}\,\mathsf{s}m \\ \qquad\qquad\qquad \forall e_1\colon 0 \simeq \mathsf{s}m \\ \qquad\qquad\qquad \forall e_2\colon t \simeq x \\ \qquad\qquad\qquad\quad \mathsf{Vlast}_m\,x \\ \qquad\qquad \mathsf{vectAux}\,\Phi\,0\,t \\ \qquad \mathit{return}\,h \\ [f_{cz}]\ \mathsf{vlast}_0\,(\mathsf{vcons}\,h\,t) \ = \ \mathit{return}\,h \quad (h : A; t : \mathsf{vect}\,0) \end{array}$$

In the case with the nonempty tail, we make the recursive call

$$\textbf{return}\ \mathsf{vlast}_n\ t$$

This becomes

$$
\begin{array}{l}
?f_{cs}\colon \lambda n \quad : \mathbb{N} \\
\qquad \lambda h \quad : A \\
\qquad \lambda t \quad : \mathsf{vect}\ \mathsf{s}n \\
\qquad \lambda recs\colon \Sigma t_{rec}\colon \forall m\colon \mathbb{N} \\
\qquad\qquad\qquad\qquad \forall x : \mathsf{vect}\ \mathsf{s}m \\
\qquad\qquad\qquad\qquad \forall e_1\colon \mathsf{s}n \simeq \mathsf{s}m \\
\qquad\qquad\qquad\qquad \forall e_2\colon t \simeq x \\
\qquad\qquad\qquad\qquad\quad \mathsf{Vlast}_m\ x \\
\qquad\qquad\qquad \mathsf{vectAux}\ \Phi\ \mathsf{s}n\ t \\
\qquad ?v_t \quad : \mathsf{Vlast}_n\ t \\
\qquad\ \ \mathit{return}\ (\mathit{call}\ v_t)
\end{array}
$$

A quick search reveals the appropriate recursion

$$!v_t = recs.1\ n\ t\ (\mathsf{refl}\ \mathsf{s}n)\ (\mathsf{refl}\ t)$$

justifying the equation

$$
\begin{array}{rcl}
[f_{cs}]\ \mathsf{vlast}_{\mathsf{s}n}\ (\mathsf{vcons}\ h\ t) & = & \mathit{return}\ (\mathit{call}\ (\mathsf{vlast}_n\ t)) \\
 & & (n : \mathbb{N}; h : A; t : \mathsf{vect}\ \mathsf{s}n)
\end{array}
$$

We can tell when a program is finished—once all the placeholding ?s have gone. We may now solve F, call and return as in the previous case. This leaves us with a real term f whose intensional behaviour corresponds to the associated equational program, which satisfies Coquand's conditions by construction.

Our example becomes

$$
\begin{array}{rcl}
\mathsf{vlast}_0\ (\mathsf{vcons}\ h\ t) & = & h \\
\mathsf{vlast}_{\mathsf{s}n}\ (\mathsf{vcons}\ h\ t) & = & \mathsf{vlast}_n\ t
\end{array}
$$

This is a rather subtle way to write vlast which makes crucial use of the extra indexing information. Naïvely erasing the indices in the hope of recovering a function over ordinary lists yields

$$\mathsf{last}\,(\mathsf{cons}\,h\,t) \;=\; h$$
$$\mathsf{last}\,(\mathsf{cons}\,h\,t) \;=\; \mathsf{last}\,t \qquad (\times)$$

This is clearly not the right function, or indeed a function at all.

Of course, we could have split t and built the usual

$$\mathsf{vlast}_0\,(\mathsf{vcons}\,h\,\mathsf{vnil}) \;=\; h$$
$$\mathsf{vlast}_{\mathsf{s}n}\,(\mathsf{vcons}\,h\,(\mathsf{vcons}\,h'\,t)) \;=\; \mathsf{vlast}_n\,(\mathsf{vcons}\,h'\,t)$$

I do not wish to embark on a discussion of the relative merits of these two programs—I will merely point out that computation on the indices of a type behaves differently from computation on the type directly, and sometimes interestingly so.

The combined effect of these tactics is to allow a similar style of interactive program development to that available in ALF—not only can we build the same programs, but we can do so in the same way.

However, this is not enough. Having built these programs, how do we store them? For OLEG, the representation of the program is still a ghastly term involving guarded recursion, case analysis and unification, all painstakingly recorded. Why can't we just write the pattern matching equations down? The construction of programs in this section has relied on knowing more than just the equations—we have also exploited the justification that the equations satisfy Coquand's conditions, and we have recovered a process for building those justifications interactively.

In the next section, I consider how much we can do with just the equations.

6.3 recognising programs

The question asked in this section is 'for which pattern matching programs can we recover the justification?'. Sadly, as we shall see, the answer is not 'all of them'. Nonetheless, it is worth analysing at what point the problem becomes undecidable, with a view to building a system where we can store enough information to allow the recovery. The aim is to describe a class of **recognisable** programs. I have made some progress in this direction, although there is work still to be done. I feel some discussion of the problem is worthwhile, not least because the techniques described here are sufficient to recognise all the examples in this thesis, which will save me the trouble of describing the construction.

Our existing tactical presentation of program construction will be of assistance to us. We have built ourselves a structural editor for acceptable programs. Let us now imagine this editor being used not by humans but by a mechanical **recogniser** whose task is to take a set of pattern matching equations and build the program. This echoes the view I have taken of the constructions with which we may equip our datatypes—they make use of the tools we have developed for theorem-proving, ie the structural editing of OLEG terms. I know relatively little about writing compilers, but it seems to me that a promising first move is to build a structural editor for the target language.

The three tactics from the previous section divide recognition into three phases:

- identify an argument position on which the recursion in the program is guarded (and apply **program**)
- show that the patterns form a covering (by applying **split**)
- fill in the right-hand sides (with **return**)

The first and third of these are easy. We may simply check each argument position in turn for one which satisfies the guardedness condition before applying **program**. Meanwhile, **return** codes up exactly the operation we need. It is the second phase, checking the covering, where undecidability creeps in.

6.3.1 recursion spotting

Given a goal

$$?goal : \forall \vec{x} : \vec{S}.\ T$$

and a pattern matching program of arity n

$$f_i\, \vec{s}_i \ = \ t_i \qquad (\vec{y} : \vec{Y}_i)$$

It is easy to check for recursive calls to f in the t_i. It is also easy to find for each equation the set R_i of argument positions which satisfy the guardedness condition. We may say that non-recursive equations are guarded in all their argument positions. Coquand's criterion requires that the intersection of the R_i be nonempty. If so, we may apply the **program** tactic for any of the indicated positions.

It is possible that this procedure will yield a choice of positions. While any of them will do, we may still have a preference. For example, structurally recursive programs over vects or fins are necessarily also guarded on their natural number indices. It does not really matter which we choose, but I would prefer the recursion to be on the datatypes themselves, rather than the indices, as this produces a justification which seems to me more intuitive.

Now we have found r, we may apply

$$\textbf{program}\ n\ x_r$$

This leaves us with subgoal f_0 and its associated equation

$$[f_0]\ f\vec{x}\ =?\quad (\vec{x} : \vec{S})$$

Now, let us relate the **program equations** we are trying to construct with the associated equations in the current state of the construction. I call the latter the **covering equations** because their patterns are guaranteed to form a covering. In particular, we are certain that each program equation is **covered** by exactly one of the covering equations—only one of the covering patterns may be instantiated to give each program pattern.

For each covering equation, we may collect the program equations it covers—this is just a first-order matching problem. There are three possibilities:

- there is one equation, and it is covered **exactly**, meaning that the covered equation also covers its coverer—the patterns are the same, up to renaming of variables and **return** is now applicable
- there is at least one covered equation, but none covered exactly, so splitting will be necessary
- there are no equations covered—this means either that the program is incomplete, or that there is nothing to cover—an undecidable type inhabitation problem

Let us look at each case in turn.

6.3.2 exact problems

If we have reached the stage where a covering equation

$$[f_i]\; f\vec{s}_i \;=?\quad (\vec{y} : \vec{Y}_i)$$

exactly covers a program equation

$$f\vec{s}_i \;=\; t_i \quad (\vec{y} : \vec{Y}_i)$$

then we may apply tactic

return t_i

We know that the guarded recursive calls will be available to us, so we complete this branch of the justification.

6.3.3 splitting problems

We have a covering equation

$$[f_j]\; f\vec{s}_j \;=?\quad (\vec{y} : \vec{Y})$$

which covers several program equations

$$f\,\sigma_i\vec{s}_j \;=\; t_i \quad (\vec{y}_i : \vec{Y}_i)$$

where σ_i is a (matching) substitution from the $\vec{y}$ to terms over the $\vec{y}_i$.

Each time we split a covering equation, we introduce at least one more constructor symbol into its patterns (since patterns may be nonlinear, replacing a pattern variable by a constructor form may add more than one constructor symbol to the pattern). We may exploit this property to measure how far away the program equations are from being covered exactly. In order for a matching to exist, a program equation must contain at least as many constructor symbols as the covering equation, so we may simply count the excess.

Suppose f_j covers equation i and is then split into several cases, one of which, f_k say, covers i also. We know the constructor excess of equation i over f_k is strictly less than that over f_j, because the f_k patterns contain more constructor symbols. Hence, we may keep splitting problems until they become exact or empty and be sure that the process will terminate.

Which split should we make? In order to see this, we must expand each of the program equations in terms of f_j. We know

$$f\,\sigma_i\vec{s}_j \;\cong\; f_j\;\sigma_i\vec{y}\;\langle\ldots\rangle$$

where the tuple is just the collection of accessible recursive call values

If there is a y_k such that each $\sigma_i y_k$ is constructor-headed, then y_k is a candidate for splitting. If that split is successful, constructor symbols will appear at y_k, yielding simpler subproblems.

Ideally, everything will be in constructor form, splits will always yield solvable unification problems, so we may split any of the candidates and carry on. Which candidate should we choose? I would suggest we prefer candidates higher up the type dependency hierarchy, as these may induce splits in other arguments by unification. For example, if we are building a covering of the vects, splitting a vector will automatically split its length into $\mathsf{0}$ and s cases, while merely splitting the length will leave us with work still to do on the vector.

Even if there are awkward non-constructor expressions involved, there will only be a finite number of candidates at any stage, so we may keep trying to split until one works. It is conceivable that, in an impure world, splitting something too early may yield unsolvable unfication problems later. For the sake of argument, we may consider the recogniser to be nondeterministic—any justification will do. This is far from satisfactory, but it is safe.

6.3.4 empty problems

We have a subgoal

$$?f_i \colon \forall \vec{y} \colon \vec{Y}_i.\ \forall recs \colon \ldots.\ F\,\vec{s}_i$$

but no program equations to give us a clue what should go on the right-hand side, or how to do any further case splitting. This either means that the programmer has forgotten a case, or else one of the Y_{ij} is empty, and there is morally no need to explain what should happen, as the case cannot arise.

Types can be empty for arbitrarily subtle reasons—the type inhabitation problem is undecidable. Even if we restrict everything in sight to constructor forms, we will still be able to code up the halting problem as a datatype inhabitation question (see table 6.1). Some empty types, such as the simple type with one step constructor and no base constructors, require an inductive argument to prove them empty. Others may eventually disappear after enough case splitting, but there is no way of telling how much is required.

If we cannot be totally clever, can we be totally stupid? That is, can we reject these problems out of hand? Unfortunately not. The elimination rule for the $\mathbb{0}$ type has no ι-reductions and thus corresponds to an empty program—if we are to recognise the recursion operators provided for datatypes as bona fide programs, we shall have to be able to solve some empty problems.

Having rejected trying zero and infinity steps of case splitting, the only other intuitively plausible option is one. Let us try one case split on each argument in turn, and if any proves the goal, we have success. Otherwise, the problem is too hard and we fail to recognise the program. This is enough to allow us to ignore cases with arguments in types like $\mathbb{0}$, $\mathsf{fin}\ 0$ and so on. The idea is that a type is **obviously empty** if there is no constructor-headed expression which inhabits it.

Effectively, the programmer must deal with non-obviously empty types explicitly, by calling subprograms which eliminate them. If we construct a program interactively which takes several steps of splitting to dispose of a type, we may represent the last step by an obviously empty subprogram which gets called from the last case where a pattern existed. This effectively records the splitting process used to dismiss the type. If we find ourselves repeating ourselves, perhaps we should be able to register commonly used emptiness proofs in such a way that they are tried along with splitting whenever an empty problem is encountered.

It is conceivable that one search path through checking a covering may lead to only obvious empty problems, while another may lead to a non-obvious empty problem. Once again, we may save ourselves by crude nondeterminism. Whether we can do better remains to be seen.

6.4 extensions

I feel I should make brief mention of a number of obvious extensions to the class of programs we should be willing to consider, none of which is particularly controversial.

6.4.1 functions with varying arity

In simply typed languages, we are not used to seeing functions with varying arity. Certainly, the use of curried functions is commonplace, but there is nothing serious

- basic datatypes

$$\frac{}{\mathsf{start} : \mathsf{state}} \qquad \frac{}{\mathsf{halt} : \mathsf{state}} \quad \cdots \text{ other } \mathsf{states}$$

$$\frac{}{\mathsf{blank} : \mathsf{symbol}} \quad \cdots \text{ other } \mathsf{symbols}$$

$$\frac{}{\mathsf{left} : \mathsf{move}} \qquad \frac{}{\mathsf{right} : \mathsf{move}}$$

- describing the machine

$$\begin{aligned} \mathsf{transition} &= \mathsf{state} \times \mathsf{symbol} \times \mathsf{state} \times \mathsf{symbol} \times \mathsf{move} \\ \mathsf{transitions} &= \mathsf{list\ transition} \\ \mathsf{tape} &= (\mathsf{tsil\ symbol}) \times \mathsf{symbol} \times (\mathsf{list\ symbol}) \\ \mathsf{configuration} &= \mathsf{state} \times \mathsf{tape} \end{aligned}$$

(tsil is the type of lists built by adding elements at the right-hand end with the constructor snoc. I overload nil.)

- list membership (for any element type A)

$$\frac{h : A \quad t : \mathsf{list}\ A}{\mathsf{find}\ h\ t : \mathsf{member}\ h\ (\mathsf{cons}\ h\ t)} \qquad \frac{h : A \quad T : \mathsf{member}\ x\ t}{\mathsf{seek}\ h\ T : \mathsf{member}\ x\ (\mathsf{cons}\ h\ t)}$$

- updating the tape ($\mathsf{update} : \mathsf{tape} \to \mathsf{move} \to \mathsf{tape} \to \mathit{Type}$)

$$\frac{s : \mathsf{symbol} \quad r : \mathsf{list\ symbol}}{\mathsf{lblank}\ s\ r : \mathsf{update}\ \langle \mathsf{nil}; s; r\rangle\ \mathsf{left}\ \langle \mathsf{nil}; \mathsf{blank}; \mathsf{cons}\ s\ r\rangle}$$

$$\frac{l : \mathsf{tsil\ symbol} \quad t, s : \mathsf{symbol} \quad r : \mathsf{list\ symbol}}{\mathsf{lmove}\ l\ t\ s\ r : \mathsf{update}\ \langle \mathsf{snoc}\ l\ t; s; r\rangle\ \mathsf{left}\ \langle l; t; \mathsf{cons}\ s\ r\rangle}$$

$$\frac{l : \mathsf{tsil\ symbol} \quad s : \mathsf{symbol}}{\mathsf{rblank}\ l\ s : \mathsf{update}\ \langle l; s; \mathsf{nil}\rangle\ \mathsf{right}\ \langle \mathsf{snoc}\ l\ s; \mathsf{blank}; \mathsf{nil}\rangle}$$

$$\frac{l : \mathsf{tsil\ symbol} \quad s, t : \mathsf{symbol} \quad r : \mathsf{list\ symbol}}{\mathsf{rmove}\ l\ s\ t\ r : \mathsf{update}\ \langle l; s; \mathsf{cons}\ t\ r\rangle\ \mathsf{right}\ \langle \mathsf{snoc}\ l\ s; t; r\rangle}$$

- one step ($\mathsf{step} : \mathsf{transitions} \to \mathsf{configuration} \to \mathsf{configuration} \to \mathit{Type}$)

$$\frac{tr : \mathsf{member}\ \langle q; s; q'; s'; d\rangle\ trs \quad u : \mathsf{update}\ \langle l; s'; r\rangle\ d\ tape}{\mathsf{do}\ tr\ u : \mathsf{step}\ trs\ \langle q; \langle l; s; r\rangle\rangle\ \langle q'; tape\rangle}$$

- halting problem ($\mathsf{halts} : \mathsf{transitions} \to \mathsf{configuration} \to \mathsf{tape} \to \mathit{Type}$)

$$\frac{trs : \mathsf{transitions} \quad tape : \mathsf{tape}}{\mathsf{stop}\ trs\ tape : \mathsf{halts}\ trs\ \langle \mathsf{halt}; tape\rangle\ tape}$$

$$\frac{step : \mathsf{step}\ trs\ X\ Y \quad halts : \mathsf{halts}\ trs\ Y\ tape}{\mathsf{go}\ step\ halts : \mathsf{halts}\ trs\ X\ tape}$$

Table 6.1: coding the halting problem

happening that $\beta\eta$-equivalence cannot explain. There seems to be little motivation for allowing functions to be *defined* with arity varying between pattern equations.

By contrast, there are some dependently typed functions for which such a relaxation in the syntax would be of genuine benefit. These tend to arise when we write one function to compute types involved in another. For example

$$
\begin{aligned}
&\mathsf{Sum} : \mathbb{N} \to \mathit{Type} \\
&\mathsf{Sum}\,\mathsf{0} \ = \ \mathbb{N} \\
&\mathsf{Sum}\,\mathsf{s}n \ = \ \mathbb{N} \to \mathsf{Sum}\,n
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{sum} : \forall n\!:\!\mathbb{N}.\ \mathsf{Sum}\,n \\
&\mathsf{sum}\,\mathsf{0} \ = \ \mathsf{0} \\
&\mathsf{sum}\,\mathsf{s0}\,x \ = \ x \\
&\mathsf{sum}\,\mathsf{ss}n\,x\,y \ = \ \mathsf{sum}\,\mathsf{s}n\,(\mathsf{plus}\,x\,y)
\end{aligned}
$$

The first argument of sum is the number of subsequent arguments, and the function computes their sum. You might well point out that I could make the arities uniform by λ-abstraction, but that is because I am not doing any pattern matching on the newly exposed arguments. Of course, in any case I can always introduce subprograms, but why should I have to?

You might also suspect that such functions are uncommon in practice, and thus not worth the trouble. There are three things to say to that:

- Dependently typed programming is still in its infancy—we do not know which techniques will turn out to be common in practice.
- This is the kind of technique which is used somewhat less frivolously in strong normalisation proofs—we compute a meta-level function type from an object-level function type, then we compute the appropriate metal-level function to inhabit it.
- This sort of behaviour is already supported in as industrial a programming language as C. The remarkably common `printf` command takes a formatting string, followed by arguments appropriate to the fields to be printed—you hope. Of course, there is no check to see that it makes sense. C compilers do not blink twice at

```
printf(``%s%s%s'');
```

but the effect is seldom benign. Dependent types sanitise these rather frightening functions.

We may accommodate this behaviour by adjusting the definition of covering to allow **lengthening** of pattern sequences by fresh pattern variables, provided the result type beforehand is functional. These extended patterns may then be split as before. Each lengthening can, of course, be replaced by a call to a subprogram in order to recover the uniform arity of the original. The treatment of recursion is as before. Recursive calls can be recognised provided they have at least the arity of the pattern to which the guarded recursion principle was applied. Longer sequences of arguments can be cut in two, leaving a recursive *call* of the right length which is then applied further.

6.4.2 more exotic recursion

While it is sufficient to facilitate functions which are only recursive on one argument position, it is nonetheless convenient to allow more complex structures to be built into a single function, rather than forcing the programmer to break them up. The traditional example is Ackermann's function:

$$\begin{array}{l}
\mathsf{ack} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
\mathsf{ack}\,\mathsf{0}\,n \;=\; \mathsf{s}n \\
\mathsf{ack}\,\mathsf{s}m\,\mathsf{0} \;=\; \mathsf{ack}\,m\,\mathsf{s0} \\
\mathsf{ack}\,\mathsf{s}m\,\mathsf{s}n \;=\; \mathsf{ack}\,m\,(\mathsf{ack}\,\mathsf{s}m\,n)
\end{array}$$

The recursion in this function is **lexicographic** in the sense that either the first argument decreases structurally, or else it stays the same, but the second argument decreases. It can be split into a pair of Coquand-accepted primitive recursive functionals as follows:

$$\begin{array}{l}
\mathsf{ack}_{sm} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N} \\
\mathsf{ack}_{sm}\,ack_m\,\mathsf{0} \;=\; ack_m\,\mathsf{s0} \\
\mathsf{ack}_{sm}\,ack_m\,\mathsf{s}n \;=\; ack_m\,(\mathsf{ack}_{sm}\,ack_m\,n) \\
\mathsf{ack} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
\mathsf{ack}\,\mathsf{0} \;=\; \mathsf{s} \\
\mathsf{ack}\,\mathsf{s}m \;=\; \mathsf{ack}_{sm}\,(\mathsf{ack}\,m)
\end{array}$$

What has happened here? For a start, the main ack function has been η'd into a functional. This enables the $\mathsf{s}m$ case to be delegated to the auxiliary function ack_{sm}.

This receives as an argument the function $\mathsf{ack}\, m$, available by structural recursion—it is thus free to apply this function, as well as making its own guarded calls.[8]

Would not all but the most die-hard of origami programmers[9] prefer to write the lexicographic version? In fact, we already have the tools to construct it interactively—suppose we have reached the following stage:

$$
\begin{array}{lll}
?\mathsf{Ack} & : & \forall m, n : \mathbb{N}.\ \mathcal{Type} \\
?\mathit{call} & : & \forall m, n : \mathbb{N}.\ (\mathsf{Ack}\, m\, n) \to \mathbb{N} \\
?\mathit{return} & : & \forall m, n : \mathbb{N}.\ \mathbb{N} \to \mathsf{Ack}\, m\, n \\
!\mathsf{ack}_0 & = & \lambda \mathit{recs} : \mathbb{1} \\
 & & \lambda n \quad : \mathbb{N} \\
 & & \quad \mathsf{s} n \\
?\mathsf{ack}_s & : & \forall m \quad : \mathbb{N} \\
 & & \forall \mathit{recs} : (\forall n : \mathbb{N}.\ \mathsf{Ack}\, m\, n) \times (\mathbb{N}\mathsf{Aux}\ \ldots\ m) \\
 & & \forall n \quad : \mathbb{N} \\
 & & \quad \mathsf{Ack}\, \mathsf{s} m\, n \\
!\mathsf{ack} & = & \mathbb{N}\mathsf{Fix} \ldots
\end{array}
$$

The *recs* argument gives us access to guarded recursion on the first argument. We may now add guarded recursion on the second (for the same first argument) by eliminating n with $\mathbb{N}\mathsf{Fix}$, fixing m and *recs*:

$$
\begin{array}{ll}
?\mathsf{ack}_s : & \forall m \quad : \mathbb{N} \\
 & \forall \mathit{recs} : (\forall n : \mathbb{N}.\ \mathsf{Ack}\, m\, n) \times (\mathbb{N}\mathsf{Aux}\ \ldots\ m) \\
 & \forall n \quad : \mathbb{N} \\
 & \forall \mathit{recs}' : \mathbb{N}\mathsf{Aux}\ (\mathsf{Ack}\, \mathsf{s} m)\ n \\
 & \quad \mathsf{Ack}\, \mathsf{s} m\, n
\end{array}
$$

Case splitting on n now gives us

[8] We would not need to pass ack_m explicitly through the recursion if we could define ack_{sm} locally to the successor case of ack.

