Introduction

The discipline of functional programming encourages the development of concise, powerful solutions to problems using high-level abstraction techniques. Older functional languages, such as LISP and APL, offer only limited higher-order functions and usually feature a single, pre-defined aggregate data structure, such as dotted pairs or arrays. Despite these limitations, they manage to support concise programming by providing powerful functions which enable the wholesale manipulation of their particular data structures [McC*56, Ive62]. Newer functional languages, such as Hope, Miranda[1] and Standard ML, have developed and amplified these themes by offering more complete use of higher-order functions and by allowing programmers to define their own data types and aggregate structures, typically in the form of recursive algebraic type definitions [ADJ79, BMS80, Tur85, MTH90].

The evolution of functional programming languages has also seen a change in their treatment of type errors: older languages are usually *dynamically-typed*, meaning that type errors, such as the application of a function to an argument outside its proper domain, are not usually noted until the time of program execution. Newer languages are usually *statically-typed*, meaning that such errors are caught at the time of compilation of the original source code, thus shielding end-users from run-time error conditions. In making the move to static typing, the designers of modern functional languages were no doubt inspired at least in part by the success which the idea enjoyed in the context of imperative programming languages such as Pascal. Unfortunately, the type systems of most imperative languages are not sufficiently expressive for the needs of functional programming, especially in the context of user-defined type structures; in particular, early imperative language type systems were *monomorphic*, meaning that any value must have a specific, fixed type. For example, a function taking a list argument and returning its length can be written only for a single, fixed component type: different length functions are needed for different list component types (e.g., list of integers, list of employee records, list of bit-maps). This is especially unacceptable in functional languages where, given standard implementation techniques, the object code generated for these "different" functions would actually be identical. In fact, the source code for the functions would also be identical, but for the explicit type declarations for the different component types that are required by the type system. This needless repetition of identical code makes monomorphic type systems too rigid for use in functional programming languages.

In contrast, the type systems of modern functional languages are *polymorphic*, meaning that a value may have several "compatible" types simultaneously. More precisely, the type of such a value is expressed in terms of *type variables*, which can be instantiated with different specific types for different uses of the value. To continue our example from above, we can write under a polymorphic type system a single length function that can be used for lists with arbitrary component types. The type for such a function is expressed, for example, as:

$$\text{length} : \forall\alpha.\ \text{List}\ \alpha \rightarrow \text{Nat}$$

where Nat is the type of natural numbers and α is a type variable that can be instantiated to various component types (integer, employee record, bit-map, etc.) as needed.

Another pleasant property of polymorphic type systems, perhaps just as important in practice, is that polymorphic types can be automatically inferred for functional programs which lack any explicit type declarations. This result, already known in logic [Hin83], was reinvented independently for use in functional programming by Milner [Mil78]. The process of type inference proceeds by assigning type variables as placeholders to represent the types of newly-encountered program variables; these type variables are then progressively resolved into more specific types using unification, according to the constraints implied by the uses of program variables, constants of known type, and the manner of their combination. When the process is finished, any remaining type variables are quantified out, as their continued presence signifies that the program is valid, up to those points, independent of any specific type instantiation. Because of restrictions on the form of quantification and the parametric nature of the type variables, this type system is called *first-order parametric polymorphism*.

Despite the freedom from run-time errors, the flexibility of polymorphism and the reduction of effort which result from automated typed inference, there are some disadvantages inherent in static type systems in general and in first-order parametric polymorphism in particular. For example, in most static type systems data structures must be homogeneous, i.e., their components must be uniformly-typed: in dynamically-typed languages, components of disparate types can be mixed in heterogeneous data structures[2]. Static type systems must also perforce reject as ill-typed some programs which would otherwise compile and run without type errors; technically, any static type system for a reasonably expressive functional language must be semantically incomplete, though of course more and more expressive systems are possible. Finally, the restrictions which must be imposed on quantification in first-order polymorphic systems in order to make type inference feasible can be relaxed if we are willing to write type constraints and instantiations explicitly: such second- and higher-order type systems offer considerable expressiveness at the expense of the burden of program annotation [FLO83, BöBe85].

It is a different shortcoming of traditional first-order polymorphic type systems which provides the motivation for the present study, however, and it is to this matter that we now turn. Imagine that a programmer has defined, in some modern functional language, a number of different algebraic data structures such as lists and binary trees—the following definitions would be typical of the syntax of such languages:

$$\textbf{data}\ \text{List}\ \alpha = \text{nil} \mid \text{cons}\ (\alpha \times (\text{List}\ \alpha))$$

$$\textbf{data}\ \text{BTree}\ \alpha = \text{empty} \mid \text{branch}\ ((\text{BTree}\ \alpha) \times \alpha \times (\text{BTree}\ \alpha))$$

(we will give detailed accounts of this style of type definition and the function definitions below in later chapters; most readers' intuitions should be sufficient for our current motivational purposes).

As her work progresses, our programmer develops various utility functions for manipulating the structures she has defined—soon she notices certain parallels between functions which are defined for different structures, but which are used

[1] Miranda is a trademark of Research Software Ltd.

[2] Union types can mitigate this problem to an extent; recent research [Tha88] may also prove helpful.

for common purposes and which share certain patterns of definition. For example, to determine the sizes of various data structures she might define functions which count the number of components they contain. In the specific case of lists, she would define the familiar length function; in the case of binary trees, she would define a similar "size" function:

```
fun length nil = 0
  | length(cons(head,tail)) = 1+(length tail)

fun size empty = 0
  | size(branch(left,node,right)) = 1+(size left)+(size right)
```

In a language with automated type inference, types like the following would be inferred for these function definitions; note that these types are even more similar than the function definitions themselves, differing only in the specific type of structure in their argument:

$$\text{length} : \forall\alpha.\ \text{List}\ \alpha \rightarrow \text{Nat}$$
$$\text{size} : \forall\alpha.\ \text{BTree}\ \alpha \rightarrow \text{Nat}$$

In fact, the abstract idea of a component count is comprehensible quite independently of lists or trees, or of any specific structure for that matter. For every recursive data structure, there is a natural method of performing such a count, based on a recursive traversal of the structure. After writing a few such functions for different structures, our programmer might offer the following general scheme according to which a "count" function could be written for any other data structure she later defined:

- for each component, count one;
- for each non-component "leaf" node, count zero;
- for each pair or other aggregate, add together constituent counts;
- and for each recursive structure, apply the function recursively.

As another example, consider a functional which applies a function argument to every component of a structure, as the familiar *map* function does for the specific case of lists: following is a standard definition for map along with the type that would be inferred from it:

```
fun map f nil = nil
  | map f (cons(head,tail)) = cons(f head,map f tail)
```
$$\text{map} : \forall\alpha.\ \forall\beta.\ (\alpha \rightarrow \beta) \rightarrow \text{List}\ \alpha \rightarrow \text{List}\ \beta$$

Once again, the abstract notion of point-wise function application is quite clear, regardless of the specific notion of structure involved; after writing different "map" functionals for several data structures, our programmer might offer the following general scheme of definition, again based on a recursive traversal of the structure:

- for each component, apply the function argument;
- for each non-component "leaf" node, return the node unchanged;
- for each pair or other aggregate, return aggregated results;
- and for each recursive structure, apply the functional recursively.

The reader is invited to test our claim by writing a definition of an appropriate map functional for binary trees based on this general sketch.

Now one of the hallmarks of a successful functional language is that it should allow a programmer to capture as an abstraction expressed in the language any such pattern of definition which can be made lucid and rigorous. Unfortunately for our programmer, the "general schemes" above, while they seem at least to show potential in this regard, cannot be codified in any current language with static types and parametric polymorphism, for these languages offer no mechanism for traversing arbitrary recursive data structures, and no means of expressing the types of the resulting functions. Thus, even though there might be a sense in which these functions (and others as well) could be defined generally, independent of the specific data structure concerned, our programmer must write separate functions for each particular case that she needs. The parallel with the difference between monomorphic and polymorphic type systems is intentional: the standard parametric polymorphism displays the same sort of unreasonable rigidity from this perspective as the monomorphic systems do from the perspective of parametric polymorphism.

Our agenda is by now, I hope, both clear and compelling: we seek to devise a linguistic mechanism which will capture the sort of definitions sketched above, along with a type system which will enforce its proper use. Should we be successful, we will have extended the scope and power of polymorphic type systems in a direction particularly well-suited to the continued development of functional languages, by fulfilling the promise of polymorphism for user-defined algebraic types. This extended type system could help to revitalize, in a modern context, the established notion of wholesale manipulation of aggregate data structures. In order to suggest the feasibility of our endeavor, however, we should immediately address two critical issues, namely the forms we will use to express the new types and the mechanism we will use to traverse arbitrary data structures.

As regards the forms of the new types, we can directly adapt the approach of the standard parametric variety of polymorphism: under parametric polymorphism, functions which are generic with respect to an individual type are given types universally quantified with respect to a type variable; since the functions we seek to describe are generic with respect to a data structure, we give them types universally quantified with respect to variables ranging over data structures. Letting Σ be such a variable, we write the types of our two general functions above as follows:

$$\text{count} : \forall\Sigma.\ \forall\alpha.\ \Sigma\ \alpha \rightarrow \text{Nat}$$
$$\text{map} : \forall\Sigma.\ \forall\alpha.\ \forall\beta.\ (\alpha \rightarrow \beta) \rightarrow \Sigma\ \alpha \rightarrow \Sigma\ \beta$$

Individual uses of these functions at different structure and component types can then be seen as instantiating Σ to, e.g., List or Tree, as well as instantiating α and β to, e.g., Nat or Bool. Note that the instantiation of Σ in the type of count to List or BTree yields precisely the types of the functions length and size above, and that the the instantiation of Σ in the type of map to List yields precisely the type of the specific map for lists above.

As regards the form of definition of the functions themselves, the informal descriptions we gave above foreshadow our approach: we recursively traverse a structural argument, analyzing it according to cases based on its form. More precisely, we will use the *type* of a structural argument as the basis for distinguishing the various cases, thus guiding the ultimate computation which is performed on the value itself.

As it happens, the functional languages we seek to extend already provide a standard means of expressing case-based

computation over values of user-defined types, namely by the use of *patterns*. Patterns are a special kind of formal parameter whose structure reflects the form of potential arguments to the function being defined.

The sample function definitions we saw above used patterns in the left-hand sides of their defining clauses, for example the patterns built from the constructors nil and cons in the definitions of map and length. While recursive definitions are convenient and familiar, it is formally simpler to consider them as "syntactic sugar" for certain combinations of an explicit recursion construct and a form called a *pattern abstraction* which generalizes the familiar lambda abstraction. Whereas a lambda abstraction (written $\lambda x.e$) comprises a single variable and a single expression body, a pattern abstraction comprises multiple clauses, each consisting of a pattern and an expression body:

$$(p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n) : \tau \rightarrow \sigma$$

The whole abstraction is interpreted as a function whose argument type is determined by the forms of the patterns and whose result type is the determined by the corresponding expression bodies, with each pattern accounting for one possible form that a value of the argument type might take.

When a pattern abstraction is applied as a function, some pattern with a form congruent to the actual argument is chosen (various protocols for choosing a single pattern from several candidates are used in practice); the variables in the pattern are then bound to the corresponding constituents of the actual argument structure by an operation called *pattern-matching* and, finally, these bindings are used as a substitution in the corresponding expression body to obtain the function's result.

We adapt the use of patterns to our purposes by using patterns over types and type functions to distinguish and analyze data structure arguments and their constituents, thereby guiding the recursive traversals sketched above. The use of type functions, in particular, will allow us to distinguish the crucial notion of a component type through its correspondence with the parameter of the type function (e.g., the α in List α). By analogy with the pattern abstractions described above, we add a new form of *type-pattern abstractions* consisting of clauses formed from pairs of type patterns and expression bodies.

$$(\rho_1 \Rightarrow e_1 \mid \ldots \mid \rho_n \Rightarrow e_n) : \forall \alpha.\tau$$

This form generalizes the usual form of explicit type abstraction (often written $\Lambda\alpha.e$) as a degenerate case, just as the pattern abstraction form generalizes the usual lambda abstraction on values. The type of such an abstraction is polymorphic, and is expressed as a universal quantification ranging over types or type functions, depending on the forms of the patterns used.

We call this notion of polymorphism *analytical polymorphism*, since it is based on the recursive analysis of types rather than on the parametric independence of a typing with respect to a type variable. Note that although the analysis is based on the type, which therefore *guides* the traversal, the *extent* of the traversal is determined rather by the value itself.

Although these ideas seem straight-forward, several complications arise both in their formalization and in their adaptation to the context of automated type inference. The use of patterns over type *functions*, for example, engenders complex variable-scoping and pattern-matching problems. Furthermore, the natural type inference system for the language with type patterns does not admit principal types, and the associated unification problem, while not clearly equivalent to second-order unification, seems to be undecidable. Nevertheless, with appropriate restrictions and compromises we can provide a useful extension of the standard system of parametric polymorphism which captures the notion of functions that act on arbitrary data structures.

In the remainder of the document we proceed as follows: in Chapter 1 we explore a formal approach to algebraic data types, patterns and constructors in the relatively tame setting of simple types, more as a preparatory exercise than as an end in itself. We review here the simply-typed lambda calculus, in order to provide context and fix notation, and then introduce pattern abstraction as an extension to this basis. In Chapter 2 we introduce the notion of higher-order data structures in the context of an explicitly-typed, polymorphic calculus with kinds. Chapter 3 continues this account of the higher-order calculus, introducing patterns over types and type functions and the consequent notion of analytical polymorphism, using the methods developed for simply-typed patterns in Chapter 1 as a guide. In Chapter 4 we discuss operational and semantic issues which arise from the incorporation of type-pattern abstractions, including revisions to the syntax of constructors. In Chapter 5 we provide some extended examples of polymorphic functions which can be defined in the higher-order calculus and describe some variations and extensions related to the map functional.

We return to more pragmatic issues in Chapter 6, exploring the application of analytical polymorphism to realistic functional programming languages. In keeping with the requirements of first-order polymorphic types and the usual unification-based approach to automated type inference, we introduce a restricted notion called *structural polymorphism*, which allows instantiation of polymorphic entities only to outermost algebraic type constructors, rather than to arbitrary type functions. Even with this restriction, however, type inference from fully-implicit terms does not seem feasible, as we discover in Chapter 7. We identify there the core unification problem associated with type inference in this context, called *functional-variant unification*, and give a complete but potentially non-terminating algorithm for it.

The problems associated with functional-variant unification motivate us to modify the implicitly-typed language in Chapter 8 to require minimal explicit type constraints from the programmer. We present experimental evidence that this approach yields a useful and realistic compromise between feasibility and expressiveness, by way of several examples generated by an implementation of the type-checking system for the language with minimal constraints. The different approaches taken in Chapters 7 and 8 suggest a range of possible compromises which might be designed into a type system for realistic programming applications, trading off between conciseness, amenability to formal analysis and freedom from ambiguity.

In the conclusions section we assess our contributions, discuss connections with related work, and survey ideas for further research. We also provide more complete formal definitions for our pattern calculi in the three appendices, which have been organized to serve as reference sources.

Chapter 1
Simple Types and Pattern Abstraction

Typed lambda calculi form the natural basis for the study of type systems for functional programming languages: lambda calculus is the standard theory of higher-order functions and its sparse syntax allows us to focus on essential issues rather than on convenient syntax or the various derivable notions that prove useful in programming practice. Several styles of type systems have been studied in this context, including monomorphic and polymorphic systems of various complexities: we begin with a brief review of the most basic monomorphic system, the *simply-typed lambda calculus*, and then extend this system to incorporate algebraic data types and patterns. A complete set of definitions for the simply-typed pattern calculus is given in Appendix A, in an order determined by technical precedence: here in the body of the chapter we treat the material less formally and use an order determined more by rhetorical considerations.

1.1 The standard simply-typed lambda calculus

The syntax of the simply-typed lambda calculus is arranged into two levels of terms, the *expressions* (also called *value terms*) and the *types*. We say "levels" because the formation of expressions is predicated on the existence of suitably related types, whereas the syntax of types is context-free: expressions represent the actual values of computational interest, whereas types are needed only as checks to guarantee that expressions are well-formed and meaningful. Thus expressions are prior to types in terms of motivation, but types are *technically* prior to expressions, in that they are necessary to describe expression formation (in fact, types are also technically prior because they occur as subterms of expressions, whereas expressions do not occur in types).

The forms of types include set of constant types, for which we use the meta-variable B, and functional types built from an argument type and a result type using the infix symbol $\rightarrow$. The forms of expressions normally include variables, applications and lambda abstractions; for practical purposes we add a set of value constants, for which we use the meta-variable c.

$$\tau ::= B \mid \tau \rightarrow \tau$$
$$e ::= c \mid x \mid e\ e \mid \lambda x{:}\tau.\, e$$

Lambda abstractions of the form $\lambda x{:}\tau.\, e$ bind a variable with an explicit type constraint and represent functions: the intended meaning of an abstraction $\lambda x{:}\tau.e$ is given by the meaning of its *body* subterm e, considered as a function of the value associated with the bound variable x when the function is applied to an actual argument. The *constraint* τ serves to indicate that the actual argument is intended to be of the type τ; semantically, it should be an element of the set of values denoted by τ.

As we remarked above, term formation is not merely context-free: terms are considered well-formed only if they can be related to some type according to a system of axioms and inference rules. This system specifies the valid derivations of *well-typing judgements* with the form:

$$A \vdash e : \tau$$

where e is an expression, τ a type and A a set of *type assumptions* about variables; the judgement form is read "e has type τ under the assumptions A".

Formally, the syntax of assumptions is given inductively as follows:

$$A ::= \{\} \mid A, x : \tau$$

Thus assumptions are just lists of variable/type pairs—we will write them with the abbreviated syntax $\{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}$. Furthermore, we will often treat assumptions more semantically, as functions from variables to types, in the usual fashion. We use the following notation for the *projection* of the type associated with a given variable, the *merge* of two associations and the *domain* of an association:

$$A(x) = \{x{:}\tau, y{:}\sigma, \ldots\}(x) = \tau$$
$$A{+}A' = \{x{:}\tau \mid A'(x){=}\tau \text{ or } (A(x){=}\tau \text{ and } x{\notin}\mathrm{dom}(A'))\}$$
$$\mathrm{dom}(\{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}) = \{x_1, \ldots, x_n\}$$

In the remainder of the work we will use several other sorts of *finite associations* between atomic symbols and terms (or other objects). Although different formal treatments of associations are possible (and in some cases even necessary), our modest needs allow us to treat them rather informally, either as lists of pairs or as functions, using the notation above, according to convenience. We also use a generic notation for various sorts of associations, based on the one introduced above for assumptions, but with different brackets and delimiters for particular sorts of associations.

For example, we have an immediate need for a sort of associations which map constants to their associated types; we call these *(value) signatures* in keeping with the usual terminology from algebra, and write them as follows:

$$C_V ::= \{c_1{:}\tau_1, \ldots, c_n{:}\tau_n\}$$

To more completely parameterize the language by the constants used at both the type and value levels, we can construe the set of basic type constants from above as a *type signature* (a set of symbols is, in some sense, a degenerate association) and then combine the two signatures, type and value, into one conglomerate structure:

$$C_T ::= \{B_1, \ldots, B_m\}$$
$$C ::= \langle C_T, C_V \rangle$$

Although this conglomerate structure may seem heavy-handed and artificial, it will prove useful as a basis for successive generalizations later on.

As an example of a complete signature, consider the following simple one with basic types and constants for both natural numbers and truth-values:

$$C = \langle \{\mathrm{Int}, \mathrm{Bool}\},$$
$$\{0{:}\mathrm{Int},\ 1{:}\mathrm{Int},\ +{:}\mathrm{Int}{\times}\mathrm{Int}{\rightarrow}\mathrm{Int},\ T{:}\mathrm{Bool},\ F{:}\mathrm{Bool},$$
$$\neg{:}\mathrm{Bool}{\rightarrow}\mathrm{Bool},\ \wedge{:}\ \mathrm{Bool}{\times}\mathrm{Bool}{\rightarrow}\mathrm{Bool}\}\ \rangle$$

Since we presume that the sets of symbols used for variables and for constants are disjoint and that the set of constants will remain fixed for any given language, we can view the signature as a parameter of the language definition (variable assumptions, by contrast, are local to a particular term context, and are expanded and contracted according to the introduction and elimination of bound variables from the scope of the expression). If desired, signatures could be made explicit in the well-typing judgements with some syntax, e.g., $C, A \vdash e : \tau$ or $A \vdash_C e : \tau$; we will instead leave them implicit for the sake of conciseness and readability.

Returning now to the question of expression formation, we

present the axiom and inference rules which are used to derive well-typing judgements:

$$A \vdash c : C(c)$$

$$A \vdash x : A(x)$$

$$\frac{A+\{x{:}\tau\} \vdash e : \sigma}{A \vdash (\lambda x{:}\tau.\, e) : \tau \to \sigma}$$

$$\frac{A \vdash f : \tau \to \sigma \qquad A \vdash e : \tau}{A \vdash (f\ e) : \sigma}$$

The two axiom schemes simply state that constants and variables are well-formed with the types assigned to them by the global signature or local assumptions, respectively. We assume here that the typing judgement cannot be formed if the projection of the type fails, i.e., if the particular constant or variable is not in the domain of the corresponding signature or assumption (we will consider this a general policy later on when we introduce a variety of other operations into the statements of axioms and rules).

The first of the two rules can be understood as follows: an abstraction $\lambda x{:}\tau.\, e$ is considered well-formed with type $\tau \to \sigma$ under assumptions A, just when its body e is well-formed with type σ under those assumptions which extend A so that the variable x has type τ (note that any contextually prior assumptions concerning x are "over-written"). The second rule allows that an application is well-formed just when both its function and argument constituents are well-formed with certain types, so that the type of the argument term agrees with the argument part of the functions type and the type of the whole application is the functions result type.

A *well-typing derivation* is a tree, each node of which is a well-typing judgement, such that each parent node and its children together constitute an instance of one of the inference rules (derivations are traditionally written with the children above the parent, though some newer systems such as NuPrl reverse this tradition). The *subject* of the derivation is the judgement which occurs as the root of the tree; the leaves of the tree thus consist of occurrences of instances of the axiom schemes for constants and variables. We consider well-formed only those value terms for which a complete well-typing derivation can be constructed with the given term as subject.

The ultimate significance of this is that only well-formed terms are considered to have meaning; in particular, we now discuss the computational interpretation of expressions, which will allow us to compute the results of the functions and applications expressed by well-formed terms. In so doing, it is useful to contrast the simply-typed lambda calculus with the *untyped lambda calculus*, which includes similar forms, according to the following grammar:

$$e' ::= c \mid x \mid e'\ e' \mid \lambda x.\, e'$$

In the untyped calculus, there are no types and thus no formation rules or explicit constraints on variables: all terms freely-generated by the grammar are well-formed. We can relate the terms of the two languages in an obvious way by defining the *erasure* of a typed term so as to eliminate type constraints on variables:

$$\mathcal{E}(c) = c$$

$$\mathcal{E}(x) = x$$

$$\mathcal{E}(f\ e) = \mathcal{E}(f)\ \mathcal{E}(e)$$

$$\mathcal{E}(\lambda x{:}\tau.\, e) = \lambda x.\, \mathcal{E}(e)$$

The main semantic difference between the two systems is that terms of the pure, untyped calculus can be used to represent arbitrary computable functions. Without the use of recursion constants, terms of the typed calculus can represent only certain total functions (a standard encoding of natural numbers as certain higher-order functionals provides the representation necessary to state these properties formally). Operationally, calculations intended to evaluate untyped terms may proceed forever, whereas calculations intended to evaluate typed terms always terminate in irreducible *normal forms*, given suitable behavior for the constants.

More formally, we consider certain relations on pairs of terms, called *one-step reductions*, defined by the following rules (we may also think of these as functional operations by orienting the rules in the direction of the arrows):

$$(\lambda x{:}\tau.\, e)\ d \to_\beta e\,[d/x]$$

$$(\lambda x{:}\tau.f\ x) \to_\eta f, \quad \text{if } x \notin FV(f)$$

In the first rule, the notation e[d/x] represents substitution of the term d for the variable x inside the term e; as usual, a certain care must be taken in doing this to avoid the capture of bound variables. In the second rule, the condition requires that x not occur free (unbound) in the term f; the free variable operation is defined as follows for typed terms and similarly for untyped terms:

$$FV(c) = \{\}$$

$$FV(x) = \{x\}$$

$$FV(\lambda x{:}\tau.\, e) = FV(e) \setminus \{x\}$$

$$FV(f\ e) = FV(f) \cup FV(e)$$

In general, we should also give rules that specify the behavior of the constants given in the signature (these are historically called δ*-rules*); care must be taken, however, to avoid destroying the pleasant properties enjoyed by the β and η rules as outlined below (see [Br88, BrGa89] for results on the interaction between the standard rules and various "external" ones used for constants).

Taking the reflexive, transitive and congruent closure of the union of the one-step reduction relations gives us a relation called *many-step reduction*, written $e \twoheadrightarrow d$; adding symmetry to the closure gives us the *conversion* relation, written $e = d$, which is taken as the standard equivalence relation on terms of the calculus. Terms which cannot be further reduced are called *normal forms*, or said to "be in normal form", and represent the final results of computations.

From the statement of the two rules above, it is clear that reduction is independent of the type constraints on variables bound in lambda abstractions (at least barring any such atrocities in the rules given for constants): we can therefore define reduction for untyped terms in a manner exactly parallel to that for typed terms. As remarked above, some untyped terms can be reduced indefinitely, even growing in size as the "reduction" progresses; the reduction of typed terms, however, always terminates in a normal form, a property known as *strong normalization*:

Proposition (strong normalization):

For all assumptions A, expressions e and type types τ such that $A \vdash e : \tau$, $e \twoheadrightarrow d$ for some d in normal form.

The characteristic property of static type systems, as exploited in implementations of functional languages, is that once a term is determined to be well-typed (at "compile-time"), all type information can be erased from it before it is reduced (at "run-time") and the same normal form or computational result obtained, without risk of non-termination:

Proposition (static typing):

For all assumptions A, expressions e and type types τ such that $A \vdash e : \tau$, if $e \twoheadrightarrow d$ in normal form then $\mathcal{E}(e) \twoheadrightarrow \mathcal{E}(d)$ in normal form.

Another crucial property of reduction, one that is shared by both untyped and simply-typed calculi, concerns the order in which reductions are made. Note that in our description of reduction we maintained an essentially nondeterministic view, i.e., a given term might be reduced in several different ways, depending on the choice of which sub-terms are reduced in what order; to give a deterministic view of reduction, we must specify more precisely the order in which reduction steps are made.

A *reduction strategy* is a uniform way of choosing the order of reductions. Two well-known reduction strategies are called *normal order* and *applicative order*: the first always chooses the leftmost/outermost candidate for reduction, while the second always reduces argument terms before substituting them for bound variables in lambda-bodies. Whereas the second approach is more typical of programming language implementation techniques, the first can be shown to produce a normal form whenever one exists, even in the case of untyped terms. The *Church-Rosser property* of reduction, which holds for both the typed and untyped calculi, states that any two convertible expressions can be reduced to some common term; in combination with the properties of a normal order strategy, this gives some justification for using reduction to compute equivalence between terms. For the typed calculus, Church-Rosser in combination with strong normalization has the consequence that normal forms are unique, and thus that conversion between terms is decidable.

As a final note on reduction, we will have occasion to consider the η rule described above as operating in the reverse order, that is, so that function-typed variables are expanded into lambda abstractions with the function appearing at the head of the body:

$$f \rightarrow_{\eta} (\lambda x{:}\tau.\, f\ x)\ , \quad \text{for } x \notin FV(f)$$

The corresponding notion of many-step reduction, in combination with the usual β rule above, leads to a different notion of normal form, called $\beta\eta$*-long form*; terms in this form look much the same as they do in the usual normal form, except that function variables are syntactically identifiable as lambda abstractions.

1.2 Algebraic data types, constructors and patterns

We now proceed from the standard, simply-typed lambda calculus to an extension which will include algebraic data types and pattern abstractions. The basic concepts underlying algebraic types and patterns are taken from formal language theory, initial algebraic semantics and domain theory, and they have been incorporated into most modern functional languages starting with Hope [BMS80], and including Standard ML [MTH90], Miranda [Tur85], Haskell and others.

In these languages, new types are introduced by equations which define a *(basic) data type* in terms of several alternative clauses, each tagged by a corresponding *value constructor*. Clauses in the right-hand sides of the equations are built up from other data types using the product and function space; typically the equations are also recursive, in that the data type being defined occurs among the types in the right-hand side clauses. All of the usual types can be defined in this fashion, including enumerated types, etc. We have already seen some examples of definitions of this kind in the Introduction; the following further examples are written in roughly the style of Standard ML data type declarations:

data Bool = true | false

data Nat = zero | succ Nat

data NList = nil | cons(Nat × NList)

data Vector = mkvector(Nat→Nat)

(we defer until the next chapter a discussion of a more general form of *data structures* which allows, for example, lists to be parameterized with an arbitrary component type, as in the examples in the Introduction).

Values of data types are expressed using the value constructors associated with these types. Although value constructors are applied syntactically as if they were functions, they have no active computational import: rather, they serve as information-bearing tags or markers, just as the operations of an initial algebra. In the presence of the above definitions, for example, we can write the following terms using the value constructors zero, succ, nil and cons (we show also the type that would be associated with each of the expressions):

succ (succ zero) : Nat

cons (zero, cons (succ zero, nil)) : NList

The first of these examples represents the natural number two; the second, the two-element list containing the natural numbers zero and one, respectively.

Conversely, constructed values can be analyzed and their component parts utilized via the mechanism of *pattern-based function definitions*. The following definitions, again in roughly the style of Standard ML, define the functions *not* and *length* by cases, based on the outermost constructors of their arguments (as before, we also show the corresponding types):

fun not true = false
  | not false = true
not : Bool→Bool

fun length nil = zero
  | length (cons (x:Nat,y:NList)) = succ (length y)
length : NList→Nat

(note that the definition for length differs from the one given in the Introduction in that it contains explicit type constraints on the variables x and y: we work for now in an explicitly-typed language, but will explore implicit types in Chapters 6

and 7).

Rather than formalizing this particular mode of recursive function definition, we will consider a more direct generalization of the usual simply-typed lambda abstraction, namely the *pattern abstraction* alluded to in the Introduction; we can easily recover the form of the preceding function declarations by combining pattern abstraction with a standard recursion construct. Instead of a single type-constrained variable, as would be bound by the usual typed lambda abstraction, a pattern abstraction contains one or more clauses separated by vertical bars, where each clause consists of a pattern/expression pair. Following are two simple examples, contrasted in each case with more familiar lambda terms representing the same function:

$(n{:}\mathrm{Nat} \Rightarrow \mathrm{succ}\ n)$ *versus* $(\lambda n{:}\mathrm{Nat}.n+1)$

$(\mathrm{true} \Rightarrow \mathrm{false} \mid \mathrm{false} \Rightarrow \mathrm{true})$ *versus* $(\lambda b{:}\mathrm{Bool}.\ \neg b)$

Each of the first pair of terms defines a function which adds one to a natural number argument; the first term is a degenerate pattern abstraction with only a single clause, its pattern consisting of a single variable. Note also that the first term uses the constructor succ rather than the constants + and 1 used in the traditional function; in a simply-typed lambda calculus we would either need to encode numbers as functions (à lá the traditional Church numerals) or give separate reduction rules for the constant function +.

Each of the second pair of terms defines the usual boolean negation; the first term contains two pattern clauses, but each consists of just a single value constructor (in general, patterns abstractions will include several pattern/expression clauses, with each pattern built from a number of variables and constructors). Once again, the use of constructors and patterns allows us to avoid either an encoding of Booleans as functions or the introduction of new reduction rules for constants like ¬ : the pattern-based versions thus allow us to internalize the term-rewriting rules we would otherwise need to describe the computational behavior of constants.

Our first step in formalizing pattern abstractions is to account for basic data types (e.g., Nat, Bool) and value constructors (e.g., zero, succ, true). In modern functional languages these identifiers are normally introduced with a form of statically-scoped recursive definition, such as the ones given as examples above—we take a simpler approach based on global signatures like those introduced in the last section. We call signatures for constructors *(constructor) settings* in order to distinguish them from the signatures for arbitrary constants, which have a more general form and which might require extra reduction rules. The use of settings allows us to avoid certain distracting difficulties with scoped constructor definitions, including the shadowing of constructor names and the escape of constructed values from their scope of definitions (see [MTH90] for more on these difficulties). We could easily revise our system (and the ones described in later chapters) to accommodate locally-scoped data type definitions rather than fixed, global settings.

Letting C range over constructor settings, B over basic data types, k over value constructors and τ over types, we use the following syntax for settings and types:

$$C_T ::= \{B, \ldots\}$$

$$C_V ::= \{k{:}\tau \to B, \ldots\}$$

$$C ::= \langle C_T, C_V \rangle$$

$$\tau ::= 1 \mid \tau \times \tau \mid B \mid \tau \to \tau$$

Here C_T and C_V are finite associations, with the usual syntax and operations; we refer to them as the *upper* and *lower halves*, respectively, of a constructor setting of the form $\langle C_T, C_V \rangle$. The forms of types include a unit type and products in addition to the usual functional and basic data types, where latter are the provided via the setting. Note that the types of value constructors have a specialized form: they are always functions mapping some fixed argument type to an associated basic data type. Formally, constructor settings should really be constrained by formation rules which ensure that the basic data types mentioned in the lower half of the setting actually occur in the upper half, and that neither half contains repeated identifiers; we will simplify the presentation in the present context by glossing over such details (we provide a more complete and formal description in Appendix A).

As an example of a typical constructor setting, consider the following:

$$\begin{aligned} C \equiv \langle & \{\mathrm{Bool}, \mathrm{Nat}, \mathrm{NList}\}, \\ & \{\mathrm{true}{:}1 \to \mathrm{Bool},\ \mathrm{false}{:}1 \to \mathrm{Bool}, \\ & \ \mathrm{zero}{:}1 \to \mathrm{Nat},\ \mathrm{succ}{:}\mathrm{Nat} \to \mathrm{Nat}, \\ & \ \mathrm{nil}{:}1 \to \mathrm{NList},\ \mathrm{cons}{:}\mathrm{Nat} \times \mathrm{NList} \to \mathrm{NList}\} \rangle \end{aligned}$$

This setting provides basic data types and constructors for the types of booleans, natural numbers and lists of natural numbers. Note the use of the unit type (1) to give the value constructors true, false, zero and nil trivial domains: this coerces them to be functions rather than basic constants, as required by the constraints on settings.

Given a fixed but arbitrary setting, C, the remaining syntax of the language consists of two sorts called expressions and patterns; letting e, d, f range over expressions and p, q range over patterns we have:

$$e ::= * \mid x \mid e, e \mid k \mid (p \Rightarrow e)^+ \mid \mu x{:}\tau.\, e$$

$$p ::= * \mid x{:}\tau \mid p, p \mid k\ p$$

Expressions include the familiar forms for variables and applications; in addition, we have the trivial value $*$, pairs, value constructors, pattern abstractions and a recursion construct (informally, the value denoted by the recursion form $\mu x{:}\tau.\, e$ is the value denoted by e when x is associated with the value of the whole expression).

In the grammar we have used the notation "$(p \Rightarrow e)^+$" to indicate a non-zero number of clauses separated by vertical bars, since the explicit use of the vertical bar might otherwise be confused with its use as a symbol in the meta-language of context-free grammars. Thus pattern abstractions actually have the form:

$$p \Rightarrow e \mid \cdots \mid p \Rightarrow e$$

for some non-zero number of pattern/expression pairs.

Note that the syntax of patterns is essentially similar to that for expressions, but with these exceptions:

- occurrences of variables in patterns are decorated with explicit type constraints, as were the bound variables of simply-typed lambda abstractions;
- there are no forms for pattern abstractions or recursions within patterns themselves;
- the only form of application for patterns is for a

constructor applied to an argument pattern.

Once again, we describe the context-sensitive aspects of the syntax by way of a type inference system, though in the presence of patterns the system is necessarily somewhat more complicated; we now have two forms of well-typing judgements, one for expressions and another for patterns:

$$A \vdash e : \tau$$
$$\vdash p : \tau$$

(here A ranges over type assumptions, formed just as in the simply-typed lambda calculus, but allowing now for the expanded notion of types). Note that pattern formation does not depend on type assumptions for variables: rather, all variables in a pattern are locally bound by that pattern. Note also that we still consider formation relative to a global constructor setting, although neither judgment form makes this dependence explicit.

The axioms and rules for the trivial value, variables and pairs are straightforward; we state them here without further explanation:

$$A \vdash * : 1$$

$$A \vdash x : A(x)$$

$$\frac{A \vdash e : \tau \qquad A \vdash d : \sigma}{A \vdash (e,d) : \tau \times \sigma}$$

The rules for value constructors and recursion require only a little discussion (we will take up patterns and pattern abstractions below):

$$A \vdash k : C_V(k)$$

$$\frac{A+\{x:\tau\} \vdash e : \tau}{A \vdash (\mu x{:}\tau.e) : \tau}$$

According to the first rule, nullary constructors, such as zero and nil from our example setting, must be applied to the trivial constant $*$ in order to result in values of type Nat and Bool, respectively. Since such applications are fully determined ($*$ is the only one expression with unit type 1) and can thus be easily reconstructed, we will systematically abbreviate in such cases to an occurrence of the constructor alone, as exemplified here (we will use the same convention when writing patterns which include nullary constructors):

$$nil \equiv nil\ *$$

As for the recursion rule, we would only point out that the variable x is bound by the μ form and note that the same type is required for the expression as a whole as is assumed for the constrained variable: this is natural when we recall our intended interpretation of the form, i.e., that the the value denoted by the recursion form $\mu x{:}\tau.\, e$ is the value denoted by e when x is associated with the value of the whole expression.

The formation of pattern abstractions depends on the proper formation of their constituent patterns and expressions, among other things. We exploit the essential similarity between patterns and expressions by writing the pattern formation rules in terms of expression formation; although this may seem circular, recall that pattern abstractions never occur within patterns themselves, so the entire system can be founded on the rules for the non-abstraction expressions, also called *ground expressions* (ground expressions denote individual entities, as opposed to functions, and are thus representative of the ultimate results of computation).

In order to exploit the similarity of patterns and expressions, we need two auxiliary operations, one to project out the type assumptions on variables in a pattern (written $\mathcal{A}(p)$) and one to erase their type constraints, producing the corresponding expression (written $\mathcal{E}(p)$). The formal definition of these operations is straight-forward, though we will remark on our treatment of variable clashes in the case of the assumptions operation (see Appendix A for the other details). Essentially, we wish to ensure that patterns are linear, i.e., that no single variable occurs twice. This restriction allows us to avoid the undecidable problem of function equality which arises when repeated variables are instantiated to functional values [O'Don85]. The restriction could actually be relaxed for variables of non-functional type, given a general notion of equality over arbitrary ground types, but we retain the stricter version here for the sake of simplicity (the definition, within the language, of such a general notion of equality is in fact one of our goals in later chapters).

Our approach to this problem is to employ a variation on the usual merge of associations: whereas the usual merge is asymmetric, preferring the associations in the second argument when the domains intersect, the variation is symmetric, and fails explicitly when the domains intersect. We call the new variation the *weak merge*, with notation and definition as follows:

$$A \oplus A' = \textbf{if } dom(A) \cong dom(A') \textbf{ then } A+A' \textbf{ else fail}$$

Given this variation on merge we can define the assumption operation on pattern as follows:

$$\mathcal{A}(*) = \{\}$$
$$\mathcal{A}(x{:}\tau) = \{x{:}\tau\}$$
$$\mathcal{A}(p,q) = \mathcal{A}(p) \oplus \mathcal{A}(q)$$
$$\mathcal{A}(k\ p) = \mathcal{A}(p)$$

By way of an example, we have the following:

$$\mathcal{A}(cons(h{:}Nat,t{:}NList)) = \{h{:}Nat,\ t{:}NList\}$$

On the other hand, a non-linear pattern would cause the operation to fail:

$$\mathcal{A}(cons(h{:}Nat,h{:}NList)) = \textbf{fail}$$

Now we can be state the rule for pattern formation: a pattern p is well-formed with type τ when p's erasure, as an expression, is well-formed with type τ, relative to the assumptions contained in p as constraints on variables. The constraint that the pattern is linear is enforced by our presumption that the rule can only be applied if the assumption operation in the antecedent succeeds. Thus the rule for pattern formation reads simply as follows:

$$\frac{\mathcal{A}(p) \vdash \mathcal{E}(p) : \tau}{\vdash p : \tau}$$

Before discussing formation rules for pattern abstractions themselves, we first sketch their operational interpretation, in order to provide the background necessary to motivate certain restrictions. Recall first that pattern abstractions represent

functions, just as the lambda abstractions they generalize, but with several constituent patterns serving as structured formal parameters. When an actual argument is supplied to an abstraction through application, the argument is compared to the various patterns for structural similarity; some particular pattern is chosen to be matched against the argument. *(Pattern) matching* is an operation which "deconstructs" the actual argument into constituent pieces, according to the common structure of the pattern and the expression, and associates the residual constituents of the expression with those variables of the pattern which occur in corresponding positions.

Using the notation $\mathcal{M}(p,e)$ for the matching operation, we have:

$$\mathcal{M}(p,e) = [e_1/x_1, \ldots, e_n/x_n]$$

where the e_i are those subterms of e corresponding to the occurrences of the variables x_i in p. The result of matching is an association of expressions to variables, called a *substitution*, which is used in the evaluation of the expression body corresponding to the particular pattern chosen for the match; this evaluation yields the final result of the function application.

More formally, we will describe operational semantics for the calculus using reduction rules, just as we did for the simply-typed lambda calculus above; in particular, the steps just outlined are captured by a β rule similar to the β rule used to reduce applications in the simply-typed and untyped calculi:

$$(p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)\ d \rightarrow_\beta e_i\,[\mathcal{M}(p_i,d)],\ \text{some } i$$

(we will develop this and several other reduction rules at length in Section 1.4). Note that the matching operation is partial, since patterns will only match expressions of similar form, and thus a proper choice of i is crucial—we leave the details of how i is chosen to the next two sections. The postfix use of matching indicates substitution of expressions for free variables in the usual manner—this is just a generalization to multiple variables of the usual notation $e[d/x]$, though here we construe the results of matching as an entity unto itself, namely the substitution result of $\mathcal{M}$.

The matching operation is formally defined as follows:

$$\mathcal{M}(*,*) = []$$
$$\mathcal{M}(x{:}\tau,e) = [e/x]$$
$$\mathcal{M}((p_1,p_2),(e_1,e_2)) = \mathcal{M}(p_1,e_1) + \mathcal{M}(p_2,e_2)$$
$$\mathcal{M}(k\ p,\ k\ e) = \mathcal{M}(p,e)$$
$$\mathcal{M}(p,\ e) = \textbf{fail},\ \text{otherwise}$$

Here the use of the explicit failure has the same affect on the use of the rule as it did in the case of the assumptions operation above, namely that it blocks any use of the rule.

Following are a few examples of the results of matchings:

$$\mathcal{M}(x{:}\text{NList},\ \text{cons}(\text{zero},\text{nil})) = [\text{cons}(\text{zero},\text{nil})/x]$$
$$\mathcal{M}(\text{cons}(\text{succ}(n{:}\text{Nat}),y{:}\text{NList}),\ \text{cons}(\text{succ zero},\text{nil})) = [\text{zero}/n,\ \text{nil}/y]$$
$$\mathcal{M}(\text{cons}(\text{succ}(n{:}\text{Nat}),y{:}\text{NList}),\ \text{cons}(\text{zero},\text{nil})) = \textbf{fail}$$
$$\mathcal{M}(\text{succ}(n{:}\text{Nat}),\ m) = \textbf{fail}$$

In the first example, the simple pattern consisting of just the variable x matches any expression; thus the whole expression is associated to x in the result. Both the pattern and the expression are structured terms in the second example, and since the structure matches up to the point where variables occur in the pattern, those variables are associated to the corresponding constituent expressions in the resultant substitution. The third and fourth examples fail, the third because the sub-pattern succ(n:Nat) is constructed differently than the corresponding sub-expression zero and the fourth because the pattern is specifically structured while the expression is just a variable—note thus that matching does *not* instantiate variables from its expression argument, unlike the similar but more symmetric operation of unification.

We can easily prove the following fact by induction on the forms of patterns:

Proposition (correctness of matching):

If $\mathcal{M}(p,e) \neq \textbf{fail}$, then $e = p[\mathcal{M}(p,e)]$

1.3 Pattern formation: coverage and compatibility

We return now to the question of which of possibly several similar patterns in an abstraction should be used for the match in the β reduction rule (i.e., the crucial issue of the choice of i, which was deferred at the end of the previous section). In fact, there are several reasonable approaches to this issue, and in practice at least two are used in actual programming language implementations. In order to accommodate all of the approaches, we discuss three different pattern-matching *regimens* from a a general perspective, giving a variation of the rule for each regimen—this approach allows for taste, convenience, efficiency or other design considerations which might affect the actual choice of regimen in any given context. The particular regimens we describe are *first-match* , *only-match* and *best-match*; corresponding to each is a different notion of *compatibility* on sets of patterns, which figures into the formation rule for pattern abstractions.

More specifically, we have:

- *first-match*: the first pattern (sequentially) which matches is chosen; any set of patterns is compatible
- *only-match*: a unique pattern matches and is chosen; a set of patterns is compatible iff the patterns are pair-wise independent, i.e., no two will match any single expression
- *best-match*: the most-specific pattern which matches is chosen; sets are compatible iff overlaps can be resolved, i.e., if any expression always has a most-specific match

The formation rule for pattern abstractions will depend on the notion of compatibility corresponding to a given choice of pattern-matching regimen, which in turn determines how matches will be chosen for the β reduction rule. We defer a formal description of the matching choice mechanism for each regimen to the next section, since we will need to develop some supporting concepts in this section first.

In addition to the compatibility conditions associated with the pattern-matching regimens described above, the formation of pattern abstractions also depends on a dual notion of *coverage*: whereas compatibility is concerned with our ability to choose a *single* pattern when matching several candidate patterns against an expression, coverage is concerned with our ability to find *at least one* pattern with which to match against any possible argument expression. More specifically, we will define compatibility as a predicate on sets of patterns, according to the ideas sketched above, and coverage as a relation between sets of patterns and a type, such that if the set of patterns covers the type, then at least one pattern in the set will match any given expression of that type.

A couple of auxiliary notions are useful in formalizing compatibility and coverage; first among these is a natural notion of *subsumption* between patterns. Subsumption can be defined conveniently in terms of the matching operation described above and an erasure operation taking a pattern to the corresponding expression without type constraints; we begin by defining subsumption for expressions and patterns, as follow:

$p \sqsubseteq e$ iff $\mathcal{M}(p,e) \neq$ **fail**

Note that the relation is defined only when the left argument is a ground expression, although variables on the left subsume arbitrary expressions on the right. As the other rules simply extend the relation compatibly with the syntax of ground expressions, one (ground) expression subsumes another just when the two share a common structure except that the first may have variable constituents in places where the second has (possibly) more refined structure. Subsumption is thus a partial-ordering on ground expressions.

We then extend subsumption to patterns using the following notion of pattern erasure:

$\mathcal{E}(*) = *$
$\mathcal{E}(x{:}\tau) = x$
$\mathcal{E}(p,q) = (\mathcal{E}(p), \mathcal{E}(q))$
$\mathcal{E}(k\ p) = k\ (\mathcal{E}(p))$

Subsumption extends naturally to a relation between pairs of patterns in a similar way. As mentioned above we then define:

$p \sqsubseteq q$ iff $p \sqsubseteq \mathcal{E}(q)$

We can easily prove the following:

Proposition (subsumption ordering):

The subsumption relation $p \sqsubseteq q$ is a partial ordering on linear patterns, up to the equivalence induced by renaming of variables (this is needed for anti-symmetry).

We say that two patterns *overlap* ($p \approx q$) if there is an expression which both of the patterns subsume, i.e., which both will successfully match; otherwise we say that they are *independent* ($p \not\approx q$)[3]. For example, we have:

$(x,nil) \approx (nil,y)$
$(cons(zero,y), z) \not\approx (cons(succ\ a,b), z)$

In the first case, the expression (nil,nil) will match both patterns; in the second, there can be no such expression, since the zero and succ constructors are distinct.

We can define overlap relation more directly and effectively as follows:

$\mathcal{O}(*,*) = true$
$\mathcal{O}(x{:}\tau,q) = true$
$\mathcal{O}(p,y{:}\sigma) = true$
$\mathcal{O}((p_1,q_1),(p_2,q_2)) = \mathcal{O}(p_1,p_2)$ and $\mathcal{O}(q_1,q_2)$
$\mathcal{O}(k\ p,k\ q) = \mathcal{O}(p,q)$
$\mathcal{O}(p,q) = false$, otherwise

We can then show:

Proposition (adequacy of overlap):

For all patterns p and q, $\mathcal{O}(p,q)$ iff $p \approx q$.

Given the definitions of subsumption, overlap and independence, we can now state more formally the compatibility predicates for sets of patterns appropriate to each of the three pattern-matching regimens:

- *first-match*: any set of patterns is compatible
- *only-match*: a set of patterns $\{p_1, \ldots, p_n\}$ is compatible iff for all distinct i and j, $p_i \not\approx p_j$
- *best-match*: a set of patterns $\{p_1, \ldots, p_n\}$ is compatible iff each $\approx$–comparable subset has a $\sqsubseteq$–greatest member

Turning now to the notion of coverage, if we consider the natural definition of this relation we are once again faced with a non-effective definition over an infinite domain, since there are, in general, an infinite number of well-typed expressions which might be supplied as an actual argument to a pattern abstraction (we need consider only well-typed arguments, but. this does not solve the problem). The definition of an effective version of coverage is more complex and less direct than that for overlap, but is still based on fairly simple reasoning.

Note first that if a pattern matches some argument, it will match any refinement thereof, in the sense made formal by the subsumption ordering. Therefore, when considering a finite number of patterns of finite size, it will suffice to consider potential arguments only up to a certain size as well. A finite set of well-typed expressions of a certain type, up to a fixed size, can be recursively generated by referring to the global constructor setting to determine which value constructors build particular basic data types and to basic properties of the unit and product types—the structure of non-ground expressions (i.e., functions) is immaterial in issues of matching.

We can afford to be somewhat liberal in defining the notion of "size" to be used, since we are interested here more in effectiveness than in efficiency; thus we define the *depth* of a pattern as follows:

$\mathcal{D}(*) = 0$
$\mathcal{D}(x{:}\tau) = 0$
$\mathcal{D}(p,q) = 1+\max(\mathcal{D}(p), \mathcal{D}(q))$
$\mathcal{D}(k\ p) = 1+\mathcal{D}(p)$

We can now define the aforementioned operation to generate (finite) sets of well-typed patterns, based on the desired type, the global setting and a specified depth:

$\mathcal{G}^0(\tau) = \{x{:}\tau\}$, fresh x
$\mathcal{G}^n(1) = \{*\}$
$\mathcal{G}^{n+1}(B) = \{k\ p \mid C_V(k) = \tau \to B \text{ and } p \in \mathcal{G}^n(\tau)\}$
$\mathcal{G}^{n+1}(\tau \times \sigma) = \{(p,q) \mid p \in \mathcal{G}^n(\tau) \text{ and } q \in \mathcal{G}^n(\sigma)\}$
$\mathcal{G}^{n+1}(\tau \to \sigma) = \{x{:}\tau \to \sigma\}$, fresh x

As examples of expression generation, consider the following (we assume here the previously defined example constructor setting):

$\mathcal{G}^0(Nat) = \{n{:}Nat\}$
$\mathcal{G}^1(Nat \times Nat) = \{(n{:}Nat, m{:}Nat)\}$

[3] Our use of the symbol $\approx$ is a bit misleading: note that overlap is *not* an equivalence relation.

$\mathcal{G}^2$(Nat×Nat) = {(zero,zero), (zero,succ (n:Nat)), (succ (m:Nat),zero),

(succ (i:Nat),succ (j:Nat))}

Finally, we can define the relation of coverage between sets of patterns and (single) types, written $\{p_1,\ldots,p_n\} \succ \tau$, using depth, subsumption and the pattern generating operation:

$\{p_1,\ldots,p_n\} \succ \tau$ iff for all $q \in \mathcal{G}^m(\tau)$, some $p_i \sqsubseteq q$ where $m = max\{\mathcal{D}(p_i) \mid 1 \leqslant i \leqslant n\}$

Our definition of coverage can be justified by the following proposition, which states that any expression of the covered type will be matched by some pattern in the set:

Proposition (adequacy of coverage):

If $\{p_1,\ldots,p_n\} \succ \tau$ then for all well-type expressions e of type τ, there is some $1 \leqslant i \leqslant n$ such that $p_i \sqsubseteq e$.

Here the intention is that the patterns in the set will be the constituents of some well-formed pattern abstraction and thus all of a common type; by taking the maximum depth of the patterns in the abstraction, we account for all possible arguments of that type which might be supplied to the abstraction, up to the distinctions observable by the patterns.

Continuing from the previous example, we have:

{(zero,zero), (succ(i:Nat),succ(j:Nat)), (n:Nat,m:Nat)} ≻ (Nat×Nat)

The relation would not hold, for example, if we dropped the last of the patterns listed, since expressions such as (zero,succ k) would not be accounted for.