[9] An origami programmer only uses pattern matching to define fold operators.

$$
\begin{array}{ll}
?\mathsf{ack}_{s0} : & \forall m \quad : \mathbb{N} \\
& \forall \mathit{recs} : (\forall n : \mathbb{N}.\ \mathsf{Ack}\ m\ n) \times (\mathbb{N}\mathsf{Aux}\ \ldots\ m) \\
& \forall \mathit{recs}' : \mathbb{1} \\
& \quad \mathsf{Ack}\ \mathsf{s}m\ 0 \\
?\mathsf{ack}_{ss} : & \forall m \quad : \mathbb{N} \\
& \forall \mathit{recs} : (\forall n : \mathbb{N}.\ \mathsf{Ack}\ m\ n) \times (\mathbb{N}\mathsf{Aux}\ \ldots\ m) \\
& \forall n \quad : \mathbb{N} \\
& \forall \mathit{recs}' : (\mathsf{Ack}\ \mathsf{s}m\ n) \times (\mathbb{N}\mathsf{Aux}\ (\mathsf{Ack}\ \mathsf{s}m)\ n) \\
& \quad \mathsf{Ack}\ \mathsf{s}m\ \mathsf{s}n
\end{array}
$$

For the ack_{s0} case, we may project the appropriate component of *recs*. Looking at ack_{ss} in more detail, the nested right-hand side translates by *call* and *return* to

$$
\begin{array}{ll}
?\mathsf{ack}_{ss} : & \lambda m \quad : \mathbb{N} \\
& \lambda \mathit{recs} : (\forall n : \mathbb{N}.\ \mathsf{Ack}\ m\ n) \times (\mathbb{N}\mathsf{Aux}\ \ldots\ m) \\
& \lambda n \quad : \mathbb{N} \\
& \lambda \mathit{recs}' : (\mathsf{Ack}\ \mathsf{s}m\ n) \times (\mathbb{N}\mathsf{Aux}\ (\mathsf{Ack}\ \mathsf{s}m)\ n) \\
& ?\mathit{rec}_1 : \mathsf{Ack}\ \mathsf{s}m\ n \\
& ?\mathit{rec}_2 : \mathsf{Ack}\ m\ (\mathit{call}\ \mathit{rec}_1) \\
& \quad \mathit{return}\ (\mathit{call}\ \mathit{rec}_2)
\end{array}
$$

rec_1 is solved from recs' and rec_2 is solved from *recs*. The definition is complete.

We can build quite complex structures with multiple eliminations by guarded recursion—more even than lexicographic recursion on a number of argument positions. For example, we may define a function on lists of trees which at each recursion replaces the head tree by its subtrees—some steps may make the list longer, but the decomposition of the head tree guarantees termination.

The question of how to extend the class of recognisable pattern matching programs into this more exotic territory is an important and interesting one. Much attention has already been paid to the simply typed case, for example, in Manoury and Simonot's 'ProPre' [MS94] system. Further, Cristina Cornes has equipped COQ with a substantial package translating equational programs with relatively interesting recursive structure into constructor guarded fixpoint expressions [Cor97].

Further investigation is beyond the scope of this thesis. However, I shall nonetheless write such equational programs in the following chapter, since they are shorter and clearer than their expanded versions where each recursion has its own subfunction. When I do so, I shall always be careful to point out the justification, imagining that we are deriving the function interactively.

Chapter 7

Some Programs and Proofs

We have now developed substantial technology for constructing dependently typed functional programs, and also for reasoning about them. Let us now put that technology to work.

In the course of this chapter, I offer some examples which I believe illustrate the advantages afforded by working with more informative types. We shall see new versions of old programs which are tidier and easier to prove correct. We shall see applications of our elimination rule technology which aid program discovery as well as verification. Hopefully, we shall see sense.

Later, I shall focus on the manipulation of syntax as a programming domain which shows off to great effect the expressive power of dependent pattern matching. In particular, I shall construct and prove correct a first-order unification algorithm which has the novel merit of being structurally recursive.

7.1 concrete categories, functors and monads

In the examples which follow, we shall examine methods of working with syntax via dependently typed functional programming. The behaviour of the functions we shall develop fits neatly into a categorical treatment, so it is worthwhile building some tools for packaging these functions and their properties categorically.

We shall not need any particularly heavy category theory, which is just as well, as far as I am concerned. For a substantial formalisation of category theory, see [SH95]. In fact, we may restrict our attention to **concrete** categories—those whose objects can be interpreted as a family of types and whose arrows can be interpreted as functions between types in the family.

7.1.1 records for categories

So, what shall we say is a category?

The idea is not that objects are types and arrows functions, but that both are data which can be *interpreted* as such. Imagine we are modelling a programming language categorically: we might have OLEG datatypes representing the types and functions of that language, together with translations which model those types and functions as OLEG types and functions. Those datatypes give us the objects and arrows of a concrete category, and the translations their interpretations.

Let us fix the types of objects and arrows.

$$O : \mathcal{Type}$$
$$\rightarrowtail : O \to O \to \mathcal{Type}$$

Now, let us define a record type $\mathsf{Concrete}$ to contain the things we must supply to have a meaningful category:

$$\begin{array}{r l}
\iota : & \forall S\!:\!O.\; S \rightarrowtail S \\
\circ : & \forall R, S, T\!:\!O.\; (S \rightarrowtail T) \to (R \rightarrowtail S) \to (R \rightarrowtail T) \\
[\![\cdot]\!] : & O \to \mathcal{Type} \\
[\![\cdot]\!] : & \forall S, T\!:\!O.\; (S \rightarrowtail T) \to [\![S]\!] \to [\![T]\!] \\
\mathsf{RespI} : & \forall S\!:\!O.\; \forall s\!:\![\![S]\!].\; [\![\iota_S]\!]\; s \simeq s \\
\mathsf{RespC} : & \forall R, S, T\!:\!O.\; \forall f\!:\!S \rightarrowtail T.\; \forall g\!:\!R \rightarrowtail S.\; \forall r\!:\![\![R]\!].\; [\![f \circ g]\!]\; r \simeq [\![f]\!]\; ([\![g]\!]\; r)
\end{array}$$

I think it is safe to overload $[\![\cdot]\!]$. Confusion between the interpretations of objects and arrows will not arise in these examples.

Saunders MacLane [Mac71] defines a **concrete category** to be a category equipped with a **faithful** functor into **Set**. That is, the interpretations must not only preserve identity and composition, but must also *embed* the objects and arrows in **Set**. I have given no such condition. It begs the question 'what is the appropriate equality on objects and arrows?'.

In type theory, as in marriage, fidelity comes down to the way you see things. OLEG's intensional equality is too discriminating to be useful here. I propose to consider two arrows the same if their interpretations are extensionally equal: interpretations are thus trivially faithful. Consequently, it makes little sense to consider the category separately from the functor which interprets it—the functor properties are how we know the category has the traditional absorption and associativity laws with respect to this extensional notion of equality.

Correspondingly, if $f, g : S \rightarrowtail T$, let us make the abbreviation

$$f \approx g \implies \forall s : [\![S]\!] .\ [\![f]\!]\ s \simeq [\![g]\!]\ s$$

The usual absorption and associtivity properties

$$\begin{aligned} f \circ \iota &\approx f \\ \iota \circ g &\approx g \\ (f \circ g) \circ h &\approx f \circ (g \circ h) \end{aligned}$$

all follow by reflexivity.

Discharging the parameters, we have our notion of category. I shall typically write

$$\mathsf{Concrete}\ \rightarrowtail$$

to mean a category for a given notion of arrow, leaving the object type implicit.

For any type family $Fam : O \to \mathcal{T}ype$, we may define

$$\overset{\rightarrowtail}{Fam} = \lambda S, T : O.\ Fam\ S \ \to\ Fam\ T \quad : \quad O \to O \to \mathcal{T}ype$$

We may easily define an operation $[\cdot]$ on such families such that

$$[Fam] : \mathsf{Concrete}\ \overset{\rightarrowtail}{Fam}$$

with objects interpreted via *Fam* and arrows, identity and composition as themselves. This is the usual notion of functions between types in a family, represented within our defined class of category.

In particular, if we let Type be the identity function on $\mathcal{T}ype$, then [Type] is the category of OLEG types.

As a special case, we may pretend any type T is a $\mathbb{1}$-indexed type family and manufacture the one-object category $[T]$ of its endofunctions.

We will encounter categories whose arrows are not represented directly as OLEG functions. A rather glib example is the $\mathsf{Concrete}\ \overset{\rightarrowtail}{\mathbb{N}}$ whose arrows live in $\mathbb{N}$ (actually, $\mathbb{1} \to \mathbb{1} \to \mathbb{N}$, but never mind), with n interpreted as $\mathsf{plus}\ n$. The identity is $\mathsf{0}$, the composition is plus, and hence the property that interpretation respects composition is just associativity.

7.1.2 records for functors

A **functor** takes objects and arrows from one category to objects and arrows of another, preserving identity and composition. We can certainly write these requirements down as a parameterised record.

Let us fix source and target categories, then open them:

$$\begin{array}{l} O^s : \mathcal{T}\!ype \\ \rightarrowtail^s : O^s \to O^s \to \mathcal{T}\!ype \\ C^s : \mathsf{Concrete}\ \rightarrowtail^s \\ O^t : \mathcal{T}\!ype \\ \rightarrowtail^t : O^t \to O^t \to \mathcal{T}\!ype \\ C^t : \mathsf{Concrete}\ \rightarrowtail^t \end{array}$$

$$C^s[\iota^s; \circ^s; [\![\cdot]\!]^s; [\![\cdot]\!]^s; \mathsf{RespI}^s; \mathsf{RespC}^s]\ C^t[\iota^t; \circ^t; [\![\cdot]\!]^t; [\![\cdot]\!]^t; \mathsf{RespI}^t; \mathsf{RespC}^t]$$

Relative to these, let us define a record type Functor with fields

$$\begin{array}{r} \mathsf{Fo}{:}O^s \ \to\ O^t \\ \mathsf{Fa}{:}\ldots(S \rightarrowtail^s T) \ \to\ (\mathsf{Fo}\,S) \rightarrowtail^t (\mathsf{Fo}\,T) \\ \mathsf{PresEq}{:}\ldots f \approx g \ \to\ \mathsf{Fa}\,f \ \approx\ \mathsf{Fa}\,g \\ \mathsf{PresI}{:}\ldots\mathsf{Fa}\ \iota^s_S \ \approx\ \iota^t_{(\mathsf{Fo}\ S)} \\ \mathsf{PresC}{:}\ldots\mathsf{Fa}\,(f \circ^s g) \ \approx\ (\mathsf{Fa}\,f) \circ^t (\mathsf{Fa}\,g) \end{array}$$

Note that I have left out some human-inferrable universal quantifiers for the sake of readability.

The extra condition—preservation of extensional equality of arrows—is necessary. It is possible for two extensionally equal source arrows to be distinguished computationally, and hence mapped to different arrows in the target category, unless we expressly forbid it.

Of course, when writing functor types, I shall suppress all the details and just leave Functor $C^s\ C^t$.

By way of example, every polymorphic[1] type family has an associated functor. It would be nice if these were manufactured automatically. I shall outline the functor for maybe.

[1]in the ML sense

$$
\begin{array}{l}
\mathsf{maybeF} : \mathsf{Functor}\ [\mathrm{Type}]\ [\mathrm{Type}] \\
\mathsf{maybeF}\ =\ \langle \quad
\begin{array}{rl}
\mathsf{Fo} = & \mathsf{maybe} \\
\mathsf{Fa} = & \lambda S, T : \mathit{Type}.\ \lambda f : S \to T.\ \lambda x : \mathsf{maybe}\ S. \\
& \begin{array}{|c|c}\hline x & \\ \hline \mathsf{yes}\ s & \mathsf{yes}\ (f\,s) \\ \mathsf{no}\ S & \mathsf{no}\ T \\ \hline \end{array} \\
\mathsf{PresEq} = & \ldots \\
\mathsf{PresI} = & \ldots \\
\mathsf{PresC} = & \ldots \qquad \rangle
\end{array}
\end{array}
$$

This functor just lifts functions to their exception-propagating images. I use a 'table' notation for case expressions: the column heading x indicates what is being analysed, underneath it are the patterns, and to the right are the corresponding return values—this notation is easily interpreted by pattern matching. The three remaining fields may easily be proven by inverting Fa, ie case analysis on the maybe-typed argument implicit in the extensional equations.

Finally, one irritating aspect of intensional type theory is that we may have to work with several implementations of, extensionally speaking, the same function. Suppose we have another candidate Fa' for the arrow part Fa of a given functor, with the same type and extensional behaviour. It would be really annoying if we had to redo the proofs of the properties for the functor with Fa replaced by Fa', but fortunately, we may make this argument once and for all.

The point is that the functor properties concern only the extensional behaviour of Fa, so we may construct a function $\mathsf{sameFunctor}$ which takes our source functor, Fa' and a proof that Fa and Fa' have the same extension, returning the functor with Fa' on arrows and all the same properties. I shall not give the details here—they amount only to unremarkable rewriting.

7.1.3 records for 'concrete' monads

The formalisation of monads I shall give is a 'concrete' version of the Kleisli triple presentation due to Manes [Man76], which he showed equivalent to the convention definition [Mac71] by an endofunctor T with natural transformations η and μ.

DEFINITION: **Kleisli triple**

A **Kleisli triple** $(T, \eta, \langle\!|)^2$ on a category C is given by

[2] $\langle\!|$ is pronounced 'bind'.

- a function T from C-objects to C-objects
- an object-indexed family of morphisms $\eta \in C(X, TX)$, interpreting the elements of X in its T-image
- a family of functions $\langle\!|$, indexed by a pair of objects X, Y, from $C(X, TY)$ to $C(TX, TY)$

satisfying the equations

- $\eta\langle\!| \; = \; \mathit{id}$
- $(f\langle\!|) \circ \eta \; = \; f$
- $((f\langle\!|) \circ g)\langle\!| \; = \; (f\langle\!|) \circ (g\langle\!|)$

The **Kleisli category** arising from such a structure has the same objects as C, and X to Y arrows given by the $C(X, TY)$. η gives each object its identity, and the composition $\diamond$ is

$$f \diamond g \; = \; f\langle\!| \circ g$$

Consequently, $\eta \circ \cdot$ gives a functor from C to the Kleisli category, and $\cdot\langle\!|$ gives a functor from the Kleisli category to the image of C under T. The composition of the two is thus a functor which does T to objects.

The presentation of monads given below is based on the idea of a functor which is split into $\eta \circ \cdot$ (below $|\!\rangle \cdot$) and $\cdot\langle\!|$.

Given two concrete categories and a functor, we may describe what it means to be a **concrete monad** which **splits** that functor. Let us keep the same target and source categories opened as above and fix further

$$F : \mathsf{Functor}\; C^s\; C^t$$

Let us open F with the names given by the fields.

A concrete monad splitting F captures a class of 'diagonal arrows', $S \searrow T$ $(S, T : O^s)$, which are interpreted in $[\![S]\!]^s \;\to\; [\![\mathsf{Fo}\; T]\!]^t$. These will be the arrows of the Kleisli category, and they must be equipped with a notion of composition $\diamond$ which behaves under interpretation like the composition in the Kleisli category.

Think of the maybeF functor, viewing yes as packaging data and no as representing an error condition. Arrows in the source category are 'reliable' functions acting on actual data; arrows in the target category are 'error-aware' functions—they may handle errors or create them. The functor takes reliable functions to 'error-propagating' functions—they will give actual output for actual input and transmit error conditions. A 'diagonal arrow' is an unreliable function—it accepts actual data, but may result in an error. A monad splitting maybeF characterises a class of these unreliable functions such that

- every reliable function f has an unreliable image (which just packages the output with yes) given by $\rrparenthesis f$

  if $f : S \to T$ then $\rrparenthesis f : S \to \mathsf{maybe}\ T$

- every unreliable function g in the class has an error-aware image (which propagates input errors, but may make new output errors) given by $g\llparenthesis$

  if $g : S \to \mathsf{maybe}\ T$ then $g\llparenthesis : \mathsf{maybe}\ S \to \mathsf{maybe}\ T$

- the combination $\rrparenthesis f\llparenthesis$ does the same thing to source arrows as maybeF

  if $f : S \to T$ then $\rrparenthesis f\llparenthesis : \mathsf{maybe}\ S \to \mathsf{maybe}\ T$

More formally, let us fix the carrier type for diagonal arrows

$$\searrow : O^s \to O^s \to \mathit{Type}$$

and collect the relevant details in a record type Monad with fields

$$
\begin{array}{r@{}l}
\rrparenthesis\cdot : \ldots & (S \rightarrowtail^s T) \;\to\; S \searrow T \\
\cdot\llparenthesis : \ldots & (S \searrow T) \;\to\; (\mathsf{Fo}\ S) \rightarrowtail^t (\mathsf{Fo}\ T) \\
\diamond : \ldots & (S \searrow T) \;\to\; (R \searrow S) \;\to\; (R \searrow T) \\
[\![\cdot]\!] : \ldots & (S \searrow T) \;\to\; [\![S]\!]^s \;\to\; [\![T]\!]^t \\
\mathsf{MonadI} : \ldots & [\![f\llparenthesis]\!]^t\ ([\![\rrparenthesis\iota^s_S]\!]\ s) \simeq [\![f]\!]\ s \\
\mathsf{MonadC} : \ldots & [\![f \diamond g]\!]\ r \simeq [\![f\llparenthesis]\!]^t\ ([\![g]\!]\ r) \\
\mathsf{Split} : \ldots & \rrparenthesis f\llparenthesis \;\approx\; \mathsf{Fa}\ f \\
\mathsf{FrontEq} : \ldots & f \approx g \;\to\; \rrparenthesis f \;\approx\; \rrparenthesis g \\
\mathsf{FrontC} : \ldots & \rrparenthesis(f \circ^s g) \;\approx\; (\rrparenthesis f) \diamond (\rrparenthesis g) \\
\mathsf{BackEq} : \ldots & f \approx g \;\to\; f\llparenthesis \;\approx\; g\llparenthesis \\
\mathsf{BackC} : \ldots & (f \diamond g)\llparenthesis \;\approx\; (f\llparenthesis) \circ^t (g\llparenthesis)
\end{array}
$$

This may look like a lot of stuff, but remember that the diagonal arrows might not be represented functionally—they might be something really concrete like association lists. The operations $\rrparenthesis\cdot$, $\cdot\llparenthesis$ and $\diamond$ should be viewed as syntactic. We have to ensure that they have the right semantics. Of course, if they are just functions and their interpretation is application, then this is very easy to do.

The maybeF functor has trivial functional representations of arrows source and target. For the corresponding $\mathsf{maybeM} : \mathsf{Monad}\ \mathsf{maybeF}$, we take the diagonal arrow type to be $S \to \mathsf{maybe}\ T$ and the interpretation as application. $\rangle\!\rangle\cdot$ just composes yes on the back of its argument whilst $\cdot\langle\!\langle$ is defined by case analysis:

$$
\begin{aligned}
&f\langle\!\langle\ (\mathsf{yes}\, s) = f\, s \\
&f\langle\!\langle\ (\mathsf{no}\ S) = \mathsf{no}\ T
\end{aligned}
$$

Composition is defined in accordance with the requirement on its interpretation:

$$(f \diamond g)\ r\ =\ f\langle\!\langle\ (g\, r)$$

As for the properties

- MonadI, MonadC and FrontC hold by reflexivity.
- Split and BackC hold by case analysis then reflexivity.
- $\mathsf{FrontEq}$ holds, rewriting by the premise.
- BackEq holds by case analysis, then reflexivity in the no case and rewriting by the premise in the yes case.

There is a function which constructs the (concrete) **Kleisli** category for a given concrete monad.

$$\mathsf{Kleisli} : \mathsf{Monad}\ \searrow \to\ \mathsf{Concrete}\ \searrow$$

The $\mathsf{Concrete}$ so constructed has operations:

$$
\begin{aligned}
\iota^k_S\ &=\ \rangle\!\rangle\iota^s_S \\
\circ^k\ &=\ \diamond \\
[\![S]\!]^k\ &=\ [\![\mathsf{Fo}\ S]\!]^t \\
[\![f]\!]^k\ &=\ [\![f\langle\!\langle]\!]^t
\end{aligned}
$$

Observe

$$
\begin{aligned}
\left[\!\left[\iota^k_{\mathsf{S}}\right]\!\right]^k\ s &\simeq [\![\rangle\iota^s_{\mathsf{S}}]\!]^k\ s && \text{(definition of } \iota^k\text{)}\\
&\simeq [\![\rangle\iota^s_{\mathsf{S}}\langle]\!]^t\ s && \text{(definition of } [\![\cdot]\!]^k\text{)}\\
&\simeq [\![\mathsf{Fa}\ \iota^s_{\mathsf{S}}]\!]^t\ s && \text{(Split)}\\
&\simeq \left[\!\left[\iota^t_{(\mathsf{Fo}\ \mathsf{S})}\right]\!\right]^t\ s && \text{(PresI)}\\
&\simeq s && \text{(RespI}^t\text{)}
\end{aligned}
$$

$$
\begin{aligned}
\left[\!\left[f \circ^k g\right]\!\right]^k\ r &\simeq [\![f \diamond g]\!]^k\ r && \text{(definition of } \circ^k\text{)}\\
&\simeq [\![(f \diamond g)\langle]\!]^t\ r && \text{(definition of } [\![\cdot]\!]^k\text{)}\\
&\simeq [\![(f\langle) \circ^t (g\langle)]\!]^t\ r && \text{(BackC)}\\
&\simeq [\![f\langle]\!]^t\ ([\![g\langle]\!]^t\ r) && \text{(RespC}^t\text{)}\\
&\simeq [\![f]\!]^k\ ([\![g]\!]^k\ r) && \text{(definition of } [\![\cdot]\!]^k\text{)}
\end{aligned}
$$

7.2 substitution for the untyped λ-calculus

In this section, I shall develop the technology to give a monadic [Man76, Mog91] presentation of substitution for terms with binding—in particular, the untyped λ-calculus with de Bruijn indices [deB72]. Bellegarde and Hook [BH94] suggest the following datatype, which Altenkirch and Reus [AR99] describe as 'heterogeneous', and Bird and Paterson [BP99] describe as 'nested'.

$$
\frac{X : \mathcal{T}ype}{\mathsf{Lam}\ X : \mathcal{T}ype}
$$

$$
\frac{x : X}{\mathsf{var}\ x : \mathsf{Lam}\ X} \qquad \frac{s, t : \mathsf{Lam}\ X}{\mathsf{app}\ s\ t : \mathsf{Lam}\ X} \qquad \frac{t : \mathsf{Lam}\ (\mathsf{maybe}\ X)}{\mathsf{lam}\ t : \mathsf{Lam}\ X}
$$

This datatype relativises terms to an arbitrary type of variables.[3] It can be defined in SML, but recursion over it is necessarily polymorphic and hence unavailable. However, Haskell now allows functions over such datatypes, so long as their types are supplied explicitly.

In such languages, terms may not appear in types—this apartheid policy is advisable because the terms often engage in such criminal activities as nontermination. Hence, if we want to make some kind of indexed family, the indices must themselves be types. This presentation works by using maybe as a kind of type-level s, corresponding to the idea that there is some number of variables and that abstraction introduces one more. Also, $\mathsf{Lam}\ \mathbb{O}$ is a type of closed terms. However, this hacked-up type-level

[3] In fact, our scheme of definitions restricts the variable type to inhabit a smaller universe than the terms over it.

$\mathbb{N}$ has only introduction rules: no computation on indices is available. Fortunately, substitution is structural on terms.

We need no such Group Areas Act. In our system, terms are as trustworthy as types. We can use the $\mathbb{N}$ God invented, and then fin to make sets of variables.

$$\frac{n : \mathbb{N}}{\mathsf{Lam}\, n : \mathit{Type}}$$

$$\frac{x : \mathsf{fin}\, n}{\mathsf{var}\, x : \mathsf{Lam}\, n} \qquad \frac{s, t : \mathsf{Lam}\, n}{\mathsf{app}\, s\, t : \mathsf{Lam}\, n} \qquad \frac{t : \mathsf{Lam}\, \mathsf{s}n}{\mathsf{lam}\, t : \mathsf{Lam}\, n}$$

$\mathsf{Lam}\, n$ is the type of λ-terms with n free variables. Later, we shall see operations on syntax which are made structural by the availability of recursion on this index.

Placing these types in our categorical setting, we have

$$[\mathsf{fin}] : \mathsf{Concrete}\, \overset{\rightarrowtail}{\mathsf{fin}}$$
$$[\mathsf{Lam}] : \mathsf{Concrete}\, \overset{\rightarrowtail}{\mathsf{Lam}}$$

The objects in these categories are elements of $\mathbb{N}$, interpreted via fin and Lam respectively. The arrows are function spaces interpreted by application. Hence we effectively abbreviate:

$$m \rightarrowtail^{f} n \implies \mathsf{fin}\, m \rightarrow \mathsf{fin}\, n$$
$$m \rightarrowtail^{L} n \implies \mathsf{Lam}\, m \rightarrow \mathsf{Lam}\, n$$

In this section, we shall be looking to build a functor

$$\mathsf{Rename} : \mathsf{Functor}\, [\mathsf{fin}]\, [\mathsf{Lam}]$$

which, for every arrow on a variable space in $[\mathsf{fin}]$, gives us the operation on terms from $[\mathsf{Lam}]$ over those variables renaming them as indicated. The object part of the functor is just the identity on $\mathbb{N}$. We may then view functions in the type

$$m \searrow n \implies \mathsf{fin}\, m \rightarrow \mathsf{Lam}\, n$$

as simultaneous substitutions from m variables to terms over n variables and seek a monadic implementation

$\mathsf{SubstM} : \mathsf{Monad}\ \mathsf{Rename}\ \searrow$

Note that Rename is not an endofunctor, as in the conventinal notion of monad, but we can still think of splitting it in a monadic way. The consequent Kleisli category will thus interpret substitutions as functions from terms over one set of variables to terms over another.