Given these accounts of compatibility and coverage, we can state the rule for formation of pattern abstractions. A given abstraction is well-formed with type $\tau \to \sigma$ if and only if each of the following holds: its constituent patterns are compatible; they cover the type τ; the pattern half of each clause is well-formed with type τ; and the expression half of each clause is well-formed with type σ, under the global assumptions plus those projected out of the corresponding pattern. Thus a derivation that an n-ary pattern abstraction is well-formed will have 2n+2 antecedents: n for the formation of the patterns, n for the formation of the expressions and 2 for the compatibility and coverage properties of the patterns.

$$\frac{\begin{array}{c}\{p_1,\ldots,p_n\}\ \text{compatible} \\ \{p_1,\ldots,p_n\} \succ \tau \\ \vdash p_i : \tau \quad (1 \leqslant i \leqslant n) \\ A + \mathcal{A}(p_i) \vdash e_i : \sigma \quad (1 \leqslant i \leqslant n)\end{array}}{A \vdash (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) : \tau \to \sigma}$$

Note the relationship between this rule and the similar formation rule for simply-typed lambda abstractions: in the simply-typed rule, only a single type-constrained variable is involved, and the local assumptions are merely extended with that constraint assumption to show the body of the abstraction well-formed. In this rule, we have multiple patterns for which compatibility, coverage and formation must be checked, and yet the final set of antecedents above corresponds to the single antecedent of the simply-typed rule, in that local assumptions are extended to include constraint information about the variables (in each pattern), in order to show the (corresponding) body of the abstraction well-formed.

As an extended example of formation for pattern abstractions, consider the following well-typing for a length function, defined in the context of our example constructor setting:

{} ⊢ (μL:NList→Nat. (nil⇒zero | cons(h:Nat,t:NList)⇒succ(L t))) : NList→Nat

The function is recursive, taking a list argument and returning a natural number result; in order to show the whole recursion expression well-typed with respect to the empty set of assumptions, we show that the the body has the desired type under the assumption that the recursion variable (here L) has that same type:

{L:NList→Nat} ⊢ (nil⇒zero | cons(h:Nat,t:NList)⇒succ(L t)) : NList→Nat

To show the pattern abstraction well-typed we must establish six antecedents, two concerning the compatibility and coverage restrictions for the patterns used and four concerning the well-typing of the patterns and expressions constituting the two clauses. The two patterns involved are independent, so they satisfy the compatibility requirements regardless of our choice of pattern-matching regimen:

{nil, cons(h:Nat,t:NList)} compatible

To establish coverage of the type NList by the two patterns, we must check that all of the generated expressions up to depth two are accounted for by one of the patterns, since two is the maximum depth of the patterns; there are two such expressions, and they are in fact equivalent to the two patterns, up to erasure and changes of variables.

{nil, cons(h:Nat,t:NList)} ≻ NList
max{$\mathcal{D}$(nil), $\mathcal{D}$(cons(h:Nat,t:NList))} = 2
$\mathcal{G}^2$(NList) = {nil, cons(x,y)}

We must also establish that both patterns are well-typed:

⊢ nil : NList
⊢ cons(h:Nat,t:NList) : NList

For the second pattern, recall that a pattern is well-typed just when it is linear and its erasure is well-typed (as an expression) with respect to the type assumptions it contains.

{h:Nat,t:NList} ⊢ cons(h,t) : NList

Finally, we must show that both of the expressions occurring as clause bodies are well-typed with respect to the global assumptions when extended by the assumptions contained in thir respective patterns. In the case of the second clause, we use the assumption for the recursion variable L to establish that recursively taking the length of the tail of the list is well-typed wit result type Nat.

{L:NList→Nat} ⊢ zero : Nat
{L:NList→Nat,h:Nat,t:NList} ⊢ succ(L t) : Nat

1.4 Operational semantics via reduction

In section 1.3 we briefly sketched a β-reduction rule

appropriate for pattern abstractions—we now return to full consideration of the operational semantics of the simply-typed pattern calculus. As for the untyped and simply-typed calculi, we first specify a number of simple "one-step" rules relating terms; we can then derive many-step reduction, representing operational computation, as their reflexive, transitive and congruent closure, and conversion, representing the standard equivalence relation on terms, by including symmetry in the closure as well.

We recall now the previously stated β rule for reduction:

$$(p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)\ d \rightarrow_\beta e_i\ (\mathcal{M}(p_i,d)),\ \text{some } i$$

As we have already discussed informally, the particular choice of i here depends on the pattern-matching regimen used; more formally, we have:

- *first-match*: i = the least j such that $p_j \sqsubseteq d$
- *only-match*: i = the unique j such that $p_j \sqsubseteq d$
- *best-match*: p_i = the most specific p_j such that $p_j \sqsubseteq d$

where the notion of "most specific" pattern is in the sense of the subsumption ordering $\sqsubseteq$.

This rule is a fairly complex one, depending as it does on the notions of constructor-based patterns, matching and subsumption. Although formal rules can sometimes be motivated purely on their own merits, through an appeal to intuition, these technical aspects are complex enough that we feel obliged to seek some further justification from more basic principles. There are simpler notions of sums and products, closely related to constructors, which can be incorporated into typed lambda calculus, in both direct and pattern-based variations: the rules for sums and products in these calculi can be derived in a systematic way governed by certain symmetries of form. In order to motivate the reduction rules we propose below, we make a brief digression to explore how these rules might be developed successively from ones used in more basic calculi; the reader may wish to skip this digression on first reading, or if she finds such motivational material unnecessary.

Beginning of digression

We proceed according to the following plan: we first develop a simple calculus of sums and products with symmetric forms and argue for the correctness of its rules by appeal to symmetry considerations. Next we transform this calculus of sums and products systematically into a variation based on simple, binary patterns, i.e., without any appeal to data type definitions or constructors. We will derive the rules for this second calculus by generalizing the corresponding rules from the first one and then analyzing the consequences. Finally, we will propose rules for our constructor-based calculus based on analogy with the simpler patterns of the second calculus.

We could perhaps motivate formal conversion rules on the basis of semantic notions, but we feel that this approach would tend to bias us in the direction of model-specific choices which might not be the most general or most desirable properties to be set out in an axiomatic description. This is particularly true with respect to extensionality issues, about which there is no general agreement even in the case of the pure lambda calculus; in fact, it is these extensionality issues which we hope will benefit most from the analogical approach we take here.

Consider then a lambda calculus with binary sums and products, with syntax according to the following grammars; we use a set of indices, here including only the constants L and R, to suggest possible extensions to multiple or labelled versions of sums and products:

$$\begin{array}{lcl} \tau & ::= & \bigcirc \mid \tau\rightarrow\tau \mid \tau\times\tau \mid \tau+\tau \\ e & ::= & x \mid \lambda x{:}\tau.e \mid e\ e \mid e\ ,\ e \mid e\ \#\ i \mid e\uparrow i \mid e \parallel e \\ i & ::= & L \mid R \end{array}$$

The syntax of types includes an atomic form used as a basis and then binary forms for functions, products and sums. The syntax of expressions includes variables and then introduction and elimination forms for each of the three type constructors (functions, products and sums).

The familiar expression form (e,e') represents pairs, thus constituting the introduction form for products; similarly, the forms $(e\#i)$ represent left and right projections, for i=L or R, respectively, and constitute the corresponding elimination forms. The injections $(e\uparrow i)$ represent "tagged" values, which introduce sums, and the form $(e \parallel e')$ combines two functions, one for each injection, which will be applied to the appropriate tagged value, thus effecting elimination of sums by way of functions.

The formation rules for expressions include an axiom scheme for variables and pairs of rules for each of the type constructors, corresponding to the introduction and elimination forms for these types; we state only the new rules here for brevity, since the axiom for variables and rules for abstraction and application are the same as for the standard simply-typed lambda calculus:

$$[\times I] \qquad \frac{A \vdash e : \tau_L \qquad A \vdash d : \tau_R}{A \vdash (e,d) : \tau_L\times\tau_R}$$

$$[\times E] \qquad \frac{A \vdash e : \tau_L\times\tau_R}{A \vdash (e\ \#\ i) : \tau_i}$$

$$[+I] \qquad \frac{A \vdash e : \tau_i}{A \vdash (e\uparrow i) : \tau_L+\tau_R}$$

$$[+E] \qquad \frac{A \vdash f : \tau_L\rightarrow\rho \qquad A \vdash g : \tau_R\rightarrow\rho}{A \vdash (f \parallel g) : \tau_L+\tau_R\rightarrow\rho}$$

Note here how the elimination of sums inherently involves functions: alternative formulations are possible, for example one which uses variable-binding forms for sum elimination, but we prefer to keep all the binding issues associated with the lambda abstraction form.

The conversion rules for expressions also fall into two classes (apart from the usual rule for α-conversion, which we leave unspecified): the β rules, which describe computations proper and the η rules which specify that all terms of appropriate type are equivalent to some canonical form. We state all the conversion rules here, including those for functions, to facilitate comparison between them. Note that these equations should be taken as *typed* equations, i.e., that both sides of the equation should be well-formed with the same type in order for the equation to apply.

$$[\rightarrow\beta] \qquad (\lambda x{:}\tau.e)\ d = e\ [d/x]$$

$$[\rightarrow\eta] \qquad (\lambda x.\ f\ x) = f \qquad (x \notin FV(f))$$

$$[\times\beta] \qquad (e_L, e_R) \# i = e_i$$

$$[\times\eta] \qquad (e\#L, e\#R) = e$$

$$[+\beta] \qquad (f_L \parallel f_R)\ (e\uparrow i) = f_i\ e$$

$$[+\eta] \qquad ((\lambda x{:}\tau.x\uparrow L) \parallel (\lambda x{:}\sigma.x\uparrow R))\ e = e$$

Here again the elimination of sums is factored through function application, but the symmetry between the rules for the various type constructs should still be clear: the β rules specify how non-canonical forms are simplified in the presence of their inverse forms: for example, applications can be simplified when the function part is an abstraction, and products can be simplified in the presence of a projection. The η rules, on the other hand, specify equivalences between non-canonical and canonical terms in a sense inverse to the β rules: for example, a pair of projections from the same term is equivalent to that term, and the summed combination of two appropriate injection functions is equivalent to the identity function on the corresponding sum type.

Note that the restriction to typed equations is especially important in the interpretation of the η rules, since otherwise, for example, the η rule for products would allow that any term is equivalent to some pair of terms. In the context of the pure, untyped calculus, the η rule for functions has the affect that all terms are equivalent to functions; this property is not always desirable, however, since we may not wish, for example, to equate all algorithms which compute the same function. For discussions of extensionality in pure lambda calculus, see [Mey82, Bar84].

Next we consider a variation on this language which allows for arbitrary patterns built from type-constrained variables and binary pairings and injections; we keep the same syntax for types and indices, but use instead the following syntax for expressions, now with patterns as well:

$$\begin{aligned} e &::= x \mid (p \Rightarrow e)^+ \mid e\ e \mid e, e \mid e\uparrow i \\ p &::= x : \tau \mid p, p \mid p\uparrow i \end{aligned}$$

Essentially, we have dropped the elimination forms for products and sums in favor of pattern pattern forms which mirror the corresponding introduction forms (injection and pairing). We take advantage of essentially the same sort of transformation in the constructor-based pattern calculus, expressing elimination forms through patterns, although here we have simple binary injections instead of constructor applications which mediate more general recursive types.

As an example of the use of binary patterns based on sums and products alone, consider the following function which takes first a default value of the base type $\circ$, and then a pair of injected values; it returns the left-injected half of the pair if only one half is so formed, and returns the default value if both or neither half of the pair is a left-injection (we write the function here along with its type for reference):

$$\begin{array}{l} (a{:}\circ \Rightarrow \\ \quad (\\ \qquad ((x{:}\circ)\uparrow L,\ (y{:}\circ)\uparrow R) \Rightarrow x \\ \qquad \mid \\ \qquad ((x{:}\circ)\uparrow R,\ (y{:}\circ)\uparrow L) \Rightarrow y \\ \qquad \mid \\ \qquad (z{:}\circ + \circ) \Rightarrow a\ )\ ) \\ :\ \circ \rightarrow ((\circ + \circ) \times (\circ + \circ)) \rightarrow \circ \end{array}$$

In order to completely specify the formation rules for the expressions of this calculus we will need appropriate notions of coverage and compatibility; these can be defined in a fairly obvious way, based on our account of these issues for the constructor-based pattern calculus described earlier in the chapter. We will define explicitly here only the matching function, as follows:

$$\begin{aligned} &\mathcal{M}(x{:}\tau, e) = [e/x] \\ &\mathcal{M}((p,q),(e,d)) = \mathcal{M}(p,e) + \mathcal{M}(q,d) \\ &\mathcal{M}(p\uparrow i, e\uparrow i) = \mathcal{M}(p,e) \\ &\mathcal{M}(p,e) = \textbf{fail} \text{ otherwise} \end{aligned}$$

Given suitable definitions for definitions for coverage and compatibility, we can state a formation rule for pattern abstractions in this simplified pattern calculus; we refrain from repeating the axiom and rules for variables, applications, injections and pairs, since they are the same as for the non-pattern-based calculus of sums and products above.

$$[\rightarrow I] \qquad \frac{\begin{array}{c} \{p_1, \ldots, p_n\} \succ \tau \\ \{p_1, \ldots, p_n\} \text{ compatible} \\ \mathcal{A}(p_i) \vdash \mathcal{E}(p_i) : \tau \qquad (1 \leqslant i \leqslant n) \\ A + \mathcal{A}(p_i) \vdash e_i : \sigma \qquad (1 \leqslant i \leqslant n) \end{array}}{A \vdash (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) : \tau \rightarrow \sigma}$$

Note that this is precisely the pattern abstraction rule from our constructor-based pattern calculus: this identity results from our use of a common technology for patterns and matching in the two calculi and lends support to the idea that insight on conversion rules gleaned from this calculus will be applicable to the constructor-based pattern calculus.

End of digression

Turning then to the issue of reduction for expressions in the pattern calculus, we propose the following four rules, based on the six rules of the symmetric sums and products calculus above, as reasonable candidates—as discussed below, we will not ultimately accept all four candidates in exactly this form.

$$[\rightarrow\beta] \qquad (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n)\ d = e_i[\mathcal{M}(p_i,d)]$$

$$[\rightarrow\eta] \qquad (p_1 \Rightarrow f\ \mathcal{E}(p_1) \mid \cdots \mid p_n \Rightarrow f\ \mathcal{E}(p_n)) = f \qquad (FV(f) \ncong FV(p_i),\ 1 \leqslant i \leqslant n)$$

$$[\times\eta] \qquad f[(p \Rightarrow x_1)\ e, \ldots, (p \Rightarrow x_n)\ e] = e \qquad (\mathcal{E}(p) = f[x_1, \ldots, x_n])$$

$$[+\eta] \qquad (p_1 \Rightarrow \mathcal{E}(p_1) \mid \cdots \mid p_n \Rightarrow \mathcal{E}(p_n))\ e = e$$

The first of these rules captures the β rule for functions in the original (non-pattern-based) version of the calculus, but also captures the β rules for both sums and products as well, under the natural translation which takes, for example, "e # i"

to "$((x_L{:}\tau, y_L{:}\sigma) \Rightarrow x_i)\ e$" for fresh variables x_L and y_L. This consolidation of all three β rules into a single one stems from the fact that in the pattern-based calculus, all three type constructs are eliminated through function application; i.e., pairs and injections are "destructured" through matching with the corresponding forms of patterns. The consolidation of the three β rules into one rule leaves the remaining three η rules for our consideration.

The second conversion rule above is an attempt to translate the η rule for functions into the style of pattern abstractions: the translation is essentially straight-forward, although we note the more complex free-variable conditions required for patterns, which introduce multiple bound variables simultaneously, and the fact that the term f must be the same in all clauses of the abstraction (more precisely, f must be equivalent, in the sense of conversion, but this follows separately from the appropriate congruence rules for conversion).

The third conversion rule is an attempt to translate the usual η rule for products into the context of patterns; in the original calculus, the significance of the corresponding η rule is that any term of product type is equivalent to the pairing of the separate projections of its left and right parts. Here we try to generalize this to a rule we might informally describe as: "any term is equivalent to a term constructed in a particular shape from various components, when those components are extracted by patterns of the same particular shape". For example, we would like to be able to derive the following equivalence, which is an extension of the case represented by the η rule for products in the non-pattern-based calculus:

$$((p \Rightarrow x)\ e, ((p \Rightarrow y)\ e, (p \Rightarrow z)\ e)) = e$$
$$\text{where } p \equiv (x{:}\tau,\ (y{:}\sigma,\ z{:}\rho))$$

(note that by the requirement of well-typing for the equation, e must have a type of the form $\tau \times (\sigma \times \rho)$).

The generalized $[\times\eta]$ rule above unfortunately fails, in the sense that it is not sound with respect to our intended interpretation; more specifically, in the case of discriminative patterns, the rule would force a value of a particular type to be equivalent to any other value of that type, no matter whether it is constructed in the same particular discriminative fashion or not. For example, the following equivalence would also be forced by the corresponding rule in the constructor-based pattern calculus (there is no particular dependence on constructors here, it is simply easier to see the point we are making in the more familiar context of constructors):

$$\mathrm{succ}((\mathrm{succ}\ (n{:}\mathrm{Nat}) \Rightarrow n)\ e) = e$$

This last equation would have the significance that every well-typed term of type Nat is equivalent to a term of the form (succ e') for some term e'. Clearly we do not want such consequences, since under our intended interpretation some well-formed terms of type Nat will instead be constructed from zero. In the case of the sums and products calculus, the rule would have the consequence that all well-formed values of sum types would be simultaneously equivalent to values of both injection forms, thus collapsing the intended distinction between the two separate injections.

It might be desirable to preserve the effects of the generalized η rule over those non-disjunctive types, i.e., not involving sums or constructed types in the appropriate calculi: an appropriate restriction on the types can easily be devised along the lines of the type/data type distinction we will develop in Chapter 3. It may also be that all the desirable consequences of the generalized rule (i.e., those restricted to non-disjunctive types) can be derived from the original η rule for products and the congruence rules for conversion, although we have not verified this fact.

Finally, we turn to the fourth candidate rule: it is intended to capture the essence of the η rule for sums. Essentially, it states that any term is equivalent to a pattern-based destructuring and restructuring of the same term and is thus in some sense the inverse of the $[\times\eta]$ rule. This rule, however, seems to be derivable from the $[\rightarrow\beta]$ and the correctness property of pattern matching, provided that we interpret the β rule in a certain fashion. More specifically, we would normally take the $[\rightarrow\beta]$ rule to mean that when a match is possible, then the equivalence holds; but by interpreting the occurrence of i *existentially*, we can take it to mean that *every* term has *some* equivalent decomposition via patterns, in which case the rule takes on more of the usual flavor of the η rules.

Under this interpretation, we derive the $[+\eta]$ rule as follows; assume we have an application of the form of the left-hand side of the rule:

$$(p_1 \Rightarrow \mathcal{E}(p_1) \mid \cdots \mid p_n \Rightarrow \mathcal{E}(p_n))\ e$$

Under the existential interpretation of the meta-variable i in the $[\rightarrow\beta]$ rule, we have:

$$(p_1 \Rightarrow \mathcal{E}(p_1) \mid \cdots \mid p_n \Rightarrow \mathcal{E}(p_n))\ e = \mathcal{E}(p_i)[\mathcal{M}(\mathcal{E}(p_i), e)],$$
$$\text{some i}$$

But by the correctness property of pattern matching, we have:

$$\mathcal{E}(p_i)[\mathcal{M}(\mathcal{E}(p_i), e)] = e$$

thus establishing the import of the $[+\eta]$ rule as stated above.

Thus we take as the conversion rules for the constructor-based pattern calculus only the $[\rightarrow\beta]$ and $[\rightarrow\eta]$ rules from above, according to the interpretation of matching and subsumption we have given for constructor-based expressions and patterns. We may also wish to take a form of the $[\times\eta]$ rule, but restricted to non-disjunctive types (in the context of constructors, those which contain no use of occurrences of basic data types), or at the very least the simpler $[\times\eta]$ rule from the first calculus of sums and products, hoping to derive any more complex equivalences from this and the usual congruence rules.

As a concrete example of the application of the $[\times\eta]$ rule in the context of constructors, consider the following instance of the rule for any term f of appropriate type:

$$(\mathrm{nil} \Rightarrow f\ \mathrm{nil} \mid \mathrm{cons}(x{:}\mathrm{Nat}, y{:}\mathrm{NList}) \Rightarrow f\ (\mathrm{cons}(x,y))) = f$$

We should at this point consider rules that will determine the behavior of the remaining forms we of the constructor-based pattern calculus, namely the trivial value of unit type and the recursion construct.

The unit type 1 does not share the same duality of form common to function spaces, products and constructors, as it is introduced by the trivial value $*$ and "eliminated" in a degenerate sense by pattern abstraction. It is, nevertheless, our intention that the trivial value be the *only* value of unit type; this uniqueness could be expressed by an inference rule like the following:

$$\frac{A \vdash e : 1}{e = *}$$

Note, though, that the same affect can be achieved by a(n admittedly rather deceptive) well-typed rule of the form:

$$e = *$$

since the common well-typing requirement forces that the left-hand side have unit type.

Our informal interpretation of the recursion form can be formalized by a conversion rule which allows for the "unfolding" of one level of recursion as follows:

$$\mu x{:}\tau.e = e\ [\mu x{:}\tau.e/x]$$

Having discussed the individual "one-step" reduction or conversion rules, we can define their extension to a transitively-closed notion of many-step reduction in the usual fashion. A certain amount of extra care is needed, however, in developing a reduction strategy for the pattern calculus, because of interactions between the choice of a pattern-matching regimen and choices about reduction order. Consider for example the behavior of the following term under first-match regimen:

$$(a{:}\tau\times\sigma \Rightarrow P\ a)\ (e,d)$$
$$\text{where } P \equiv ((x{:}\tau, y{:}\sigma) \Rightarrow \text{true} \mid z{:}\tau\times\sigma \Rightarrow \text{false})$$

If we reduce the application of P to the variable a first , we get the following sequence of reductions, leading to the result false:

$$(a{:}\tau\times\sigma \Rightarrow P\ a)\ (e,d)$$
$$\twoheadrightarrow (a{:}\tau\times\sigma \Rightarrow \text{false})\ (e,d)$$
$$\twoheadrightarrow \text{false}$$

If, on the other hand, we choose to reduce the application of the outer pattern abstraction to the pair (e,d) first, we get the following sequence of reductions, leading to the result true:

$$(a{:}\tau\times\sigma \Rightarrow P\ a)\ (e,d)$$
$$\twoheadrightarrow P\ (e,d)$$
$$\twoheadrightarrow \text{true}$$

The underlying problem in choosing the first reduction order is that we have not exposed enough of the ultimate structure of the argument (the variable a) to make a proper choice of matching pattern; that is, since the structure of an actual argument determines the choice of matching pattern to be used in reducing an application, we should take care to delay reductions of applications until a sufficient amount of argument structure has been exposed by other reductions. This restriction can be expressed easily in terms of the subsumption relation between patterns and expressions.

Note that the same problem will occur under the best-first matching regimen as well, since the order of reductions taken will affect the question of which match is best. The problem does arise under the only-match regimen, however, since there is never any ambiguity over which match to choose in this case.

Our inclusion of a recursion construct also has implications for reduction strategy, since it introduces the possibility of non-terminating reduction sequences. To avoid non-termination, we can simply choose to expose as little of the structure of a recursion expression as possible, though some reduction will be necessary in cases where a recursion expression occurs as the actual argument of a pattern abstraction (i.e., the considerations above will take precedent).

Since the issues surrounding reduction in the presence of recursion are rather complex (consider for example the interactions with our $[\rightarrow\eta]$ rule), we will limit our discussion to the previous development of appropriate one-step rules and to our brief remarks about the impact of the choice of matching-regimen on reduction strategies—we leave any further development of the operational semantics of the pattern calculus to further research.

Chapter 2
Higher-order Analytical Polymorphism and Type Patterns

In the last chapter we developed a formal account of patterns over values; now we wish to apply the ideas of this formalism to the development of a notion of pattern abstraction over types. Just as we can view value patterns as an extension of lambda abstraction on values in the simply-typed case, so we can view type patterns as an extension of the explicit abstraction operation on types found in the second-order and higher-order polymorphic lambda calculi. We introduce in this chapter the basic notions required for a higher-order calculus extended with both value-pattern abstractions and type-pattern abstractions, with the latter providing the more general form of analytical polymorphism we need to express functions which operate on arbitrary data structures. We continue the development of this formal aspects of the calculus over the next two chapters and present some extended examples and applications in Chapter 5.

Although the explicitly-typed pattern calculus is quite expressive, it does not form a reasonable basis for a programming language since, like the standard explicit lambda calculi, it foists too great a burden on the programmer by requiring her to supply numerous explicit type constraints and instantiations throughout her code. Since applications to type systems in realistic languages are our ultimate goals, we use the formal calculus of these next chapters rather as a means of clarifying various aspects of type patterns, thus developing a basis for the introduction of an implicitly-typed language in Chapter 6.

As in Chapter 1, we attempt to present the higher-order pattern calculus of this chapter in a natural rhetorical order; this order requires a certain amount of backtracking in order to properly motivate concepts such as, e.g., the coercive partial ordering on kinds. Appendix B gathers all of the formal definitions for the higher-order language in an order and style better suited to reference purposes: readers who would rather see the formalism in its final form are directed to look there first.

2.1 The standard polymorphic lambda calculus

The basic innovations in explicitly-typed, polymorphic lambda calculi are the addition of variables to the syntax of types and the introduction of a form of abstraction over these variables in a manner that parallels ordinary lambda abstraction. These extensions allow us to express *polymorphic* values, which may have several compatible typings, whereas in the simply-typed calculus we could only express *monomorphic* values, those with a single, fixed type. Consider, for example, that in the simply-typed calculus we can write the identity function over values of type τ, for any particular type τ, as $(\lambda x{:}\tau.x)$; we cannot, however, write an identity function that captures the generality of the untyped term $(\lambda x.x)$, which can be applied to any argument. Of course, part of the value of a type system is that it forbids the formation of problematic terms whose reduction might lead to errors or non-termination; in the case of the identity function, however, there is clearly no such danger. We are thus moved to seek a more flexible type system which will allow us to express reasonable functions like a general identity, while still preventing the formation of arbitrary (and possibly nonsensical) terms.

In a polymorphic calculus, we express the identity function over an arbitrary type by constraining the argument x with a *type variable*; for example, letting α be a type variable, we can form the term $(\lambda x{:}\alpha.x)$. In order to manage the introduction and use of type variables like α, we bind them using a new form of lambda abstraction which abstracts an expression with respect to a type variable rather than a value variable (we will use a capital lambda to avoid confusion between the two forms of abstraction; some authors use the same letter for both forms). For example, we form the polymorphic version of the identity function as follows, presented here along with its type:

$$(\Lambda\alpha.(\lambda x{:}\alpha.x)) : \forall\alpha.\alpha\to\alpha$$

The symbol $\forall$ in the type given to this term is a quantifier which also binds the type variable α, just as the Λ does. We use the usual symbol for universal quantification to convey the intuition that the term has the function type $\alpha\to\alpha$ for *all* possible instantiations of the type variable α to particular types such as Nat or Bool (some authors use the symbols Π or Δ in place of $\forall$).