Before we can work with terms, we need some basic tools for working with variables in the de Bruijn style.

7.2.1 lift, thin and thick

de Bruijn's insight was to see a variable not just as an indentifier, but as a reference to a binding. Variable indices count outwards through the λ-bindings, 0 for the most local, 1 for the next and so on. For example,

$\lambda f.\ \lambda x.\ f\,x$ becomes $\lambda\lambda 1\ 0$

Every time we go under a binder, the new variable is 0 and the old ones get incremented. We may represent this distinction by the constructors of the fin family.

Now, suppose we have a **renaming**—an arrow $f : m \rightarrowtail^{f} n$. In order to apply such a renaming across a term, we must explain what to do with the expanded variable space under a lam—it must affect only the free variables embedded by fs, leaving the newly bound fz variable alone.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{f' : \mathsf{s}m \rightarrowtail^{f} \mathsf{s}n} \\
f'\ \ (\mathsf{fz}\,m) & = & \mathsf{fz}\,n \\
f'\ \ (\mathsf{fs}\,x) & = & \mathsf{fs}\,(f\,x)
\end{array}
$$

This is a recognisable program.

Discharging over arbitrary n, m and f, we obtain the functional lift which takes any such f to the appropriate f'. I suppress the boring arguments when I apply it.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathsf{lift} : \forall m, n{:}\mathbb{N}.\ \forall f{:}\,\mathsf{fin}\,m \ \to\ \mathsf{fin}\,n.\ \mathsf{fin}\,\mathsf{s}m \ \to\ \mathsf{fin}\,\mathsf{s}n} \\
\mathsf{lift}\,f\ \ (\mathsf{fz}\,m) & = & \mathsf{fz}\,n \\
\mathsf{lift}\,f\ \ (\mathsf{fs}\,x) & = & \mathsf{fs}\,(f\,x)
\end{array}
$$

lift gives us the arrow part of the functor

$$\mathsf{Lift} : \mathsf{Functor}\,[\mathsf{fin}]\,[\mathsf{fin}]$$

$$\mathsf{Lift} \;=\; \left\langle \begin{array}{r@{\;=\;}l} \mathsf{Fo} & \mathsf{s} \\ \mathsf{Fa} & \mathsf{lift} \\ \mathsf{PresEq} & \ldots \\ \mathsf{PresI} & \ldots \\ \mathsf{PresC} & \ldots \end{array} \right\rangle$$

There is a recursion induction principle for lift which we may regard as generated automatically from its equational definition. lift is not a recursive function, so it is perhaps more informative to call it an **inversion** principle $\mathsf{liftInv}$:

$$\boxed{\begin{array}{l} m, n : \mathbb{N} \\ f : m \rightarrowtail^{f} n \\ \Phi : \mathsf{fin}\,\mathsf{s}m \to \mathsf{fin}\,\mathsf{s}n \to \mathit{Type} \\ \\ \quad \dfrac{\Phi\ (\mathsf{fz}\,m)\ (\mathsf{fz}\,n) \qquad \Phi\ (\mathsf{fs}\,x)\ (\mathsf{fs}\ (f\,x))}{\forall x : \mathsf{fin}\,\mathsf{s}m.\ \Phi\ x\ \boxed{\mathsf{lift}\ f\,x}} \end{array}}$$

The three functor properties left elliptic above follow easily by inversion. I shall show PresC and leave the other two to your imagination.

$$\begin{array}{ll} ?\mathsf{PresC} : & \forall r, s, t : \mathbb{N} \\ & \forall f \quad : s \rightarrowtail^{f} t \\ & \forall g \quad : r \rightarrowtail^{f} s \\ & \forall \boxed{x} \quad : \mathsf{fin}\,\mathsf{s}r \\ & \quad \mathsf{lift}\ (f \circ g)\ x \simeq \mathsf{lift}\ f\ \boxed{\mathsf{lift}\ g\,x} \end{array}$$

Inverting the boxed lift application, we acquire two subgoals

$$\begin{array}{ll} ?\mathsf{PresC}_{z} : & \forall r, s, t : \mathbb{N} \\ & \forall f \quad : s \rightarrowtail^{f} t \\ & \forall g \quad : r \rightarrowtail^{f} s \\ & \forall x \quad : \mathsf{fin}\,\mathsf{s}r \\ & \quad \mathsf{lift}\ (f \circ g)\ (\mathsf{fz}\,r) \simeq \mathsf{lift}\ f\,(\mathsf{fz}\,s) \\ ?\mathsf{PresC}\ : & \forall r, s, t : \mathbb{N} \\ & \forall f \quad : s \rightarrowtail^{f} t \\ & \forall g \quad : r \rightarrowtail^{f} s \\ & \forall x \quad : \mathsf{fin}\,r \\ & \quad \mathsf{lift}\ (f \circ g)\ (\mathsf{fs}\,x) \simeq \mathsf{lift}\ f\,(\mathsf{fs}\ (g\,x)) \end{array}$$

The two conclusions then reduce respectively to

$$\mathsf{fz}\, t \simeq \mathsf{fz}\, t$$
$$\mathsf{fs}\,(f\,(g\,x)) \simeq \mathsf{fs}\,(f\,(g\,x))$$

As you can see, these are both reflexive.

We can use lift to define an important class of renamings—the **thinnings**. These add a new variable to the set, but not necessarily at the top.[4] If there are n old variables, there are $\mathsf{s}n$ choices for the new variable x. $\mathsf{thin}\, x$ is the renaming which shuffles the old variables in around the new one, without changing their order.

The idea is, morally:

$$\begin{array}{lcl} \mathsf{thin}\, x\, y & = & y, \text{ if } y < x \\ & & y+1, \text{ if } y \geq x \end{array}$$

In particular, $\mathsf{thin}\, x\, y \;\neq\; x$.

Now, if the new variable is $\mathsf{fz}\, n$, then thinning is just the fs embedding. Otherwise, it is a lifted thinning!

$$\mathsf{thin} : \forall n\!:\!\mathbb{N}.\ \mathsf{fin}\ \mathsf{s}n \;\to\; (n \rightarrowtail^{f} \mathsf{s}n)$$
ie
$$\begin{array}{lcl} \multicolumn{3}{l}{\mathsf{thin} : \forall n\!:\!\mathbb{N}.\ \mathsf{fin}\ \mathsf{s}n \;\to\; \mathsf{fin}\, n \;\to\; \mathsf{fin}\ \mathsf{s}n} \\ \mathsf{thin}\quad (\mathsf{fz}\, n) & = & \mathsf{fs}_n \\ \mathsf{thin}\quad (\mathsf{fs}\, x) & = & \mathsf{lift}\,(\mathsf{thin}\, x) \end{array}$$

Thinning provides us with an alternative view of $\mathsf{fin}\,\mathsf{s}n$. Every variable is either the new one, x, or an embedded old one, $\mathsf{thin}\, x\, y$ for some $y : \mathsf{fin}\, n$. We may imagine a partial inverse to thin which makes the distinction, with the following extensional behaviour:

$$\begin{array}{llcl} \multicolumn{4}{l}{\mathsf{thick} : \forall n\!:\!\mathbb{N}.\ \mathsf{fin}\ \mathsf{s}n \;\to\; \mathsf{fin}\ \mathsf{s}n \;\to\; \mathsf{maybe}\,(\mathsf{fin}\, n)} \\ \mathsf{thick}\quad x & (\mathsf{thin}\, x\, y) & \simeq & \mathsf{yes}\, y \\ \mathsf{thick}\quad x & x & \simeq & \mathsf{no}\,(\mathsf{fin}\, n) \end{array}$$

thick is a refinement of the decidable equality for the finite sets—it not only tells us whether two elements differ, but also in what way.

We can get some help writing thick if we try to prove the above pair of equational laws (for a common abstracted x) by recursion induction on thin, as defined in the obvious way. We thus seek:

[4] 'Thinning' is a liquid metaphor.

$$
\begin{array}{l}
?\mathsf{thick} : \forall n \quad : \mathbb{N} \\
\qquad\quad \forall x, y \colon \mathsf{fin}\ \mathsf{s}n \\
\qquad\qquad \mathsf{maybe}\ (\mathsf{fin}\ n) \\
?\mathsf{thick}_i\colon \forall \boxed{n} \quad : \mathbb{N} \\
\qquad\quad \forall \boxed{x} \quad : \mathsf{fin}\ \mathsf{s}n \\
\qquad\quad \Sigma\mathsf{thick}_y\colon \forall y\colon \mathsf{fin}\ n \\
\qquad\qquad\qquad\qquad \mathsf{thick}\ x\ (\boxed{\mathsf{thin}\ x}\ y) \simeq \mathsf{yes}\ y \\
\qquad\qquad \mathsf{thick}\ x\ x \simeq \mathsf{no}\ (\mathsf{fin}\ n)
\end{array}
$$

The abstraction of x outside both equations allows them to be transformed simultaneously. The induction yields subgoals:

$$
\begin{array}{l}
?\mathsf{thick} \ : \forall n \quad : \mathbb{N} \\
\qquad\quad \forall x, y \colon \mathsf{fin}\ \mathsf{s}n \\
\qquad\qquad \mathsf{maybe}\ (\mathsf{fin}\ n) \\
?\mathsf{thick}_{iz}\colon \forall n \qquad : \mathbb{N} \\
\qquad\quad \Sigma\mathsf{thick}_y\colon \forall y\colon \mathsf{fin}\ n \\
\qquad\qquad\qquad\qquad \mathsf{thick}\ (\mathsf{fz}\ n)\ (\mathsf{fs}\ y) \simeq \mathsf{yes}\ y \\
\qquad\qquad \mathsf{thick}\ (\mathsf{fz}\ n)\ (\mathsf{fz}\ n) \simeq \mathsf{no}\ (\mathsf{fin}\ n) \\
?\mathsf{thick}_{is}\colon \forall n \qquad : \mathbb{N} \\
\qquad\quad \forall x \qquad : \mathsf{fin}\ \mathsf{s}n \\
\qquad\quad \forall f \qquad : \mathsf{fin}\ n \ \rightarrow\ \mathsf{fin}\ \mathsf{s}n \\
\qquad\quad \forall hyp \quad : \Sigma\mathsf{thick}_y\colon \forall y\colon \mathsf{fin}\ n \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{thick}\ x\ (f\,y) \simeq \mathsf{yes}\ y \\
\qquad\qquad\qquad\qquad \mathsf{thick}\ x\ x \simeq \mathsf{no}\ (\mathsf{fin}\ n) \\
\qquad\quad \Sigma\mathsf{thick}_y\colon \forall \boxed{y}\colon \mathsf{fin}\ \mathsf{s}n \\
\qquad\qquad\qquad\qquad \mathsf{thick}\ (\mathsf{fs}\ x)\ \boxed{\mathsf{lift}\ f\ y} \simeq \mathsf{yes}\ y \\
\qquad\qquad \mathsf{thick}\ (\mathsf{fs}\ x)\ (\mathsf{fs}\ x) \simeq \mathsf{no}\ (\mathsf{fin}\ \mathsf{s}n)
\end{array}
$$

We now know how to thick at $\mathsf{fz}\ n$. We can gain further information about the fs case by inverting the lift. Allowing that we can do this inside the Σ-binding by appropriate algebraic manipulation, we obtain

$$
\begin{array}{l}
?\mathsf{thick}_{is}\colon \forall n \quad : \mathbb{N} \\
\qquad\qquad \forall x \quad : \mathsf{fin}\ \mathsf{s}n \\
\qquad\qquad \forall f \quad : \mathsf{fin}\ n \rightarrow \mathsf{fin}\ \mathsf{s}n \\
\qquad\qquad \forall hyp \quad : \Sigma\mathsf{thick}_y\colon \forall y\colon \mathsf{fin}\ n \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{thick}\ x\ (f\,y) \simeq \mathsf{yes}\ y \\
\qquad\qquad\qquad\qquad \mathsf{thick}\ x\ x \simeq \mathsf{no}\ (\mathsf{fin}\ n) \\
\qquad\qquad \Sigma\mathsf{thick}_y\colon \Sigma\mathsf{thick}_{sz}\colon \mathsf{thick}\ (\mathsf{fs}\ x)\ (\mathsf{fz}\ \mathsf{s}n) \simeq \mathsf{yes}\ (\mathsf{fz}\ n) \\
\qquad\qquad\qquad\quad \forall y \quad : \mathsf{fin}\ n \\
\qquad\qquad\qquad\qquad \mathsf{thick}\ (\mathsf{fs}\ x)\ (\mathsf{fs}\ (f\,y)) \simeq \mathsf{yes}\ (\mathsf{fs}\ y) \\
\qquad\qquad\quad \mathsf{thick}\ (\mathsf{fs}\ x)\ (\mathsf{fs}\ x) \simeq \mathsf{no}\ (\mathsf{fin}\ \mathsf{s}n)
\end{array}
$$

Stripping away the excess notation, we have certainly found the base cases to our function:

$$
\begin{array}{llcll}
\mathsf{thick} & (\mathsf{fz}\ n) & (\mathsf{fz}\ n) & = & \mathsf{no}\ (\mathsf{fin}\ n) \\
\mathsf{thick} & (\mathsf{fz}\ n) & (\mathsf{fs}\ y) & = & \mathsf{yes}\ y \\
\mathsf{thick} & (\mathsf{fs}\ x) & (\mathsf{fz}\ \mathsf{s}n) & = & \mathsf{yes}\ (\mathsf{fz}\ n) \\
\ \ \ldots & & & &
\end{array}
$$

We have also found out some useful information about the step case. It must satisfy:

$$
\frac{\bigwedge \begin{array}{l} \forall y\colon \mathsf{fin}\ n.\ \mathsf{thick}\ x\ (f\,y) \simeq \mathsf{yes}\ y \\ \mathsf{thick}\ x\ x \simeq \mathsf{no}\ (\mathsf{fin}\ n) \end{array}}{\bigwedge \begin{array}{l} \forall y\colon \mathsf{fin}\ n.\ \mathsf{thick}\ (\mathsf{fs}\ x)\ (\mathsf{fs}\ (f\,y)) \simeq \mathsf{yes}\ (\mathsf{fs}\ y) \\ \mathsf{thick}\ (\mathsf{fs}\ x)\ (\mathsf{fs}\ x) \simeq \mathsf{no}\ (\mathsf{fin}\ \mathsf{s}n) \end{array}}
$$

Effectively, each branch of the conclusion propagates the result of the corresponding recursive call: yes stays yes and no stays no[5]. That is, the recursive value is passed on by the appropriate monadic lifting $\rangle\mathsf{fs}_n\langle$. Hence the whole program is

$$
\begin{array}{llcll}
\mathsf{thick} & (\mathsf{fz}\ n) & (\mathsf{fz}\ n) & = & \mathsf{no}\ (\mathsf{fin}\ n) \\
\mathsf{thick} & (\mathsf{fz}\ n) & (\mathsf{fs}\ y) & = & \mathsf{yes}\ y \\
\mathsf{thick} & (\mathsf{fs}\ x) & (\mathsf{fz}\ \mathsf{s}n) & = & \mathsf{yes}\ (\mathsf{fz}\ n) \\
\mathsf{thick} & (\mathsf{fs}\ x) & (\mathsf{fs}\ y) & = & \rangle\mathsf{fs}_n\langle\ (\mathsf{thick}\ x\ y)
\end{array}
$$

By construction, this satisfies the three base case equations and reduces the step case to

$$
\frac{\bigwedge \begin{array}{l} \forall y\colon \mathsf{fin}\ n.\ \mathsf{thick}\ x\ (f\,y) \simeq \mathsf{yes}\ y \\ \mathsf{thick}\ x\ x \simeq \mathsf{no}\ (\mathsf{fin}\ n) \end{array}}{\bigwedge \begin{array}{l} \forall y\colon \mathsf{fin}\ n.\ \rangle\mathsf{fs}_n\langle\ (\mathsf{thick}\ x\ (f\,y)) \simeq \mathsf{yes}\ (\mathsf{fs}\ y) \\ \rangle\mathsf{fs}_n\langle\ (\mathsf{thick}\ x\ x) \simeq \mathsf{no}\ (\mathsf{fin}\ \mathsf{s}n) \end{array}}
$$

[5] Matthew 5:37

This holds by rewriting the conclusions with the hypotheses. The desired extensional introduction rules have thus been satisfied. The corresponding non-computational inversion rule, $\mathsf{thickInv}$, is the real prize:

$$
\boxed{
\begin{array}{l}
n : \mathbb{N} \\
x : \mathsf{fin}\,\mathsf{s}n \\
\Phi : \mathsf{fin}\,\mathsf{s}n \;\rightarrow\; \mathsf{maybe}\,(\mathsf{fin}\,n) \;\rightarrow\; \mathit{Type} \\
\\
\qquad \dfrac{\Phi\, x\,(\mathsf{no}\,(\mathsf{fin}\,n)) \qquad \Phi\,(\mathsf{thin}\,x\,y)\,(\mathsf{yes}\,y)}{\forall y\!:\!\mathsf{fin}\,\mathsf{s}n.\; \Phi\; y \,\boxed{\mathsf{thick}\,x\,y}}
\end{array}
}
$$

$\mathsf{thickInv}$ tells us that there are two possible outcomes from thick and under what circumstances they arise. Fixing 'new variable' x, then any y is either x (in which case thick returns no) or an 'old variable' thinned (in which case thick identifies it). It is a very useful rule, because it effectively performs a constructor case analysis on the *output* of the function. We will see just why this is so helpful later on.

Can you guess how we prove this rule? That's right: by thick's recursion induction principle, making sure to keep Φ in the scheme, so that any inductive hypotheses are themselves elimination rules. We start with

$$
\begin{array}{ll}
?\mathsf{thickInv}\!: & \forall \boxed{n} : \mathbb{N} \\
& \forall \boxed{x} : \mathsf{fin}\,\mathsf{s}n \\
& \forall \Phi\;\, : \mathsf{fin}\,\mathsf{s}n \;\rightarrow\; \mathsf{maybe}\,(\mathsf{fin}\,n) \;\rightarrow\; \mathit{Type} \\
& \forall \phi_{\mathrm{n}} : \Phi\; x\,(\mathsf{no}\,(\mathsf{fin}\,n)) \\
& \forall \phi_{\mathrm{y}} : \forall y\!: \mathsf{fin}\,n \\
& \qquad\quad \Phi\,(\mathsf{thin}\,x\,y)\,(\mathsf{yes}\,y) \\
& \forall y\;\;\, : \mathsf{fin}\,\mathsf{s}n \\
& \quad \Phi\; y \,\boxed{\mathsf{thick}\,x\,y}
\end{array}
$$

I have indicated by boxing how the recursion induction scheme is abstracted. We acquire three base subgoals, corresponding to the base cases of the function, and their conclusions all follow directly from ϕ_{y} (off the diagonal) or ϕ_{n} (for $x = y = (\mathsf{fz}\,n)$). It is on the step subgoal where you should concentrate any remaining interest you can muster.

$$
\begin{array}{l}
?\mathsf{thickInv}_{ss}: \forall n \quad : \mathbb{N} \\
\qquad\qquad \forall x, y : \mathsf{fin}\ \mathsf{s}n \\
\qquad\qquad \forall y' \quad : \mathsf{maybe}\ (\mathsf{fin}\ n) \\
\qquad\qquad \forall hyp : \forall \Phi : \mathsf{fin}\ \mathsf{s}n \ \rightarrow\ \mathsf{maybe}\ (\mathsf{fin}\ n) \ \rightarrow\ \mathit{Type} \\
\qquad\qquad\qquad \forall \phi_{\mathrm{n}} : \Phi\ x\ (\mathsf{no}\ (\mathsf{fin}\ n)) \\
\qquad\qquad\qquad \forall \phi_{\mathrm{y}} : \forall y : \mathsf{fin}\ \mathsf{s}n \\
\qquad\qquad\qquad\qquad \Phi\ (\mathsf{thin}\ x\ y)\ (\mathsf{yes}\ y) \\
\qquad\qquad\qquad \Phi\ y\ y' \\
\qquad\qquad \forall \Phi \quad : \mathsf{fin}\ \mathsf{ss}n \ \rightarrow\ \mathsf{maybe}\ (\mathsf{fin}\ \mathsf{s}n) \ \rightarrow\ \mathit{Type} \\
\qquad\qquad \forall \phi_{\mathrm{n}} \ : \Phi\ (\mathsf{fs}\ x)\ (\mathsf{no}\ (\mathsf{fin}\ \mathsf{s}n)) \\
\qquad\qquad \forall \phi_{\mathrm{y}} \ : \forall y : \mathsf{fin}\ \mathsf{s}n \\
\qquad\qquad\qquad \Phi\ (\mathsf{thin}\ (\mathsf{fs}\ x)\ y)\ (\mathsf{yes}\ y) \\
\qquad\qquad \forall y \quad : \mathsf{fin}\ \mathsf{s}n \\
\qquad\qquad \boxed{\Phi\ (\mathsf{fs}\ \boxed{y})\ (\rangle\!\mathsf{fs}_n\!\langle\ \boxed{y'})}
\end{array}
$$

We are not yet in a position to use either ϕ_{n} or ϕ_{y}, because we do not yet know which applies. In the conclusion, the computation is blocked at the point where $\rangle\!\mathsf{fs}_n\!\langle$ is applied to y', the result of the recursive call, not yet in constructor form. However, case analysis on the result of the recursive call is exactly the effect of the inductive hypothesis. Eliminating with the indicated scheme, we obtain:

$$
\begin{array}{l}
?\mathsf{thickInv}_{ssn}: \\
\qquad\qquad \forall \phi_{\mathrm{n}} : \Phi\ (\mathsf{fs}\ x)\ (\mathsf{no}\ (\mathsf{fin}\ \mathsf{s}n)) \\
\qquad\qquad \Phi\ (\mathsf{fs}\ x)\ (\rangle\!\mathsf{fs}_n\!\langle\ (\mathsf{no}\ (\mathsf{fin}\ n))) \\
?\mathsf{thickInv}_{ssy}: \\
\qquad\qquad \forall \phi_{\mathrm{y}} : \forall y : \mathsf{fin}\ \mathsf{s}n \\
\qquad\qquad\qquad \Phi\ (\mathsf{thin}\ (\mathsf{fs}\ x)\ y)\ (\mathsf{yes}\ y) \\
\qquad\qquad \forall z \ : \mathsf{fin}\ n \\
\qquad\qquad \Phi\ (\mathsf{fs}\ (\mathsf{thin}\ x\ z))\ (\rangle\!\mathsf{fs}_n\!\langle\ (\mathsf{yes}\ z))
\end{array}
$$

The lifted fs now reduces, propagating the two cases correctly. Both conclusions now follow from the indicated hypotheses. The elimination rule holds.

In fact, the way the inductive step was proven shows us how this rule is useful in the wider setting. Applying this rule unblocks computations which are waiting to do case analysis on the result of a call to thick, and these are very common. For example, we may define the following function:

$$[\cdot \mapsto \cdot] : \forall n\!:\!\mathbb{N}.\ \forall x\!:\!\mathsf{fin}\ \mathsf{s}n.\ \forall t\!:\!\mathsf{Lam}\ n.\ (\mathsf{s}n \searrow n)$$

$$[x \mapsto t]y \quad = \quad \begin{array}{|c|c|}\hline \mathsf{thick}\ x\ y & \\ \hline \mathsf{no}\ (\mathsf{fin}\ n) & t \\ \mathsf{yes}\ y' & \mathsf{var}\ y' \\ \hline \end{array}$$

$[\cdot \mapsto \cdot]$ (pronounced 'knockout') generates a substitution (function from variables to terms) which removes x, replacing it by a term t over the 'remaining variables'. A source variable y other than x, ie a $(\mathsf{thin}\ x\ y')$, is mapped to the y' given by removing x from the variable set without reordering the others.