We resolve type abstractions in polymorphic calculi through a form of application to actual type arguments, just as we resolved value abstractions in the untyped and simply-typed calculi by function application to actual value arguments. To avoid confusion between the two forms of application, we call the new form *instantiation* and write it with a down-arrow, as in $e{\downarrow}\tau$ (some authors use juxtaposition or brackets around the type argument). A new β reduction rule, similar to the β rule of the untyped and simply-typed calculi, describes the reduction of instantiated type abstractions through the substitution of actual type arguments for type variables:

$$(\Lambda\alpha.e){\downarrow}\tau \;\to_\beta\; e\,[\tau/\alpha]$$

As for β reduction in the calculi of Chapter 1, we must take care here to avoid capture of bound variables during the substitution.

To continue the example from above, we can explicitly instantiate the polymorphic identity function to a specific type, say the type Nat of natural numbers, and then reduce it using the above rule to obtain the identity function on that type:

$$(\Lambda\alpha.(\lambda x{:}\alpha.x)){\downarrow}\mathrm{Nat} \;\to_\beta\; (\lambda x{:}\mathrm{Nat}.x)$$

Just as the formation rule for function applications ensures that only functions are applied only to (appropriately typed) arguments, a type instantiation rule in the polymorphic calculus is used to ensure that only polymorphically-typed terms are instantiated with type arguments.

The second-order polymorphic calculus includes these three forms, namely type variables, type abstraction and type instantiation, and was independently developed by Girard and Reynolds [Gir72, Rey74]. Girard also studied a more general, higher-order polymorphic calculus which includes, in addition to these "second-order" features, forms of type abstraction and application which allow the expression of functions from types to types. In order to ensure that expressions involving the application of type functions to types are well-formed requires the introduction of "types for types", a third level of terms which we will call *kinds*. For example, the following term represents a function from types to their diagonal products,

also types, and is thus given a kind representing functions from types to types:

$$(\lambda\alpha{:}T.\ \alpha\times\alpha) : T\to T$$

(here we have used the same symbol λ for function abstraction over types as we have used for function abstraction over values; context will have to suffice to disambiguate the two).

The formation of types is predicated on the existence of suitably-related kinds, just as the formation of expressions is predicated on the existence of suitably related types (the formation of kinds, at the highest level, is context-free). We distinguish between types proper, i.e., those which are the types of expressions, and the more general *type functionals*, which include also functions from types to types, etc. The subset of proper types will be characterized by the fact that have kind T.

The capability to express functions over types is useful in applications, such as our own, which involve polymorphic data structures, i.e., structures which may contain components of some fixed but arbitrary type: structures of this sort can be viewed as functions which take type arguments and return type results. For example, the data structure for lists of arbitrary components can be construed as a function which takes a type argument (such as Nat) and returns a type result (e.g., List(Nat)), which is appropriate to all lists whose components are of the type represented by the argument. Note that this conception of data structures as type functions does not directly allow for structures which contain different components of different types: the latter are called *heterogeneous* structures, in contrast with the *homogeneous* ones we study here.

The higher-order calculus is thus organized into three levels, including kinds, types and type functionals, and value expressions. The following diagram illustrates the formation relations between levels with vertical arrows:

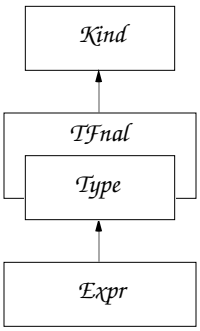

Figure 2.1.1: The three levels of the higher-order calculus

Note that whereas the technical priority of the three sorts runs from the top of the diagram downwards, the more natural rhetorical or motivational order runs from the bottom of the chart upwards, in the sense that the expressions are the sort of primary interest, and the types and ultimately the kinds are necessary only to preserve a certain amount of order in the formation of expressions.

The higher-order calculus also incorporates three different sorts of lambda abstractions: value-function abstractions, which bind value variables in value expressions; polymorphic type abstractions, which bind type variables in value expressions; and type-function abstractions, which bind type variables in type expressions. A final form of abstraction, which binds value variables in type expressions, is a natural candidate to complete the set: such abstractions are in fact studied in systems of *dependent* types. In these systems, the strict stratification of levels between value terms and types breaks down, as do various other properties which may be important in programming language applications. For these reasons we will refrain from further consideration of dependent types, except for some short remarks later about the relationship between algebraic constructors and dependent sums. A general, schematic way of describing the various forms of lambda abstraction and the calculi derived from them (as well as many others) has recently been developed by Barendregt [BH90, Bar9?].

With this brief introduction to the concepts of polymorphism aside, we now proceed to a more formal presentation of the higher-order lambda calculus. As mentioned above, the syntax of this calculus is arranged into three levels, including now also kinds in addition to the familiar types and expressions. The syntax of kinds is essentially context-free, as was the syntax of types in Chapter 1. It includes the constant T, representing the "kind of all types", and the function space constructor, $\to$ (we will use the same symbol for the function space in both the languages of types and kinds; some authors use a different arrow symbol, e.g., $\Rightarrow$, for the latter). The syntax of types and type functionals includes variables, function spaces and polymorphic quantifications. Expressions include the familiar variables, value abstractions and applications, but now also type abstractions and instantiations.

$$\begin{aligned}
\kappa &::= T \mid \kappa\to\kappa\\
\theta &::= \alpha \mid \theta\to\theta \mid \forall\alpha{:}\kappa.\theta \mid \lambda\alpha{:}\kappa.\theta \mid \theta\ \theta\\
e &::= x \mid \lambda x{:}\tau.e \mid e\ e \mid \Lambda\alpha{:}\kappa.e \mid e{\downarrow}\theta
\end{aligned}$$

Here we use the meta-variable θ to range over the type functionals; we will continue to refer loosely to "types" in general when we do not need to draw attention to the distinction.

In saying that kinds serve as the "types of types", we mean that the formation of types is predicated on the existence of suitably related kinds. In fact, types now serve in two distinct rôles, a descriptive one, in which they constrain the formation of expressions, and a more active one, in which they signify functions over types and are subjects of the abstractions and reductions of this (higher) level.

Since types now include essentially the forms of the simply-typed lambda calculus, their syntax is now no longer context-free: as we have stated above, their formation is dependent on suitably-related kinds. Thus the three levels of the calculus exhibit a technical priority, with kinds as the (context-free) basis, followed by types and then by value expressions.

The formation rules for types are also essentially the same as those for the simply-typed lambda calculus; we give just two rules here, for the functional and universally-quantified types, since these forms do not have direct correspondents in the simply-typed language. Note that we need a new sort of assumptions, mapping type variables to their kinds, to give context to free type variables; we will use A_T to range over these assumptions; thus we have $A_T ::= \{\alpha{:}\kappa, \ldots\}$:

$$\frac{A_T \vdash \tau : T \qquad A_T \vdash \sigma : T}{A_T \vdash (\tau\to\sigma) : T}$$

$$\frac{A_T+\{\alpha:\kappa\} \vdash \tau : T}{A_T \vdash (\forall\alpha:\kappa.\tau) : T}$$

(NB: we use the *index* T in the meta-language to distinguish type-level assumptions; this should not be confused with out use of the *constant* T in the object language as the kind of types). Note that both of these forms are given the kind T, i.e., they constitute proper types, as opposed to type functionals; we will use the meta-variables τ and σ to range over proper types, reserving θ and ω (possibly with subscripts, etc.) for the more general type functionals.

With regard to reduction and conversion, the situation is again similar to that of the simply-typed calculus: we give rules for beta and eta reduction on types as follows, where the substitution operation $\theta[\omega/\alpha]$ and the free variable operation on types $FV(\theta)$ are defined as the analogous operations on value variables and expressions in the simply-typed calculus.

$$(\lambda\alpha:\kappa.\theta)\ \omega \ \rightarrow_\beta\ \theta\ [\omega/\alpha]$$
$$(\lambda\alpha:\kappa.\theta\ \alpha) \ \rightarrow_\eta\ \theta\ , \quad \text{if } \alpha\notin FV(\theta)$$

In order to express formation for value expressions we will need the usual notion of an assumption which provides types for free variables; in addition, since type variables may appear in value expressions as constraints or instantiations, we also need the context of the type assumptions from above. In fact, since the types in the range of the value assumptions will in general include type variables, we will need type assumptions as contexts for the formation of value assumptions (note that in the simply-typed calculus, the formation of assumptions was context-free). We approach these issues by defining combined assumptions as pairs:

$$A \ ::= \ \langle A_T,\ \{x:\tau,\ \ldots\}\rangle$$

We can then use the following rule to express the formation of these combined assumptions:

$$\frac{A_T \vdash \tau_1 : T \quad \cdots \quad A_T \vdash \tau_n : T}{\vdash \langle A_T, \{x_1:\tau_1, \ldots, x_n:\tau_n\}\rangle\ \text{assm}}$$

Now, when describing the formation of expressions, we will use these combined assumptions to give context to both kinds of variables. We will also take care to ensure that the formation of assumptions is preserved by the rules, as we do in the following rule for formation of lambda abstractions:

$$\frac{A_T \vdash \tau : T \qquad A+_V\{x:\tau\} \vdash e : \sigma}{A \vdash (\lambda x:\tau.e) : \tau\rightarrow\sigma}$$

(here we have used indices T and V to project the appropriate half of a pair of assumptions and to restrict the merge operation in a similar way).

The following rules express the formation constraints for type abstraction and instantiation expressions:

$$\frac{A+_T\{\alpha:\kappa\} \vdash e : \tau}{A \vdash (\Lambda\alpha:\kappa.e) : \forall\alpha:\kappa.\tau} \quad (\alpha\notin FV(A))$$

$$\frac{A \vdash e : \forall\alpha:\kappa.\tau \qquad A_T \vdash \theta : \kappa}{A \vdash (e\downarrow\theta) : \tau[\theta/\alpha]}$$

The rule for formation of abstractions is very similar to the rule for formation of value abstractions in the simply-typed calculus, except that the universally quantified type in the consequent binds a type variable in its body. The rule for instantiation is in some sense a combination of the usual rule for application and the beta reduction rule, the latter in that it involves substitution of the instantiated type for all free occurrences of the quantified variable.

The side condition on the formation rule for type abstractions warrants some discussion: intuitively, it would only be sound to the consider a type variable universal if there were no dependence on that variable in the current context. The condition prevents us from forming such a quantification with respect to a variable which appears in the constraints on some value variable in the current assumptions; i.e., we have:

$$FV(\{x_1:\tau_1, \ldots, x_n:\tau_n\}) = \bigcup_{1\leqslant i\leqslant n} FV(\tau_i)$$

The only other significant difference between the formation rules for the higher-order calculus, relative to the simply-typed one, is that we must allow for the fact that types are now subject to conversion; i.e., they may be represented by type functional terms not in normal form. We use the following *subject-conversion* rule to express that expressions can be typed with convertible types:

$$\frac{A \vdash e : \tau \qquad \tau = \sigma}{A \vdash e : \sigma}$$

The notion of reduction for well-typed value expressions includes the usual beta and eta rules for value abstractions, but also rules for beta and eta reduction on type abstractions, as follows:

$$(\Lambda\alpha:\kappa.e)\ \tau \ \rightarrow_{\beta'}\ e\ [\tau/\alpha]$$
$$(\Lambda\alpha:\kappa.f\downarrow\alpha) \ \rightarrow_{\eta'}\ f\ , \quad \text{if } \alpha\notin FV(f)$$

The eta rule depends on an operation for finding the free type variables of an expression, just as the usual eta rule for value abstractions depends on an operation for finding the free value variables of an expression. We can succinctly define these operations by combining them into a single operation returning a pair of results, and then projecting the necessary component:

$$FV(x) = \langle\{\}, \{x\}\rangle$$
$$FV(\lambda x:\tau.e) = FV(e) \setminus_V \{x\} \cup_T FV(\tau)$$
$$FV(f\ e) = FV(f) \cup FV(e)$$
$$FV(\Lambda\alpha:\kappa.e) = FV(e) \setminus_T \{\alpha\}$$
$$FV(e\downarrow\theta) = FV(e) \cup_T FV(\theta)$$

(here we have used indices T and V on set operations to indicate that they should be applied only to the corresponding half of a pair of free variable sets, leaving the other half unchanged).

Surprisingly, this more complex and expressive calculus still retains the best properties of the simply-typed calculus: the formation relation between terms and types is decidable and computable, in the sense that a type can be computed for a

term; typing is strong, in that all well-typed terms can be sensibly reduced, and static, in that types can be erased or ignored for the purposes of reduction; and finally, reduction of terms is strongly-normalizing, so that the reduction of well-typed terms always terminates in a normal form. We will not prove any of these results (some are surprisingly difficult), but instead refer the reader to the literature on the subject [Gir72, FLO83, Gall90].

In the pure form in which we have presented it, the higher-order calculus might seem ill-suited to practical applications in programming languages, since it lacks the usual base types and constants one would normally want for these purposes. In fact, though, many of the desired types and constants can be encoded in the calculus as it stands. For example, consider the following definitions of a boolean type and its truth values:

$$\text{Bool} \equiv \forall\alpha{:}\text{T}.\ \alpha\to\alpha\to\alpha$$
$$\text{true} \equiv (\Lambda\alpha{:}\text{T}.\ \lambda x{:}\alpha.\ \lambda y{:}\alpha.\ x)$$
$$\text{false} \equiv (\Lambda\alpha{:}\text{T}.\ \lambda x{:}\alpha.\ \lambda y{:}\alpha.\ y)$$
$$\textbf{if}\ b\ \textbf{then}\ e\ \textbf{else}\ d \equiv b{\downarrow}\tau\ e\ d$$

These constants true and false can be shown to be the only closed normal-form terms of type Bool and they can be used, as here in the definition of the conditional, to select between two alternatives of common type (in this case, τ). Study has shown that this sort of encoding can be used to represent all of the usual data types and structures in a general way, including natural numbers, polymorphic lists, etc. (see, e.g., the work of Berarducci et al. [BöBe86]). In fact, via an appropriate representation of natural numbers, the higher-order calculus is expressive enough to allow the definition and manipulation of any function which can be proved total in higher-order arithmetic [Gir72].

For the purposes of programming practice, however, we would like to incorporate arbitrary recursions in addition to just the total functions which can be expressed in the pure higher-order calculus. Furthermore, the sorts of encodings described above are neither very intuitive nor very efficient., and so we are moved to consider additional features which will allow for a more natural expression of basic data types.

Our approach will be based, of course, on the algebraic types and recursion form used in the simply-typed calculus developed in Chapter 1. In the context of a higher-order calculus, we will be able to incorporate not only basic data types such as the natural numbers, but also generic lists such as were described earlier in this section. We will also introduce a notion of patterns over types, which will allow us to express the analytical polymorphism we motivated in the Introduction. In the next section we begin our description of the higher-order pattern calculus with a description of kinds and type functionals which comprise its uppermost levels of terms.

Our higher-order pattern calculus is organized into three levels, just as the higher-order lambda calculus of the previous section, including kinds, types and value expressions. Once again, the lower levels of the calculus are predicated upon the higher ones, in the sense that the formation of terms at lower levels is predicated on the existence of suitably related terms of the higher levels.

Our language of kinds include a constant T for the kind of all types, unit and product constructors for expressing multiple arguments, and a function space arrow (here κ ranges over arbitrary kinds):

$$\kappa\ ::=\ 1\ \mid\ \text{T}\ \mid\ \kappa\times\kappa\ \mid\ \kappa\to\kappa$$

(in the next Chapter we add *kind variables* to this language, but we defer to that point when we can make proper motivation).

The forms of type functionals include atomic type constants, constructors and variables, pairing, function application and lambda abstraction, and universal quantification:

$$\theta\ ::=\ \chi\ \mid\ \text{S}\ \mid\ \Sigma\ \mid\ \alpha\ \mid\ \theta,\theta\ \mid\ \theta\ \theta\ \mid\ \lambda\pi.\theta\ \mid\ \forall\pi.\theta$$

As before, we use θ, ω, τ, and σ for type functionals, reserving τ and σ for proper types (those of kind T). We will use any of the symbols α, β, ε, φ, ψ for type variables, reserving some symbols for special purposes, e.g., α and β for variables of kind T and φ and ψ for functions over various kinds. Of the remaining atomic forms, χ ranges over type constants, S over the special class of type constructors (such as, e.g., List or Tree) and Σ over a class of type constructor variables, whose motivation and description we defer to Chapter 3.

Our lambda abstractions and quantifications over types employ a degenerate form of type patterns called *(type) parameter patterns*; parameter patterns allow for a natural way of introducing multiple formal parameters, including the trivial case of no parameters at all (here π ranges over parameter patterns):

$$\pi\ ::=\ *\ \mid\ \alpha : \kappa\ \mid\ \pi\ ,\ \pi$$

These patterns are degenerate both in that they lack some of the forms of the full type patterns we will introduce later, but also in that they are *non-discriminative*; that is, a single parameter pattern is sufficient to match any argument of the corresponding kind (for arbitrary, *discriminative* patterns, several patterns are generally required for full coverage, as in the case of the value patterns of Chapter 1).

By way of some examples, the following type functional expressions, shown here with their kinds, represent a trivial function (i.e., of no arguments) returning the type of natural numbers, a function of one type argument returning the "diagonal product" type for that argument, and a function of two arguments, one a type function and one a type, returning a function type built from these arguments:

$$(\lambda *.\ \text{Nat}) : 1\to\text{T}$$
$$(\lambda\alpha{:}\text{T}.\ \alpha\times\alpha) : \text{T}\to\text{T}$$
$$(\lambda\varphi{:}\text{T}\to\text{T},\ \alpha{:}\text{T}.\ \alpha\to\varphi\ \alpha) : (\text{T}\to\text{T})\times\text{T}\to\text{T}$$

Before proceeding with a description of the formation and conversion rules for type functionals in Section 2.3, we first digress in the next section to discuss the nature and use of the type constructors introduced into the syntax of types above.

2.2 Higher-order data structures and constructor settings

In the syntax of types and type functionals we make a distinction between type *constants* and type *constructors:* this distinction is analogous to the distinction between constants and constructors at the value level of the simply-typed pattern calculus described in Chapter 1. More specifically, whereas constants include more arbitrary constructs with individual rules for reduction, constructors behave in the usual initial-algebraic fashion, applied as functions to build terms, but with no computational (reductive) import.

In our language of types, the constants will include at least the unit type 1, the trivial type $*$ and the product and function space operators $\times$ and $\rightarrow$; other constants might also be included, but their behavior would have to be separately specified. Note that we could introduce a general class of constants at the level of kinds also, thus allowing us to construe the unit kind and the product and function space operators for kinds as constants. To do so would, however, make the syntax of kinds context-sensitive, if only to the point of requiring ranks for kind constants, thus de-emphasizing the rôle of kinds as the limit of our hierarchy of term levels. We are motivated to take the current approach for reasons as much rhetorical as technical.

Our use of constructors is again based on the idea of algebraic data type definitions, although we will now allow for data *structures*, such as lists and trees, which take a type component parameter. In addition to the usual first-order structures definable in most programming languages, we will allow for *higher-order data structures*, i.e., structures which take type function arguments; such structures are not available in any current functional language. Although we will not use explicit declarations of data types and structures, we present the following example declarations, written in the style of Standard ML, in order to provide some context.

Nat = zero of 1 | succ of Nat
List α = nil of 1 | cons of $\alpha \times$List α
WTree α = wnode of Nat$\times\alpha$ | grow of List(WTree α)
STree(Σ,α) = leaf of α | spread of Σ(STree(Σ,α))

The first of these declarations defines the usual "Peano-style" natural numbers; it yields a basic data type with no type arguments and the usual value constructors for zero and successor. The second declaration is the familiar recursive definition for lists, yielding List as type constructor of one type argument, and the value constructors nil and cons for empty and structured lists, respectively. The third declaration is another first-order data structure definition, now for a form of weighted tree: it yields the type constructor WTree, which takes a single type parameter and returns the type of weighted trees with components of that type. Its value constructors include wnode, which takes a component and a natural number weight and returns a weighted leaf node, and grow, which constructs a weighted tree from a list of weighted trees, understood as its children.

The final declaration is for a higher-order data structure, since the variable Σ is intended to range over other data structures; it yields the type constructor STree, which takes a pair of arguments, one a type constructor and one a basic type. Note that the parameter structure Σ is applied in the type of the spread constructor so that a well-typed use of spread will take a Σ structure as argument. and return a structure of type STree (Σ,α), for some α (the leaf constructor builds a leaf node from an argument of the component type α).

In order to provide a "kinding" context for constants and constructors, we again use special algebraic signatures in place of the preceding style of declarations. In contrast to the simply-typed settings of Chapter 1, we use here (*higher-order) settings*, which associate a kinds with each type constants and type constructors. Our settings will actually be conglomerate structures, similar to the assumptions we used for type and value variables in the higher-order lambda calculus, including also associations which provide types to value constants and constructors. Using indices to distinguish the various components, we have thus:

$$C_T ::= \langle C_{TC}, C_{TS} \rangle$$
$$C_V ::= \langle C_{VC}, C_{VK} \rangle$$
$$C ::= \langle C_T, C_V \rangle$$

where C_T ranges over type-level settings, C_{TC} over associations for type constants, C_{TS} over associations for type constructors, C_V over value-level settings, C_{VC} over associations for value constants, C_{VK} over associations for value constructors and C over the complete conglomerate settings. We will occasionally refer to the two major components of settings as the upper and lower halves.

For the sake of uniformity, we construe all type constructors as functions, taking the basic ones to be degenerate functions on the unit kind (e.g., Nat:$1\rightarrow$T). Value constructors will be similarly construed as functions of a single argument, but will also now be polymorphic in a type argument; for technical reasons we will discuss below, it is easiest to express the necessary information associated with value constructors as a pair consisting of a type function and a type constructor. Thus we have the following more specific syntax for the various associations which constitute settings:

$$C_{TC} ::= \{\chi{:}\kappa, \ldots\}$$
$$C_{TS} ::= \{S{:}\kappa\rightarrow T, \ldots\}$$
$$C_{VC} ::= \{c{:}\tau, \ldots\}$$
$$C_{VK} ::= \{k{:}\theta\mapsto S, \ldots\}$$

where c and k range over value constants and constructors, respectively. Note that type constructors always produce a type, though they may take higher-order arguments, and that value constructors always produce a value of an associated type constructor.

We will take the following type and value constant associations as our initial constant settings, providing kinds for the standard type constants mentioned above, as well as a type for for the (single) value constant $*$:

$$C_{TC_0} \equiv \{*{:}1,\ 1{:}T,\ \times{:}T\times T\rightarrow T,\ \rightarrow{:}T\times T\rightarrow T\}$$
$$C_{VC_0} \equiv \{*{:}1\}$$

We warn the reader that we have used the same symbols for units, products and function spaces at both the levels of kinds and types, and the same symbol for the trivial value of unit type as for the trivial type of unit kind. In particular, this leads to the circular-looking $\times{:}T\times T\rightarrow T$ and $\rightarrow{:}T\times T\rightarrow T$. Note, however, that the occurrences of these constants on the right sides of the colon are forms of kinds, whereas the ones on the left are the type constants. A product of types takes two type arguments and returns a type result, whereas the product of kinds is used to express the kind of the pair of type arguments. Likewise, the trivial value has unit type ($*{:}1$), but the trivial type is a proper type ($1{:}T$), whereas the trivial type has a unit kind ($*{:}1$).

In order to specify a particular higher-order pattern calculus, we simply extend this initial setting with associations for type constructors and their kinds and for value constructors and their types, corresponding to the desired user-defined algebraic types and data structures, to obtain a fixed, global type setting. We will consider the complete setting as a parameter to the language definition, fixing a particular choice of data types and structures globally; in actual practice, type and structure definitions would normally be scoped, but as in

Chapter 1 we prefer the use of a global setting in order to avoid the peripheral issues raised by scoping (for a description of the problems of scoped data type and constructor definitions, see *The Definition of Standard ML*[MTH90]).

Since the formation of kinds is context-free, the formation of the upper half of a setting is also context-free, modulo the usual restriction that associations be functions (i.e., that they associate unique terms to the symbols in their domain). But since the lower half of a setting includes types, whose formation is dependent on kinds, we will need to give explicit formation rules for this component of the setting. We expect the types and type functions appearing in the range of a value setting to be closed, i.e., without free occurrences of variables, so we do not need type assumptions to appear explicitly in the formation rules for value settings; rather, we simply use empty assumptions in the antecedent formation judgements for the appropriate types.

Note, however, that we *will* need the context of type settings in the formation rules for value settings, since we expect that the type functions in the range of the lower half of the setting will refer to type constructors in the upper half—this is necessary, for example, to capture recursive data type definitions. Thus a complete well-formed setting $\langle C_T, C_V \rangle$ will consists of an upper half C_T and a lower half C_V which is well-formed with respect to C_T. Respecting our policy of considering settings as global entities, we will not include type settings explicitly in the formation judgments for value settings

Given these remarks on the inter-dependence of the various sorts with respect to setting formation, we will need the following three judgement forms for value constant settings, value constructor settings and conglomerate settings as wholes:

$\vdash C_{VC}$ vcsett
$\vdash C_{VK}$ vksett
$\vdash C$ sett

The following rule describes the formation of value constant settings; essentially, all the types appearing in the range of the association must be proper types with respect to the empty set of assumptions (and the implicit, global type setting):

$$\frac{\{\} \vdash \tau_1 : T \quad \cdots \quad \{\} \vdash \tau_n : T}{\vdash \{c_1{:}\tau_1, \ldots, c_n{:}\tau_n\} \text{ vcsett}}$$

This next rule describes the formation of value constructor settings; for constructors we require that their associated type functionals have the same kind as is given to the corresponding type constructor by the global type setting (note that by our conventions on associations, the projection operation will fail, and thus the rule not apply, should any of the type constructors mentioned in the range of the value setting not appear in the domain of the type setting):

$$\frac{\{\} \vdash \theta_1 : C_{TS}(S_1) \quad \ldots \quad \{\} \vdash \theta_n : C_{TS}(S_n)}{\vdash \{k_1{:}\theta_1 \mapsto S_1, \ldots, k_n{:}\theta_n \mapsto S_n\} \text{ vksett}}$$

Finally, the rule for formation of complete settings is as follows (we could of course state all three of these as a single rule, but we refrain from doing so on both rhetorical and formatting grounds):

$$\frac{\vdash C_{VC} \text{ vcsett} \quad \vdash C_{VK} \text{ vksett}}{\vdash \langle C_T, \langle C_{VC}, C_{VK} \rangle \rangle \text{ setting}}$$

Given a well-formed constructor setting as a global parameter, the following axiom schemes are used to give kinds to type constants and type constructors as well-formed type functional terms (we will address the structure of assumptions in detail in the next section, but they are essentially just the combined type and value assumptions of the higher-order lambda calculus):

$$A_T \vdash \chi : C_{TC}(\chi)$$

$$A_T \vdash S : C_{TS}(S)$$

As usual, our conventions on projection from associations will prevent the use of these axioms in cases where the constant or constructor do not appear in the appropriate setting domain.

With regard to the level of values, we use a similar axiom to give types to value constants, as follows (we will give more details on expression formation in Chapter 3, following a discussion of discriminative type patterns):

$$A \vdash c : C_{VC}(c)$$

The rule for value constructors is a little more interesting, as it shows how each value constructor is considered a polymorphic function of a single (possibly structured) argument, whose result has the type of the associated type constructor:

$$\frac{C_V(k) = \theta \mapsto S}{A \vdash k : \forall \alpha{:}\kappa.\ \theta\ \alpha \to S\ \alpha}$$

Note that the polymorphic parameters of the constructor correspond to the type parameters of the associated argument functions, and that the associated type structure has components in the same polymorphic parameters.

As in the last chapter, we will systematically abbreviate to avoid explicitly writing trivial arguments to type constructors and trivial instantiations for polymorphic value constructors, as follows:

$S \equiv S\ *$, for $S{:}1 \to \kappa$
$k \equiv k{\downarrow}*$, for $k{:}\forall *.\tau$

We will complete this section with brief examples of constructor settings for polymorphic data structures and of the use of the associated value constructors. In particular, we will draw our inspiration from the sample ML-style declarations we gave at the beginning of the section. The following constructor settings correspond to those declarations and, in combination with the standard constant settings we gave earlier, would constitute a complete setting for a particular version of the higher-order pattern calculus:

$C_{TS_0} \equiv$
$\{ \text{Nat}{:}1 \to T,\ \text{List}{:}T \to T,\ \text{WTree}{:}T \to T,$
$\ \ \text{STree}{:}(T \to T) \times T \to T \}$
$C_{VK_0} \equiv$
$\{\ \text{zero}{:}(\lambda *.1) \mapsto \text{Nat},\ \text{succ}{:}\text{Nat} \mapsto \text{Nat},$

nil:(λα:T.1)↦ List, cons:(λα:T.α×List α)↦ List,
wnode:(λα:T.Nat×α)↦ WTree,
grow:(λα:T.List (WTree α))↦ WTree,
leaf:(λΣ:T→T, α:T.1)↦ STree,
cons:(λΣ:T→T, α:T.Σ(STree(Σ,α)))↦ STree }

Note how the separation of the constructor setting into upper and lower halves allows for a straightforward specification of recursive and even mutually recursive types, since all the type constructors are given separately, first, and are only then used in the argument and result types of value constructors in the lower half. Thus no explicit recursion is necessary in constructor settings, which simplifies their formation rules.

Although we have not yet discussed the syntax and formation rules for expressions, it may be useful at this point to have a concrete example of the use of constructors; the syntax of expressions is, in any case, similar to the syntax of expressions in the higher-order polymorphic calculus, at least with regard to application, instantiation and pairing. Relative to the above setting, we can write the following term using just these application, pairing and some abbreviations:

```
grow(cons(wnode(2,5),
  cons(grow(cons(wnode (4,7), cons(wnode(6,9), nil))),
  cons(wnode(8,11), nil)))) : WTree Nat
```

In particular, we have abbreviated by using numerals in favor of terms built from successor and zero (for brevity). We have also left out the type instantiations which are necessary to fix the polymorphic constructors at specific types—the same term, with type instantiations now restored, should serve to motivate our ultimate goal of an implicitly-typed language:

```
grow↓Nat
  (cons↓(WTree Nat) (wnode↓Nat(2,5),
  cons↓(WTree Nat) (grow↓Nat
    (cons↓(WTree Nat) (wnode↓Nat (4,7),

    cons↓(WTree Nat) (wnode↓Nat(6,9),
    nil↓(WTree Nat) ))),
  cons↓(WTree Nat) (wnode↓Nat(8,11),
  nil↓(WTree Nat) )))) : WTree Nat
```

Note how the size and complexity of the term have grown with the use of explicit instantiations. Note also that uses of grow and wnode are instantiated with the type Nat, since the final term is of type WTree Nat; uses of cons and nil, on the other hand, are instantiated with type WTree Nat, since the constructor grow builds a term of type WTree Nat from a list of children of that same type.

More importantly, note that the component type and the type of the weights here is the same, namely Nat, and that this is entirely coincidental. In general, a weighted tree might contain components of some other type besides Nat, yet it must always contain weights of type Nat, since this is specified as a fixed type, not a parameter, in the declaration for WTree. In this example, all of the even numbers occur as weights, whereas all the odd numbers occur as components. This distinction between fixed and parametric uses of types will be crucial later in our development of analytical polymorphism.

2.3 Formation and conversion of type functionals

As we have mentioned in passing above, the structure of assumptions in the higher-order pattern calculus is the same as the structure of assumptions in the higher-order lambda calculus; that is, assumptions are paired associations, with the upper part giving kinds to type variables and the lower part giving types to value variables. As with assumptions in the higher-order lambda calculus and the higher-order settings we introduced in the last section, assumptions in the pattern calculus are "telescoped", in the sense that while only the upper part is needed for the formation of types, both parts are needed for the formation of expressions. Finally, as expected for "telescoped" structures, while the syntax of the upper halves is context-free, we need the context provided by an upper half in the formation for the corresponding lower half (in order to ensure that the range of a value assumption includes only proper types), and thus to specify formation of the assumption as a whole.[4]

The grammar and rules for the syntax and formation of assumptions are therefore just the ones we have already given for assumptions in the last section.

Our use of parameter patterns as a non-discriminative binding mechanism for type variables is more novel; it is derived from our treatment of normal discriminative patterns (i.e., in Chapter 1), albeit now at the level of types. Recall from above our syntax for parameter patterns and the way in which they are used to bind type variables in type functional expressions:

$$\pi \;::=\; * \;\mid\; \alpha : \kappa \;\mid\; \pi\,,\,\pi$$
$$\theta \;::=\; \cdots \;\mid\; \lambda\pi.\theta \;\mid\; \forall\pi.\theta$$

Parameter patterns are thus simply a mechanism which allows for the simultaneous introduction of multiple, kind-constrained type variables, via a binary product structure, with allowance for the degenerate case of no variables at all (represented by the pattern $*$).

We might just as well have chosen to use "flat" patterns of multiple variables, but the specific choice of binary patterns allows for easier technical treatment of coverage and other issues later. In any case, one would systematically abbreviate multiple-variable patterns using the product and unit structure in the obvious way if desired; i.e.:

$$(\alpha_1{:}\kappa_1,\ \ldots,\ \alpha_n{:}\kappa_n) \triangleq (*,\ (\alpha_1{:}\kappa_1,\ (\ \ldots\ ,\ \alpha_n{:}\kappa_n)\ \ldots))$$

The choice to allow multiple, simultaneous formal arguments in addition to successive, "curried" arguments is not especially crucial for lambda abstraction over types (the latter can in fact be used as well), but is important in the case of universal quantification, as it allows us to consider all type constructors to be functions of a single argument and all value constructors to be polymorphic with respect to a single quantified parameter—this uniformity makes the treatment of patterns ranging over constructed types and values easier.

In the standard lambda calculus, assumptions are extended by a single kind-constrained type variable whenever we enter the scope of a lambda-binding, i.e., in the antecedent rule for

[4] Note that our telescoped association structures are nevertheless strictly separated into upper and lower halves, in contrast with some systems of dependent types that require a "finer-grained" inter-dependence, where each successive element of an association may depend for its formation on all previous ones (the structures we use are simpler because we are concerned with a simpler formalism).

typing the body of the abstraction. Our use of structured parameter patterns in the pattern calculus will require a little bit of technical machinery to analogously extend assumptions with the relevant variable and constraint information. In particular, we will use a device similar to that developed in the simply-typed case of value patterns, namely an operation which extracts the relevant information and returns a "flat" assumption; we have thus:

$$\mathcal{A}(*) = \{\}$$
$$\mathcal{A}(\alpha{:}\kappa) = \{\alpha{:}\kappa\}$$
$$\mathcal{A}(\pi,\upsilon) = \mathcal{A}(\pi) \oplus \mathcal{A}(\upsilon)$$

We also use a notion of erasure for parameter patterns, as follows; both of these operations will be extended later to a broader set of discriminative patterns.

$$\mathcal{E}(*) = *$$
$$\mathcal{E}(\alpha{:}\kappa) = \alpha{:}\kappa$$
$$\mathcal{E}(\pi,\upsilon) = \mathcal{E}(\pi)\ ,\ \mathcal{E}(\upsilon)$$

Given these two operations, we can now state formation rules for type functionals; we have already stated the rules for type constants and type constructors in the last section: the rule for variables is also based on projection, but from a local type assumption rather then a global type setting:

$$A_T \vdash \alpha : A_T(\alpha)$$

Conversely, type variables are added to local assumptions using the operation defined above in the rules for formation of lambda abstractions over types and universal quantification:

$$\frac{\mathcal{A}(\pi) \vdash \mathcal{E}(\pi) : \kappa \qquad A_T + \mathcal{A}(\pi) \vdash \theta : \eta}{A_T \vdash (\lambda\pi.\theta) : \kappa \to \eta}$$

$$\frac{\mathcal{A}(\pi) \vdash \mathcal{E}(\pi) : \kappa \qquad A_T + \mathcal{A}(\pi) \vdash \theta : T}{A_T \vdash (\forall\pi.\theta) : T}$$

Two other formation rules for type functionals concern the forms for type application and pairing; the first is the same as the corresponding rule for the higher-order lambda calculus and the second is the obvious counterpart of the corresponding rule for pairs of values in the simply-typed lambda calculus; we state the two rules here just for the sake of completeness:

$$\frac{A_T \vdash \theta : \kappa \qquad A_T \vdash \omega : \eta}{A_T \vdash (\theta,\omega) : \kappa\times\eta}$$

$$\frac{A_T \vdash \omega : \kappa \to \eta \qquad A_T \vdash \theta : \kappa}{A_T \vdash (\omega\ \theta) : \eta}$$

We defer to the next chapter a discussion of the formation rules for type constructor variables, following a section motivating and describing their use; we will in any case slightly modify our approach to constructor variables in Chapter 4 in order to facilitate certain technical aspects of these forms under reduction.

As examples of formation derivations for type functionals, consider the following:

$$\vdash (\lambda\varnothing.\ \mathrm{Nat}\times\mathrm{Bool}) : 1 \to T$$

$$\vdash (\lambda\alpha{:}T,\ \varphi{:}T\to T.\ \varphi\ \alpha\times\mathrm{Nat}) : T\times(T\to T)\to T$$

$$\{\psi{:}T\to T,\ \zeta{:}T\to T\} \vdash (\lambda\alpha{:}T.\ \psi\ \alpha\times\zeta\ \alpha) : T\to T$$

Chapter 3
Type Patterns and Analytical Polymorphism

So far in our endeavors we have developed an extension of simply-typed lambda abstraction on values to pattern abstraction on values (Chapter 1) and reviewed the expression of polymorphism by higher-order type abstraction (Chapter 2). We now return to our primary objective, that of finding a means of extending traditional polymorphism to encompass functions which are generic with respect to data structures: we realize this goal in the current chapter by extending higher-order lambda abstraction on types to pattern abstraction on types, guided by the analogous extension at the level of values from Chapter 1.

In the next section we sketch the development of some sample definitions using type-pattern abstraction in order to motivate the subsequent formalism and to convey some intuition about the main problems that we must resolve. In Section 3.2 we describe a refinement of the kind inference system that supports a classification of types according to their use of certain forms, namely function spaces and polymorphic quantifications, which we bar from discriminative coverage by type patterns. In Section 3.3 we the proceed to a treatment of coverage and compatibility constraints for type patterns: these issues are complicated in this context by the use of a lambda abstraction form within the patterns themselves. In Section 3.4 we present the syntax and formation rules for value expressions, including type-pattern abstractions.

The higher-order calculus introduced in Chapter 2 and further developed here is also the subject of Chapters 4 and 5, where we will address operational issues (reduction and conversion), semantics and applications. Recall that we study this higher-order, explicitly-typed calculus mainly as a basis for the more practical, implicitly-typed languages that we will develop in the final chapters of the dissertation.

3.1 Patterns over types and type functionals

In this section we develop informally some first examples of type patterns and analytically-polymorphic functions, in order to better motivate various aspects of our approach to these language features. As we mentioned above, we intend the formation and use of type patterns to be analogous to the formation and use of value patterns. Although we have seen value constructors and value patterns in the context of the simply-typed language of Chapter 1, we have not yet seen examples of the corresponding forms as they appear in our polymorphic pattern calculus: we therefore briefly preview the use of the polymorphic versions before turning to the development of type patterns by analogy.

Assume a global constructor setting which provides us with some type and value constructors, say those for lists and natural numbers such as we saw in the last chapter. Then we can express the 2-element list of natural numbers containing the components zero and one respectively, with the following term:

$$L \equiv \mathrm{cons}{\downarrow}\mathrm{Nat}(\mathrm{zero},\ \mathrm{cons}{\downarrow}\mathrm{Nat}(\mathrm{succ}\ \mathrm{zero},\ \mathrm{nil}{\downarrow}\mathrm{Nat})) : \mathrm{List}\ \mathrm{Nat}$$

Note that the constructors cons and nil must be instantiated to the list component type (here Nat) before they are applied to their arguments. Note also that we have abbreviated uses of nullary constructors to avoid tedious explicit applications to the unit term or the unit type: i.e., we should more properly write "Nat $*$" instead of just "Nat" and "nil↓(Nat $*$) $*$" instead of just "nil↓Nat" .

Now consider a typical function which we might apply to the list L, e.g., one which sums up the natural number components of its list argument: we can define such a function as follows:

$$\begin{aligned}\mathrm{sum} \equiv\ & \mu S{:}\mathrm{List}\ \mathrm{Nat}\to\mathrm{Nat}.\\ & \mathrm{nil}{\downarrow}\mathrm{Nat} \Rightarrow \mathrm{zero}\\ & \mid \mathrm{cons}{\downarrow}\mathrm{Nat}(n{:}\mathrm{Nat},\ l{:}\mathrm{List}\ \mathrm{Nat}) \Rightarrow n{+}(S\ l)\end{aligned}$$

The definition employs the usual recursion construct with an explicit type constraint for the recursion variable S; since the intent is to define a function, the body of the recursion form is a pattern abstraction (thus occurrences of S in the body of the form represent recursive uses of the function being defined). Note that we must instantiate the value constructors nil and cons to an appropriate parameter type before we apply them in the patterns of the abstraction, just as was required for their rôle in forming the list data structure above.

Were we to apply the function sum to the term L, we would expect, during the course of the reduction, to match the cons pattern of sum's definition against the term L, since the outer form of the term L is an application of cons. The result of this match should be a substitution which binds zero to the variable n and the tail part of the list L to the variable l; that is, we would expect to compute:

$$\mathcal{M}(\mathrm{cons}{\downarrow}\mathrm{Nat}(n{:}\mathrm{Nat},\ l{:}\mathrm{List}\ \mathrm{Nat}),\ L) = [\mathrm{zero}/n,\ \mathrm{cons}{\downarrow}\mathrm{Nat}(\mathrm{succ}\ \mathrm{zero},\ \mathrm{nil}{\downarrow}\mathrm{Nat})/l]$$

Now we wish to use type-pattern abstractions to express polymorphic entities, just as we have used value-pattern abstractions to express functions over algebraic types. These uses of type-pattern abstractions will typically involve recursion, just as the use of value-pattern abstractions do. Furthermore, most of the examples we consider will be polymorphic functions, so that we will actually express them using both type-pattern abstractions and value-pattern abstractions, with the latter nested within the former.

Whereas value-pattern abstractions are comprised of multiple clauses, each consisting of a pattern/expression pair, type-pattern abstractions are comprised of multiple clauses, but with each clause consisting of a type-pattern/value expression pair. Note that clauses consisting of type-pattern/*type*-expression pairs would instead yield functions from types to types defined by recursive analysis: we will not pursue the corresponding notion of computation over types further in the dissertation, although it would be natural in a dependently-typed calculus with a uniform space of types and values (see, e.g., [Card86]).

Lambda abstraction	**Pattern abstraction**
$(\lambda x{:}\tau.e) : \tau\to\sigma$	$(p\Rightarrow e \mid \ldots \mid p\Rightarrow e) : \tau\to\sigma$
$(\Lambda\alpha{:}\kappa.e) : \forall\alpha{:}\kappa.\sigma$	$(\rho\Rightarrow e \mid \ldots \mid \rho\Rightarrow e) : \forall\alpha{:}\kappa.\sigma$

Figure 3.1.1: Abstractions in lambda and pattern calculi

In the case of value patterns, the constituent patterns p_i of an abstraction form range over the values of its domain type τ; we took care in the typing rules to guarantee that all values of the domain type would be matched by some pattern (coverage)

and that there would be a unique way to match each one relative to a given pattern-matching regimen (compatibility). In the case of type-pattern abstractions, the constituent type patterns ρ_i range over the kind κ which constrains the quantified variable in the type of the abstraction. We will consider coverage and compatibility issues for type patterns with respect to the corresponding kind constraint by analogy with value patterns and type constraints, though the type case will be somewhat more complex.

As regards the syntax of type patterns themselves, recall from Chapter 1 that value patterns have forms which mirror a restricted subset of the forms of value expressions; likewise, type patterns will have forms which mirror a restricted subset of the forms of types and type functionals. Just as value patterns are formed from constructors, pairs, the trivial value and type-constrained value variables, type patterns are formed from type constructors, pairs, the trivial type and kind-constrained type variables, thus generalizing the type parameter patterns of the last chapter. We also include our standard type constants 1, × and → among the forms of type patterns. Note that these type constants behave more like constructors than do typical value constants, i.e., they have no associated reduction rules and thus no real computational import; in essence they are type constructors which are (necessarily) primitive.

We use the following example of a type pattern to illustrate their syntax and use:

$$(\gamma{:}T){\times}(\mathrm{List}\ (\eta{:}T))$$

Here γ and η are (kind-constrained) type variables, List is a type constructor and × is the constant used to represent product types. We would expect to be able to match this type pattern against any particular type of similar form, such as Nat×(List Bool), just as we match value patterns against similarly-formed value expressions; from such a match we would expect to obtain as a result a substitution of type functionals for type variables, e.g., as follows:

$$\mathcal{M}((\gamma{:}T){\times}(\mathrm{List}\ (\eta{:}T)),\ \mathrm{Nat}{\times}(\mathrm{List}\ \mathrm{Bool})) = [\mathrm{Nat}/\gamma,\ \mathrm{Bool}/\eta]$$

In addition to the forms listed above, we also include lambda abstractions over types among our forms of type patterns. Note that this is specifically *not* analogous with the syntax of value patterns: although we could express function abstractions over functional domain types in the simply-typed pattern calculus, they were in some sense degenerate, consisting only of a single variable, since we had no good way of matching patterns against functional values. In contrast with the value level, we specifically need compositional type-pattern abstractions which range over functions from types to types, since we wish to capture polymorphism over data structures, which we have construed as functions over their component types.

We will address the details of the syntax and formation of patterns over type functions in more detail later in this section; for the moment we wish merely to point out that the addition of lambda abstractions to the syntax of type patterns introduces a distinction between variables bound by the type pattern itself, like γ and η above, and those bound by lambda abstractions *within* the pattern. This new distinction between uses of variables is the main source of extra complexity in type patterns .

As a preliminary example, consider the following instance of a match of a type-functional pattern against a similarly-formed type function:

$$\mathcal{M}(\lambda\alpha{:}T.(\varphi\ \alpha{:}T){\times}(\psi\ \alpha{:}T),\ \lambda\beta{:}T.\mathrm{Nat}{\times}(\mathrm{List}\ \beta)) = [(\lambda\beta{:}T.\ \mathrm{Nat})/\varphi,\ \mathrm{List}/\psi]$$

Here α is a type variable bound within the pattern by a lambda abstraction form, just as β is bound within the type function term, whereas φ and ψ are variables ranging over type functions that occur "freely" within the pattern in the usual fashion. Thus it is φ and ψ which constitute the domain of the resultant substitution of type functions. Note also that the occurrences of φ and ψ within the type pattern are applied to the lambda-bound variable α: this allows us to maintain the overall kind (a function from types to types) in the results returned in the substitution, a property which is crucial in the recursive analysis of type functions, since it provides for a common kind at each stage of the recursion.

Although the use of type functional patterns complicates the matching operation, it is nevertheless at least *feasible*, unlike the analogous use of value function patterns, since our language of type functions is essentially just a simply-typed lambda calculus, lifted to the level of types and kinds. That is, we do not use recursion or patterns to specify functions from types to types, but rather just kind-constrained lambda abstraction, so the terms representing these functions will have unique normal forms which we can find effectively with finite computations.

We now proceed with a longer example of a definition of a polymorphic function expressed using type patterns, which we will develop by analogy with the definition of the value function sum above. In particular, we consider a size function which will return, for an argument of arbitrary type, the number of value constructors used to form the argument: this function can be thought of as a simple complexity measure for data values—as we discuss below, it is not so clear how to interpret it for functional arguments.

Our function size should clearly have a polymorphic, functional type of the following form:

$$\mathrm{size}{:}\ \forall\alpha{:}T.\ \alpha{\to}\mathrm{Nat}$$

Note that we cannot define such a function in a general and uniform way in any current functional language based on Milner-style polymorphism, since the informal description above clearly requires that we analyze the structure of an argument of arbitrary type: this sort of analysis is not supported by Milner-style polymorphism, which gives polymorphic types only to functions which manipulate their arguments "extensionally", i.e., in a manner which is parametrically independent of the argument type.

It is nevertheless intuitively clear how we might code such a function, say in an untyped context: we recursively traverse the argument, accumulating a count of the constructors used by adding one to the count each time any constructor is encountered. In contrast to an untyped version, however, we seek a definition to which we can give a sound typing: toward this goal we use a form of definition which employs type patterns to guide the recursive traversal.at each stage, thus accounting in a well-typed way for the differently-typed arguments which might be encountered at different stages of the recursion.

Based on this sketch, we can make a first approximation to the definition of size as follows:

$$\mathrm{size}\ \equiv\mu\mathrm{S}{:}\forall\alpha{:}T.\ \alpha{\to}\mathrm{Nat}.$$

$$
\begin{array}{l}
(\ \rho_1 \Rightarrow\!\!\!> (p_{11} \Rightarrow e_{11} \mid \cdots \mid p_{1m_1} \Rightarrow e_{1m_1}) \\
\quad \mid \ \cdots \\
\quad \mid \ \rho_n \Rightarrow\!\!\!> ((p_{n1} \Rightarrow e_{n1} \mid \cdots \mid p_{nm_n} \Rightarrow e_{nm_n})\ )
\end{array}
$$

Here size is defined using recursion, just like sum above, though now the body of the recursion is not a value-pattern abstraction, but rather a type-pattern abstraction, since we intend that the function have a polymorphic type. Since we intend it furthermore to be a polymorphic *function*, we know that the bodies of the various type-pattern clauses ought to be in turn value-pattern abstractions, each of which will consist of one or more clauses of its own. Note that the number of clauses comprising each of the inner abstractions will, in general, be different, being based on the particular type matched by the corresponding type pattern.

At this point we can also recognize that uses of the size function will require first an instantiation to some type and then an application to a value of that type, for example as follows:

$$\mathrm{size}{\downarrow}(\mathrm{List\ Nat})\ \mathrm{L}$$

These same remarks apply to any uses of the recursion variable S inside the body of the definition, since it (necessarily) has the same type as the defined function.

Returning to the definition, we can continue by writing in some specific type patterns for the left-hand sides of the type-pattern abstraction:

$$
\begin{array}{l}
\mathrm{size} \equiv \mu S{:}\forall\alpha{:}T.\ \alpha \rightarrow \mathrm{Nat}. \\
\quad (\ 1 \Rightarrow\!\!\!> (p_{11} \Rightarrow e_{11} \mid \cdots \mid p_{1m} \Rightarrow e_{1m}) \\
\quad \mid (\beta{:}T \times \gamma{:}T) \Rightarrow\!\!\!> (p_{21} \Rightarrow e_{21} \mid \cdots \mid p_{2m} \Rightarrow e_{2m}) \\
\quad \mid \mathrm{List}(\delta{:}T) \Rightarrow\!\!\!> (p_{31} \Rightarrow e_{31} \mid \cdots \mid p_{3m} \Rightarrow e_{3m}) \\
\quad \mid \ \cdots \\
\quad \mid \ \rho_n \Rightarrow\!\!\!> ((p_{n1} \Rightarrow e_{n1} \mid \cdots \mid p_{nm} \Rightarrow e_{nm})\ )
\end{array}
$$

Here we have allowed for (at least) three of the possible outer forms that an actual type argument might have, namely the unit type, the product constant and the list constructor. Before considering other possible outer forms, we will develop the right-hand sides of the clauses we have specified so far. By the type constraint on the recursion form, all the right-hand side value-pattern abstractions must denote functions from the type given by the corresponding left-hand side to a natural number. This is so because the domain part of the functional type in the constraint on S is just α, i.e., the type variable bound by the universal quantifier, which thus represents the actual type argument against which the type patterns will ultimately be matched.

Using this information about the typing of the right-hand sides of the type-pattern abstractions, we can write further:

$$
\begin{array}{l}
\mathrm{size} \equiv \mu S{:}\forall\alpha{:}T.\ \alpha \rightarrow \mathrm{Nat}. \\
\quad (\ 1 \Rightarrow\!\!\!> (* \Rightarrow \mathrm{zero}) \\
\quad \mid (\beta{:}T \times \gamma{:}T) \Rightarrow\!\!\!> ((b{:}\beta, c{:}\gamma) \Rightarrow (S{\downarrow}\beta\ b) + (S{\downarrow}\gamma\ c)) \\
\quad \mid \mathrm{List}(\delta{:}T) \Rightarrow\!\!\!> \\
\\
\quad (\ \mathrm{nil}{\downarrow}\delta \Rightarrow \mathrm{succ\ zero} \\
\\
\quad \mid \mathrm{cons}{\downarrow}\delta(p{:}\delta \times \mathrm{List}\ \delta) \Rightarrow \mathrm{succ}(S{\downarrow}(\delta \times \mathrm{List}\ \delta)\ p)) \\
\quad \mid \ \cdots \\
\quad \mid \ \rho_n \Rightarrow\!\!\!> ((p_n \Rightarrow e_n)\ )
\end{array}
$$

Note that only a single value-pattern clause is needed for the unit and pair cases, whereas we must use two clauses, one each for the constructors nil and cons, to cover the possibilities in the case for the type constructor List. By analogy, in order to get complete coverage of the type argument by the various type-pattern clauses, we should include clauses at that level which will match all other data type constructors (e.g., Nat, Bool, Tree, etc.), as well as functional and polymorphic types. Since a given version of the pattern calculus (i.e., one that is associated with a given global constructor setting) might have an arbitrary set of type constructors, we need to find some way of covering arbitrary type constructors if we wish to write functions which are truly general (i.e., not tied to some particular setting).