When proving properties of $[\cdot \mapsto \cdot]$, we will see it reduce to the case analysis on thick. At this point, elimination by $\mathsf{thickInv}$ has exactly the effect required to unblock the computation. We are interested in what comes *out* of thick, so the more conventional elimination of what goes *in* is a clumsy way to proceed.

Now that we have the tools to work with variables, let us turn our attention to terms.

7.2.2 the substitution monad splits the renaming functor

We have already decided that the object part of the functor Rename is just the identity on $\mathbb{N}$. It is also fairly clear that a renaming becomes a substitution just by composing var on the back, ie

$$\rangle\!\rangle f\ x \ = \ \mathsf{var}\ (f\, x)$$

Hence var is the identity for substitution.

The remaining programming consists of the arrow part of Rename and the $\cdot\langle\!|$ operation of SubstM—the effect on terms.

It is fairly clear that we shall have

$$\begin{array}{lcll} f\langle\!|\ (\mathsf{var}\ x) & = & f\, x & \qquad\qquad \text{(a monad law)} \\ f\langle\!|\ (\mathsf{app}\ s\ t) & = & \mathsf{app}\ (f\langle\!|\ s)\ (f\langle\!|\ t) & \\ \vdots & & & \end{array}$$

It is not so clear how to push f under a binder. We need something like

$$\begin{array}{lcl} \vdots & & \\ f\langle\!|\ (\mathsf{lam}\ t) & = & \mathsf{lam}\ (f'\langle\!|\ t) \end{array}$$

where f' is the lifting of f which takes the source bound variable to (a reference to) the target bound variable, and whose behaviour on the free variables respects that of f.

Now, we have already defined lift to lift renamings. How do we lift substitutions? The bound/free case analysis on the source variable is easy enough. We know what to do with the bound variable, otherwise the case analysis also tells us which 'old' variable f should be applied to. The latter yields a term over the old variables, which must then be renamed to the free variables in the target set. Now, we know that the variable renaming is just fs_n, but we need this lifted to terms. That is, we need something like

$$
\begin{aligned}
\mathsf{slift}\, f\, (\mathsf{fz}\, m) &= \mathsf{var}\, (\mathsf{fz}\, n) \\
\mathsf{slift}\, f\, (\mathsf{fs}\, x) &= \rangle\!\!\rangle \mathsf{fs}_n \langle\!\!\langle\ (f\, x)
\end{aligned}
$$

However, it is $\cdot\langle\!\!\langle$ which we are trying to define, and applying it recursively to the result of f is not structural.

One solution is to define the renaming Fa operation in advance—we already know how to lift renamings:

$$
\begin{aligned}
\mathsf{Fa}\, f\, (\mathsf{var}\, x) &= \mathsf{var}\, (f\, x) \\
\mathsf{Fa}\, f\, (\mathsf{app}\, s\, t) &= \mathsf{app}\, (\mathsf{Fa}\, f\, s)\, (\mathsf{Fa}\, f\, t) \\
\mathsf{Fa}\, f\, (\mathsf{lam}\, t) &= \mathsf{lam}\, (\mathsf{Fa}\, (\mathsf{lift}\, f)\, t)
\end{aligned}
$$

Once we have this, we can define slift with $\mathsf{Fa}\,\mathsf{fs}_n$ for $\rangle\!\!\rangle \mathsf{fs}_n \langle\!\!\langle$, leaving us free to define $\cdot\langle\!\!\langle$ in terms of it.

As Altenkirch and Reus point out, this involves writing two very similar functions over terms, where one nonstructural function would do. Of course, the nonstructural function saves three lines of code at the expense of a well-founded induction on an ordering which they must exhibit and prove satisfactory. They suggest that, turning a blind eye to the proof obligations, the nonstructural function is preferable, expressing the vague hope that the carpet under which they are sweeping the actual work will one day become magic.

As it happens, no carpets are necessary, magic or otherwise. $\cdot\langle\!\!\langle$ and Fa can be implemented with a single structurally recursive function, provided it is made sufficiently parametric. Suppose that for some type family T we have a function

$$
f : \mathsf{fin}\, m \;\rightarrow\; T\, n
$$

We can map this function across terms, provided we know

- how to convert f output from $T\,n$ to terms $\mathsf{Lam}\;n$
- how to represent variables in $T\,n$
- how to lift functions between fin sets and T sets

We already know how to do these things when T is fin, so we have renaming—we can then build the three operations for use when T is $\lambda\colon.\,.$

In fact, we will have an easier time proving the monadic behaviour of substitution if we take this opportunity to generalise lifting from inserting new variables at fz to inserting them anywhere—thick and thin make this just as easy to implement. We only ever use thick on variables, so the 'how to lift' requirement becomes 'how to thin'.

The goal is

$$
\begin{array}{ll}
\lambda T & : \mathbb{N} \to \mathit{Type} \\
\lambda vT & : \forall n\!:\!\mathbb{N}.\ \mathsf{fin}\,n \ \to\ T\,n \\
\lambda TLam & : \forall n\!:\!\mathbb{N}.\ T\,n \ \to\ \mathsf{Lam}\,n \\
\lambda thinT & : \forall n\!:\!\mathbb{N}.\ \forall x\!:\!\mathsf{fin}\,\mathsf{s}n.\ T\,n \ \to\ T\,\mathsf{s}n \\
?\mathsf{map} & : \forall m, n\!:\ \mathbb{N} \\
 & \quad \forall f \quad : \mathsf{fin}\,m \ \to\ T\,n \\
 & \quad \forall t \quad : \mathsf{Lam}\,m \\
 & \qquad \mathsf{Lam}\,n
\end{array}
$$

Subject to these parameters, we may first build lifting for T from the thinning parameter:

$$\mathsf{liftT} : \forall m, n\!:\!\mathbb{N}.\ \forall x\!:\!\mathsf{fin}\,\mathsf{s}m.\ \forall x'\!:\!\mathsf{fin}\,\mathsf{s}n.\ \forall f\!:\!\mathsf{fin}\,m \ \to\ T\,n.\ \mathsf{fin}\,\mathsf{s}m \ \to\ T\,\mathsf{s}n$$

$$
\mathsf{liftT}\;x\;x'\;F\,y \;=\;
\begin{array}{|c|l|}
\hline
\mathsf{thick}\;x\;y & \\
\hline
\mathsf{no}\;(\mathsf{fin}\,m) & vT\;x' \\
\mathsf{yes}\;y' & thinT\;x'\;(f\,y') \\
\hline
\end{array}
$$

x is the 'new' source variable and x' is the corresponding target variable. The lifted function uses thick to distinguish new from old, and either embeds x' via vt or thins the result of f with $thinT$.

The map function may now be written

$$
\begin{array}{lcl}
\mathsf{map}\;f\,(\mathsf{var}\;x) & = & TLam\;(f\,x) \\
\mathsf{map}\;f\,(\mathsf{app}\;s\;t) & = & \mathsf{app}\;(\mathsf{map}\;f\,s)\;(\mathsf{map}\;f\,t) \\
\mathsf{map}\;f\,(\mathsf{lam}\;t) & = & \mathsf{lam}\;(\mathsf{map}\;(\mathsf{liftT}\;(\mathsf{fz}\;m)\;(\mathsf{fz}\;n)\;f)\;t)
\end{array}
$$

Once the parameters are discharged, we may take:

$$
\begin{aligned}
\mathsf{Fa} &= \mathsf{map\ fin}\ \iota^{f}\ \mathsf{var\ thin} \\
\mathsf{thinL}_{n}\ x &= \mathsf{Fa}\ (\mathsf{thin}\ x) \\
\cdot\langle\!| &= \mathsf{map\ Lam\ var}\ \iota^{L}\ \mathsf{thinL}
\end{aligned}
$$

Note that the notion of lifting used in renaming

$$\mathsf{liftT\ fin}\ \iota^{f}\ \mathsf{var\ thin}\ (\mathsf{fz}\ m)\ (\mathsf{fz}\ n)$$

is extensionally the same as the lift function we defined earlier. This follows easily by inverting the thick contained in liftT. It therefore inherits all the same functor properties via sameFunctor.

Our task is now to plug these into the relevant functor and monad. I am afraid to say that a little forward planning at this point will pay dividends later. I will motivate it as best I can. Both Functor and Monad require the extensional equality of arrows to be respected: conditions which will apply to both Fa and $\cdot\langle\!|$. Since these are both implemented by map, it is worth proving this property for map while the parameters are still abstracted. The goal is

$$
\begin{aligned}
?\mathsf{mapEq} :\ & \forall m, n : \mathbb{N} \\
& \forall f, g : \mathsf{fin}\ m \to T\ n \\
& \forall hyp : \forall x : \mathsf{fin}\ m \\
& \qquad f\,x \simeq g\,x \\
& \forall \boxed{x} : \mathsf{Lam}\ m \\
& \quad \mathsf{map}\ f\,x \simeq \boxed{\mathsf{map}\ g\,x}
\end{aligned}
$$

You will, I hope, be unsurprised to learn that the technique I recommend is recursion induction on map. Either map will do—I have chosen the second. Three subgoals, one at a time:

$$
\begin{aligned}
?\mathsf{mapEq}_{v} :\ & \forall m, n : \mathbb{N} \\
& \forall f, g : \mathsf{fin}\ m \to T\ n \\
& \forall hyp : \forall x : \mathsf{fin}\ m \\
& \qquad f\,x \simeq g\,x \\
& \forall x \quad : \mathsf{fin}\ m \\
& \quad TLam\ (f\,x) \simeq TLam\ (g\,x)
\end{aligned}
$$

Rewrite by *hyp*. Next ...

$$
\begin{array}{l}
?\mathsf{mapEq}_a : \forall m, n : \mathbb{N} \\
\qquad\qquad \forall f, g \;: \mathsf{fin}\, m \;\rightarrow\; T\, n \\
\qquad\qquad \forall hyp \;: \forall x \colon \mathsf{fin}\, m \\
\qquad\qquad\qquad\quad f\, x \simeq g\, x \\
\qquad\qquad \forall s, t \;\;: \mathsf{Lam}\, m \\
\qquad\qquad \forall s', t' : \mathsf{Lam}\, n \\
\qquad\qquad \forall shyp \colon \forall f \quad : \mathsf{fin}\, m \;\rightarrow\; T\, n \\
\qquad\qquad\qquad\quad \forall hyp \colon \forall x \colon \mathsf{fin}\, m \\
\qquad\qquad\qquad\qquad\qquad f\, x \simeq g\, x \\
\qquad\qquad\qquad\quad \mathsf{map}\; f\, s \simeq s' \\
\qquad\qquad \forall thyp \colon \\
\qquad\qquad\quad \mathsf{app}\; (\mathsf{map}\; f\, s)\; (\mathsf{map}\; f\, t) \simeq \mathsf{app}\; s'\; t'
\end{array}
$$

If we plug *hyp* into *shyp*, we can turn $(\mathsf{map}\; f\, s)$ into s'. The same thing happens with $(\mathsf{map}\; f\, t)$. In fact, all the inductive proofs (implicitly) on Lam we shall encounter in this thesis have an app case whose proof is ‘rewrite by the inductive hypotheses’. From now on, I shall omit them.

Of course, the real interest is in the lam case:

$$
\begin{array}{l}
?\mathsf{mapEq}_l : \forall m, n : \mathbb{N} \\
\qquad\qquad \forall f, g \;: \mathsf{fin}\, m \;\rightarrow\; T\, n \\
\qquad\qquad \forall hyp \;: \forall x \colon \mathsf{fin}\, m \\
\qquad\qquad\qquad\quad f\, x \simeq g\, x \\
\qquad\qquad \forall t \qquad : \mathsf{Lam}\, \mathsf{s}m \\
\qquad\qquad \forall t' \qquad : \mathsf{Lam}\, \mathsf{s}n \\
\qquad\qquad \forall thyp \colon \forall f \quad : \mathsf{fin}\, \mathsf{s}m \;\rightarrow\; T\, \mathsf{s}n \\
\qquad\qquad\qquad\quad \forall hyp \colon \forall x \colon \mathsf{fin}\, \mathsf{s}m \\
\qquad\qquad\qquad\qquad\qquad f\, x \simeq \mathsf{liftT}\; (\mathsf{fz}\, m)\; (\mathsf{fz}\, n)\; g\, x \\
\qquad\qquad\qquad\quad \mathsf{map}\; f\, t \simeq t' \\
\qquad\qquad\quad \mathsf{lam}\; (\mathsf{map}\; (\mathsf{liftT}\; (\mathsf{fz}\, m)\; (\mathsf{fz}\, n)\; f)\; t) \simeq \mathsf{lam}\; t'
\end{array}
$$

Now, equation respects function application, so we may strip off those λs. The conclusion is now

$$
\mathsf{map}\; (\mathsf{liftT}\; (\mathsf{fz}\, n)\; (\mathsf{fz}\, m)\; f)\; t \simeq t'
$$

and this is ripe for the inductive hypothesis, leaving us with

$$
\begin{array}{l}
?hyp' : \forall x : \mathsf{fin}\;\mathsf{s}m \\
\qquad \mathsf{liftT}\;(\mathsf{fz}\;m)\;(\mathsf{fz}\;n)\;f\,x \simeq \mathsf{liftT}\;(\mathsf{fz}\;m)\;(\mathsf{fz}\;n)\;g\,x
\end{array}
$$

Expanding liftT, we find this is really

$$
\begin{array}{l}
?hyp' : \forall x : \mathsf{fin}\;\mathsf{s}m \\
\qquad \begin{array}{|c|l|}\hline \mathsf{thick}\;(\mathsf{fz}\;m)\;x & \\ \hline \mathsf{no}\;(\mathsf{fin}\;m) & vT\;(\mathsf{fz}\;n) \\ \mathsf{yes}\;y & thinT\;(\mathsf{fz}\;n)\;(f\,y) \\ \hline \end{array}
= \begin{array}{|c|l|}\hline \mathsf{thick}\;(\mathsf{fz}\;m)\;x & \\ \hline \mathsf{no}\;(\mathsf{fin}\;m) & vT\;(\mathsf{fz}\;n) \\ \mathsf{yes}\;y & thinT\;(\mathsf{fz}\;n)\;(g\,y) \\ \hline \end{array}
\end{array}
$$

The computation is blocked by the two thick applications, but we know how to invert them. Indeed, since they have the same arguments, we may invert them simultaneously. Of course, in this instance, a case analysis on x would have the same effect, but that is only because we are thickening at $(\mathsf{fz}\,n)$, and we know how thick is implemented—we want the effect of inversion, so we do inversion. We are left with two cases:

$$
\begin{array}{l}
?case_n : vT\;(\mathsf{fz}\;n) \simeq vT\;(\mathsf{fz}\;n) \\
?case_y : \forall y : \mathsf{fin}\;n \\
\qquad thinT\;(\mathsf{fz}\;n)\;(f\,y) \simeq thinT\;(\mathsf{fz}\;n)\;(g\,y)
\end{array}
$$

The first is reflexive; the second becomes so after rewriting with *hyp*. We have proven mapEq and may now discharge the parameters.

Let us prove that renaming is functorial—we have already supplied Fo and Fa. It remains to prove the properties. PresEq is just a special case of mapEq.

The PresI property gives us the goal

$$
\begin{array}{l}
?\mathsf{PresI} : \forall m : \mathbb{N} \\
\qquad \forall t \;: \mathsf{Lam}\;m \\
\qquad\quad \mathsf{Fa}\;\iota^f_m\;t \simeq t
\end{array}
$$

Here, at last, my devotion to recursion induction comes unstuck. The trouble is twofold:

- The scheme for map recursion induction is abstracted over different source and target objects and here they are unified. The elimination tactic will supply a constraint to resolve this, but it is a little clumsy.

- The scheme is abstracted over an arbitrary renaming, but we are concerned with a very particular one. Again the tactic will give us a constraint—that the function is intensionally equal to ι^f. We will only have extensional equality, so the proof will not go through.

There is still much work to do to come to an understanding of the correct manipulation of constraints for this kind of inductive proof. In the meantime, let us do structural induction on t! The var and app cases are easy.[6] Here is lam:

$$
\begin{array}{ll}
?\mathsf{PresI}_l\colon & \forall m \quad : \mathbb{N} \\
& \forall t \quad\; : \mathsf{Lam}\;\mathsf{s}m \\
& \forall thyp\colon \mathsf{Fa}\;\iota^f_{\mathsf{s}m}\;t \simeq t \\
& \quad \mathsf{lam}\;(\mathsf{Fa}\;(\mathsf{liftT}\ldots(\mathsf{fz}\;m)\;(\mathsf{fz}\;m)\;\iota^f_m)\;t) \simeq \mathsf{lam}\;t
\end{array}
$$

We may introduce the hypotheses and strip off the lams. This leaves us with $\ldots \simeq t$. The inductive hypothesis looks a bit like that, so let us try transitivity (or rewriting backwards).

$$
?\mathsf{PresI}'_l\colon \mathsf{Fa}\;(\mathsf{liftT}\ldots(\mathsf{fz}\;m)\;(\mathsf{fz}\;m)\;\iota^f_m)\;t \simeq \mathsf{Fa}\;\iota^f_{\mathsf{s}m}\;t
$$

Now we get a bonus for proving mapEq in advance. The goal asks us to show that two renamings do the same thing to a term t. If we apply mapEq, it is enough to show that they agree at every variable:

$$
\begin{array}{ll}
?same\colon & \forall x\colon \mathsf{fin}\;\mathsf{s}m \\
& \mathsf{liftT}\ldots(\mathsf{fz}\;m)\;(\mathsf{fz}\;m)\;\iota^f_m\;x \simeq \iota^f_{\mathsf{s}m}\;x
\end{array}
$$

But $\mathsf{liftT}\ldots(\mathsf{fz}\;m)\;(\mathsf{fz}\;m)$ has the same functor properties as lift, including preservation of identity—exactly the goal here.

Next, PresC:

$$
\begin{array}{ll}
?\mathsf{PresC}\colon & \forall t, r, s\colon \mathbb{N} \\
& \forall f \quad : s \rightarrowtail^f t \\
& \forall g \quad : r \rightarrowtail^f s \\
& \forall \boxed{x} \;\; : \mathsf{Lam}\;r \\
& \quad \mathsf{Fa}\;(f \circ^f g)\;x \simeq \mathsf{Fa}\;f\;\boxed{\mathsf{Fa}\;g\;x}
\end{array}
$$

[6] Trust me, I'm doing the proof as I write this.

Recursion induction is once more our friend. Eliminating the boxed application, we again find easy var and app cases. The lam case is very similar to that in the previous proof:

$$
\begin{array}{l}
?\mathsf{PresC}_l\colon \forall t, r, s\colon \mathbb{N} \\
\qquad \begin{array}{ll}
\forall f & : s \rightarrowtail^f t \\
\forall g & : r \rightarrowtail^f s \\
\forall x & : \mathsf{Lam}\ \mathsf{s}r \\
\forall x' & : \mathsf{Lam}\ \mathsf{s}s \\
\forall hyp & : \forall t\colon \mathbb{N} \\
 & \quad \forall f\colon \mathsf{s}s \rightarrowtail^f t \\
 & \qquad \mathsf{Fa}\ (f \circ^f (\mathsf{liftT} \ldots g))\ x \simeq \mathsf{Fa}\ f\, x'
\end{array} \\
\qquad\quad \mathsf{lam}\ (\mathsf{Fa}\ (\mathsf{liftT} \ldots (f \circ^f g))\ x) \simeq \mathsf{lam}\ (\mathsf{Fa}\ (\mathsf{liftT} \ldots f)\ x')
\end{array}
$$

Once again, strip the lams, apply transitivity with the inductive hypothesis on the right, and then mapEq, leaving:

$$
\begin{array}{l}
?\mathsf{PresC}_l\colon \forall x\colon \mathsf{fin}\ \mathsf{s}r \\
\qquad\quad \mathsf{liftT} \ldots (f \circ^f g)\ x \simeq \mathsf{liftT} \ldots f\, (\mathsf{liftT} \ldots g\, x)
\end{array}
$$

Quelle surprise! The property that the lifting functor preserves composition! Renaming is a functor!

Now let us turn to showing that substitution is monadic. We have already supplied $\rangle\!\rangle\cdot$ (composition with var) and $\cdot\langle\!\langle$. Since the representation of $\searrow$ is functional, we interpret these arrows by application. We may also supply directly the Kleisli $\diamond$ demanded by MonadC:

$$f \diamond g\, x \;=\; f\langle\!\langle\ (g\ mx)$$

MonadI reduces to reflexivity and MonadC is true by construction. $\mathsf{FrontEq}$ follows because var respects equality whilst BackEq is an instance of mapEq. FrontC is reflexive. Only Split and BackC require any real work.

Split says

$$
\begin{array}{l}
?\mathsf{Split}\colon \forall m, n\colon \mathbb{N} \\
\qquad \begin{array}{ll}
\forall f & : m \rightarrowtail^f n \\
\forall \boxed{t} & : \mathsf{Lam}\ m
\end{array} \\
\qquad\ \ \rangle\!\rangle f \langle\!\langle\ t \simeq \boxed{\mathsf{Fa}\ f\ t}
\end{array}
$$

We can prove this with exactly the same plan as before. Recursion induction leaves easy var and app cases. The lam case reduces by the same strategy as before to

$$\begin{array}{l} ?\mathsf{Split}_l \colon \forall x \colon \mathsf{fin}\ \mathsf{s}m \\ \qquad\qquad \mathsf{liftT}\ldots\mathsf{thinL}\ (\mathsf{fz}\ m)\ (\mathsf{fz}\ n)\ (\rangle f)\ x \simeq \mathsf{var}\ (\mathsf{liftT}\ldots\mathsf{thin}\ (\mathsf{fz}\ m)\ (\mathsf{fz}\ n)\ f\,x) \end{array}$$

That is, composing var and lifting must commute. Both liftTs, on expansion, are blocked at $(\mathsf{thick}\ (\mathsf{fz}\ m)\ x)$. Inverting thick leaves two trivial subgoals.

BackC starts the same way:

$$\begin{array}{ll} ?\mathsf{BackC} \colon & \forall r, s, t \colon \mathbb{N} \\ & \forall f \quad : s \searrow t \\ & \forall g \quad : r \searrow s \\ & \forall \boxed{x} \quad : \mathsf{Lam}\ r \\ & \quad (f \diamond g)\langle\!|\ x \simeq f\langle\!|\ \boxed{g\langle\!|\ x} \end{array}$$

But after the usual story, the lam case is reduced to

$$\begin{array}{l} ?\mathsf{BackC}_l \colon \forall x \colon \mathsf{fin}\ \mathsf{s}r \\ \qquad\qquad \mathsf{liftT}\ldots(f \diamond g)\ x \simeq \mathsf{liftT}\ldots f\,(\mathsf{liftT}\ldots g\,x) \end{array}$$

This says that lifting for substitutions must respect composition—we only know this result for renamings. We can boil the goal down a little further by expanding the outer liftTs and inverting their blocked thicks. This give us two cases: one for the newly bound variable, just a reflexive equation, indicating that it is correctly propagated by lifting; the other, for the free variables, is still awkward.

$$\begin{array}{l} ?\mathsf{BackC}_l \colon \forall x \colon \mathsf{fin}\ r \\ \qquad\qquad \mathsf{thinL}\ (\mathsf{fz}\ t)(f\langle\!|\ (g\,x)) \simeq (\mathsf{liftT}\ldots\ f)\langle\!|\ (\mathsf{thinL}\ (\mathsf{fz}\ s)\ (g\,x)) \end{array}$$

This is a special case of the last lemma we need to prove—a crucial fact about the relationship between thinning and substitution:

$$
\begin{array}{l}
?\mathsf{thinSubst} \colon \forall m, n \colon \mathbb{N} \\
\qquad \forall x \quad \colon \mathsf{fin}\ \mathsf{s}m \\
\qquad \forall x' \quad \colon \mathsf{fin}\ \mathsf{s}n \\
\qquad \forall f \quad \colon m \searrow n \\
\qquad \forall \boxed{t} \quad \colon \mathsf{Lam}\ m \\
\qquad\quad \mathsf{thinL}\ x'\ \boxed{f \langle\!| \ t} \simeq (\mathsf{liftT} \ldots\ x\ x'\ f) \langle\!|\ (\mathsf{thinL}\ x\ t)
\end{array}
$$

That is, substituting then thinning has the same effect as thining first, then applying the lifted substitution.