Our approach to this problem is to use *type-constructor variables* which range over arbitrary type constructors—we had introduced these atomic symbols briefly in the discussion of the syntax of type functionals in Chapter 2. The use of variables, as opposed to specific type constructors, has the desired affect that we can cover all possible data structures uniformly, with a single pattern. A potential problem with the use of type-constructor variables is that an arbitrary type constructor may have an arbitrary domain kind; i.e., a type-constructor variable might be bound to an actual type constructor which takes a type argument, or a pair of type arguments, or a function on types, etc. (we should stress that, by contrast, the *range* kind of a type constructor is always just T, the kind of all types).

Our solution to this problem is to introduce variables into the syntax of kinds: we use uppercase letters V, W, X, etc. to range over *kind variables*. Now we can constrain binding occurrences of type-constructor variables with kind variables in their domains, thus allowing for an arbitrarily-kinded type constructor to be matched as an actual argument.

For example, the following type pattern should match a type constructed with any type constructor, binding Σ to that type constructor, V to its domain kind and the type variable α to the actual argument of the type constructor in the matched type term:

$$(\Sigma{:}V \rightarrow T)\ (\alpha{:}V)$$

In particular, we would expect the following match to succeed in our standard constructor setting with the indicated substitution as its result:

$$\mathcal{M}((\Sigma{:}V \rightarrow T)\ (\alpha{:}V),\ \mathrm{List(Nat)}) = [\mathrm{List}/\Sigma,\ T/V,\ \mathrm{Nat}/\alpha]$$

since the type constructor List has kind $T \rightarrow T$ in that setting. Note that we must now accommodate the binding of kinds to kind variables and the binding of type constructors to type-constructor variables in the structure of our substitutions (we have simplified the result above to a flat, many-sorted substitution only for the sake of readability).

Returning to our running example of the size function, we can now revise the structural clause as follows:

$$
\begin{array}{l}
\mathrm{size} \equiv \mu S{:}\forall\alpha{:}T.\ \alpha \rightarrow \mathrm{Nat}. \\
\quad (\ 1 \Rightarrow\!\!\!> (* \Rightarrow \mathrm{zero}) \\
\quad \mid (\beta{:}T \times \gamma{:}T) \Rightarrow\!\!\!> ((b{:}\beta, c{:}\gamma) \Rightarrow (S{\downarrow}\beta\ b) + (S{\downarrow}\gamma\ c)) \\
\quad \mid (\Sigma{:}X \rightarrow T)(\chi{:}X) \Rightarrow\!\!\!> (p \Rightarrow e \mid \cdots\ )) \\
\quad \mid \ \cdots \\
\quad \mid \ \rho_n \Rightarrow\!\!\!> ((p_{n1} \Rightarrow e_{n1} \mid \cdots \mid p_{nm} \Rightarrow e_{nm})\ )
\end{array}
$$

The astute reader will foresee a potential problem here: in the product clause we have instantiated recursive uses of S to the types β and γ, but with what type argument should we instantiate recursive uses of S in the structural clause? The obvious candidate, χ, has an arbitrary kind X, rather than the specific kind T which we need for a well-typed instantiation; that is, χ may be an arbitrarily high-order function over types rather than just an individual type as required by the constraint on the universally quantified variable α.

It happens that the correct instantiation in this case will not be to χ, but rather to the *application* to χ of the argument type function associated with the value constructor used to construct the value of structural type Σ. Although we will not fully justify this answer to the question until later in the chapter, where we will have the proper background, the reader's intuition may be aided by thinking about an example: consider a list of natural numbers formed from a value constructor, say cons, and the argument of that value constructor. The domain type of this particular use of cons is equivalent to the corresponding domain type function from the setting, in this case $(\lambda\alpha{:}T.\alpha\times List\ \alpha)$, applied to the type argument of List, in this case Nat. Thus in this particular instance, the argument of cons will have type $(\lambda\alpha{:}T.\alpha\times List\ \alpha)Nat = Nat\times List\ Nat$, i.e., it will be a pair consisting of a natural number and another list of natural numbers. So, in recursively traversing the list of natural numbers, we will visit first an argument of type List Nat, expressed as an application of cons, and then an argument of type Nat×(List Nat), comprising the argument to cons. So the proper instantiation in the recursive call is not to the argument *type function* $(\lambda\alpha{:}T.\alpha\times List\ \alpha)$ associated with cons, but rather to the *application* of that type function to an argument appropriate to the particular use of cons.

Returning now to our example definition of the size function, consider what form should be taken by the value-pattern abstraction occurring on the right-hand side of the structural clause. By the afore-mentioned equivalence between the domain type of the function and the parameter of the polymorphic quantification (both are the variable α), we know that within the scope of the structural clause the value-pattern abstraction will have to be of type $(\Sigma\ \chi)\rightarrow Nat$, since α will be specialized to the form $\Sigma\ \chi$ if the structural type pattern is chosen in the match.

Thus the pattern abstraction on the right-hand side should cover an arbitrary structural type of the form $\Sigma\ \chi$: once again we are faced with the problem of covering an arbitrarily-constructed term, though now at the level of values rather than at the level of types. By analogy with our solution to the similar problem with type constructors, we introduce *value constructor variables;* we will use the uppercase letters K and H to range over these variables. Just as we were forced to introduce kind variables to constrain the domain kinds of type-constructor variables, we will need to use type function variables in the domain types of value constructor variables, since an arbitrary actual value constructor will have an arbitrary type functional domain associated with it by the global constructor setting.

Using these ideas, we can now refine our definition of size to the following:

$$
\begin{aligned}
size \equiv\ & \mu S{:}\forall\alpha{:}T.\ \alpha\rightarrow Nat.\\
& (\ \ 1 \Rightarrow (* \Rightarrow zero)\\
& \ \mid\ (\beta{:}T\times\gamma{:}T) \Rightarrow ((b{:}\beta,c{:}\gamma) \Rightarrow (S{\downarrow}\beta\ b)+(S{\downarrow}\gamma\ c))\\
& \ \mid\ (\Sigma{:}X\rightarrow T)(\chi{:}X) \Rightarrow\\
& \qquad ((K{:}\varphi\mapsto\Sigma){\downarrow}\chi\ (x{:}\varphi\ (\chi{:}X)) \Rightarrow succ(S{\downarrow}(\varphi\ \chi)\ x))\\
& \ \mid\ \ldots\\
& \ \mid\ \rho_n \Rightarrow ((p_n \Rightarrow e_n)\ )
\end{aligned}
$$

Note here that we have constrained the variable x, which occurs as the argument of the constructor variable K, with the type constraint $\varphi\mapsto\Sigma$: this is the same notational device we used in specifying the polymorphic functional types associated with value constructors in value constructor settings. In particular, it allows us to focus on the essential information about the type, since the (implicit) quantification is already determined by the kind of the corresponding type-constructor variable Σ. It also allows us to more easily specify the matching of the constraint $\varphi\mapsto\Sigma$ and the corresponding type associated with the actual constructor against which K is matched, since we have a simple algebraic form rather than a more complex one involving quantification and bound variables.

We refer to the variables used in the domain parts of constructor variables as *skolem variables*, since they are essentially existential in nature, and are instantiated in concert with the corresponding constructor variable (for every instantiation of a constructor variable, there exists an appropriate instantiation of the corrsponding skolem variable). This terminology is meant merely to be suggestive: there is no deep or clear connection to the traditional notion of a skolem function.

By way of example, if the actual constructor matched against the constructor variable K were cons for lists, we would expect to compute as follows:

$$
\begin{aligned}
&\mathcal{M}((K{:}\varphi\mapsto List){\downarrow}Nat\ (x{:}\varphi\ Nat),\ cons{\downarrow}Nat(h,t))\\
&\quad = [cons/K,\ (\lambda\alpha{:}T.\alpha\times List\ \alpha/\varphi,\ (h,t)/x]
\end{aligned}
$$

since we have $C_{VC}(cons) = (\lambda\alpha{:}T.\alpha\times List\ \alpha)\mapsto List$. Here h and t are some expressions of appropriate type; we presume that the type-constructor variable Σ and type variable χ would have been matched previously against the actual type constructor List and argument type Nat, and then been appropriately substituted into the body of the type-pattern clause.

Although the introduction of constructor and skolem variables allows us to account in a single clause for all possible structured types, in order to completely cover the possible range of types, and thus finish the definition of size, we must still account for functional and polymorphic types in the left-hand sides of the type-pattern abstraction. Consider first how we might add a pattern for function types: as a first approximation, we might try a clause like this:

$$(\alpha{:}T)\rightarrow(\beta{:}T) \Rightarrow (\lambda a{:}\alpha.\ b{:}\beta) \Rightarrow e[a,b]$$

But this first approximation is not workable, for we have already seen that we cannot properly match such lambda abstraction patterns against actual value function arguments, for a variety of reasons (see Chapter 1 for details). Even given a solution to the problems associated with patterns over functions, there is a question of what we would then write on the right-hand side of the clause in order to count the number of constructors used to build a function. In particular, it seems sensible to consider the "size" of a data value of ground type in terms of the number of constructors used to express it, since the expression (a term) and its denotation (in some initial algebra) are at least isomorphic. But it does *not* seem sensible to consider the "complexity" of an arbitrary, recursively-

defined function as being given in the same way by the number of constructors used in some term denoting that function, since the meaning of the function is an infinitary entity and not so closely tied to any of its many syntactic representations.

From a broader perspective, many of the functions which we have used to motivate the development of analytical polymorphism cannot be easily extended to cover the case of function types: equality over recursively-defined functions, for example, is not decidable. In the case of functions which depend on a type parameter for a notion of structural component, problems arise especially in regard to negative occurrences of a type parameter within the functional type. For example, we wish to define the count function so as to return the number of components in its structural argument: we might count the components of type α in a function of type $\beta\rightarrow\alpha$ according to the cardinality of the type β, though we would still need to develop a means for expressing the cardinality of a type including, in general, infinite cardinalities.

But how should we count the number of components of type α in a function of type $\alpha\rightarrow\beta$ or in a function of type $\tau\rightarrow\alpha$ when α occurs free in τ? Our intuitions do not seem to extend well to these cases where the component type occurs in the domain part of a function type.

Similar problems arise with the map functional: we might define map on function types for positive occurrences of a type parameter using composition, so that a function f of type $\alpha\rightarrow\beta$ would be mapped across an argument function g of type $\tau\rightarrow\alpha$ by returning $g\circ f$ of type $\tau\rightarrow\beta$. But it is not clear that there is any good sense in which a function of type $\alpha\rightarrow\sigma$ has "components" of type α. Similar difficulties with negative occurrences of a type parameter within functional types have been encountered by other authors in different contexts: see for example [Hag86]. These problems seem to be related to the well-established difficulties concerning the anti-monotonicity of the functional type construct with respect to its first argument. Although we expect that a full and rigorous semantic analysis could provide more insight into the problem, we will take a straight-forward approach which simply disallows refined patterns over functional types.

More specifically, we use an order-sorted extension of kinds to ensure that the universally quantified variables of type-pattern abstractions are restricted so as to preclude any functional types built from their domain. In particular, we use a new kind constant, written D for *data type*, to denote a subset of the types of kind T, namely those which do not contain any uses of the function space form.

Note that in our calculus, terms of polymorphic type are also expressed using recursion and patterns, just as functions are: therefore we would expect to encounter similar problems with patterns over polymorphic types (in a non-pattern-based polymorphic calculus, on the other hand, type abstractions are much simpler and might be amenable to the same treatment as we use for patterns over type functions).

We will present a detailed treatment of the type/data type distinction in the next section; for our current purposes it suffices to note that the restriction of the quantified variable α in the definition of the size function allows us to finally complete this definition:

```
size ≡ μS:∀α:D. α→Nat.
    (  1 ⇒» (∗ ⇒ zero)
     | (β:D×γ:D) ⇒» ((b:β,c:γ) ⇒ (S↓β b)+(S↓γ c))
     | (Σ:X→D)(χ:X) ⇒»
          ((K:φ↦Σ)↓χ (x:φ (χ:X)) ⇒ succ(S↓(φ χ) x)) )
```

Given the new, more refined restriction on α, we see that we have indeed covered all possible cases of actual data type arguments which might be supplied to size, as these comprise only the unit type, products and constructed types (note that even a type constructed from a type-constructor variable will match the third type pattern).

Before proceeding to the next section, we consider now an example of a definition which makes use of functional type patterns, namely a definition for the count function. Informally, we want to define this function for arbitrary structures so that it will count the number of components in a structural argument. This is clearly closely related to the size function above, though in the case of count we should increment our running total on the basis of a new type case for structural components, rather than on the basis of the constructor case.

More formally, the type of the count function should be:

```
count : ∀α:T. ∀φ:T→D. φ α→Nat
```

Here the quantification over the type variable α allows for an arbitrary component type, since we do not need to analyze the structure of components in order to count them. The constraint $T\rightarrow D$ on the type function φ involves an extension of the notion of data types to higher order kinds: we will develop this extension at length in the next section, but in essence it restricts the instantiation of count to only those type functions (think: data structures) which makes no use of the problematic functional or quantificational forms. Note that the ultimate domain type, $\varphi\ \alpha$, will in general include uses of these problematic constructs, since the appropriate beta substitution will carry any such occurrences into the resultant type term: attention to these interactions between the kinds of type functions, their arguments and the results of applications will be crucial for the development of the data type notion.

As an initial approximation to the definition of the count function, we can mimic the definition of size above, though we will now need to use patterns over a type function of kind $T\rightarrow D$, corresponding to the quantification over φ. Rather than motivating and developing each aspect of the definition of count separately as we did for size above, we simply present the final definition all at once and then briefly discuss the major differences on a point-by-point basis. The definition is:

```
count ≡ α:T ⇒» μC:∀φ:T→T. φ α→Nat.
    (  λε:T. 1 ⇒» (∗ ⇒ zero)
     | λε:T. (ψ ε:D)×(ζ ε:D) ⇒»
          ((y:ψ α, z:ζ α) ⇒ (C↓ψ y)+(C↓ζ z))
     | λε:T. (Σ:X→D)(χ ε:X) ⇒»
          ((K:η↦Σ)↓(χ α) (x:η (χ α:X))
             ⇒ C↓(λε:T. η (χ ε)) x)
     | λε:T. ε ⇒» (a:α ⇒ succ(zero)) )
```

In regards this definition, note first that we now need an extra type abstraction at the outermost level, corresponding to the quantification over the arbitrary component type α. This degenerate, single-variable abstraction does not analyze the structure of its argument and thus has precisely the same "extensional" sort of interpretation as the standard type abstraction $\Lambda\alpha:T.e$ in the original polymorphic lambda

calculus, which is warranted since we do not need to analyze the structure of a component in order to count it. This usage also demonstrates that type-pattern abstraction is a proper generalization of type-lambda abstraction.

Next note that all of the type patterns now have the form $(\lambda\varepsilon{:}T.\rho)$, for some type pattern ρ, thus allowing for the fact that they range over type functions rather than basic types. The type variables in the patterns themselves have now also been "lifted" to variables over type functions, thus facilitating proper recursive instantiations which respect the same notion of component. Within the patterns these type functions are applied to the lambda-bound type variable ε, whereas on the right-hand sides of the pattern clauses they are applied instead to the type parameter α: this reflects the fact that the type patterns range over the possible forms of the type *function* argument φ, independent of the component type α, whereas the value patterns on the right-hand sides range over domain types of the form $\varphi\ \alpha$ for each corresponding form of φ. This distinction is the key to understanding how type-functional patterns allow us to capture a notion of quantification over data structures.

Also not that recursive instantiations on the right-hand sides of the value-pattern abstractions are made just to the appropriate type function, e.g. ψ or ζ, rather than a type application of the form $(\psi\ \alpha)$ or $(\zeta\ \alpha)$, since here again we are concerned with supplying a type function argument to the (recursive core of) the count function. In the particular case of the structural clause, the recursive instantiation is to the *composition* of the value constructor's domain function η and the pattern variable χ, expressed here as $(\lambda\varepsilon{:}T.\ \eta\ (\chi\ \varepsilon))$. This corresponds to the instantiation of S to a type application in the structural case of the size function, but is now "lifted" one functional level so as to provide the appropriate type function argument to the recursion variable C and thus continue to account for the appropriate notion of component in recursive calls.

We have also added a clause for the case of components, with a type pattern $(\lambda\varepsilon{:}T.\ \varepsilon)$, which mimics the identity function on types. In the recursive analysis of a type function, the identity function will correspond to an occurrence of the type argument of the function, and thus to a component of the corresponding data structure, so this type pattern properly captures the component case. The use of the identity function as the type pattern for this clause implies a type of α for the corresponding value pattern $(a{:}\alpha)$, since if $\varphi = (\lambda\varepsilon{:}T.\ \varepsilon)$, then $\varphi\ \alpha = (\lambda\varepsilon{:}T.\ \varepsilon)\ \alpha = \alpha$. Since there is no more recursive structure to be handled, we return a count of one here (expressed as succ(zero)) rather than incrementing a recursive count.

Finally, we should consider the issue of coverage with regard to the type-pattern abstraction used in the definition of count: according to the quantification on φ, we need to cover all cases of functions from types to types which, when expressed as lambda abstractions, have no occurrences of functional or quantificational forms in their bodies. We have in fact done this, since we have accounted for the outermost forms of the body of such a function which correspond to the data types of unit, products and constructors, and also for the identity function, with a body consisting just of an occurrence of the lambda-abstracted variable ε. We will make this observation rigorous later in the chapter with a formal definition of coverage of kinds by sets of type patterns.

3.2 Classification of types by data kinds

In this section we revise the syntax of kinds to include an extra constant D representing the set of data types, as was briefly motivated in the last section, and then revise the formation rules for type functionals to implement an order-sorted notion of classification which hereditarily extends the type/data type distinction to higher-order type functionals. These revisions have consequences in turn for the syntax of various auxiliary sorts including settings, assumptions and parameter patterns.

Recall the motivation for these extensions to the calculus from the discussion in the last section: given an analytically-polymorphic function such as size or count defined above, we want to restrict its occurrences in expressions so that it is never instantiated with a type term that contains certain problematic forms, namely functional types and universal quantifications.

One approach to this problem would be to simply make the restriction explicit, along the lines of a free variable condition, using a predicate which is true of a type term just when that term has no occurrences of the problematic forms. This approach will not work, however, when the term in question is a type variable, for example one bound by a type-pattern abstraction or universal quantification: in these circumstances we have access only to the kind of the variable, as given by the current type assumption, but not to the term which will be ultimately bound to that variable. Based on this observation, we propose to encode the relevant information about occurrences of the problematic forms in an actual term (or about potential occurrences in a formal parameter) through an extended notion of kind for type functional terms.

More specifically, we use an *order-sorted* extension of kinds to classify well-formed type functionals as to their inclusion of the problematic forms: for example, a data type will have kind D, representative of a subset of the full set of types of kind T; the order-sorted aspect arises because a data type is also a type (intuitively, $[\![D]\!] \subseteq [\![T]\!]$, where $[\![\kappa]\!]$ denotes the meaning of a kind κ).

The primary sorts at the various levels of our calculus will thus now reveal a finer structure of "subsorts": at the level of values, we distinguish between arbitrary well-formed expressions and a restricted subset of ground expressions, namely those which can be expressed without using functional or polymorphic forms. The former are well-formed relative to a proper type, the latter to a restricted subset of data types. At the level of types, data types are well-formed with a data kind D and proper types with kind T, with a formal ordering relation relating the two used to represent the underlying subset relation. The following diagram illustrates these relationships graphically:

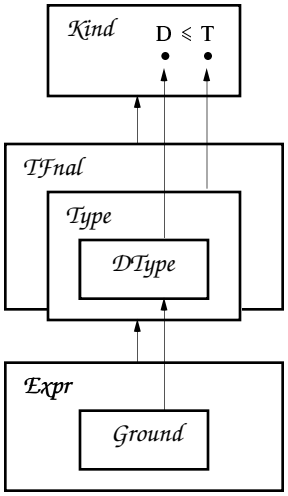

Figure 3.2.1: Subsorts and formation relations

In order to formalize these ideas, we seek a system of axioms and rules which will allow us to derive judgements of the form $A \vdash \theta : \delta$, where δ ranges over the set of data kinds (we leave open the exact syntax of data kinds for the moment, but it should at least extend the original language of kinds with the new constant D). The crucial problem, beyond choosing a syntax for the data kind terms, involves the classification of application terms: consider, for example, the application of a data type function, i.e. one which makes no explicit use of the problematic forms, to a proper type argument: such an application leads to the introduction of one of the problematic forms into the resulting type term by β reduction.

$$\varphi \equiv (\lambda\alpha{:}T.\ \alpha \times \mathrm{List}\ \alpha)$$
$$\tau \equiv (\mathrm{Nat} \to \mathrm{Bool})) : T$$
$$\varphi\ \tau = (\mathrm{Nat} \to \mathrm{Bool}) \times \mathrm{List}(\mathrm{Nat} \to \mathrm{Bool}) : T$$

One criterion for evaluating a solution to the data type classification problem is that it should be complete and "static", i.e., such that we can determine the data kind of a type application just from the data kinds of the function and argument parts. Unfortunately this is difficult, since we must then record in the data kind terms rather complex information about the way in which type arguments occur in the bodies of type functions. For example, assume that τ is a proper type and that σ is a data type and consider the following type functions[5]:

$$\psi \equiv \lambda\varphi{:}T \to T \to T.\ \varphi\ \tau\ \sigma$$
$$\varphi_1 \equiv \lambda\alpha{:}T.\ \lambda\beta{:}T.\ \alpha$$
$$\varphi_2 \equiv \lambda\alpha{:}T.\ \lambda\beta{:}T.\ \beta$$

Now the application $\psi\ \varphi_1$ will result in a proper type, since:

$$\begin{aligned}\psi\ \varphi_1 &= (\lambda\varphi{:}T \to T \to T.\ \varphi\ \tau\ \sigma)\ (\lambda\alpha{:}T.\ \lambda\beta{:}T.\ \alpha)\\ &= (\varphi\ \tau\ \sigma)\ [(\lambda\alpha{:}T.\ \lambda\beta{:}T.\ \alpha)/\varphi]\\ &= (\lambda\alpha{:}T.\ \lambda\beta{:}T.\ \alpha)\ \tau\ \sigma\\ &= \alpha\ [\tau/\alpha,\ \sigma/\beta]\\ &= \tau : T\end{aligned}$$

On the other hand, the application $\psi\ \varphi_2$ will result in a data type, since:

$$\begin{aligned}\psi\ \varphi_2 &= (\lambda\varphi{:}T \to T \to T.\ \varphi\ \tau\ \sigma)\ (\lambda\alpha{:}T.\ \lambda\beta{:}T.\ \beta)\\ &= (\varphi\ \tau\ \sigma)\ [(\lambda\alpha{:}T.\ \lambda\beta{:}T.\ \beta)/\varphi]\\ &= (\lambda\alpha{:}T.\ \lambda\beta{:}T.\ \beta)\ \tau\ \sigma\\ &= \beta\ [\tau/\alpha,\ \sigma/\beta]\\ &= \sigma : D\end{aligned}$$

This means that we will have to classify φ_1 and φ_2 with different data kinds, since ψ has just one fixed data kind and we wish to distinguish between the kinds of its applications to φ_1 and φ_2. From this example it is clear that any complete, static system of classification must use data kind terms which incorporate the equivalent of variable-binding operators, in order to track the precise points in a type functional term at which the formal parameters of an abstraction occur.

We have in fact explored such a system of classification, but the introduction of binding forms to the level of kinds (along with attendant conversions and substitutions needed to effect the classification of type application forms) seems too complex: essentially this system performs an "abstract interpretation" of the full reduction or conversion mechanism for type terms, but performs it at the level of kinds. A simpler and more efficient approach is to instead allow for the reduction of types to feed back information into the classification system. That is, rather than implementing type reduction twice, once at the level of types and again, more abstractly, at the level of kinds, we simply rely upon the interaction of type reduction and classification through a subject-conversion rule with the following form:

$$\frac{A \vdash \theta : \delta \qquad \theta = \theta'}{A \vdash \theta' : \delta}$$

Using this rule, we can construct the following derivation to account for the example above:

$$\frac{A \vdash \sigma : D \qquad \psi\ \varphi_2 = \sigma}{A \vdash (\psi\ \varphi_2) : D}$$

Although this approach based on algebraic data kinds and the subject-conversion rule is not static, in the sense that reduction may be required to determine the least possible data kind for a type term, it is complete "by fiat". It also seems to be adequate for practical applications and is much simpler than the approach based on variable-binding mechanisms at the level of kinds; we therefore proceed with a formal description of this approach in the remainder of the section.

We begin with the introduction of the new constant D into the language of kinds. It will be convenient on occasion to consider the set of constants {D,T} independently; we call these the *basic kinds* and use the meta-variable B to range over them; thus we have:

$$B ::= D \mid T$$

Rather than directly extending the language of kinds to include the new constant, we define the new language of data kinds, ranged over by δ, as a superset of the original language of kinds; more specifically, the language of data kinds is defined in terms of the original, *proper* kinds as follows (we reproduce here the grammar for proper kinds for easier reference):

[5] It may be helpful to recognize that these functions are just the usual encodings of the boolean conditional and truth-values, lifted to the level of types.

$$\kappa ::= 1 \mid T \mid V \mid \kappa\times\kappa \mid \kappa\rightarrow\kappa$$
$$\delta ::= 1 \mid D \mid T \mid V \mid \delta\times\delta \mid \kappa\rightarrow\delta$$

Thus proper kinds include no occurrences of the new constant D; the more inclusive data kinds are free from occurrences of D on the left-hand sides of the function kinds $\kappa\rightarrow\delta$, although D may occur freely in product kinds and on the (ultimate) right-hand side of the function arrow. The reasons for these restrictions will become clear as we proceed through this section.

Next we formalize the interpretation of the constant D as representing a subset of the set of all types by way of a partial ordering on data kinds. This partial ordering is founded on the basic axiom $D \leqslant T$ and then extended to other data kinds as follows:

$$\frac{\delta \leqslant \delta' \qquad \eta \leqslant \eta'}{(\delta\times\eta) \leqslant (\delta'\times\eta')}$$

$$\frac{\delta \leqslant \delta'}{(\kappa\rightarrow\delta) \leqslant (\kappa\rightarrow\delta')}$$

$$\delta \leqslant \delta$$

Here the first two rules simply extend the ordering to products and functional types in the obvious way (note that there is no need to extend the ordering to the domain part of the function form, since this part is just a proper kind); the third rule assures the reflexivity necessary to make the relation a partial ordering.

Proposition (partial ordering of data kinds):

The relation $\delta \leqslant \eta$ is in fact a partial ordering on data kinds; i.e., is is reflexive, transitive and anti-symmetric.

Given the new sort of data kinds and the partial ordering over it, we can now address the necessary changes to the inference rules for type formation: we will use this goal to drive a development of the necessary changes to the syntax of settings, assumptions and parameter patterns.

Recall the we seek axioms and rules for deriving judgements of the form $A \vdash \theta : \delta$; in the case of atomic forms of types, i.e., variables, constants and constructors, we normally use projection from an appropriate association to obtain the kind of the atomic symbol. Since we wish now to give data kinds to terms, we should change the forms of these associations to allow for data kinds in their ranges. We can retain the forms of the original projection-based rules by making the following changes to the syntax of settings and assumptions:

$$C_T ::= \langle\{\chi{:}\delta, \ldots\}, \{S{:}\kappa\rightarrow B, \ldots\}\rangle \equiv \langle C_{TC}, C_{TS}\rangle$$
$$A_T ::= \langle\{\alpha{:}\delta, \ldots\}, \{\Sigma{:}\kappa\rightarrow B\}\rangle \equiv \langle C_{TV}, C_{SV}\rangle$$

Here the restriction on the form of kinds for type constructors allows for either form of basic type in the range: this means that we can accommodate both *data type* constructors, like the lists and trees we have seen earlier, but also full-fledged (proper) *type* constructors, such as would correspond to the following ML-style declaration[6]:

type vector α = mkvector (Nat$\rightarrow\alpha$)

That is, our data kind mechanism should allow the use of proper type constructors in the language, but also prevent analytically polymorphic functions, with their restricted domains, from being instantiated to types or type functions which involve uses of these constructors.

Note that we must change the rules for the formation of value-level settings to ensure that every value constructor associated with a given data type constructor has a type-functional domain with a data kind which is less than that of the type constructor, in the sense of the ordering on data kinds (in the last chapter we asked only that the domain functional of each value constructor have the same kind as the type constructor). In particular, we replace the original rule for formation of value constructor settings with the following one:

$$\frac{\delta_1 \leqslant C_{TS}(S_1) \quad \ldots \quad \delta_n \leqslant C_{TS}(S_n) \qquad \{\} \vdash \theta_1 : \delta_1 \quad \ldots \quad \{\} \vdash \theta_n : \delta_n}{\vdash \{k_1{:}\theta_1 \mapsto S_1, \ldots, k_n{:}\theta_n \mapsto S_n\}\ \text{vksett}}$$

These changes also affect the kindings we give to the type constants in our standard type constant setting, as follows:

$$C_{TC_0} \equiv \{*{:}1,\ 1{:}D,\ \times{:}T\times T\rightarrow D,\ \rightarrow{:}T\times T\rightarrow T\}$$

In the revised syntax for assumptions we have allowed for the possibility of proper types as constraints on type-constructor variables, since these variables range over type constructors from the setting. In the other half of the conglomerate type-level assumptions, we have used data kinds as constraints for type variables, since we will need to record information about the restrictions on variables in the assumptions. This change also affects the syntax of type parameter patterns, since these are the source from which the constraints on type variables are introduced into the assumptions:

$$\pi ::= * \mid \alpha : \delta \mid \pi\ ,\ \pi$$

We will need this change in any case to allow us to form universal quantifications which represent restrictions to data types, e.g., the quantifications $\forall\alpha{:}D.\ \alpha\rightarrow$Nat and $\forall\alpha{:}T.\ \forall\varphi{:}T\rightarrow D.\ \varphi\ \alpha\rightarrow$Nat that we used in the definitions for the size and count functions, respectively, of the last section.

This change in the syntax of type parameter patterns also has an impact on the forms of lambda abstractions over types; it is not so clear that this is necessary or desirable, since we have no need to restrict the formation of lambda abstractions according to the type/data type distinctions. That is, we do not need to distinguish between a function expressed as $\lambda\alpha{:}D.\tau[\alpha]$ and one expressed as $\lambda\alpha{:}T.\tau[\alpha]$, since the formation of functional abstractions does not depend on the presence or absence of the problematic type forms in potential type arguments[7].

[6] Note that in the terminology of Standard ML, the type/data type distinction concerns the *generativity* of a type definition (its introduction of constructor names), whereas in our terminology the distinction concerns the presence or absence of the problematic type forms.

[7] A possible exception would arise if we chose, as some authors do, to describe universal and other quantifications as applications of constants to lambda-abstracted type terms; under this approach our rule for abstractions below

More to the point, recall that we would like to derive the least possible data kind for an abstraction term, in the sense of the ordering $\leqslant$, since we will make restrictions to instantiation on the basis of the type/data type distinction; since our coercion rule allows us to data kinds coerce "upward", we can always instantiate an analytically-polymorphic expression to a type with a lesser data kind, though we do not want to allow the reverse coercion.

Our approach to the classification of type-level lambda abstractions is based on intuitions drawn from the context of Milner-style type inference for value terms: when we wish to infer the best possible type for a value term under the Milner system, we assume a minimal type for the value variables, i.e., we constrain them with a type variable in the assumptions, since type variables are minimal in an appropriate instantiation ordering on types. We then depend on constraints implied by the uses of the value variables within the term to force the minimal type constraints assumed for them "upward" toward more refined forms.

By analogy, we can derive a minimal data kind for a type lambda abstraction relative to its parameters by constraining those parameters in the antecedent judgements with the least possible corresponding data kind. Any occurrences of the problematic forms (i.e., functions and quantifications) will force the data kind derived for the result to increase appropriately; if no problematic forms occur, then no increase will be necessary. In either case, we can derive the best (i.e., least) possible data kind for the abstraction.

In order to constrain the formal parameters of a type lambda abstraction with the least possible data kind corresponding to their explicitly-declared constraint, we need a simple auxiliary operation on data kinds which will retract its argument to the corresponding least data kind under the $\leqslant$ ordering. We define this operation below, along with the inverse retraction which takes data kind arguments to the corresponding greatest data kind under the same ordering. We call these operations retractions because they are identities on the subsets of minimal and maximal data kinds, respectively.

$$1^- = 1 \qquad 1^+ = 1$$

$$D^- = D \qquad D^+ = T$$

$$T^- = D \qquad T^+ = T$$

$$V^- = V \qquad V^+ = V$$

$$(\delta \times \eta)^- = \delta^- \times \eta^- \qquad (\delta \times \eta)^+ = \delta^+ \times \eta^+$$

$$(\kappa \to \delta)^- = \kappa \to (\delta^-) \qquad (\kappa \to \delta)^+ = \kappa \to (\delta^+)$$

Given these definitions we can easily show that the operations behave as expected:

Proposition (retractions on data kinds):

The operations δ^+ and δ^- are in fact retractions on data kinds; i.e., for all data kinds δ, we have both $(\delta^+)^+ = \delta^+$ and $(\delta^-)^- = \delta^-$.

Proposition (retractions and ordering):

The operations δ^+ and δ^- retract data kinds upward and downward in the ordering, respectively; i.e., for all data kinds δ, we have $\delta^- \leqslant \delta \leqslant \delta^+$.

Using the retraction operations and based on the discussion above, we can now state the rule which governs the formation of type lambda abstraction terms:

$$\frac{\vdash \pi : \kappa \qquad A +_T \mathcal{A}(\pi)^- \vdash \theta : \delta}{A \vdash (\lambda\pi.\theta) : \kappa \to \delta}$$

(here we have implicitly extended the downward retraction operation point-wise to the range of the projected parameter assumptions $\mathcal{A}(\pi)$).

Since this rule effectively erases any type/data type distinctions made in the kind constraints of lambda abstraction parameters, we will follow the convention of writing always the more familiar maximal data kind constraints for lambda abstractions, e.g., we will write always $\lambda\alpha{:}T.\tau$ instead of $\lambda\alpha{:}D.\tau$. Note that we could take the opposite tack and use minimal data kinds, but we prefer to reserve the explicit use of data kinds for contexts such as quantification where it is important. We could also formalize this convention as a syntactic restriction, but it does not seem especially worthwhile to do so.

Returning to the formation rules for type functionals, we now proceed to the crucial case of application. Consider first a simple application of a type function to a type: we have four possible cases of data kindings, since both the type and the type function might introduce a problematic type form into the resultant type term after β reduction. Given a context corresponding to the following type-variable assumption

$$\{\varphi{:}T \to D,\ \psi{:}T \to T,\ \sigma{:}D,\ \tau{:}T\}\ ,$$

the following table shows the data kinds of the results of the four possible combinations of type functions applied to types:

	$\sigma : D$	$\tau : T$
$\varphi : T \to D$	$(\varphi\ \sigma) : D$	$(\varphi\ \tau) : T$
$\psi : T \to T$	$(\psi\ \sigma) : T$	$(\psi\ \tau) : T$

Figure 3.2.2: Data kinds for some simple type applications

Note that these result kind for the case $\varphi\ \tau$ is actually only a conservative approximation, since our language of data kinds is not rich enough to express the distinction between data type functions which have or lack free occurrences of their parameter in their body; i.e., for $\varphi \equiv (\lambda\alpha{:}T.\ \sigma)$ we would have that $(\varphi\ \tau) : D$ even though $\tau : D$. This approximation is in keeping with our decision to use an algebraic notion of data kind and is in any case ameliorated by our subject-conversion rule.

From this simple table it seems that we might be able to express the rule for the formation of type applications using just the notion of the *join* of two data kinds relative to the partial ordering on kinds (let us use infix $\sqcup$ for this join operation). That is, since $B_1 \sqcup B_2 = D$ if and only if $B_1 = B_2 = D$, we might think we could use a rule which allows us to derive for the data kind of an application just the join of the argument data kind and the range part of the functional data kind, as follows:

would have to be modified.

$$\frac{A \vdash \omega : \kappa \to \eta \qquad A \vdash \theta : \delta}{A \vdash (\omega\ \theta) : \kappa \sqcup \delta}$$

Although this rule would correctly handle all the cases in the table, it would not always be sensible; specifically, it is inadequate for more complex cases where the result kind and argument kind are incomparable in the ordering, i.e., where they have different overall "shapes". Consider, for example, the application of a function expression such as $\lambda\alpha{:}T,\beta{:}T.\ \alpha\times List\ \beta$ to a pair of arguments such as $(Nat,Nat) : D\times D$: in this case we cannot simply take the join of D and D×D since these data kinds are incomparable in the ordering; similar problems arise with functions that take higher-order functional arguments and return basic types.

The best we can do in these circumstances is to test the data kind of the actual argument for minimality: this gives us very rough information about the use of problematic constructs in its expression, information which we can then use to conditionally retract upward the range part of the data kind derived for the type functional. More formally, we note that the minimality of a data kind can be checked by comparing it to its downward retraction; we then define the following conditional retraction operation on pairs of data kinds:

$$\delta / \gamma = \textbf{if } (\gamma{=}\gamma^{-}) \textbf{ then } \delta \textbf{ else } \delta^{+}$$

Now we can express the formation rule for type applications in terms of conditional retraction as follows:

$$\frac{A \vdash \omega : \kappa \to \eta \qquad A \vdash \theta : \delta \qquad \delta \leqslant \kappa}{A \vdash (\omega\ \theta) : \eta / \delta}$$

Here we use separate data kinds δ and κ in the formation constraints for the argument and the domain part of the function in order to keep any type/data type distinctions intact in δ for the purposes of the minimality check; this in turn requires us to use the extra ordering antecedent to ensure that the "shapes" of δ and κ match.

Although this rule is incomplete on its own, its shortcomings are compensated for by the subject-conversion rule. For example, if $\varphi\equiv(\lambda\alpha{:}T.\ Nat)$ and $\tau\equiv Nat\to Bool$, then using the above rule we can only derive that $\vdash (\varphi\ \tau) : T$ but not that $\vdash (\varphi\ \tau) : D$ (we presume here that $Nat : D$). We can, however, derive the latter judgement using the subject-conversion rule, since $(\varphi\ \tau) = Nat$ by the β rule and the fact that α does not occur free in the body of φ.

To complete our review of the new rules for classification of type functionals, we note that no significant changes are necessary in the remaining rules, i.e. those for products and universal quantifications, except for the obvious change from proper types to data types in formation constraints.

3.3 Syntax and formation of type patterns

With a mechanism for classifying type functionals in place, we now proceed to a more formal description of the syntax and formation of type patterns. Our main concerns in this section include the choice of a specific syntax for type patterns, the statement of a formation rule for type patterns (based on our usual approach to pattern formation using assumption and erasure operations), the definition of the matching operation for type patterns (which yields a corresponding notion of pattern subsumption) and an explication of coverage and compatibility issues for type patterns (this will include the usual operations for depth and pattern-set generation as well as an overlap predicate).

As should be clear from this summary, our approach is based on a direct generalization of the technical mechanisms we developed for handling value-level patterns in Chapter 1; the main difference in the treatment of type patterns results from the inclusion of the variable-binding lambda-abstraction form in the syntax of type patterns, which is in turn needed to allow the expression of general type-functional patterns. A single type-functional pattern typically includes occurrences of both lambda-bound variables and the usual pattern-bound variables, which complicates the formalism beyond the level required previously for value patterns.

We have already seen in the last sections some examples of the syntax of type patterns; the following grammar describes this syntax more formally.

$$\begin{aligned} \rho &::= \rho\ ,\ \rho \mid \Phi\ \rho \cdots \rho \mid \lambda\pi^{\circ}.\rho \mid \psi\ \rho \cdots \rho : \delta \\ \Phi &::= \alpha \mid \chi \mid S \mid \Sigma{:}V\to D \end{aligned}$$

Here Φ ranges over atomic type patterns, comprising type variables, type constants, type constructors and constrained type-constructor variables. The type patterns proper include forms for pairing, applications of atomic type patterns to zero or more successive arguments, lambda abstractions over *flat parameter patterns* (see below) and data kind-constrained applications of pattern-bound type variables to zero or more arguments. We use the meta-variable ψ for pattern-bound type variables here and below in order to emphasize their special rôle, even though they are drawn from the same set of symbols as the usual type variables. Formally, we use the data kind constraint form as a means of marking pattern-bound variables to distinguish their occurrences from those of lambda-bound variables.

Note that a type variable applied to zero arguments is a type pattern in the above syntax: context-sensitive restrictions enforced by the formation rules will ensure that such occurrences are allowed only for lambda-bound variables, and that all pattern-bound variables will be applied to the full preceding context of lambda-bound parameters. The latter restriction forces a simpler and more regular form for type patterns: essentially, it amounts to requiring that they be expressed $\beta\eta$-long normal form. For example, we require the second of the following two forms, though when erased as types they would be equal:

$$\begin{aligned} &\psi : (T\to D)\to(T\times T)\to D \\ &\quad \equiv \lambda\varphi{:}T\to D.\ \lambda\alpha{:}T,\beta{:}T.\ \psi\ \varphi\ (\alpha,\beta) : D \end{aligned}$$

The $\beta\eta$-long form restriction is useful since it allows a more systematic use of type constants and lambda-bound variables in the bodies of type patterns; for example, consider the following two type patterns:

$$\lambda\alpha{:}T.\ (\psi\ \alpha : D) \times (\zeta\ \alpha : D) \equiv (\psi : T\to D) \otimes (\zeta : T\to D)$$

Here we have used a combinator $\otimes$ to stand for a term which basically lifts the product construct to the level of functions over types. Without the restriction to $\beta\eta$-long form, we would need similar "lifted" combinators for the unit type, type constructors, etc. While there is some economy of expression in such a combinator-based approach, it requires at the least a schematically-kinded set of combinators for each possible domain kind. It also is not clear that higher-order combinators

would be the most natural way to express such patterns, so we prefer to use instead the explicit lambda abstractions.

The use of flat parameter patterns above is motivated by a desire to more easily express type patterns over product kinds and to more easily specify coverage restrictions for the same. By way of example, assume that we wish to write a set of type patterns which cover the kind $T\times T\rightarrow T$; we might use patterns of the form $(\lambda\varepsilon{:}T\times T.\ \rho)$, but we would then have to cover separately any possible arguments of the forms $(\lambda\alpha{:}T,\beta{:}T.\ \alpha)$ and $(\lambda\alpha{:}T,\beta{:}T.\ \beta)$, which represent the left and right component types, respectively. Since we have only a single variable as a name for what is essentially two type parameters, we would have to use patterns of the form $(\lambda\varepsilon{:}T\times T.\ \pi_i\ \varepsilon)$, where π_i would be a lambda abstraction expressing the ith projection function for two type arguments.

On the other hand, if we use patterns of the form $(\lambda\gamma{:}T,\eta{:}T.\ \rho)$, then we can cover separately the argument forms corresponding to the two component types using just the simple patterns $(\lambda\gamma{:}T,\eta{:}T.\ \gamma)$ and $(\lambda\gamma{:}T,\eta{:}T.\ \eta)$. In addition, the specification of the coverage restrictions for sets of type patterns over data kinds is also considerably simplified if we can be assured that we need not handle complex patterns involving projection functions for products.

Although it might be possible to define the coverage predicate so as to handle the first example above, we feel that it is better design practice to simply rule out the more complex projection-based patterns as a matter of syntactic restriction, since this brings the issue out at the level of the language specification rather than burying it in what would probably be obscure and difficult error messages about coverage problems in an implementation.

We can enforce the use of the simpler forms of patterns by way of a restriction that constraints on type parameters should contain no explicit uses of products; to this end we define a notion of *flat kinds* according to the following grammar:

$$\kappa^\circ\ ::=\ 1\ \mid\ T\ \mid\ V\ \mid\ \kappa\rightarrow\kappa^\circ$$

These are just proper kinds without a product form; note that we do not need the corresponding notion of flat data kinds, since we only need flat kinds in parameter constraints, where by convention we use only proper kinds.

Using flat kinds as a basis, we can specify the appropriate notion of flat parameter patterns simply by using flat kinds in place of arbitrary kinds, as follows:

$$\pi^\circ\ ::=\ *\ \mid\ \alpha:\kappa^\circ\ \mid\ \pi^\circ\ ,\ \pi^\circ$$

Having resolved this rather minor issue, we can now proceed to describe formation rules for type patterns: recall that our approach to expression formation for pattern abstractions requires a (rather natural) notion of formation for the patterns constituting the abstraction. For value patterns, we expressed the notion of pattern formation in terms of erasure and assumption operations: erasure allowed us to convert a pattern with explicit type constraints into the corresponding unconstrained term. The erased term was then required to be well-formed with respect to the set of assumptions corresponding to its own constraints, which were gathered out by an assumption projection operation.

Erasure for type patterns is straightforwardly defined as follows:

$$\mathcal{E}(\Phi) = \textbf{if } \Phi\equiv\Sigma{:}V\rightarrow D \textbf{ then } \Sigma \textbf{ else } \Phi$$
$$\mathcal{E}(\rho,\rho') = \mathcal{E}(\rho)\ ,\ \mathcal{E}(\rho')$$
$$\mathcal{E}(\Phi\ \rho_1\ \cdots\ \rho_n) = \Phi\ \mathcal{E}(\rho_1)\ \cdots\ \mathcal{E}(\rho_n)$$
$$\mathcal{E}(\lambda\pi^\circ.\rho) = \lambda\pi^\circ.\ \mathcal{E}(\rho)$$
$$\mathcal{E}(\psi\ \rho_1\ \cdots\ \rho_n : \delta) = \psi\ \mathcal{E}(\rho_1)\ \cdots\ \mathcal{E}(\rho_n)$$

Note here that we must remove constraints from both the usual pattern-variable case (the last one) and the explicitly-constrained type-constructor variables. Also note that we do not need to erase constraints on lambda-bound parameters, since these would occur constrained in the corresponding type term anyway.

The definition of the assumption operation for type patterns is rather more complicated than the corresponding operation on value patterns, for several reasons: first, we must handle two kinds of variables in the resultant assumptions; second, we must account for the abbreviated form of explicit constraint we have used for pattern-bound type variables; third, we must distinguish between lambda- and pattern-bound type variables, checking that they do not clash and enforcing different linearity restrictions on them; finally, we must enforce the restriction that all type function patterns be expressed in η-long form.

Since the results of assumption projection for type patterns are a conglomerate structure , it is useful to make the following abbreviation for the empty assumption:

$$\varnothing_T\ \triangleq\ \langle\{\},\ \{\}\rangle$$

We also extend the usual operation for the domain of an assumption to a tuple of type parameters by taking the point-wise projections of the parameters and then weakly merging the results (the weak merge ensures that lambda-bound variables are used only once in any given context; this is necessary for the η-long forms to be sensible). This use of assumption projection before its own definition may look ill-founded, but recall that type parameters are just a subset of type patterns in general: we have already defined all the cases we need to handle parameters (the non-subscripted version of projection is defined below as simply using an initially empty context).

$$\mathrm{dom}(\langle\pi_1,\ \ldots,\ \pi_n\rangle) = \mathrm{dom}(\mathcal{A}(\pi_1)\oplus\cdots\oplus\mathcal{A}(\pi_n))$$

Using this notation, we can define the first few cases of assumption projection: here the extra subscripted argument Π is a tuple of type parameter patterns holding the current context of lambda-bound type variables:

$$\mathcal{A}_\Pi(\chi) = \varnothing_T$$
$$\mathcal{A}_\Pi(\alpha) = \textbf{if } \alpha\in\mathrm{dom}(\Pi) \textbf{ then } \varnothing_T \textbf{ else fail}$$
$$\mathcal{A}_\Pi(\Sigma{:}V\rightarrow D) = \langle\{\},\ \{\Sigma{:}V\rightarrow D\}\rangle$$
$$\mathcal{A}_\Pi(\lambda\pi.\rho) = \mathcal{A}_{\Pi,\pi}(\rho)$$

The first clause of the definition simply returns the empty assumption for type constants. The second clause assumes that an undecorated type variable is lambda-bound: if so, it must be contained in the current context, in which case we return the empty assumption, since the variable is bound, not free (if the variable is not in the current context the operation fails). The third clause handles kind-constrained type-constructor variables, simply recording the variable and constraint in the appropriate component of the conglomerate assumption[8].

[8] The third clause here is actually inadequate, since we

The fourth clause returns for abstractions just the assumptions contained in their body, but with respect to an extended context. For a more usual approach we might change the second and fourth clauses to include lambda-bound

variables in the assumptions of the body and then eliminate them again at the point of binding, but our definition here seems to better convey the different rôles of lambda- and pattern-bound variables by keeping them separate throughout.

Another difference in our treatment of type variables and type-constructor variables is that we wish to enforce linearity restrictions for the former but not for the latter. The reasons for allowing non-linearity of type-constructor variables are two-fold: first, it is at least feasible, since type-constructor variables are bound to type constructors, atomic forms which we can easily compare for equality; type variables, on the other hand, are bound to types, which we can compare for equality, but only at the expense of some reduction. The second and overriding reason is that non-linear uses of type-constructor variables allow us to express some useful functions, such as the equality function on data types (we present a definition for equality in Section 5.1); we know of no such motivating examples which require non-linear type variables, so we prefer to avoid any of the complications which non-linear type variables might entail, even if these are limited to pragmatic ones involving the inefficiency of type reduction or the extra potential for programming errors.

Based on these considerations, we define a conglomerate merge operation for type-level assumptions, with infix notation $\oplus'$, in order to simultaneously merge the type-variable component with a standard merge (failing for overlapping argument domains) and the type-constructor variable component with a weak merge (which allows overlapping argument domains).

$$A \oplus' B \triangleq \langle A_{TV} \oplus B_{TV},\ A_{TS} + B_{TS} \rangle$$

Using this notation, we can define assumption projection for pair and application forms of type patterns as follows:

$$\mathcal{A}_\Pi(\rho,\rho') = \mathcal{A}_\Pi(\rho) \oplus' \mathcal{A}_\Pi(\rho')$$
$$\mathcal{A}_\Pi(\Phi\ \rho_1 \cdots \rho_n) = \mathcal{A}_\Pi(\Phi) \oplus' \mathcal{A}_\Pi(\rho_1) \oplus' \cdots \oplus' \mathcal{A}_\Pi(\rho_n)$$

The definition of assumption projection for the last form of type patterns, pattern-bound variables, requires some further machinery. First we define an operation which simply extracts the kind information from a type parameter pattern:

$$\mathcal{K}(*) = 1$$
$$\mathcal{K}(\alpha{:}\delta) = \delta$$
$$\mathcal{K}(\pi,\nu) = \mathcal{K}(\pi) \times \mathcal{K}(\nu)$$

We extend this operation to tuples of type parameters pointwise, in the obvious fashion:

$$\mathcal{K}^*\ \langle \pi_1, \ldots, \pi_n \rangle = \langle \mathcal{K}(\pi_1), \ldots, \mathcal{K}(\pi_n) \rangle$$

Given a tuple of kinds, we use the following notation to denote a curried functional kind based on the tuple components, along with a given data kind as range:

$$\langle\rangle \to \delta \triangleq \delta$$
$$\langle \kappa_1, \ldots, \kappa_n \rangle \to \delta \triangleq \kappa_1 \to (\cdots (\kappa_n \to \delta) \cdots)$$

Given all this, we can define assumption projection for pattern-bound variables as follows:

$$\mathcal{A}_\Pi((\psi\ \rho_1 \cdots \rho_n){:}\delta) =$$
if $\mathcal{E}^*(\Pi) = \langle \rho_1, \ldots, \rho_n \rangle$ **and** $\psi \notin \mathrm{dom}(\Pi)$ **then**
$\langle \{\psi{:}\ \mathcal{K}^*(\Pi) \to \delta\}, \{\} \rangle$ **else fail**

This

As mentioned above, it is convenient to define a version of projection for use at the outer level of call that provides an initial empty context of type parameters:

$$\mathcal{A}(\rho) = \mathcal{A}_{\langle\rangle}(\rho)$$

The following examples illustrate some simple calls of assumption projection:

$$\mathcal{A}(\lambda\alpha{:}T.\ \lambda\beta{:}T.\ \varphi\ \alpha\ \beta\ :\ D) = \{\varphi{:}T \to T \to D\}$$
$$\mathcal{A}(\lambda\psi{:}T \to T.\ (\lambda\alpha{:}T, \beta{:}T.\ \varphi\ \psi\ (\alpha, \beta)\ :\ D\ ,\ \zeta\psi\ :\ D)) =$$
$$\{\varphi{:}(T \to T) \to (T \times T) \to D,\ \zeta{:}(T \to T) \to D\}$$
$$\mathcal{A}(\lambda\alpha{:}T.\ (\lambda\alpha{:}T, \beta{:}T.\ \varphi\ \alpha\ (\alpha, \beta)\ :\ D)) = \textbf{fail}$$

Only the third case here really bears mention: it fails because of the two successive bindings of the same type variable, α, which makes the later application of the pattern-bound φ to two occurrences of α nonsensical (the failure itself results from the weak merge of the the two type parameters containing α).