There is no point inventing a new proof plan when an old one will do. var and app are easy as before. Modulo the need to switch between a liftTed thin and a lifted thin (ie another thin), we can again reduce the lam case to an equation involving blocked liftTs which we simplify by inversion, leaving us with the free variable case:

$$
\begin{array}{l}
?\mathsf{thinSubst}_l \colon \forall y \colon \mathsf{fin}\ m \\
\qquad \mathsf{thinL}\ (\mathsf{fs}\ x')\ (\mathsf{thinL}\ (\mathsf{fz}\ n)\ (f\,y)) \simeq \mathsf{thinL}\ (\mathsf{fz}\ \mathsf{s}n)\ (\mathsf{thinL}\ x'\ (f\,y))
\end{array}
$$

Now thinL is just renaming via thin, so what we really have is

$$
\begin{array}{l}
?\mathsf{thinSubst}_l \colon \forall y \colon \mathsf{fin}\ m \\
\qquad \mathsf{Fa}\ (\mathsf{lift}\ (\mathsf{thin}\ x'))\ (\mathsf{Fa}\ (\mathsf{fs}_n)\ (f\,y)) \simeq \mathsf{Fa}\ (\mathsf{fs}_{\mathsf{s}n})\ (\mathsf{Fa}\ (\mathsf{thin}\ x')\ (f\,y))
\end{array}
$$

We may rewrite both sides by the property that renaming preserves composition (backwards):

$$
\begin{array}{l}
?\mathsf{thinSubst}_l \colon \forall y \colon \mathsf{fin}\ m \\
\qquad \mathsf{Fa}\ ((\mathsf{lift}\ (\mathsf{thin}\ x')) \circ^f \mathsf{fs}_n)\ (f\,y) \simeq \mathsf{Fa}\ (\mathsf{fs}_{\mathsf{s}n} \circ^f (\mathsf{thin}\ x'))\ (f\,y)
\end{array}
$$

But all the lift does is shuffle fs through $(\mathsf{thin}\ x')$. The two sides of the equation are intensionally the same. We have proven that substitution is monadic.

7.3 a correct first-order unification algorithm

This is the main example of dependently typed functional programming in this thesis.

I propose to study unification for 'trees with holes'. The algorithm is a variation on the theme which goes back to Alan Robinson [Rob65]. It is the program implementing the algorithm which is new, and which benefits from the dependent type system in a way which is just not available in the simply typed world, even with the remarkable higher-order polymorphic extensions which are becoming available in the more upmarket sorts of programming language. Here we shall make critical use of the fact that our types depend on data—real data with elimination as well as introduction rules.

Just as with Lam, let us represent variables via fin, but since trees have no binding, we may fix the number of variables as a parameter of the type.

- formation rule
$$\frac{n : \mathbb{N}}{\mathsf{tree}\; n : \mathcal{T}ype}$$
- constructors
$$\frac{x : \mathsf{fin}\; n}{\mathsf{var}\; x : \mathsf{tree}\; n} \qquad \frac{}{\mathsf{leaf}_n : \mathsf{tree}\; n} \qquad \frac{s, t : \mathsf{tree}\; n}{\mathsf{fork}\; s\; t : \mathsf{tree}\; n}$$
- elimination rule
$$\boxed{\begin{array}{l} n : \mathbb{N} \\ \Phi : \mathsf{tree}\; n \;\rightarrow\; \mathcal{T}ype \\ \\ \dfrac{\Phi\;(\mathsf{var}\; x) \qquad \Phi\; \mathsf{leaf}_n \qquad \begin{array}{c} \Phi\, s \quad \Phi\, t \\ \cdots\cdots\cdots \\ \Phi\;(\mathsf{fork}\; s\; t) \end{array}}{\forall \boxed{t : \mathsf{tree}\; n}.\; \Phi\; t} \end{array}}$$

We may construct the renaming functor and substitution monad for tree following much the same path as for Lam, but without the work required to cope with binding. For this section, let us have

$$\begin{array}{l} m \rightarrowtail^{t} n \;\Longrightarrow\; \mathsf{tree}\; m \;\rightarrow\; \mathsf{tree}\; n \\ m \searrow n \;\Longrightarrow\; \mathsf{fin}\; m \;\rightarrow\; \mathsf{tree}\; n \\ \mathsf{Rename} : \mathsf{Functor}\; [\mathsf{fin}]\; [\mathsf{tree}] \\ \mathsf{SubstM} : \mathsf{Monad}\; \mathsf{Rename}\; \searrow \\ \mathsf{SubstK} \;=\; \mathsf{Kleisli}\; \mathsf{SubstM} \end{array}$$

Within this framework, we may equip substitutions with the preorder induced by prior composition:

$$f \diamond g \;\leq\; g$$

The task of unifying some $s, t : \mathsf{tree}\ m$ is to find (an n and) an arrow $f : m \searrow n$ such that

$$f\langle\!|\ s \;=\; f\langle\!|\ t$$

if any exists, and in particular, to find one which is maximal with respect to the above ordering.

Unification is thus an optimisation problem, and it is worth spending a little time thinking about such problems in general, before proceeding with this particular example.

7.3.1 optimistic optimisation

Unification is just one example of a problem involving optimisation with respect to a conjunction of constraints. I should like to draw your attention to a particular class of constraint which makes such problems vulnerable to a reassuringly naïve technique—optimism.

That is, we begin by guessing that the optimum is the best thing we can think of. Then, as we encounter each constraint in turn, we continue to think the best that it allows, reducing our current guess by only so much as is necessary. Once we have worked our way through all the constraints, it is to be hoped that our final guess, however battered by bitter experience, is genuinely optimal.

This hope holds true if each constraint has the property that once a solution has been found, anything smaller remains a solution. Let us call such constraints **downward-closed**, or **closed** for short. This property of constraints gives the underlying rationale to the transformation of recursive optimisation algorithms which relativises them to an accumulating solution—a technique which has already found its way into the automated synthesis of (parts of) a unification algorithm in [ASG99].

We can give a record type characterising such properties for arrows ordered by composition. Fixing a category and a source object S, we may represent a closed constraint on S out-arrows as inhabitants of the record type $\mathsf{Closed}\ S$ with fields:

$$\begin{array}{r l}
\mathsf{Why} : & \forall T.\ (S \rightarrowtail T) \ \rightarrow\ \mathit{Type} \\
\mathsf{ClosedEq} : & \forall T.\ \forall f, g : S \rightarrowtail T.\ f \approx g \ \rightarrow\ \mathsf{Why}\ f \ \rightarrow\ \mathsf{Why}\ g \\
\mathsf{Closure} : & \forall T.\ \forall g : S \rightarrowtail T.\ \mathsf{Why}\ g \ \rightarrow\ \forall U.\ \forall f : T \rightarrowtail U.\ \mathsf{Why}\ f \circ g
\end{array}$$

Note the extra condition that the constraint must not distinguish extensionally equal arrows. This is the price of allowing functional representations of arrows in intensional type theory.

We may further define what it means to be maximal with respect to such a constraint. Fixing and opening a $\mathsf{Closed}\,S$ record, and also fixing a target T and an arrow $f : S \rightarrowtail T$, we may collect the relevant conditions in a record $\mathsf{Maximal}\ f$ with fields:

$$
\begin{array}{rl}
\mathsf{Holds} : & \mathsf{Why}\ f \\
\mathsf{Factors} : & \forall U.\ \forall g : S \rightarrowtail U.\ \mathsf{Why}\ g\ \to\ \Sigma h : T \rightarrowtail U.\ g \approx h \circ f
\end{array}
$$

That is, f must be a solution, and every other solution g must be smaller than f, with a witness h such that $g \approx h \circ f$. We may easily prove that maximality respects extensional equality of arrows.

Next, let us define an operator which conjoins closed constraints.

$$
\begin{array}{l}
\mathsf{AND} : \forall S.\ \forall P, Q : \mathsf{Closed}\ S.\ \mathsf{Closed}\ S \\
\mathsf{AND}\ \langle \mathsf{Why} = P; \mathsf{ClosedEq} = PEq; \mathsf{Closure} = PCl \rangle \\
\qquad \langle \mathsf{Why} = Q; \mathsf{ClosedEq} = QEq; \mathsf{Closure} = QCl \rangle \\
\quad = \langle \mathsf{Why} = \lambda T.\ \lambda f.\ (P\ f) \times (Q\ f); \ldots \rangle
\end{array}
$$

The proofs of the properties are unremarkable.

The optimistic strategy at each constraint P extends an accumulated guess g by enough of an f that $P.\mathsf{Why}\ f \circ g$ holds. We may regard this as effectively constraining the witnesses f to the existence of solutions to P bounded by g. The constraint on f is closed provided P is. Let us therefore construct an operator

$$
\begin{array}{l}
\mathsf{Bound} : \forall S, T.\ \forall g : S \rightarrowtail T.\ \mathsf{Closed}\ S\ \to\ \mathsf{Closed}\ T \\
\mathsf{Bound}\ g\ \langle \mathsf{Why} = P; \mathsf{ClosedEq} = PEq; \mathsf{Closure} = PCl \rangle \\
\qquad = \langle \mathsf{Why} = \lambda U.\ \lambda f : T \rightarrowtail U.\ P\ f \circ g; \ldots \rangle
\end{array}
$$

Again, the properties are easily proven.

We are now ready to prove the optimist's lemma:

$$
\begin{array}{lll}
?\mathsf{Optimist} : & \forall R, S, T & : O \\
& \forall clP, clQ & : \mathsf{Closed}\ R \\
& \forall g & : R \rightarrowtail S \\
& \forall gMax & : \mathsf{Maximal}\ P\ g \\
& \forall f & : S \rightarrowtail T \\
& \forall fMax & : \mathsf{Maximal}\ (\mathsf{Bound}\ g\ P)\ f \\
& \multicolumn{2}{l}{\mathsf{Maximal}\ (\mathsf{AND}\ P\ Q)\ (f \circ g)}
\end{array}
$$

This is the key step in the correctness proof for the optimistic strategy. It tells us that a conjunction $(\mathsf{AND}\; P\; Q)$ may be optimised by extending the optimum g for P with just enough f to satisfy Q. The proof is not very difficult, which is one of the reasons why I like it.

First, let us unpack the definitions by the elimination rules for the argument records and introduce the hypotheses:

$$
\begin{array}{ll}
\lambda R, S, T & : O \\
\lambda P & : \ldots \\
\lambda PEq & : \ldots \\
\lambda PCl & : \ldots \\
\lambda Q & : \ldots \\
\lambda QEq & : \ldots \\
\lambda QCl & : \ldots \\
\lambda g & : R \rightarrowtail S \\
\lambda gHolds & : P\; g \\
\lambda gFactors & : \forall U : O \\
 & \quad \forall k\; : R \rightarrowtail U \\
 & \quad \forall Pk : P\; k \\
 & \quad \Sigma h\; : S \rightarrowtail U \\
 & \quad\quad k \approx h \circ g \\
\lambda f & : S \rightarrowtail T \\
\lambda fgHolds & : Q\; f \circ g \\
\lambda fgFactors & : \forall U \;\; : O \\
 & \quad \forall k \;\;\; : S \rightarrowtail U \\
 & \quad \forall Qkg : Q\; k \circ g \\
 & \quad \Sigma h \;\;\; : T \rightarrowtail U \\
 & \quad\quad k \approx h \circ f \\
?max & : \mathsf{Maximal}\; (\mathsf{AND}\; P\; Q)\; (f \circ g)
\end{array}
$$

We may also attack the goal with the introduction rules for records and implications:

$$
\begin{array}{ll}
?Pfg & : P\; f \circ g \\
?Qfg & : Q\; f \circ g \\
\lambda U & : O \\
\lambda k & : R \rightarrowtail U \\
\lambda Pk & : P\; k \\
\lambda Qk & : Q\; k \\
?h & : T \rightarrowtail U \\
?kEq & : k \approx h \circ (f \circ g)
\end{array}
$$

Now, $Q\ f \circ g$ is already known to hold, and $P\ f \circ g$ follows by *PCl* from *gHolds*, so we have certainly found a solution to the composite problem. It remains to show the optimality by expressing the hypothetical solution k as some $h \circ f \circ g$.

The proof successively exploits the optimality of the solution to each subproblem. Firstly, we use *gFactors* to acquire for some $h' : S \rightarrowtail U$

$$k \approx h' \circ g$$

By *QEq*, we now know that $Q\ h' \circ g$, hence *fgFactors* gives us an $h : T \rightarrowtail U$ with

$$h' \approx h \circ f$$

We supply this h as the witness, for we have

$$k \approx h' \circ g \approx (h \circ f) \circ g \approx h \circ (f \circ g)$$

as required.

Note that the proof does not make use of *QCl*. In effect, we can optimise with respect to a collection of constraints all but one of which are downward-closed, as long as we address the non-closed constraint last—it is not a freedom we shall need.

The $\mathsf{Optimist}$ lemma allows us to solve a complex closed constraint by recursively decomposing it into an equivalent conjuction of simpler closed constraints, each of which we address in turn, accumulating the solution. Accordingly, we shall need a book equivalence on closed constraints

$$\begin{array}{l}
\mathsf{Equiv} : \forall S.\ \mathsf{Closed}\ S \rightarrow \mathsf{Closed}\ S \rightarrow \mathit{Type} \\
\mathsf{Equiv}\ \langle \mathsf{Why} = P; \ldots \rangle\ \langle \mathsf{Why} = Q; \ldots \rangle \\
\quad = \forall T.\ \forall f{:}S \rightarrowtail T.\ (P\ f \rightarrow Q\ f) \times (Q\ f \rightarrow P\ f)
\end{array}$$

together with a proof $\mathsf{EquivMax}$ that for equivalent constraints an arrow maximising one also maximises the other—this is easy.

Lots of algorithms follow the optimistic strategy, from finding the largest element of a nonempty list of numbers to principal type inference for ML. Let us see how it works for unification.

7.3.2 optimistic unification

A unifier for $s, t : \mathsf{tree}\, m$ is a substitution $f : m \searrow n$ subject to the constraint $f\llparenthesis\, s \simeq f\llparenthesis\, t$. We may thus consider the computation of most general unifiers to be an optimisation problem over the Kleisli category SubstK induced by the substitution monad. Fortunately for us, the constraint is downward-closed. We may construct

$$\mathsf{Unifies}\; s\; t : \mathsf{Closed}\; m$$
$$\mathsf{Unifies}\; s\; t \;=\; \langle \mathsf{Why} = \lambda n.\, \lambda f.\; f\llparenthesis\, s \simeq\; f\llparenthesis\, t; \ldots \rangle$$

The two properties are easily proven. Extensional equality of arrows in SubstK means exactly that they have the same effect on terms. Downwards closure follows from the fact that the interpretation of arrows in the Kleisli category—substitution—respects composition.

It is easy to provide the justification for the structural decomposition of rigid-rigid problems:

$$\begin{array}{ll}\mathsf{Equiv} & (\mathsf{Unifies}\,(\mathsf{fork}\, s_1\; t_1)\;(\mathsf{fork}\, s_2\; t_2)) \\ & (\mathsf{AND}\;(\mathsf{Unifies}\, s_1\; s_2)\;(\mathsf{Unifies}\, t_1\; t_2))\end{array}$$

We may represent out-arrows from m by a dependent pair

$$\mathsf{from}\; m \;\Longrightarrow\; \Sigma n : \mathbb{N}.\; m \searrow n$$

We might well guess that the type of the unification algorithm should be

$$\mathsf{mgu} : \forall m.\, \forall s, t \,{:}\, \mathsf{tree}\; m.\; \mathsf{maybe}\;(\mathsf{from}\; m)$$

The adoption of the optimist strategy means defining mgu in terms of a subfunction bmgu computing unifiers which are most general relative to an accumulated bound.

$$\mathsf{bmgu} : \forall m.\, \forall s, t \,{:}\, \mathsf{tree}\; m.\; \mathsf{from}\; m \;\to\; \mathsf{maybe}\;(\mathsf{from}\; m)$$

The identity substitution is the biggest substitution in the composition ordering, so we take

$$\mathsf{mgu}_m\; s\; t \;=\; \mathsf{bmgu}\; s\; t\; \langle m; \iota \rangle$$

Note that for any given s and t this function is an arrow in the Kleisli category of the maybe monad—we already know how to propagate unification failures correctly. This suggests a functional definition of bmgu, with the rigid-rigid cases given by:

$$
\begin{array}{llllll}
\mathsf{bmgu} & \mathsf{leaf}_m & \mathsf{leaf}_m & = & \lambda f.\ \mathsf{yes}\ f \\
\mathsf{bmgu} & \mathsf{leaf}_m & (\mathsf{fork}\ s\ t) & = & \lambda f.\ \mathsf{no}\ (\mathsf{from}\ m) \\
\mathsf{bmgu} & (\mathsf{fork}\ s\ t & \mathsf{leaf}_m & = & \lambda f.\ \mathsf{no}\ (\mathsf{from}\ m) \\
\mathsf{bmgu} & (\mathsf{fork}\ s_1\ t_1) & (\mathsf{fork}\ s_2\ t_2) & = & (\mathsf{bmgu}\ t_1\ t_2) \diamond (\mathsf{bmgu}\ s_1\ s_2) \\
\quad\vdots
\end{array}
$$

So far, this is structural on terms. The trouble comes once we encounter a variable. How do we unify a variable with a tree, relative to a nontrivial bounding guess g? The traditional approach is to unload the accumulator g, and we may easily prove the lemma Unload

$$
\mathsf{Equiv}\quad (\mathsf{Bound}\ g\ (\mathsf{Unifies}\ s\ t))\quad (\mathsf{Unifies}\ \ g\langle\!|\ s\ \ g\langle\!|\ t)
$$

Unfortunately, applying the substitution blows up the terms, so the corresponding recursive program is not structural. This is where you might think we need to impose an external termination ordering or accessibility argument which exploits the fact that, although the substitutions blow up the terms, they do get rid of variables. In fact, this is not the case.

7.3.3 dependent types to the rescue

Incidentally, I have just noticed that Augustsson and Carlsson's paper [AC99] also contains a section with this title—I expect it to become traditional.

Now, we certainly need to exploit the property that the accumulated substitution gets rid of variables as it blows up terms. Every development of unification in the literature[7] does this externally to the program, by means of a more or less ad hoc termination ordering. This invariably requires an auxiliary function to count the distinct variables in a term and an auxiliary lemma which relates the value of this function before and after a substitution subject to the occur check.

That is to say, a vital component of the sense made by the unification algorithm has been absent from every one of its implementations until now—understandably, because the data structures which manifest that sense have not been available until now. The

[7] or at least those which care about termination

point is that by explaining terms as built over a finite context of variables, we have equipped them with exactly the natural recursive behaviour which we need. To count the number of variables in a term is to make a posterior phenomenon of what is, at least to structural linguists [Sau16], a prior requirement for the terms to be considered meaningful. The number of variables has finally arrived where it belongs—in the *type* of terms.

Look again at the type of bmgu:

$$\mathsf{bmgu} : \forall m.\ \forall s, t : \mathsf{tree}\ m.\ \mathsf{from}\ m \ \rightarrow\ \mathsf{maybe}\ (\mathsf{from}\ m)$$

This entitles us to proceed not only by structural recursion on trees, but also by structural recursion on m. I cannot stress too strongly that it is the indexing of types with *terms* which allows this. Parametric polymorphism is not enough, because we cannot compute on types. There are structural forms of computation available in our dependently typed setting which just cannot be found in simply typed languages.

The recursive structure I therefore suggest is lexicographic, first on m and then on s. If we are unifying trees over $\mathsf{s}m$ variables, we are entitled to make recursive calls for any trees over m variables, however large.

Of course, the number of variables must not merely be decreasing—it must do so in a structural way, one at a time if we are to avoid further appeals to well-founded recursion. We have already seen how to define a substitution which gets rid of a single variable via the $[\cdot \mapsto \cdot]$ function. Here it is again:

$$[\cdot \mapsto \cdot] : \forall n : \mathbb{N}.\ \forall x : \mathsf{fin}\ \mathsf{s}n.\ \forall t : \mathsf{tree}\ n.\ (\mathsf{s}n \searrow n)$$

$$[x \mapsto t]y \quad = \quad \begin{array}{|c|l|} \hline \mathsf{thick}\ x\ y & \\ \hline \mathsf{no}\ (\mathsf{fin}\ n) & t \\ \mathsf{yes}\ y' & \mathsf{var}\ y' \\ \hline \end{array}$$

$[\cdot \mapsto \cdot]$ can easily be shown to have extensional behaviour (or, thinking relationally, introduction rules):

$$\overline{[x \mapsto t]x \simeq t} \qquad \overline{[x \mapsto t](\mathsf{thin}\ x\ y) \simeq \mathsf{var}\ y}$$

These follow directly from the established extensional behaviour of thick. The corresponding inversion rule, $\mathsf{knockoutInv}$, follows from $\mathsf{thickInv}$:

$$\boxed{\begin{array}{l}\mathsf{knockoutInv}\\[1em] m : \mathbb{N}\\ x : \mathsf{fin}\,\mathsf{s}m\\ t : \mathsf{tree}\,m\\ \Phi : \mathsf{fin}\,\mathsf{s}m \to \mathsf{tree}\,m \to \mathit{Type}\\[1em] \qquad \dfrac{\Phi\,x\,t \quad \Phi\,(\mathsf{thin}\,x\,y)\,(\mathsf{var}\,y)}{\Phi\,y\,[x \mapsto t]y}\end{array}}$$

If our accumulator is a composition of $[\cdot \mapsto \cdot]$s, we may apply it one step at a time whenever we reach a variable. In fact, this is not merely a structural way to do unification, but also quite an efficient one. Of course, we must constrain the accumulator to take this form, and the easiest way to do this means abandoning our functional representation of substitution in favour of a more concrete 'association list' treatment.

Let us then define the following datatype

- formation rule $\dfrac{m, n : \mathbb{N}}{\mathsf{alist}\,m\,n : \mathit{Type}}$
- constructors $\dfrac{}{\mathsf{anil}_n : \mathsf{alist}\,n\,n} \quad \dfrac{x : \mathsf{fin}\,\mathsf{s}m \quad t : \mathsf{tree}\,m \quad g : \mathsf{alist}\,m\,n}{\mathsf{acons}\,x\,t\,g : \mathsf{alist}\,\mathsf{s}m\,n}$

This datatype is a combination of a conventional association list and the $\geq$ relation. It is definable in ALF, COQ and OLEG, but not in Agda or Cayenne because of its nonlinear base constructor type.

We may equip it with

- a composition which behaves like append for association lists and transitivity for $\geq$
  $$\begin{array}{lcl} f \ddagger \mathsf{anil} & = & f\\ f \ddagger (\mathsf{acons}\,x\,t\,g) & = & \mathsf{acons}\,x\,t\,(f \ddagger g)\end{array}$$
- an interpretation via $[\cdot \mapsto \cdot]$ into SubstK
  $$\begin{array}{rcl} \mathsf{anil} \blacktriangleleft & = & \iota\\ (\mathsf{acons}\,x\,t\,f) \blacktriangleleft & = & (f \blacktriangleleft) \diamond [x \mapsto t]\end{array}$$

Correspondingly, we may manufacture a concrete category $\mathsf{AList} : \mathsf{Concrete}\ \mathsf{alist}$ with

$$\begin{aligned} \iota &= \mathsf{anil} \\ \circ &= \ddagger \\ [\![m]\!] &= \mathsf{tree}\ m \\ [\![g]\!] &= g \blacktriangleleft \langle\!| \end{aligned}$$

There is trivially a functor from AList to SubstK which does $\blacktriangleleft$ to arrows, because the interpretations of arrows in source and target are the same.

It is amongst the arrows of AList that I propose we search for unifiers, although we should still show that any most general f computed yields a most general $f\blacktriangleleft$ in SubstK. Correspondingly, let us take

$$\mathsf{from}\ m \implies \Sigma n : \mathbb{N}.\ \mathsf{alist}\ m\ n$$

and define

$$\begin{aligned} &\mathsf{bmgu} : \forall m.\ \forall s, t : \mathsf{tree}\ m.\ \mathsf{from}\ m \ \to\ \mathsf{maybe}\ (\mathsf{from}\ m) \\ &\mathsf{mgu} : \forall m.\ \forall s, t : \mathsf{tree}\ m.\ \mathsf{maybe}\ (\mathsf{from}\ m) \\ &\mathsf{mgu}\ s\ t \ =\ \mathsf{bmgu}\ s\ t\ \mathsf{anil} \end{aligned}$$

We now have all we need to outline a structurally recursive defininition of bmgu, deferring the treatment of the base cases:

$$
\begin{array}{lllll}
\mathsf{bmgu}_m & \mathsf{leaf}_m & \mathsf{leaf}_m & f & = & \mathsf{yes}\ f \\
\mathsf{bmgu}_m & \mathsf{leaf}_m & (\mathsf{fork}\ s\ t) & f & = & \mathsf{no}\ (\mathsf{from}\ m) \\
\mathsf{bmgu}_m & (\mathsf{fork}\ s\ t) & \mathsf{leaf}_m & f & = & \mathsf{no}\ (\mathsf{from}\ m) \\
\mathsf{bmgu}_m & (\mathsf{fork}\ s_1\ t_1) & (\mathsf{fork}\ s_2\ t_2) & f & = & \\
\multicolumn{6}{l}{\quad (\lambda f.\ \mathsf{bmgu}_m\ t_1\ t_2\ f) \langle\!| \ (\mathsf{bmgu}_m\ s_1\ s_2\ f)}
\end{array}
$$

$$
\begin{array}{lllll}
\mathsf{bmgu}_{\mathsf{s}m} & (\mathsf{var}\ x) & (\mathsf{var}\ y) & f & =
\end{array}
$$

$$
\begin{array}{|c|l|}
\hline
f & \\
\hline
\mathsf{anil} & \mathsf{yes}\ (\mathsf{FlexFlex}\ x\ y) \\
\mathsf{acons}\ z\ r\ g & |\!\rangle(\mathsf{Extend}\ z\ r)\langle\!| \\
 & \quad (\mathsf{bmgu}_m\ [z \mapsto r]\langle\!|\ (\mathsf{var}\ x)\ \ [z \mapsto r]\langle\!|\ (\mathsf{var}\ y)\ g) \\
\hline
\end{array}
$$

$$
\begin{array}{lllll}
\mathsf{bmgu}_{\mathsf{s}m} & (\mathsf{var}\ x) & \mathsf{leaf}_{\mathsf{s}m} & f & =
\end{array}
$$

$$
\begin{array}{|c|l|}
\hline
f & \\
\hline
\mathsf{anil} & \mathsf{FlexRigid}\ x\ \mathsf{leaf}_{\mathsf{s}m} \\
\mathsf{acons}\ z\ r\ g & |\!\rangle(\mathsf{Extend}\ z\ r)\langle\!| \\
 & \quad (\mathsf{bmgu}_m\ [z \mapsto r]\langle\!|\ (\mathsf{var}\ x)\ \ [z \mapsto r]\langle\!|\ \mathsf{leaf}_{\mathsf{s}m}\ g) \\
\hline
\end{array}
$$

$$
\begin{array}{lllll}
\mathsf{bmgu}_{\mathsf{s}m} & (\mathsf{var}\ x) & (\mathsf{fork}\ s\ t) & f & =
\end{array}
$$

$$
\begin{array}{|c|l|}
\hline
f & \\
\hline
\mathsf{anil} & \mathsf{FlexRigid}\ x\ (\mathsf{fork}\ s\ t) \\
\mathsf{acons}\ z\ r\ g & |\!\rangle(\mathsf{Extend}\ z\ r)\langle\!| \\
 & \quad (\mathsf{bmgu}_m\ [z \mapsto r]\langle\!|\ (\mathsf{var}\ x)\ \ [z \mapsto r]\langle\!|\ (\mathsf{fork}\ s\ t)\ g) \\
\hline
\end{array}
$$

$\vdots$ and the symmetric cases . . .

where

$$\mathsf{Extend}\ z\ r\ \langle n; g\rangle \ = \ \langle n; \mathsf{acons}\ z\ r\ g\rangle$$

and $|\!\rangle\mathsf{Extend}\ z\ r\langle\!|$ is its failure-propagating image.

7.3.4 correctness of mgu

In the spirit of refinement, let us now reduce correctness of the unification algorithm to correctness of FlexFlex and FlexRigid. We have not yet defined the latter, but we can motivate the definition by seeing where we get stuck.

Here is the specification of mgu in the form of an inversion principle mguInv:

$$
\begin{array}{l}
\textsf{mguInv} \\[1ex]
m : \mathbb{N} \\
s, t : \textsf{tree}\; m \quad \Phi : \textsf{maybe}\;(\textsf{from}\; m) \;\rightarrow\; \mathit{Type} \\[2ex]
\dfrac{\dfrac{\textsf{NoUnifier}\; s\; t}{\Phi\;(\textsf{no}\;(\textsf{from}\; m))} \qquad \dfrac{\begin{array}{l} f : \textsf{alist}\; m\; n \\ \textsf{Maximal}\;(\textsf{Unifies}\; s\; t)\;(f \blacktriangleleft) \end{array}}{\Phi\;(\textsf{yes}\;\langle n; f\rangle)}}{\Phi\;(\textsf{mgu}\; s\; t)}
\end{array}
$$

where

$$\textsf{NoUnifier}\; s\; t \;\Longrightarrow\; \forall n.\, \forall f{:}\, m \searrow n.\;\; f\langle\!|\, s \not\simeq f\langle\!|\, t$$

We can prove mguInv from an inversion principle bmguInv for bmgu:

$$
\begin{array}{l}
\textsf{bmguInv} \\[1ex]
m : \mathbb{N} \\
s, t : \textsf{tree}\; m \\
\Phi : \textsf{from}\; m \;\rightarrow\; \textsf{maybe}\;(\textsf{from}\; m) \;\rightarrow\; \mathit{Type} \\[2ex]
\dfrac{\dfrac{\begin{array}{l} g : \textsf{alist}\; m\; n \\ \textsf{NoUnifier}\;\; g \blacktriangleleft\langle\!|\, s \;\; g \blacktriangleleft\langle\!|\, t \end{array}}{\Phi\;\langle n; f\rangle\;\;(\textsf{no}\;(\textsf{from}\; m))} \qquad \dfrac{\begin{array}{l} f : \textsf{alist}\; m\; n \\ g : \textsf{alist}\; n\; n' \\ \textsf{Maximal}\;(\textsf{Bound}\;(f \blacktriangleleft)\;(\textsf{Unifies}\; s\; t))\;(g \blacktriangleleft) \end{array}}{\Phi\;\langle n; f\rangle\;\;(\textsf{yes}\;\langle n'; g \ddagger f\rangle)}}{\forall f{:}\,\textsf{from}\; m.\; \Phi\; f\;(\textsf{bmgu}\; s\; t\; f)}
\end{array}
$$

The proof simply expands mgu in terms of a call to bmgu which is then inverted. This leaves bmguInv subgoals with g instantiated to anil. Recall that anil $\blacktriangleleft$ is just ι. The properties of AList and SubstK then reduce these subgoals to those of mguInv.

The interesting work is proving bmguInv. Of course, like all our other proofs of non-computational rules by recursion induction, the proof is by recursion induction on bmgu keeping Φ universally quantified. In the subgoals involving variables, let us also follow the program and do case analysis on the accumulated substitution. We may classify the subgoals as follows

- rigid-rigid off-diagonal (also known as 'conflict')

Here we are trying to unify leaf with $\mathsf{fork}\ s\ t$. bmgu returns no, so we must apply the no case. This leaves us proving

$$\begin{array}{l}
?\mathit{leafFork}: \forall n \quad : \mathbb{N} \\
\qquad\qquad\quad \forall g \quad : \mathsf{alist}\ m\ n \\
\qquad\qquad\quad \forall \mathit{bad}: g \blacktriangleleft\!\langle\!|\ \mathsf{leaf} \simeq g \blacktriangleleft\!\langle\!|\ (\mathsf{fork}\ s\ t) \\
\qquad\qquad\quad\ \ \bot
\end{array}$$

Fortunately, reducing $\cdot\langle\!|$ pushes the substitution under the constructors, leaving us with a hypothesis

$$\mathit{bad} : \mathsf{leaf} \simeq \mathsf{fork}\ \ g \blacktriangleleft\!\langle\!|\ s\ \ g \blacktriangleleft\!\langle\!|\ t$$

The goal can thus be proven by the unification tactic from chapter 5.

- rigid-rigid on-diagonal (also known as 'injectivity')

  Correctness for leaf with leaf is very easy.

  As for $(\mathsf{fork}\ s_1\ t_1)$ and $(\mathsf{fork}\ s_2\ t_2)$, the computation reduces the goal conclusion to

$$\Phi\ f\,((\lambda f.\ \mathsf{bmgu}_m\ t_1\ t_2\ f)\langle\!|\ (\mathsf{bmgu}_m\ s_1\ s_2\ f))$$

  If my propaganda has worked, you should now expect me to use the inductive hypotheses to invert the recursive calls. I shall not disappoint you. This leaves us with four subgoals.

  In three of them, the unification has failed somewhere and the ultimate value is no—the inversion will give us a proof of $\mathsf{NoUnifier}\ s_i\ t_i$ for some i. We may use this to show that the original forks have no unifier.

  Otherwise, we have substitutions h and g, together with proofs of

$$\begin{array}{l}
\lambda \mathit{gMax}: \mathsf{Maximal}\ (\mathsf{Bound}\ (f \blacktriangleleft)\ (\mathsf{Unifies}\ s_1\ s_2))\ (g \blacktriangleleft) \\
\lambda \mathit{hMax}: \mathsf{Maximal}\ (\mathsf{Bound}\ ((g \ddagger f) \blacktriangleleft)\ (\mathsf{Unifies}\ t_1\ t_2))\ (h \blacktriangleleft)
\end{array}$$

  Unification has returned $(h \ddagger g) \ddagger f$ and applying the yes case leaves us trying to prove.

$$?\mathit{goal}: \mathsf{Maximal}\ (\mathsf{Bound}\ (f \blacktriangleleft)\ (\mathsf{Unifies}\ (\mathsf{fork}\ s_1\ t_1)\ (\mathsf{fork}\ s_2\ t_2)))\ ((h \ddagger g) \blacktriangleleft)$$

  By $\mathsf{EquivMax}$ with Unload, followed by structural decomposition and $\mathsf{AList.RespC}$ this becomes

$$?goal\text{: Maximal (AND (Unifies } f \blacktriangleleft\llparenthesis\, s_1\;\; f \blacktriangleleft\llparenthesis\, s_1)$$
$$\text{(Unifies } f \blacktriangleleft\llparenthesis\, t_1\;\; f \blacktriangleleft\llparenthesis\, t_2)) \; ((h \blacktriangleleft) \diamond (g \blacktriangleleft))$$

Applying Optimist, we acquire two subgoals

$$?goal_g\text{: Maximal (Unifies } f \blacktriangleleft\llparenthesis\, s_1\;\; f \blacktriangleleft\llparenthesis\, s_1) \; (g \blacktriangleleft)$$
$$?goal_h\text{: Maximal (Bound } (g \blacktriangleleft) \text{ (Unifies } f \blacktriangleleft\llparenthesis\, t_1\;\; f \blacktriangleleft\llparenthesis\, t_2)) \; (h \blacktriangleleft)$$

In the former, Unload backwards lets us move f out as a bound, giving us a goal which follows immediately from *gMax*. In the latter, we may shuffle the bound inside, then apply composition laws to get

$$?goal_h\text{: Maximal (Unifies } (g \ddagger f) \blacktriangleleft\llparenthesis\, t_1\;\; (g \ddagger f) \blacktriangleleft\llparenthesis\, t_2)) \; (h \blacktriangleleft)$$

Now, pulling out the composition as a bound, we reduce the goal to *hMax*.

- flexible cases with acons $z\; r\; g$

  All of these work the same way. We have some

  $$\text{bmgu}\; s\; t\;\; \langle n; \text{acons}\; z\; r\; g\rangle$$

  where either s or t is a variable. This reduces to

  $$\rrparenthesis \text{Extend}\; z\; r \llparenthesis\;\; (\text{bmgu}\; [z \mapsto r]s\; [z \mapsto r]t\; g)$$

  Inverting the recursive call with the inductive hypothesis, we find one of two things

  – NoUnifier $g \blacktriangleleft\llparenthesis\, ([z \mapsto r]\llparenthesis\, s)\;\; g \blacktriangleleft\llparenthesis\, ([z \mapsto r]\llparenthesis\, t)$

    and we must prove

    $$\text{NoUnifier } (\text{acons}\; z\; r\; g) \blacktriangleleft\llparenthesis\, s\;\; (\text{acons}\; z\; r\; g) \blacktriangleleft\llparenthesis\, t$$

    But (acons $z\; r\; g$) $\blacktriangleleft$ is just $(g \blacktriangleleft) \diamond [z \mapsto r]$, so it is just a question of pushing $\cdot\llparenthesis$ through the composition.

  – h such that Maximal (Bound $(g \blacktriangleleft)$ (Unifies $[z \mapsto r]s\; [z \mapsto r]t$)) $(h \blacktriangleleft)$

    and we must prove

    $$\text{Maximal (Bound ((acons } z\; r\; g) \blacktriangleleft) \text{ (Unifies } s\; t)) \; (h \blacktriangleleft)$$

    The proof is easy bound shuffling and composition hacking.

- flex-flex base case

  The computation of

  $$\mathsf{bmgu}\ (\mathsf{var}\ x)\ (\mathsf{var}\ y)\ \mathsf{anil}$$

  has reduced to

  $$\mathsf{yes}\ (\mathsf{FlexFlex}\ x\ y)$$

  We may safely presume a yes answer, because we are in either the 'identity' or the 'coalescence' situation, according as x equals y or not. Hence, we must choose the yes case in the proof, leaving us with the obligation

  $$?goal\colon \mathsf{Maximal}\ (\mathsf{Bound}\ \iota\ (\mathsf{Unifies}\ (\mathsf{var}\ x)\ (\mathsf{var}\ y)))\ ((\mathsf{FlexFlex}\ x\ y) \blacktriangleleft)$$

  We may easily remove the trivial bound, yielding

  $$?goal\colon \mathsf{Maximal}\ (\mathsf{Unifies}\ (\mathsf{var}\ x)\ (\mathsf{var}\ y))\ ((\mathsf{FlexFlex}\ x\ y) \blacktriangleleft)$$

  Since we have not yet implemented FlexFlex, we can go no further with the proof. Let us export this goal as the specification of FlexFlex.

- flex-rigid base cases

  For these five subgoals, we are trying to unify var x with some t which is not a variable. We may collect them all together in the following rule, expressing the latter as a side condition:

$$\begin{array}{l}
m : \mathbb{N} \\
x : \mathsf{fin}\ \mathsf{s}m \\
t : \mathsf{tree}\ \mathsf{s}m \\
notVar : \forall y{:}\mathsf{fin}\ \mathsf{s}m.\ t \not\simeq \mathsf{var}\ y \\
\Phi : \mathsf{from}\ \mathsf{s}m \ \to\ \mathsf{maybe}\ (\mathsf{from}\ \mathsf{s}m)\ \to\ \mathit{Type}
\end{array}$$

$$\dfrac{\dfrac{\begin{array}{l} g : \mathsf{alist}\ m\ n \\ \mathsf{NoUnifier}\ g\blacktriangleleft x\ \ g\blacktriangleleft\!\langle\!|\ t \end{array}}{\Phi\ \langle n; g\rangle\ (\mathsf{no}\ (\mathsf{from}\ m))} \qquad \dfrac{\begin{array}{l} f : \mathsf{alist}\ m\ n \\ g : \mathsf{alist}\ n\ n' \\ \mathsf{Maximal} \\ \quad (\mathsf{Bound}\ (f\blacktriangleleft)\ (\mathsf{Unifies}\ (\mathsf{var}\ x)\ t)) \\ \quad (g\blacktriangleleft) \end{array}}{\Phi\ \langle n; f\rangle\ (\mathsf{yes}\ \langle n'; g\ddagger f\rangle)}}{\Phi\ \mathsf{anil}\ (\mathsf{FlexRigid}\ x\ t)}$$

We could regard this as an inversion rule specification for FlexRigid, but it is still a little too general. For example, the hypotheses of the rule each have arbitrary accumulators, but we know the accumulator is anil. Once we have made the accumulator anil everywhere, we no longer need to let it vary in the scheme. Let us tidy up a little.

$$\boxed{\begin{array}{l}
m : \mathbb{N} \\
x : \mathsf{fin}\ \mathsf{s}m \\
t : \mathsf{tree}\ \mathsf{s}m \\
\mathit{notVar} : \forall y{:}\mathsf{fin}\ \mathsf{s}m.\ t \not\simeq \mathsf{var}\ y \\
\Phi : \mathsf{maybe}\ (\mathsf{from}\ \mathsf{s}m) \ \rightarrow\ \mathit{Type} \\
\\
\dfrac{\dfrac{\mathsf{NoUnifier}\ (\mathsf{var}\ x)\ t}{\Phi\ (\mathsf{no}\ (\mathsf{from}\ m))} \qquad \dfrac{\begin{array}{l} g : \mathsf{alist}\ n\ n \\ \mathsf{Maximal}\ (\mathsf{Unifies}\ (\mathsf{var}\ x)\ t)\ (g \blacktriangleleft) \end{array}}{\Phi\ (\mathsf{yes}\ \langle n; g\rangle)}}{\Phi\ (\mathsf{FlexRigid}\ x\ t)}
\end{array}}$$

The tidy version proves the untidy version because the tidy hypotheses are special cases of the untidy ones, modulo some equational reasoning. Let us take this as the specification of FlexRigid.

We have proven correctness of unification, contingent on correct implementation of FlexFlex and FlexRigid. You may have noticed that we did not have to unwrap any of the Maximals in the above proof—we merely showed that the most general unifiers computed in the base cases were correctly propagated. It is in FlexFlex and FlexRigid that we create the substitutions and where we shall have to do real work proving maximality. In order to achieve this, we must come to an understanding of variable occurrence.

But even now, we have seen enough to know that our unification algorithm is terminating of its own accord.

7.3.5 what substitution tells us about the occurs check

In conventional presentations of unification, the occurs check is a boolean decision, and its role in ensuring termination is external to the program. For us, though, the situation is somewhat different—what is to happen if there is no occurrence of $(\mathsf{var}\ x)$ in some rigid t with which it is to be unified? We do not just substitute t itself for x.

We must make manifest in the program the elimination of x by computing the image of t in the syntax with one fewer variable—t' such that

$$\rrparenthesis\mathsf{thin}\, x\llparenthesis\; t' \simeq t$$

If we can find such a t', then

$$[x \mapsto t']$$

is a most general unifier for $(\mathsf{var}\, x)$ and t. Let us prove this lemma, as we shall need it several times.

$$\begin{array}{l} ?\mathsf{Knockout}\colon \forall m\colon \mathbb{N} \\ \qquad\qquad \forall x : \mathsf{fin}\; \mathsf{s}m \\ \qquad\qquad \forall t' : \mathsf{tree}\; m \\ \qquad\qquad\quad \mathsf{Maximal}\; (\mathsf{Unifies}\; (\mathsf{var}\, x)\; (\rrparenthesis\mathsf{thin}\, x\llparenthesis\, t'))\; [x \mapsto t'] \end{array}$$

Now, at last, we must do some real work. Introducing the $\mathsf{Maximal}$ record:

$$\begin{array}{l} ?\mathit{holds}\ : [x \mapsto t']\llparenthesis\, (\mathsf{var}\, x) \simeq [x \mapsto t']\llparenthesis\, (\rrparenthesis\mathsf{thin}\, x\llparenthesis\, t') \\ ?\mathit{factors}\colon \forall n \quad : \mathbb{N} \\ \qquad\quad \forall f \quad : \mathsf{s}m \searrow n \\ \qquad\quad \forall \mathit{hyp}\colon f\llparenthesis\, (\mathsf{var}\, x) \simeq f\llparenthesis\, (\rrparenthesis\mathsf{thin}\, x\llparenthesis\, t') \\ \qquad\quad \Sigma g \quad : m \rightarrowtail n \\ \qquad\qquad f \approx g \diamond [x \mapsto t'] \end{array}$$

Taking *holds* first, notice that the left-hand side is just $[x \mapsto t']x$, which we can rewrite by the 'introduction rules' to t'. Observe that the right-hand side is a composition of substitutions. After a little monadic tinkering, we obtain

$$?\mathit{holds}'\colon \rrparenthesis\iota\llparenthesis\, t' \simeq ([x \mapsto t'] \diamond \rrparenthesis\mathsf{thin}\, x)\llparenthesis\, t'$$

This says that two substitutions have the same behaviour at an arbitrary tree t'. By BackEq, it is enough to prove that they behave the same at variables.

$$\begin{array}{l} ?\mathit{holds}'\colon \forall y\colon \mathsf{fin}\; m \\ \qquad\quad \rrparenthesis\iota\, y \simeq ([x \mapsto t'] \diamond \rrparenthesis\mathsf{thin}\, x)\; y \end{array}$$

Reducing, we obtain

$$
\begin{array}{l}
?\mathit{holds}' : \forall y \colon \mathsf{fin}\ m \\
\qquad\qquad \mathsf{var}\ y \simeq [x \mapsto t']\ (\mathsf{thin}\ x\ y)
\end{array}
$$

Again, this follows by the established extensional behaviour of $[\cdot \mapsto \cdot]$.

We have found a unifier—let us now show that any other unifier factors through it. Introducing the assumptions and the pair:

$$
\begin{array}{ll}
\lambda n & : \mathbb{N} \\
\lambda \mathsf{f} & : \mathsf{s}m \searrow n \\
\lambda \mathit{hyp} & : \mathsf{f}\langle\!|\ (\mathsf{var}\ x) \simeq \mathsf{f}\langle\!|\ (|\!\rangle \mathsf{thin}\ x \langle\!|\ t') \\
?\mathsf{g} & : m \searrow n \\
?\mathit{fac} & : \mathsf{f} \approx \mathsf{g} \diamond [x \mapsto t']
\end{array}
$$

Let us try to prove *fac* first, hoping to shed some light on g. This goal also comes down to checking that the two substitutions agree at all variables:

$$
\begin{array}{l}
?\mathit{fac}' : \forall \boxed{y} \colon \mathsf{fin}\ \mathsf{s}m \\
\qquad \mathsf{f}\, y \simeq \mathsf{g}\langle\!|\ \boxed{[x \mapsto t']\ y}
\end{array}
$$

Predictably, the next step is to invert the blocked computation with $\mathsf{knockoutInv}$:

$$
\begin{array}{l}
?\mathit{fac}_x : \mathsf{f}\, x \simeq \mathsf{g}\langle\!|\ t' \\
?\mathit{fac}_y : \forall y \colon \mathsf{fin}\ m \\
\qquad \mathsf{f}\,(\mathsf{thin}\ x\ y) \simeq \mathsf{g}\, y
\end{array}
$$

The latter subgoal gives us a big clue. We can prove it by taking

$$\mathsf{g} \;=\; \lambda y \colon \mathsf{fin}\ m.\ \mathsf{f}\,(\mathsf{thin}\ x\ y)$$

We must now prove fac_x. A little monadic massage shows g is extensionally the same as the composition

$$\mathsf{f} \diamond |\!\rangle \mathsf{thin}\ x$$

Making the replacement,

$$?fac'_x \colon f\,x \simeq (f \diamond \rangle\!\!\mid \mathsf{thin}\ x) \langle\!\!\mid t'$$

Unwinding the composition reduces this goal to *hyp*.

This is progress indeed, for all the nontrivial substitutions generated by FlexFlex or FlexRigid will be most general unifiers by this lemma. Indeed, we are now in a position to write FlexFlex:

$$\mathsf{FlexFlex} : \forall m.\ \forall x, y \colon \mathsf{fin}\ \mathsf{s}m.\ \mathsf{from}\ \mathsf{s}m$$

$$\mathsf{FlexFlex}\ x\ y \ = \ \begin{array}{|c|l|} \hline \mathsf{thick}\ x\ y & \\ \hline \mathsf{no}\ (\mathsf{fin}\ m) & \langle \mathsf{s}m; \mathsf{anil}\rangle \\ \mathsf{yes}\ y' & \langle m; \mathsf{acons}\ x\ (\mathsf{var}\ y')\ \mathsf{anil}\rangle \\ \hline \end{array}$$

Recall that to establish correctness, we must prove

$$\begin{array}{ll} ?\mathsf{FlexFlex}_{max} \colon & \forall m \ \colon \mathbb{N} \\ & \quad \forall x, y \colon \mathsf{fin}\ \mathsf{s}m \\ & \quad\ \ \mathsf{Maximal}\ (\mathsf{Unifies}\ (\mathsf{var}\ x)\ (\mathsf{var}\ y))\ ((\mathsf{FlexFlex}\ x\ y)\ \blacktriangleleft) \end{array}$$

Since FlexFlex is defined with thick, it is verified by thickInv, leaving two cases

$$\begin{array}{ll} ?\mathsf{FlexFlex}_x \colon & \forall m \colon \mathbb{N} \\ & \forall x : \mathsf{fin}\ \mathsf{s}m \\ & \ \ \mathsf{Maximal}\ (\mathsf{Unifies}\ (\mathsf{var}\ x)\ (\mathsf{var}\ x))\ (\mathsf{anil}\ \blacktriangleleft) \\ ?\mathsf{FlexFlex}_y \colon & \forall m \colon \mathbb{N} \\ & \forall x : \mathsf{fin}\ \mathsf{s}m \\ & \forall y : \mathsf{fin}\ m \\ & \ \ \mathsf{Maximal}\ (\mathsf{Unifies}\ (\mathsf{var}\ x)\ (\mathsf{var}\ (\mathsf{thin}\ x\ y)))\ ((\mathsf{acons}\ x\ (\mathsf{var}\ y)\ \mathsf{anil})\ \blacktriangleleft) \end{array}$$

For the former, recall that anil ◀ is the identity substitution—this clearly unifies two equal variables, and equally clearly, every other unifier factors through it. For the latter, interpreting the association list and tidying, we get

$$\begin{array}{ll} ?\mathsf{FlexFlex}'_y \colon & \forall m \colon \mathbb{N} \\ & \forall x : \mathsf{fin}\ \mathsf{s}m \\ & \forall y : \mathsf{fin}\ m \\ & \ \ \mathsf{Maximal}\ (\mathsf{Unifies}\ (\mathsf{var}\ x)\ (\rangle\!\!\mid \mathsf{thin}\ x \langle\!\!\mid\ (\mathsf{var}\ y)))\ [x \mapsto \mathsf{var}\ y] \end{array}$$

This follows from Knockout.

The role thick plays in FlexFlex is to attempt to compute the image of y in the variable set with x removed. If this succeeds, we manufacture the corresponding knockout. If it fails, that is because y is x and the identity substitution will do.

The analogous role in FlexRigid is played by the occurs check, seen as an attempt to compute the appropriate 'thickened' tree for use in a knockout—this will fail exactly in the case of an offending occurrence. Correspondingly, the occurs check is no longer a boolean decision—it provides us with the witness which explains why it is safe to substitute. More sense has appeared in the program! The type of the occur check is

$$\mathsf{check} : \forall m.\ \forall x{:}\mathsf{fin}\ \mathsf{s}m.\ \forall t{:}\mathsf{tree}\ \mathsf{s}m.\ \mathsf{maybe}\ (\mathsf{tree}\ m)$$

Its inversion rule should be something like:

$$\boxed{\begin{array}{l}
m : \mathbb{N} \\
x : \mathsf{fin}\ \mathsf{s}m \\
\Phi : \mathsf{tree}\ \mathsf{s}m \ \to\ \mathsf{maybe}\ (\mathsf{tree}\ m)\ \to\ \mathit{Type} \\
\\
\dfrac{\Phi\ (\rangle\!|\mathsf{thin}\ x|\!\langle\ t)\ (\mathsf{yes}\ t) \qquad \begin{array}{c}\mathsf{Occurs}\ x\ t \\ \cdots\cdots\cdots\cdots \\ \Phi\ t\ (\mathsf{no}\ (\mathsf{tree}\ m))\end{array}}{\forall t{:}\mathsf{tree}\ \mathsf{s}m.\ \Phi\ t\ (\mathsf{check}\ x\ t)}
\end{array}}$$

where Occurs is some useful means of characterising when x occurs in t.

In other words, check x is the partial inverse of $\rangle\!|\mathsf{thin}\ x|\!\langle$. Hence we will implement check x by pushing thick x through trees, with any no at a variable causing a no overall. However, before we can really work with the occurs check, we must formalise the notion of occurrence.

7.3.6 positions

The idea of pattern matching is to explain decomposition by inverting construction, and I was exposed to it at such an early age that it simply refuses to wear off. We have already seen $\mathbb{N}$Eq in terms of duplication and thick in terms of thin. Since searching for an occurrence is a kind of decomposition, I cannot help asking what the corresponding construction might be.

Let us therefore identify the operation which *makes* an occurrence—the operation which puts something at a given position. In order to do this, we shall need to represent positions within a tree.

Every datatype T has an allied datatype of positions or 'one-hole contexts' within elements of T, together with an operation which puts a T in the hole. Huet gives a beautiful construction of 'zipper' types which code up one-hole contexts as paths from the hole back to the root of the term, recording the contents of other side-branches on the way. We may equivalently, and slightly more conveniently for our purposes, reverse the direction and code up paths from the root to the hole. Let us therefore define the parameterised datatype $\mathsf{pos}\, n$ of positions within n :

- formation rule $\dfrac{n : \mathbb{N}}{\mathsf{pos}\, n : \mathbb{N}}$

- constructors $\dfrac{}{\mathsf{here}_n : \mathsf{pos}\, n}$ $\dfrac{\mathit{there} : \mathsf{pos}\, n \quad t : \mathsf{tree}\, n}{\mathsf{left}\, \mathit{there}\, t : \mathsf{pos}\, n}$

  $\dfrac{s : \mathsf{tree}\, n \quad \mathit{there} : \mathsf{pos}\, n}{\mathsf{right}\, s\, \mathit{there} : \mathsf{pos}\, n}$

The constructors may be interpreted as directions for finding the position from the root, respectively 'stop here', 'go left' and 'go right'. Consequently, the function which puts a term at a position is

$$\cdot\cdot\,\mathsf{goes} : \forall n.\ \forall \mathit{there}{:}\mathsf{pos}\, n.\ \forall \mathit{it}{:}\mathsf{tree}\, n.\ \mathsf{tree}\, n$$

Allow me to break with convention and write goes postfix—its definition is:

$$\begin{array}{rlll}
\mathsf{here} & \mathit{it} & \mathsf{goes} & = & \mathit{it} \\
(\mathsf{left}\, \mathit{there}\, t) & \mathit{it} & \mathsf{goes} & = & \mathsf{fork}\, (\mathit{there}\, \mathit{it}\, \mathsf{goes})\, t \\
(\mathsf{right}\, s\, \mathit{there}) & \mathit{it} & \mathsf{goes} & = & \mathsf{fork}\, s\, (\mathit{there}\, \mathit{it}\, \mathsf{goes})
\end{array}$$

In particular, we may now describe a term containing $\mathsf{var}\, x$ as

$$\mathit{where}\, (\mathsf{var}\, x)\, \mathsf{goes}$$

In order to reason about positions, it will be useful to have some other apparatus. Indeed, we may consider goes to interpret $\mathsf{pos}\, n$ as the arrows of a category with one object interpreted as $\mathsf{tree}\, n$. here is the identity. Let us therefore define the composition, which, in the spirit of the piece, I shall write as an infix operator called then:

$$
\begin{array}{l}
\mathsf{then} : \forall n.\ \forall \mathit{there}, \mathit{where} : \mathsf{pos}\ n.\ \mathsf{pos}\ n \\
\begin{array}{rccl}
\mathsf{here} & \mathsf{then} & \mathit{where} = & \mathit{where} \\
(\mathsf{left}\ \mathit{there}\ t) & \mathsf{then} & \mathit{where} = & \mathsf{left}\ (\mathit{there}\ \mathsf{then}\ \mathit{where})\ t \\
(\mathsf{right}\ s\ \mathit{there}) & \mathsf{then} & \mathit{where} = & \mathsf{right}\ s\ (\mathit{there}\ \mathsf{then}\ \mathit{where})
\end{array}
\end{array}
$$

The definition of goes ensures the correct interpretation of here. An easy recursion induction proves the correct interpretation of then:

$$(\mathit{where}\ \mathsf{then}\ \mathit{there})\ \mathit{it}\ \mathsf{goes} \simeq \mathit{where}\ (\mathit{there}\ \mathit{it}\ \mathsf{goes})\ \mathsf{goes}$$

By the way, datatypes (eg list, $\mathbb{N}$) with a single and constant base constructor (eg nil, 0) and linear step constructors (eg cons, s) are isomorphic to their own position types. The goes and then operations are the same (eg append, plus). This may account for their peculiarly regular behaviour.

Returning to our tree syntax, we shall also need to push substitutions through positions. Overloading slightly:

$$
\begin{array}{l}
\cdot\langle\!| : \forall m, n.\ \forall f : m \searrow n.\ \mathsf{pos}\ m \ \rightarrow\ \mathsf{pos}\ n \\
\begin{array}{ll}
f\langle\!|\ \mathsf{here}_m & = \quad \mathsf{here}_n \\
f\langle\!|\ (\mathsf{left}\ \mathit{there}\ t) & \mathsf{left}\ (f\langle\!|\ \mathit{there})\ (f\langle\!|\ t) \\
f\langle\!|\ (\mathsf{right}\ s\ \mathit{there}) & \mathsf{right}\ (f\langle\!|\ s)\ (f\langle\!|\ \mathit{there})
\end{array}
\end{array}
$$

Recursion induction on this operation gives us a proof of Coherence:

$$f\langle\!|\ (\mathit{there}\ \mathit{it}\ \mathsf{goes}) \simeq (f\langle\!|\ \mathit{there})\ (f\langle\!|\ \mathit{it})\ \mathsf{goes}$$

Now, in order to prove that occurs check failure causes unification failure, we shall need to show that the only position at which we may find a term inside itself is here:

$$
\begin{array}{ll}
?\mathsf{NoCycle}: & \forall n \quad\ : \mathbb{N} \\
& \forall \mathit{it} \quad : \mathsf{tree}\ n \\
& \forall \mathit{there} : \mathsf{pos}\ n \\
& \forall \mathit{hyp}\ \ : \mathit{it} \simeq \mathit{there}\ \mathit{it}\ \mathsf{goes} \\
& \quad \mathit{there} \simeq \mathsf{here}
\end{array}
$$

We have seen a similar theorem before. The proof goes by induction on *it*, then case analysis on *there*. A lot of impossible cases are removed by unification—there are obviously no left or right positions within var x or leaf. The only real work to be done is when *there* is not here and *it* is a fork. There are two such cases, one much like the other, so I shall just give the proof for fork and left:

$$
\begin{array}{l}
?\mathsf{NoCycle}_{fl} : \forall n \quad : \mathbb{N} \\
\qquad\qquad \forall s, t, r : \mathsf{tree}\; n \\
\qquad\qquad \forall sHyp : \forall there : \mathsf{pos}\; n \\
\qquad\qquad\qquad\qquad \forall hyp \;\; : s \simeq there\; s\; \mathsf{goes} \\
\qquad\qquad\qquad\qquad\quad there \simeq \mathsf{here} \\
\qquad\qquad \forall tHyp : \\
\qquad\qquad \forall there : \mathsf{pos}\; n \\
\qquad\qquad \forall hyp \;\; : \mathsf{fork}\; s\; t \simeq (\mathsf{left}\; there\; r)\; (\mathsf{fork}\; s\; t)\; \mathsf{goes} \\
\qquad\qquad\quad there \simeq \mathsf{here}
\end{array}
$$

The trick is to rotate the cycle. Reducing goes, *hyp* becomes

$$\mathsf{fork}\; s\; t \simeq \mathsf{fork}\; (there\; (\mathsf{fork}\; s\; t)\; \mathsf{goes})\; r$$

Unification identifies t and r and tells us that

$$s \simeq there\; (\mathsf{fork}\; s\; t)\; \mathsf{goes}$$

Now, if we are careful, we can turn this into a cycle in s and apply the relevant inductive hypothesis. Our categorical tools allow us to rewrite the above equation to

$$s \simeq (there\; \mathsf{then}\; (\mathsf{left}\; \mathsf{here}\; t))\; s\; \mathsf{goes}$$

The inductive hypothesis *sHyp* now tells us that

$$(there\; \mathsf{then}\; (\mathsf{left}\; \mathsf{here}\; t)) \simeq \mathsf{here}$$

This is manifest nonsense, but we need to make a constructor appear at the head on the left-hand side to reveal the conflict. That is to say, a further case analysis on *there*, accompanied by the unification tactic, completes the proof.

We are now in a position to fill in the last component of the unification algorithm.

7.3.7 check and FlexRigid

As suggested earlier, the check function pushes thick through a tree.

$$
\begin{array}{llll}
\mathsf{check} & x & (\mathsf{var}\; y) & = \; \rangle\!\!\rangle \mathsf{var}_n \langle\!\!\langle\; (\mathsf{thick}\; x\; y) \\
\mathsf{check} & x & \mathsf{leaf}_{sn} & = \; \mathsf{leaf}_n \\
\mathsf{check} & x & (\mathsf{fork}\; s\; t) & =
\end{array}
$$

check x s	check x t	
yes s'	yes t'	yes (fork s' t')
yes s'	no (tree n)	no (tree n)
no (tree n)	t'	no (tree n)

Now that we know how to talk about positions, we can give this function a better inversion principle, checkInv:

$$
\boxed{
\begin{array}{l}
\mathsf{checkInv} \\[1ex]
m : \mathbb{N} \\
x : \mathsf{fin}\ \mathsf{s}m \\
\Phi : \mathsf{tree}\ \mathsf{s}m \ \to\ \mathsf{maybe}\ (\mathsf{tree}\ m)\ \to\ \mathit{Type} \\[1ex]
\qquad \dfrac{\Phi\ ((\!|\mathsf{thin}\ x|\!)\ t)\ (\mathsf{yes}\ t) \qquad \Phi\ (\mathit{where}\ (\mathsf{var}\ x)\ \mathsf{goes})\ (\mathsf{no}\ (\mathsf{tree}\ m))}{\forall t : \mathsf{tree}\ \mathsf{s}m.\ \Phi\ t\ (\mathsf{check}\ x\ t)}
\end{array}
}
$$

The proof, which I omit, is by recursion induction and inversion of the blocked computations.

Now let us define FlexRigid:

$$
\mathsf{FlexRigid}\ x\ t \ = \
\begin{array}{|c|l|}
\hline
\mathsf{check}\ x\ t & \\
\hline
\mathsf{yes}\ t' & \mathsf{yes}\ \langle m; \mathsf{acons}\ x\ t'\ \mathsf{anil}\rangle \\
\mathsf{no}\ (\mathsf{tree}\ m) & \mathsf{no}\ (\mathsf{from}\ \mathsf{s}m) \\
\hline
\end{array}
$$

We must show that this function satisfies its specification:

$$
\begin{array}{lll}
?\mathsf{FlexRigid}_s : & \forall m & : \mathbb{N} \\
& \forall x & : \mathsf{fin}\ \mathsf{s}m \\
& \forall t & : \mathsf{tree}\ \mathsf{s}m \\
& \forall \mathit{notVar} & : \forall y : \mathsf{fin}\ \mathsf{s}m \\
& & \quad t \not\simeq \mathsf{var}\ y \\
& \forall \Phi & : \mathsf{maybe}\ (\mathsf{from}\ \mathsf{s}m)\ \to\ \mathit{Type} \\
& \forall \phi_{\mathrm{n}} & : \forall \mathit{occ} : \mathsf{NoUnifier}\ (\mathsf{var}\ x)\ t \\
& & \quad \Phi\ (\mathsf{no}\ (\mathsf{from}\ \mathsf{s}m)) \\
& \forall \phi_{\mathrm{y}} & : \forall n \quad\ \ : \mathbb{N} \\
& & \ \ \ \forall f \quad\ \ : \mathsf{alist}\ m\ n \\
& & \ \ \ \forall \mathit{fMax} : \mathsf{Maximal}\ (\mathsf{Unifies}\ (\mathsf{var}\ x)\ t)\ (f \blacktriangleleft) \\
& & \quad \Phi\ \mathsf{yes}\ \langle n; f\rangle \\
& \multicolumn{2}{l}{\ \ \Phi\ (\mathsf{FlexRigid}\ x\ t)}
\end{array}
$$

This we prove by expanding FlexRigid and inverting check x t, leaving two cases. The first is

$$
\begin{array}{l}
?\mathsf{FlexRigid}_y : \\
\quad \forall t \; : \mathsf{tree}\; m \\
\quad \forall \Phi : \mathsf{maybe}\;(\mathsf{from}\;\mathsf{s}m) \;\to\; \mathit{Type} \\
\quad \forall \phi_{\mathrm{y}} : \forall n \quad : \mathbb{N} \\
\qquad\quad \forall f \quad : \mathsf{alist}\; m\; n \\
\qquad\quad \forall \mathit{fMax} : \mathsf{Maximal}\;(\mathsf{Unifies}\;(\mathsf{var}\;x)\;\rangle\!\!\rangle \mathsf{thin}\;x \langle\!\!\langle\, t)\;(f \blacktriangleleft) \\
\qquad\qquad \Phi\;\mathsf{yes}\;\langle n; f \rangle \\
\quad \Phi\;(\mathsf{yes}\;\langle m; \mathsf{acons}\;x\;t\;\mathsf{anil}\rangle
\end{array}
$$

Introducing the hypotheses, refining by ϕ_{y} and unpacking the association list, we are left proving

$$
?\mathsf{FlexRigid}'_y : \mathsf{Maximal}\;(\mathsf{Unifies}\;(\mathsf{var}\;x)\;\rangle\!\!\rangle \mathsf{thin}\;x \langle\!\!\langle\, t)\;[x \mapsto t]
$$

This follows by the Knockout lemma.

Meanwhile, the other case of the inversion is

$$
\begin{array}{l}
?\mathsf{FlexRigid}_n : \\
\quad \forall \mathit{where} \; : \mathsf{pos}\;\mathsf{s}m \\
\quad \forall \mathit{notVar} : \forall y : \mathsf{fin}\;\mathsf{s}m \\
\qquad\qquad \mathit{where}\;(\mathsf{var}\;x)\;\mathsf{goes} \;\not\simeq\; \mathsf{var}\;y \\
\quad \forall \Phi \qquad : \mathsf{maybe}\;(\mathsf{from}\;\mathsf{s}m) \;\to\; \mathit{Type} \\
\quad \forall \phi_{\mathrm{n}} \qquad : \forall \mathit{occ} : \mathsf{NoUnifier}\;(\mathsf{var}\;x)\;(\mathit{where}\;(\mathsf{var}\;x)\;\mathsf{goes}) \\
\qquad\qquad \Phi\;(\mathsf{no}\;(\mathsf{from}\;\mathsf{s}m)) \\
\quad \Phi\;(\mathsf{no}\;(\mathsf{from}\;\mathsf{s}m))
\end{array}
$$

This time, introducing the hypotheses, refining by ϕ_{n} and expanding NoUnifier leaves

$$
\begin{array}{l}
?\mathsf{FlexRigid}'_n : \forall n \quad : \mathbb{N} \\
\qquad\qquad\quad \forall f \quad : \mathsf{s}m \searrow n \\
\qquad\qquad\quad \forall \mathit{bad} : f \langle\!\!\langle\, (\mathsf{var}\;x) \simeq f \langle\!\!\langle\, (\mathit{where}\;(\mathsf{var}\;x)\;\mathsf{goes}) \\
\qquad\qquad\qquad \bot
\end{array}
$$

By Coherence, we may push f through goes, telling us

$$f\,x \simeq (f\langle\!| where)\,(f\,x) \text{ goes}$$

NoCycle now tells us that

$$where \simeq \mathsf{here}$$

reducing *notVar* to

$$\forall y.\, \mathsf{var}\,x \not\simeq \mathsf{var}\,y$$

from which we may easily prove the goal.

7.3.8 comment

This verification of a unification is another in a long line of such developments. From Zohar Manna and Richard Waldinger's pioneering hand-synthesis [MW81], through Lawrence Paulson's machine verification in LCF [Pau85] to the more recent work in diverse proof systems [Coen92, Rou92, Jau97, Bove99], all have faced the same inherent problem of explaining a program which simply does not make the sense its maker intended.

Critical to the correctness of the unification algorithm is the relativisation of terms to their context of variables. Such relativised data structures occur naturally in dependent type systems. Unification has always been structurally recursive—it is just that the structure could not be made data until the right types came along. Now they have, and that is something to be pleased about, and to be vocal about.

There are three delicate aspects of unification which must be handled somehow in every treatment, and they are not entirely independent:

- the termination of the algorithm
- the propagation of a unifier computed for one part of a problem through the rest of the problem
- the failure of unification due to failure of the occur check

The termination issue has, over the years, been separated from partial correctness with increasing panache and aplomb, but the technique standard in the literature is well-founded recursion over an ad hoc ordering. Manna and Waldinger [MW81] are sensible enough to leave the choice of this ordering until they have extracted the conditions it must satisfy:

> 'We have deferred the choice of an ordering $\prec_{un}$ to satisfy the ordering conditions we have accumulated during the proof. The choice of this ordering is not so well-motivated formally as the other steps of this derivation.'

The necessary ordering combines lexicographically the size of the variable set and the structure of the problem—the different treatments manifest this in slightly different ways. Paulson [Pau85] points out that he works rather harder than he would like to, motivating the desire for 'an LCF package for well-founded induction' in order to emulate Manna and Waldinger's paper development more closely.

Implementations of what would otherwise be generally recursive programs in type theory necessarily involve computation over the proof of termination. Different strategies exist to minimise the impact of this unwelcome intrusion of proof into program. Joseph Rouyer [Rou92] manages to confine the logical component to the outermost well-founded recursion on the number of variables, the inner recursion on terms being purely structural.

Ana Bove moves the goalposts in a pleasingly systematic way [Bove99]. Her ALF program does its recursion over the proof of an accessibility relation constructed almost directly from the Haskell program she wishes to import—the arguments to the program become the indices of the relation. A single induction over this relation thus splits into cases corresponding to the left-hand sides of the original program, while the exposed sub-proofs give exactly the recursive calls. Of course, she still has to prove that all the elements are accessible by well-founded lexicographic induction, but by packaging this complicated induction into a single relation, she has not only supported the program but also effectively acquired ipso facto its recursion induction principle—useful for proving its properties.

Of course, my program does a similar lexicographic recursion, but it is internalised in the data structures. I avoid an appeal to well-founded recursion on $\leq$ for the number of variables by unloading the accumulated substitutions incrementally, which is not unreasonable as they are built incrementally, and which incidentally enables me to delay them until they become critical.

It might perhaps be interesting to consider how much more trouble it would be to use a normalised representation of substitution, applied all at once. However, normalising substitutions is, in any case, computationally quite expensive.

Manna and Waldinger work rather hard to synthesise the accumulation of a unifier across a list of subproblems. The idempotence of the unifier plays a pivotal role. Paulson's proof is apparently simpler, but he is unforthcoming about the 'occasional ugly steps'. Coen [Coen92] describes this problem as the only awkward aspect of partial correctness.

The 'optimistic' treatment of accumulators makes this problem rather easier to deal with—introducing the accumulator as an extra parameter effectively strengthens the inductive hypotheses for the subproblems in exactly the necessary manner. Armando, Smaill and Green's automated synthesis manages to profit from this without excessive prompting [ASG99]. Bove also exploits an accumulating parameter with the same benefit. As I have shown, it is a natural technique to employ when the order with respect to which we seek an optimum is induced by some notion of composition.

As for showing there is no unifier when the occur check fails, my treatment is morally the same as Manna and Waldinger's, packaged slightly more categorically. It is also slightly more concrete. The use of the datatype of positions and its attendant operations, together with thin, means that the inversion of the occur check instantiates the investigated term with patterns capturing the relevant information, rather than presenting it propositionally. However, the position datatype comes into play only in the proof, not in the program, so in this case, there is not much to choose between the two.

Nonetheless, we may one day want a unification algorithm which augments the failure response with diagnostic information, so that a PhD student desperate for cash can have an easier time finding the type errors in an undergraduate's ML program. At this point, a concrete representation of positions becomes a must. The type of check could just as easily have been

$$\mathsf{check} \;:\; \forall n.\, \mathsf{fin}\,\mathsf{s}n \;\rightarrow\; \mathsf{tree}\,\mathsf{s}n \;\rightarrow\; \mathsf{tree}\,n + \mathsf{pos}\,\mathsf{s}n$$

returning a witness in the case of failure. A treatment of positions is hardly wasted. Furthermore, as Huet points out [Hue97], there is no reason why the construction of position apparatus should not be automated for arbitrary datatypes.

Finally, I would like to comment on two 'packaging' aspects of the development of unification in this thesis. Firstly, the monadic treatment both of failure-propagation and of substitution itself seems to present the necessary equipment in a useful and orderly way.

Secondly, the use of inversion and recursion induction principles to capture the behaviour of components lent such a regularity and tangibility to the components of the correctness proof that I believe I have given substantial credence to the methodology of capturing ‘leverage’ in this way. Recall, for example, how the inversion of the occur check not only exposed the information pertinent to the two possibilities but performed the consequent rewriting, allowing still further progress by computation. Further, the whole effect was triggered by asking a single high-level question about a program component

> ‘what can have happened in that occur check?’

rather than a low-level question about data

> ‘what values can that $\mathsf{maybe}\ (\mathsf{tree}\ n)$ have?’

We have been able to stare at unification without going blind.

Chapter 8

Conclusion

What are the contributions of this thesis?

Firstly, and somewhat tangentially, it introduced OLEG, a type theory with holes which has two advantages:

- separation of partial constructions from the core computational terms in such a way that the partial constructions—where the holes live—behave well enough to have the **replacement** property
- internalisation of the account of the holes within the judgments of the theory, allowing the state of a theorem prover to be represented exactly by a valid context

Of course, relative to systems which explain holes with the aid of explicit substitution, it has the disadvantage of forbidding certain interactions between holes and computation. For the work presented here, this has not troubled us at all. Admittedly, this has not involved the kind of higher-order problem for which the banned interactions might help.

On the other hand, the resemblance to Miller's 'mixed prefix' [Mil92] treatment is strong enough to suggest that his brand of higher-order unification might be feasible. He too handles the interaction between holes and computation by 'raising' the holes to the functionality required to ensure that the computation happens entirely within their scope. Nonetheless, deeper exploration is needed before we can say that OLEG is a suitable basis for sophisticated theorem proving. It is, however, an effective basis for the tactics and mechanised constructions on which the main work of the thesis depends.

That work was to build object-level support for pattern matching on dependent types in a conventional type theory extended with uniqueness of identity proofs. It closes the problem opened by Thierry Coquand as to the status of pattern matching [Coq92, CS93] as implemented in ALF [Mag94]: it demonstrates that uniqueness of identity proofs is sufficient to support pattern matching where the unification underpinning case analysis is for first-order constructor forms—this is the unification suggested by Coquand and implemented in ALF. The necessity was shown by Hofmann and Streicher [HoS94, Hof95].

In the course of that demonstration, I used a new 'John Major' formulation of propositional equality. This allows elements of different types to *aspire* to equality, but ensures that they are only *treated* equally if they come from the same type. John Major equality is equivalent to Martin-Löf equality, but considerably more convenient in practice. It facilitates the expression of unification problems over sequences of terms involving type dependency, without requiring any dependency in the equations.

Consequently, I was able to extend the object-level first-order unification algorithm presented for simply typed constructor forms in my MSc work [McB96] to the dependently typed case. The necessary 'no confusion' and 'no cycle' theorems for each family of types can be constructed automatically in a uniform way. This is the object-level unification required to support pattern matching, and it shows that the need for uniqueness of identity proofs is no idle coincidence.

However, following the famous dictum of Marx and Engels, it is not enough merely to show that dependently typed pattern matching can be given meaning in an almost conventional type theory—the point is to show that it is good for something. I hope I have successively argued for the principle of representing relativised data in relativised types. I believe the developments of substitution and unification in chapter seven lend tangible credence to this argument. The unification example, in particular, demonstrates the importance of allowing datatypes to depend on *terms*.

The latter may require general recursion to be abandoned for the sake of typechecking,[1] but it makes more programs structurally recursive because it gives us more structures—types indexed by terms allow computation on the indices; types indexed by types do not.

MANTRA:

> If my recursion is not structural, I am using the wrong structure.

Dependent types make sense where general recursion is made sense of, if we are lucky.

[1]Lennart Augustsson disagrees, as do a number of others. In my opinion they are trying to have their cake and eat it, but they are nonetheless convinced of the advantages of cake.

There are many examples where the 'right structure' is hard to represent internally to the program, and where an external termination argument seems the prudent course, but the expressiveness of a dependent type system nonetheless offers the improved prospect of principled structural alternatives. The functional programming community ignores dependent types at its peril.

Turning from programs to their proofs, I suspect the idea of using elimination rules to capture the behaviour of program components abstractly from their implementations to be an important one. Specifications should not only tell us what programs to write—they should tell us what we need to know about the function when it is used. The latter behaviour is clearly like elimination in character. The kind of second-order rule supported by OLEG's **eliminate** tactic exploits such information in a compact and powerful way, relativised to the goal which motivates its use.

We are quite happy to specify and write programs (derived introduction rules) in an abstract and modular fashion—we should derive the corresponding elimination rules so that we can reason about programs in an abstract and modular fashion. We have been trying far too long to prove properties of programs by fiddling about with the primitive rules for data—we would never dream of *writing* programs that way. Henrik Persson has also identified this style of reasoning as of considerable assistance in his formalisation of the polynomial ring [Per99]. First-order equational specifications only do half the job—they are inappropriate for reasoning about the *usage* of programs. That is, they are good for characterising introduction behaviour, but they need to be complemented by a more effective treatment of elimination.

I believe the technology and methodology developed in this thesis contributes not only to the writing of programs which make sense, but to the effective exploitation of that sense in reasoning about them.

8.1 further work

> *'The world will be far better when we turn things upside down.'*
> (J. Bruce Glasier)

There is a great deal to be done.

Firstly, as far as the technology supporting dependently typed programming is concerned, it is an important task to identify a *recognisable* dependently typed programming language. As things stand, the equational programs we might like to write correspond only to the deducible computational behaviour of complex proof terms—if we want to be able to check a reloaded program, we need to reload its justification.

As I pointed out in chapter six, the problem lies in ensuring that stored programs give a satisfactory account of their empty cases. I believe that a reasonable way to go about this is to make the machine capable of detecting those argument types which can be shown to be empty by one step of case analysis. If more than one step is required, then the empty type can nonetheless be split into nontrivial constructor cases, and this is something the program can and should record. In effect, the program must contain enough hints to allow the reconstruction of the emptiness proof.

We might consider insisting that types be 'filled up' with markers indicating 'badness' in regions which would otherwise be empty. What implications for the expressiveness of the type system the enforcement of this discipline would entail, it is too early to say. However, the propagation of 'badness' surely involves the same work as the propagation of emptiness. It is a question of which gives the clearest treatment, and a more explicit approach is certainly worthy of attention.

With the development of improved technology for programming with dependent types, there is an imperative to *write programs*. Despite the clear argument from principle that more precise data structures lead to tighter programs—otherwise, why have types at all?—it is not rhetoric which changes *practice* but competition.

One example, close to home, which springs to mind is the development of a polymorphically typed strongly terminating functional programming language: parser, type inference algorithm, interpreter. Delphine Terrasse has encoded Natural Semantics in Coq [Ter95a, Ter95b] using a simply typed presentation of terms and types, with inductively defined relations describing valid formation and typing. It seems reasonable to hope that these latter properties can be built directly into data structures via dependent types. The work of Altenkirch and Reus [AR99] and of Augustsson and Carlsson [AC99] is already moving positively in this direction. Further, having developed first-order unification, an ML-style type inference algorithm [DM82] seems an obvious next step, especially as finding principal types is another optimisation problem addressable by the optimistic strategy. Also, there are existing developments in simple types available for comparison [NN96, DM99].

However, in tandem with the continuing development of programming technology, the development of a strong specification methodology which includes elimination as well as introduction rules seems a task of genuine importance. The focus of that development should be on program *derivation* at least as much as *verification*. Even at the early stage reached in this thesis, we have seen a small example of elimination rules used to transform a specification towards a program—the development of thick from thin.

More than this, an area of interest not touched on in this thesis, but prominent in my thinking is the use of derived elimination rules for programming itself. As a starting point, it seems very likely that Phil Wadler's suggestion to equip datatypes with different 'views' [Wad87] supporting different notions of pattern matching for the same underlying type can be put on a sound footing.

The motivation for such a development is very straightforward. As a matter of course, we write 'derived constructors'—functions which build elements of datatypes in more abstract patterns, reflecting the macroscopic structure of the problem at hand. We write plus to add numbers together, snoc to add an element to the end of a list, and so forth. It would surely be helpful to equip programmers with the means to analyse data at the same macroscopic level.

Although a great deal of attention has been paid to developing what goes on the right-hand side of pattern equations in a principled way, the left-hand side has long been neglected. It is time the left came into its own. We have nothing to lose but our chains. We have a world to gain.

Appendix A

Implementation

A few points about the prototype implementation:

- OLEG was implemented primarily as technology for the machine construction of the standard theorems with which I equip datatypes, and to support the writing of tactics at a relatively high level. The separation of partial constructions from terms is not rigidly enforced. Further, as it uses LEGO's unification algorithm, the scoping conditions for solving holes are not enforced either. However, the complete terms generated are independently checked by LEGO's reliable type-checker before they are trusted.

  The restrictions on the positioning and behaviour of holes were not rationalised until after the implementation was complete. Nonetheless, in all the developments I implemented, I found that I obeyed them. This gives at least anecdotal support to the suggestion that they are, in some way, natural restrictions to make.

- The implementation of the **eliminate** tactic does not have the abstraction facility described in chapter three. This still makes it entirely adequate for all the programming in this thesis, as such abstractions are not necessary when working with datatype elimination rules.

  The tactic does not, however, support derived elimination rules for functions in the way that it should. Although the examples using such rules have all been implemented and machine checked, the elimination rules for functions were applied by hand.

- The invention of 'John Major' equality came some time after I stopped work on the prototype. Consequently, the traditional equality (plus uniqueness) is used. Telescopic equations are thus represented in a somewhat awkward way, with each equation in the telescope coercing by all the previous ones in order to be well typed. This significantly complicated the elimination tactic and the unification technology, but nonetheless they work.

Bibliography

[AR99] Thorsten Altenkirch, Bernhard Reus. Monadic presentations of lambda-terms using generalized inductive types. In Computer Science Logic, 1999.

[ASG99] Alessandro Armando, Alan Smaill, Ian Green. Automatic Synthesis of Recursive Programs: The Proof-Planning Paradigm. Automated Software Engineering, 6(4):329–356. October 1999.

[Aug85] Lennart Augustsson. Compiling Pattern Matching. In Conference Funcitonal Programming and Computer Architecture, 1985.

[Aug98] Lennart Augustsson. Cayenne—a language with dependent types. In Proceedings of the International Conference on Functional Programming (ICFP'98). ACM Press, September 1998.

[AC99] Lennart Augustsson, Magnus Carlsson. An exercise in dependent types: A well-typed interpreter. 1999.

[Bar92] Henk Barendregt. Lambda calculi with types. In D.M. Gabbay, S. Abramsky and T.S.E. Maibaum, editors, Handbook of Logic in computer Science, volume 1. OUP. 1992.

[Bar99] Bruno Barras. Auto validation d'un système de preuves avec familles inductives. PhD Thesis, Université Paris VII. November 1999.

[BH94] Françoise Bellegarde, James Hook. Substitution: A formal methods case study using monads and transformations. Science of Computer Programming, 23(2-3). 1994.

[BP99] Richard Bird, Ross Paterson. de Bruijn notation as a nested datatype. Journal of Functional Programming. Vol. 9, No. 1, pp77–92. 1999.

[Bove99] Ana Bove. Programming in Martin-Löf Type Theory. Unification: A non-trivial Example. Licentiate Thesis. Chalmers University of Technology, Göteborg. In preparation.

[Bro96] Jason Brown. Presentations of Unification in a Logical Framework. 1996. D. Phil Thesis. Keble College, Oxford.

[BD77] Rod Burstall, John Darlington. A Transformation System for Developing Recursive Programs. JACM, Vol. 24, No. 1, January 1997, pp44–67.

[Bur69] Rod Burstall. Proving Properties of Programs by Structural Induction. Computer Journal, 12(1). pp41–48. 1969.

[BMS80] Rod Burstall, Dave MacQueen, Don Sannella. Hope: An Experimental Applicative Language. In Proceedings of the 1980 LISP Conference. Stanford, California.

[Bur87] Rod Burstall. Inductively Defined Functions in Functional Programming Languages. Journal of Computer and System Sciences. Vol. 34, Nos. 2/3, April/June 1987, pp409–421.

[Cla78] K. Clark. Negation as Failure. Logic and Databases, editors H. Gallaire, J. Minker, pp293–322. Plenum Press. 1978.

[Coen92] Martin Coen. Interactive Program Derivation. PhD Thesis. University of Cambridge. 1992.

[CG99] Adriana Compagnoni, Healfdene Goguen. Typed Operational Semantics for Higher Order Subtyping. To appear in Information and Computation.

[Coq97] Projet Coq. The Coq Proof Assistant Reference Manual, Version 6.1. Rapport de recherche 203, INRIA. 1997.

[CPM90] Thierry Coquand, Christine Paulin-Mohring. Inductively Defined Types. In P. Martin-Löf and G. Mints editors, Proceedings of Colog '88. Springer-Verlag LNCS 417. 1990.

[Coq92] Thierry Coquand. Pattern Matching with Dependent Types. In Proceedings Types for Proofs and Programs, June 1992.

[CS93] Thierry Coquand, Jan Smith. What is the status of pattern matching in type theory? In El Wintermte, pp112–114. June 1993.

[CT95] Cristina Cornes, Delphine Terrasse. Inverting Inductive Predicates in Coq. In Types for Proofs and Programs: International Workshop TYPES '95. Springer-Verlag LNCS 1158. June 1995.

[Cor97] Cristina Cornes. Conception d'un langage de haut niveau de représentation de preuves. Doctoral Thesis, Université Paris VII. 1997.

[CF58] H.B. Curry, R. Feys. Combinatory Logic. Amsterdam: North Holland. 1958.

[DM82] Luis Damas, Robin Milner. Principal type schemes for functional programs. In Proceedings, 9th ACM Symposium, Principles of Programming Languages, pp207–212. 1982.

[deB72] N.G. de Bruijn Lambda calculus notation with nameless dummies. Indagationes mathematicae 34, pp381–392.

[deB91] N.G. de Bruijn. Telescopic Mappings in Typed Lambda-Calculus. Information and Computation 91, pp189–204. 1991.

[DM99] C. Dubois, V. Menissier-Morain. Certification of a type inference tool for ML: Damas-Milner within Coq. Journal of Automated Reasoning, Vol. 23, No. 3, pp319–346. November, 1999.

[Dyb91] Peter Dybjer. Inductive Sets and Families in Martin-Löf's Type Theory. Logical Frameworks, edited by G. Huet and G. Plotkin. CUP 1991.

[Gen35] Gerhard Dentzen. Untersuchungen über das logische Schliessen. Mathematische Zeitschrift 39, pp176–210, 405–431. 1935. (In translation, pp68–131 of The Collected Papers of Gerhard Gentzen, edited by M.E. Szabo, North-Holland, 1969.)

[Gim94] E. Giménez. Codifying guarded definitions with recursive schemes. Proceedings of Types 94, pp39—59.

[Gim96] E. Giménez. Un Calcul de Constructions Infinies et son Application à la Vérification de Systèmes Communicants. Doctoral Thesis. ENS Lyon. 1996.

[Gim98] E. Giménez. Structural Recursive Definitions in Type Theory. In Proceedings of ICALP '98. Springer-Verlag LNCS 1443. July 1998.

[Gog94] H. Goguen. A Typed Operational semantics for Type Theory. PhD Thesis. University of Edinburgh. CST-110-94.

[GS91] P.A. Gardner, J.C. Shepherdson. Unfold/Fold Transformations of Logic Programs. pp565—582 of Computational Logic: Essays in Honor of Alan Robinson, edited by Jean-Louis Lassez and Gordon Plotkin, MIT Press, 1991.

[Hal99] Thomas Hallgren. Alfa User's Guide.
`http://www.cs.chalmers.se/~hallgren/Alfa`

[HP91] Robert Harper, Robert Pollack. Type checking, universe polymorphism and typical ambiguity in the calculus of constructions. Theoretical Computer Science, 89(1). 1991.

[vHLS98] F. von Henke, M. Luther, M. Strecker. Interactive and automated proof construction in type theory. In Bibel and Schmitt (1998), chapter 3: Interactive Theorem Proving.

[HoS94] Martin Hofmann, Thomas Streicher. The groupoid model refutes uniqueness of identity proofs. Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science. pp208–212. Paris, France. July 1994. IEEE Computer Society Press.

[Hof95] Martin Hofmann. Extensional concepts in intensional type theory. PhD Thesis. University of Edinburgh. CST-117-95.

[Hue75] Gérard Huet. A Unification Algorithm for Typed λ-Calculus. Theoretical Computer Science 1, pp27—57. 1975.

[Hue97] Gérard Huet. The Zipper. Journal of Functional Programming Vol. 7, No. 5, pp549–554. 1997

[Jau97] M. Jaume. Unification : a Case Study in Transposition of Formal Properties. In Supplementary Proceedings of the 10th International Conference on Theorem Proving in Higher Order Logics: Poster session TPHOLs'97. E.L. Gunter and A. Felty, editors. pp79–93. 1997.

[KST94] Stefan Kahrs, Donald Sannella, Andrzej Tarlecki. The Definition of Extended ML. LFCS Technical Report 94-300, University of Edinburgh. 1994.

[LP92] Zhaohui Luo, Robert Pollack. The LEGO proof development system: a user's manual. LFCS Technical Report 92-211, University of Edinburgh. 1992.

[Luo94] Zhaohui Luo. Computation and Reasoning: A Type Theory for Computer Science. 1994. Oxford University Press.

[Mac71] Saunders MacLane. Categories for the Working Mathematician. Springer Verlag GTM 5. 1971.

[Mag94] Lena Magnusson. The implementation of ALF—A Proof Editor based on Martin-Löf's Monomorphic Type Theory with Explicit Substitutiton. PhD thesis, Chalmers University of Technology, Göteborg. 1994.

[Man76] E. Manes. Algebraic Theories. Springer-Verlag GTM 26. 1976.

[MW81] Zohar Manna, Richard Waldinger. Deductive Synthesis of the Unification Algorithm. Science of Computer Programming, 1:5–48. North-Holland. 1981.

[MS94] P. Manoury, M. Simonot. Automatizing Termination Proofs of Recursively Defined Functions. Theoretical Computer Science, 135, pp319–343. 1994.

[M-L71a] Per Martin-Löf. An Intuitionistic Theory Of Types. Manuscript, 1971.

[M-L71b] Per Martin-Löf. Hauptsatz for the Intuitionistic Theory of Iterated Inductive Definitions. Proceedings of the Second Scandinavian Logic Symposium. North Holland. 1971.

[M-L75] Per Martin-Löf. An Intutionistic Theory Of Types: Predicative Part. In H. Rose and J.C. Shepherdson, editors, Logic Colloquium '74. North-Holland. 1975.

[M-L84] Per Martin-Löf. Intuitionistic Type Theory. Bibliopolis, 1984.

[McB70] Fred McBride. Computer Aided Manipulation of Symbols. PhD thesis, the Queen's University of Belfast, 1970.

[McB92] Conor McBride, Chris McBride. POLYSEMY: LISP with ambigous pattern matching and first class local definitions. Experimental implementation. 1992.

[McB96] Conor McBride. Inverting Inductively Defined Relations in LEGO. Types for Proofs and Programs, International Worskshop TYPES '96. Springer-Verlag LNCS 1512. pp236–253.

[MP67] J. McCarthy, J.A. Painter. Correctness of a compiler for arithmetic expressions. Mathematical Aspects of Computer Science, pp33–41. AMS, 1967.

[McC67] John McCarthy. A Basis for a Mathematical Theory of Computation. Computer Programming anf Formal Systems, P. Braffort and D. Hirschberg editors. North Holland Publishing Company. 1967.

[Mil91] Dale Miller. A Logic Programming Language with Lambda-Abstraction, Function Variables and Simple Unification. Journal of Logic and Computation 2/4, pp497–536. 1991.

[Mil92] Dale Miller. Unification Under a Mixed Prefix. Journal of Symbolic Computation 14, pp321–358. 1992.

[MTH90] Robin Milner, Mads Tofte, Robert Harper. The Definition of Standard ML. The MIT Press, 1990.

[Mog91] E. Moggi. Notions of computation and monads. Information and Computation, 93(1). 1991.

[Muñ96] César Muñoz. Dependent Types with Explicit Substitutions: A Metatheoretical development. Types for Proofs and Programs, International Worskshop TYPES '96. Springer-Verlag LNCS 1512. pp294–316.

[NN96] Wolfgang Naraschewski, Tobias Nipkow. Type Inference Verified: Algorithm W in Isabelle/HOL. Types for Proofs and Programs, International Worskshop TYPES '96. Springer-Verlag LNCS 1512. pp317–332.

[Nor88] Bengt Nordstrom. Terminating General Recursion. BIT, Vol. 28, pp605–619. 1988.

[P-M92] Christine Paulin-Mohring. Inductive Definitions in the System Coq: Rules and Properties. In Proceedings TLCA, 1992.

[P-M96] Christine Paulin-Mohring. Définitions Inductives tn Théorie des Types d'Ordre Supérieur. Habilitation Thesis. Université Claude Bernard (Lyon I). 1996.

[Pau85] Verifying the Unification Algorithm in LCF. Science of Computer Programming, 5:143–169. North-Holland. 1985.

[Pau86] Lawrence Paulson. Constructing recursion operators in intutionistic type theory. Journal of Symbolic Computation (2), pp325–355. 1986.

[Pau87] Lawrence Paulson. Logic and computation: interactive proof with Cambridge LCF. Cambridge Tracts in Theoretical Computer Science, Vol. 2. CUP. 1987.

[Per99] Henrik Persson. An Abstract Development of the Polynomial Ring in Agda. In Type Theory and the Integrated Logic of Programs, Doctoral Thesis. Chalmers University of Technology, Göteborg. 1999.

[Poll94] Erik Poll. A Programming Logic Based on Type Theory. Doctoral Thesis. Technische Universiteit Eindhoven, 1994.

[Pol90] Robert Pollack. Implicit Syntax. In preliminary Proceedings, 1st workshop on Logical Frameworks, 1990.

[Pol94] Robert Pollack. Incremental Changes in LEGO: 1994. LFCS Report. University of Edinburgh.

[Pra65] Dag Prawitz. Natural Deduction—A proof theoretical study. Almquist and Wiksell, Stockholm. 1965.

[Pym90] David Pym. Proofs, Search and Computation in General Logic. PhD Thesis. University of Edinburgh. 1990.

[Pym92] David Pym. A Unification Algorithm for the $\lambda\Pi$-Calculus. International Journal of Foundations of Computer Science Vol. 3 No. 3, pp333–378. 1992.

[Rob65] Alan Robinson. A Machine-oriented Logic Based on the Resolution Principle. ACM, 12:23–41. 1965.

[Rou92] Joseph Rouyer. Développement de'algorithme d'unification dans le Calcul des Constructions avecs types inductifs. Technical Report 1795, INRIA-Lorraine. November 1992.

[SH95] Amokrane Saïbi and Gérard Huet. Constructive Category Theory. In Proceedings of the joint CLICS-TYPES Workshop on Categories and Type Theory, Göteborg, Sweden. 1995.

[SSB99] Masahiko Sato, Takafumi Sakurai, and Rod Burstall. Explicit Environments (Extended abstract). 1999.

[Sau16] Ferdinand de Saussure. Cours de Linguistique Générale. 1916.

[SP94] Paula Severi, Erik Poll. Pure Type Systems with Definitions. In LFCS '94. Springer-Verlag LNCS 813, pp316–328. 1994.

[Sli97] Konrad Slind. Function Definition in Higher-Order Logic. In Theorem Proving in Higher-Order Logics. 9th International Conference, TPHOLs '96. Springer-Verlag LNCS 1125. August 1996.

[Str93] Thomas Streicher. Investigations into intensional type theory. Habilitiation Thesis, Ludwig Maximilian Universität. 1993.

[Tak95] M. Takahashi. Parallel reductions in λ-calculus (Revised version). Information and Computation. 118(1), pp120–127. 1995.

[TS83] Hisao Tamaki, Taisuke Sato. A transformation system for logic programs which preserves equivalence. ICOT TR-018. 1983.

[Ter95a] D. Terrasse. Encoding Natural Semantics in Coq. Fourth International Conference on Algebraic Methodology and Software Technology, (AMAST '95). Springer-Verlag LNCS. July 1995

[Ter95b] D. Terrasse. Vers un environnement de developpement de preuves en Semantique Naturelle. PhD thesis, École Nationale des Ponts et Chaussées (ENPC). October 1995.

[Tur95] David Turner. Elementary strong functional programming. Proceedings of the first international symposium on Functional Programming Languages in Education. Springer-Verlag LNCS 1022. 1995.

[Wad87] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. 14th ACM Symposium on Principles of Programming Languages, Munich, January 1987.

Index