We can now state the inference rule governing the formation of type patterns as follows: a type pattern is well-formed with a certain data kind just when its erasure is well-formed with that data kind under the assumptions contained in the type pattern. Note that no global context is needed for the formation of type patterns since globally-bound variables have no meaning *within the pattern* (though, of course, they have meaning in the corresponding right-hand side expression body).

$$\frac{\mathcal{A}(\rho) \vdash \mathcal{E}(\rho) : \delta}{\vdash \rho : \delta}$$

The following examples of type-pattern formation, both positive and negative, should help to clarify the definitions above:

$$\vdash (\lambda\alpha{:}T.\ \lambda\beta{:}T.\ \varphi\ \alpha\ \beta\ :\ D) : T \to T \to D$$
$$\nvdash (\lambda\alpha{:}T.\ (\psi\ \alpha\ :\ D) \times (\psi\ \alpha\ :\ D)) : T \to D$$
$$\nvdash (\lambda\alpha{:}T.\ (\alpha\ \alpha\ :\ D) \times (\psi\ \alpha\ :\ D)) : T \to D$$
$$\nvdash (\lambda\alpha{:}T.\ \psi\ :\ T \to D) : T \to T \to D$$

The first example is indeed well-formed with the given type; note that the projected assumption will give the same kind for φ as for the whole pattern itself. The second example is ill-formed because there is a clash between the two pattern-bound variables: this will cause the assumption projection operation to fail, since the weak merge of the two assumptions will fail. The third example is ill-formed because there is a clash between the pattern-bound occurrence and the lambda-bound occurrence of the same variable (α), which will cause failure by way of the membership check in the last clause of the assumption projection definition. The fourth example is ill-

should check that non-linear occurrences of type constructor variables all have the same declared kind; this will be resolved by fiat in the next chapter when we change the syntax of constructors and their variables.

formed because the

Beyond simple formation, we must also address for type patterns the usual issues of coverage and compatibility, just as we did for patterns in Chapter 1. In keeping with the approach taken there, we first define a pattern-matching operation between type patterns and type terms, using this as the basis for a subsumption ordering on patterns and also, of course, for the actual pattern-matching which underlies the β reduction of type-pattern abstractions.

As for the value level, pattern-matching for type patterns and type terms will result in a substitution of terms for variables occurring in the pattern. In the case of type patterns, the variables in the domain of the substitution will be the pattern-bound type variables, as opposed to the lambda-bound variables in the pattern, since the latter do not occur freely. We must also account in these substitutions for type constructor variables, which may be bound either to concrete type constructors or to other type-constructor variables. Since there are several contexts in which either a concrete type constructor or a type constructor variable is appropriate, it will be useful to define a syntactic sort which effects the union of these two atomic sorts. Using meta-variables s, t to range over this combined sort and S_T, R_T to range over type-level substitutions, we have the following syntax:

$$s ::= S \mid \Sigma$$
$$S_T ::= \langle [\theta/\alpha, \ldots], [s/\Sigma, \ldots] \rangle \equiv \langle S_{TV}, S_{TS} \rangle$$

Here our use of subscripts on the two components of the conglomerate substitutions is meant to suggest projection of the components by labels (in fact, we will later consider type-level substitutions as conglomerated with value-level substitutions in turn, which accounts for our use of subscripts S_{TV} and S_{TS}). We warn the reader that we use the meta-variable S in two different ways here, both for concrete type constructors and for substitutions; aside from the just previous definitions, however, context should suffice to disambiguate the two uses (at the type level, the substitutions will also always be subscripted as above).

It will also be convenient to refer to the empty type-level substitution more succinctly, for which purpose we use the following abbreviation (the context of use will allow us to distinguish this from the same notation for the empty assumption we used above):

$$\varnothing_T \equiv \langle [], [] \rangle$$

For atomic type patterns, we define matching as follows: note that both constants *and* lambda-bound variables must coincide exactly in order to match.

$$\mathcal{M}(\chi,\chi) = \varnothing_T$$
$$\mathcal{M}(\alpha,\alpha) = \varnothing_T \quad \text{(NB: same } \alpha\text{)}$$

We define matching for type constructors (either concrete ones or variables) as follows; note that a match of a concrete pattern and a variable expression must fail: this has implications for reduction order, since we may want to resolve a variable on the right-hand side before matching, if possible.

$$\mathcal{M}(S,R) = \text{if } S \equiv R \text{ then } \varnothing_T \text{ else } \textbf{fail}$$
$$\mathcal{M}(\Sigma,s) = \langle [], [s/\Sigma] \rangle$$
$$\mathcal{M}(S,\Sigma) = \textbf{fail}$$

In order to properly match pairing and application forms, we need to apply substitutions from left-more constituents before matching right-more constituents: this would not normally be required, but we have allowed for non-linear uses of type-constructor variables, so we must instantiate (i.e., substitute for) occurrences of type-constructor variables which occur "to the right" of any occurrences which get instantiated to concrete constructors or other variables. For example, we want the first of the following two matches to succeed and the second to fail:

$$\mathcal{M}((\Sigma{:}V \to D,\ \Sigma{:}V \to D),\ (\text{List},\ \text{List})) = [\text{List}/\Sigma]$$
$$\mathcal{M}((\Sigma{:}V \to D,\ \Sigma{:}V \to D),\ (\text{List},\ \text{Tree})) = \textbf{fail}$$

Given this consideration, we define as follows:

$$\mathcal{M}((\rho,\rho'),(\theta,\theta')) = \textbf{let } S = \mathcal{M}(\rho,\theta) \textbf{ in } \mathcal{M}(\rho'[S],\ \theta'[S])$$
$$\mathcal{M}(\Phi\ \rho_1 \cdots \rho_n,\ \varphi\ \theta_1 \cdots \theta_n) =$$
$$\quad \textbf{let } S_0 = \mathcal{M}(\Phi,\varphi)$$
$$\quad \textbf{and for } i=1 \text{ to } n,$$
$$\quad\quad R_i = \mathcal{M}(\rho_i[S_{i-1}],\ \theta_i[S_{i-1}]) \textbf{ and } S_i = S_{i-1} \circ R_i$$
$$\quad \textbf{in } S_n$$

For lambda abstractions, we match the bodies of the two abstractions after first renaming the bound variables in the type pattern argument to match those of the type term argument:

$$\mathcal{M}(\lambda\pi.\rho,\lambda\nu.\theta) = \mathcal{M}(\rho[\mathcal{M}(\pi,\nu)],\ \theta)$$

Finally, a pattern-bound variable matches an arbitrary type, though we must account for the η-long forms we have required by first abstracting the type term argument appropriately (note that we depend here on the bound-variable renaming which was effected by the lambda-abstraction clause of the definition for matching):

$$\mathcal{M}((\psi\ \rho_1 \cdots \rho_n){:}\delta,\theta) = \langle [(\lambda\rho_1.\ \ldots\ \lambda\rho_n.\theta)/\psi],\ [] \rangle$$

Note that we commit a slight abuse here in that the ρ_i are unconstrained: this matters little, since in the context of reduction we need no kind information and in the context of subsumption we are only concerned with well-kinded types anyway.

For any other combinations of type patterns and type terms, i.e., other than those defined above, the matching operation fails. Note that we assume that type term arguments to matching will be reduced to normal form as a preliminary to matching them against some type pattern.

$$\mathcal{M}(\rho,\theta) = \textbf{fail}, \text{ otherwise}$$

We state without proof the following characteristic property of the matching operation for type patterns:

Proposition (correctness of type matching):

For all well-formed type patterns ρ and type functionals θ, if $\mathcal{M}(\rho,\theta) \neq \textbf{fail}$ then $\mathcal{E}(\rho)[\mathcal{M}(\rho,\theta)] = \theta$.

Beyond this correctness property, we expect matching to be complete in the sense that it will return a minimal substitution:

Proposition (minimality of type matching):

For all well-formed type patterns ρ and type functionals θ, if $\mathcal{M}(\rho,\theta) = S$ then S is minimal, in the sense that for any other

substitution R such that $R\rho = \theta$, there exists a substitution T such that $R = S \circ T$.

Using pattern-matching, we can define a subsumption relation between type patterns and type terms just as we did for value patterns and expressions; we then extend this in the obvious way to pairs of type patterns using erasure.

$\rho \sqsubseteq \theta$ iff $\mathcal{M}(\rho,\theta) \neq$ **fail**
$\rho \sqsubseteq \upsilon$ iff $\rho \sqsubseteq \mathcal{E}(\upsilon)$
$\rho \sqsubset \upsilon$ iff $\rho \sqsubseteq \upsilon$ and $\neg\, \upsilon \sqsubseteq \rho$

Our use of the symbol $\sqsubseteq$ should suggest that we expect subsumption to be a partial ordering; in fact, we must qualify this slightly, as follows:

Proposition (subsumption ordering):

The subsumption relation $\rho \sqsubseteq \upsilon$ is a partial ordering on linear type patterns, up to the equivalence induced by renaming of variables (for pattern-bound variables this is needed for anti-symmetry; for lambda-bound variables it is natural to treat type patterns as equivalent up to renaming).

If there is some type term which two patterns will both successfully match, we say that the two type patterns overlap:

$\rho \approx \upsilon$ iff there is a θ such that both $(\rho \sqsubseteq \theta)$ and $(\upsilon \sqsubseteq \theta)$

Expressed this way, overlap is a non-effective relation, since we must check an infinite number of type terms to see if any match against both patterns; an effective version of overlap can easily be given, however, along the lines of the effective overlap relation for value patterns, simply by examining the definition of type-pattern matching.

For atomic type patterns we have as follows:

$\mathcal{O}(\chi,\chi)$ = true
$\mathcal{O}(\alpha,\alpha)$ = true *{NB: same α}*
$\mathcal{O}(S,R)$ = if $S \equiv R$ then true else false
$\mathcal{O}(s,t)$ = true, otherwise

Note that constants and type constructors overlap only if when they coincide, i.e., they are the same same symbol; type-constructor variables overlap concrete type constructors and other type-constructor variables. What may seem surprising, at first, is that type variables also overlap only if they coincide: this is because undecorated type variables are not pattern-bound, but rather lambda-bound variables, and thus have a significance and meaning which is specific to the context of their lambda-binding, rather than the usual significance of matching an arbitrary term.

On the other hand, pattern-bound variables, which we have marked syntactically with constraints, overlap an arbitrary pattern, since they match an arbitrary type term in the definition of pattern-matching above:

$\mathcal{O}(\rho, (\zeta\ \upsilon_1 \cdots \upsilon_n){:}\gamma)$ = true
$\mathcal{O}((\psi\ \rho_1 \cdots \rho_n){:}\delta, \upsilon)$ = true

For non-atomic type patterns, we define overlap as follows:

$\mathcal{O}(\Phi\ \rho_1 \cdots \rho_n, \Theta\ \upsilon_1 \cdots \upsilon_n) = \mathcal{O}(\Phi,\Theta)$ and $\mathcal{O}(\rho_1,\upsilon_1)$ ... and $\mathcal{O}(\rho_n,\upsilon_n)$
$\mathcal{O}((\rho,\rho'),(\upsilon,\upsilon')) = \mathcal{O}(\rho,\upsilon)$ and $\mathcal{O}(\rho',\upsilon')$
$\mathcal{O}(\lambda\pi.\rho, \lambda\pi'.\upsilon) = \mathcal{O}(\rho, \upsilon[\pi/\pi'])$

Type applications overlap just when both the (atomic) function parts and the corresponding argument parts overlap; similarly, pairs overlap only when the corresponding left and right halves overlap. Lambda-abstracted patterns overlap just when their bodies overlap, after renaming bound variables in one of the patterns to match the other (this renaming justifies our insistence above that occurrences of lambda-bound variables must coincide exactly in order to overlap).

Finally, we define the overlap relation to be false for any combinations of forms that we have not given explicitly above, just as type-pattern matching fails for combinations of forms not explicitly defined as successful in its definition above:

$\mathcal{O}(\rho,\upsilon)$ = false, otherwise

It will be convenient to define the *independence* relation between type patterns as the negation of overlap (an effective definition is determined by the effective definition of overlap above):

$\rho \not\approx \upsilon$ iff $\neg(\rho \approx \upsilon)$

We can now proceed to a description of type pattern compatibility, i.e., the issue of when a set of type patterns is such that a unique match can always be found among them for any given type term. As for value patterns, this depends on the matching regimen in force—in fact, due to our use of a common conceptual and notational basis,we can use the same rules here as for value patterns. For a first-match regimen, any set of type patterns is compatible:

$$\{\rho_1, \ldots, \rho_n\} \text{ compatible}$$

For a best-match regimen, we must ensure that all pairs of type patterns are independent (thus only one pattern will ever match):

$$\frac{\rho_i \not\approx \rho_j \qquad (1 \leqslant i \neq j \leqslant n)}{\{\rho_1, \ldots, \rho_n\} \text{ compatible}}$$

For an only-match regimen, we need the same notion of pattern-set coherence as for the value case:

$$\frac{\{\rho_1, \ldots, \rho_n\} \text{ coherent}}{\{\rho_1, \ldots, \rho_n\} \text{ compatible}}$$

And, as for the value case, we define coherence rather non-effectively as follows:

$\{\rho_1, \ldots, \rho_n\}$ coherent iff for all θ of the same kind,
there is some i, $1 \leqslant i \leqslant n$, such that for all j, $1 \leqslant j \leqslant n$,
either $j = i$ or $\neg(\rho_j \sqsubseteq \theta)$ or $\rho_j \sqsubset \rho_i$

Compatibility specifies when a set of patterns allows for unique matches relative to some matching regimen: we also want to restrict sets of type patterns in an abstraction so that at least one pattern will match any possible argument. Toward this end we define a notion of *coverage* between sets of type patterns and data kinds, just as we defined coverage for sets of value patterns and types. Our first step is to define a notion of *depth* for type patterns: this allows us to account for the maximum complexity of structure we must consider in a

possible argument. Using $\mathcal{D}(\rho)$ for the depth operation, we have:

$$\mathcal{D}(\Phi\ \rho_1 \cdots\ \rho_n) = 1+\max(\mathcal{D}(\rho_1), \ldots, \mathcal{D}(\rho_n))$$
$$\mathcal{D}(\rho,\upsilon) = 1+\max(\mathcal{D}(\rho),\mathcal{D}(\upsilon))$$
$$\mathcal{D}(\lambda\pi.\rho) = \mathcal{D}(\rho)$$
$$\mathcal{D}(\psi\ \rho_1 \cdots\ \rho_n : \delta) = 0$$

Application patterns are just deeper than their maximum depth argument, since the applied type function itself constitutes matchable structure; similar remarks apply to pairs of type patterns. Lambda-abstracted patterns, on the other hand, are only as deep as their bodies. Finally, applied pattern-bound variables have depth of zero, since they display no inherent structure at all. Note that the depth of an atomic pattern occurring alone is just 1, since it has no arguments (i.e., the maximum of an empty set is taken to be just 0).

Given this notion of depth, we proceed by analogy with the value case to define an operation which generates sets of schematic terms (i.e., patterns) of a given data kind. We will need first an auxiliary function which will return fresh parameter patterns of a given kind (this is just a generalization of the usual "fresh variables" operation).

$$\mathcal{P}(1) = *$$
$$\mathcal{P}(T) = \alpha{:}T \quad (\text{fresh } \alpha)$$
$$\mathcal{P}(V) = \alpha{:}V \quad (\text{fresh } \alpha)$$
$$\mathcal{P}(\kappa\times\eta) = \mathcal{P}(\kappa)\ ,\ \mathcal{P}(\eta)$$
$$\mathcal{P}(\kappa\to\eta) = \alpha : \kappa\to\eta \quad (\text{fresh } \alpha)$$

At the value level, we generated patterns based on two arguments, the maximum depth of the patterns desired and their type; at the type level we will need an extra argument for recursive calls in order to keep track of the current context of lambda-bound type variables—we write this as a subscript, using Π again to range over tuples of parameter patterns, just as for the assumption projection operation. Given this notation, we can specify zero-depth type patterns of a given data kind as just a fresh (pattern-bound) type variable applied to the appropriate parameter context:

$$\mathcal{G}_\Pi{}^0(\delta) = \{\psi\ \rho_1 \cdots\ \rho_n : \delta\},\ \text{where } \langle\rho_1, \ldots, \rho_n\rangle = \mathcal{E}(\Pi)\ (\text{fresh } \psi)$$

Note that we must erase the parameter patterns in order to make them into proper, unconstrained arguments for the type application. Note also that we handle the zero-depth case for unit and functional kinds as separate cases below.

The unit kind has only a single pattern:

$$\mathcal{G}_\Pi{}^n(1) = \{*\}$$

We can expose no structure for patterns of variable kind or of kind T, since we have no kind structure to work from in the former case and no appropriate structure to generate in the latter (recall that we do not allow structured patterns over proper types):

$$\mathcal{G}_\Pi{}^{n+1}(V) = \{\alpha{:}V\} \quad (\text{fresh } \alpha)$$
$$\mathcal{G}_\Pi{}^{n+1}(T) = \{\alpha{:}T\} \quad (\text{fresh } \alpha)$$

We can generate patterns for product kinds by generating all combinations of pairs of patterns from the sets generated for the left and right halves of the product:

$$\mathcal{G}_\Pi{}^{n+1}(\delta\times\eta) = \{(\rho,\upsilon) \mid \rho\in\mathcal{G}_\Pi{}^n(\delta) + \upsilon\in\mathcal{G}_\Pi{}^n(\eta)\}$$

We generate sets of patterns for function kinds using a lambda abstraction over a fresh parameter pattern of the appropriate domain kind and then recursively generating pattern bodies based on the range kind; note that we must add the fresh parameter pattern to the tuple representing the current context of lambda-bound type variables:

$$\mathcal{G}_\Pi{}^n(\kappa\to\delta) = \{\lambda\pi.\rho \mid \pi=\mathcal{P}(\kappa),\ \rho\in\mathcal{G}_{\Pi,\pi}{}^n(\delta)\}$$

Finally, in the crucial case of the data type constant, we generate patterns for the unit type, products of patterns of lesser depth, structural patterns of the appropriate depth (to be defined below) and pattern-bound variables of the appropriate depth. We use auxiliary operations in these last two cases in order to simplify the presentation, since we must appeal to the constructor setting for the structural case and to an extra recursion in the variable case.

$$\mathcal{G}_\Pi{}^{n+1}(D)=\{1\} \cup \{(\rho_1\times\rho_2) \mid \rho_i\in\mathcal{G}_\Pi{}^n(D)\} \cup \mathcal{S}_\Pi{}^{n+1} \cup \bigcup \{\mathcal{V}_\Pi{}^{n+1}(\{\alpha\},\kappa) \mid \Pi(\alpha)=\kappa\}$$

For structures, we must generate all possible applications of concrete type constructors to appropriately-kinded patterns arguments, where the appropriate kinds are given by the global setting. Note that use a downward retraction on the domain kind of the type constructor for the purposes of the recursive call, in order to get as much structure as possible out in the resulting patterns (proper kinds with constants T would result in largely degenerate patterns).

$$\mathcal{S}_\Pi{}^{n+1} = \{S\ \rho \mid S\in\mathrm{dom}(C_{TS}),\ C_{TS}(S)=\kappa\to D,\ \rho\in\mathcal{G}_\Pi{}^n(\kappa^-)\}$$

The auxiliary operation for generating variable-based patterns takes two arguments, the first a set of patterns

$$\mathcal{V}_\Pi{}^{n+1}(R,T) = R$$
$$\mathcal{V}_\Pi{}^{n+1}(R,D) = R$$
$$\mathcal{V}_\Pi{}^{n+1}(R,\kappa\to\eta) = \mathcal{V}_\Pi{}^{n+1}(\{\rho\ \upsilon \mid \rho\in R,\ \upsilon\in\mathcal{G}_\Pi{}^n(\kappa^-)\},\eta)$$

Since this definition is a bit complex, it will perhaps be helpful to run through an example: assume that we have a current type-parameter context $\Pi=\langle\varphi{:}T\to D,\ \alpha{:}T\rangle$ and that the global type-constructor setting gives us the usual constructors for natural numbers and lists, along with their kinds, i.e., $C_{TS}=\{Nat{:}1\to D,\ List{:}T\to D\}$. Then we generate a complete set of type patterns of atomic data kind D up to depth 1 as follows:

$$\mathcal{G}_\Pi{}^1(D)=\{1\} \cup \{(\rho_1\times\rho_2) \mid \rho_i\in\mathcal{G}_\Pi{}^0(D)\} \cup \mathcal{S}_\Pi{}^1 \cup \bigcup \{\mathcal{V}_\Pi{}^1(\{\alpha\},\kappa) \mid \Pi(\alpha)=\kappa\}$$
$$=\{1\} \cup \{(\rho_1\times\rho_2) \mid \rho_1\in\{\psi\ \varphi\ \alpha : D\},\ \rho_2\in\{\zeta\ \varphi\ \alpha : D\}\} \cup \mathcal{S}_\Pi{}^1 \cup \bigcup \{\mathcal{V}_\Pi{}^1(\{\alpha\},\kappa) \mid \Pi(\alpha)=\kappa\}$$
$$=\{1,\ (\psi\ \varphi\ \alpha : D)\times(\zeta\ \varphi\ \alpha : D)\} \cup \{Nat\ *,\ List\ (\omega\ \varphi\ \alpha : D)\} \cup \mathcal{V}_\Pi{}^1(\{\alpha\},T) \cup \mathcal{V}_\Pi{}^1(\{\varphi\},T\to D)$$
$$=\{1,\ (\psi\ \varphi\ \alpha : D)\times(\zeta\ \varphi\ \alpha : D),\ Nat\ *,\ List\ (\omega\ \varphi\ \alpha : D)\} \cup \{\alpha\} \cup \mathcal{V}_\Pi{}^1(\{\varphi\ \upsilon \mid \upsilon\in\mathcal{G}_\Pi{}^0(T^-)\},D)$$
$$=\{1,\ (\psi\ \varphi\ \alpha : D)\times(\zeta\ \varphi\ \alpha : D),\ Nat\ *,\ List\ (\omega\ \varphi\ \alpha :$$

$D), \alpha\}$
$\quad \cup\ \mathcal{V}_{\Pi}^{1}(\{\varphi\ (\nu\ \theta\ \alpha : D)\}, D)$
$= \{1,\ (\psi\ \varphi\ \alpha : D) \times (\zeta\ \varphi\ \alpha : D),\ \mathrm{Nat}\ *,\ \mathrm{List}\ (\omega\ \varphi\ \alpha :$
$D),\ \alpha\} \cup \{\varphi\ (\nu\ \theta\ \alpha : D)\}$
$= \{1,\ (\psi\ \varphi\ \alpha : D) \times (\zeta\ \varphi\ \alpha : D),\ \mathrm{Nat}\ *,\ \mathrm{List}\ (\omega\ \varphi\ \alpha :$
$D),\ \alpha,\ \varphi\ (\nu\ \theta\ \alpha : D)\}$

Given the definition of pattern-set generation for data kinds,we can now finally state the definition of the coverage relation between sets of patterns and a data kind. The definition is precisely analogous to the definition of coverage at the value level, except that we must here provide an empty tuple as the initial context of type parameters for the purposes of the recursion on the data kind. As for the value-level definition, we need only account for type patterns which are as deep as the deepest pattern in the argument set.

$\{\rho_1, \ldots, \rho_m\} \succ \delta$ iff for all $\upsilon \in \mathcal{G}_{\langle\rangle}^{n}(\delta)$, some $\rho_i \sqsubseteq \upsilon$,
where $n = \max\{\mathcal{D}(\rho_i) \mid 1 \leq i \leq m\}$

By way of example, the following instance of the coverage relation holds, assuming the constructor setting used in the example above (the generated set of patterns is similar to that from the example above, but note that we have no type variables in context, and thus need not apply pattern-bound variables or include them as cases of patterns):

$\{1,\ (\beta : D) \times (\gamma : D),\ \mathrm{Nat}\ *,\ \mathrm{List}\ (\delta : D)\} \succ D$

An alternative means of covering the data type constant D is to use a type-constructor variable instead of the specific constructors for naturals and lists: this frees us from the context of a particular constructor setting, so that the following instance of the coverage relation will hold in an arbitrary setting:

$\{1,\ (\beta : D) \times (\gamma : D),\ (\Sigma{:}X \to D)\ (\chi : X)\} \succ D$

Finally, the next example represents in some sense an extension of the previous one, in that we now cover the first-order type functions which return a data type by lambda-abstracting each of the previous patterns. Note that all of the pattern-bound type variables are now lifted to be type functions and that we must include a case for the identity function on types, representing an occurrence of the "component type" of the type function:

$\{\lambda\varepsilon{:}T.\ 1,\ \lambda\varepsilon{:}T.\ (\psi\ \varepsilon : D) \times (\zeta\ \varepsilon : D),\ \lambda\varepsilon{:}T.\ (\Sigma{:}X \to D)\ (\chi$
$\varepsilon : X),\ \lambda\varepsilon{:}T.\ \varepsilon\} \succ T \to D$

This completes our discussion of the issues of compatibility and coverage for type patterns: we now employ these notions in the next section to give an account of formation for type-pattern abstractions.

3.4 Syntax and formation of expressions

We now proceed to consider the formation of expressions, including especially the type-pattern abstractions which realize analytical polymorphism. The context-free syntax of expressions is given by the following grammar:

$e ::= c \mid x \mid k \mid K \mid e,e \mid (p \Rightarrow e)^{+} \mid e\ e \mid \mid$
$(\rho \Rrightarrow e)^{+} \mid e{\downarrow}\theta \mid \mu x{:}\theta.e$

Here e ranges over expressions, c and x over value constants and variables, respectively, k and K over type constructor and type-constructor variables, respectively, and p and ρ over value and type patterns, respectively. The forms include the usual constants, variables, constructors, pairing, pattern abstraction, application and recursion constructs, along with new forms for type-pattern abstraction and type instantiation. Although we will restrict the type constraints in value patterns and recursion to types of kind T as part of the formation rules, note that for instantiation we allow arbitrarily-kinded type functionals: this will in particular allow us to express instantiation to (higher-order) type functions representing data structures.

The formation rules for the two pattern-abstraction forms will depend on formation rules for the two sorts of patterns. We have discussed type patterns at length in the previous section; the syntax of value patterns is much the same as for value patterns in the simply-typed calculus, but with minor changes for constructors and constructor variables:

$p ::= * \mid x{:}\tau \mid p,p \mid k{\downarrow}\theta\ p \mid (K{:}\alpha \mapsto s){\downarrow}\theta\ p$

Here we allow for the instantiation of value constructors and their variables to particular types as part of the pattern, since constructors are now always polymorphic in a single type-functional argument. Note the explicit constraints on type-constructor variables and the "skolem" variables representing the domain type functionals associated with them; the syntax of these constraints also accommodates either concrete or variable type constructors in the range.

The rules governing the formation of patterns are also much the same as in the simply-typed calculus, as they are again based on the notions of subsumption, erasure and assumption projection: we discuss below only those revisions to these operations which are necessary to account for the new forms of value patterns.

The revisions we need to make to the erasure operation to accommodate the new forms of patterns should be fairly clear: we need only erase the explicit constraints on value-constructor variables. Note, though, that we do not erase the type instantiations on constructors, since these are not just constraints on variables, but rather provide information which is necessary to typing (and reducing) the expressions which result from erasing these patterns.

$\mathcal{E}(k{\downarrow}\theta\ p) = k{\downarrow}\theta\ (\mathcal{E}(p))$
$\mathcal{E}((K{:}\alpha \mapsto s){\downarrow}\theta\ p) = K{\downarrow}\theta\ (\mathcal{E}(p))$

The remaining cases of erasure follow the definitions for the simply-typed case.

The proper extension of assumption projection is a little bit tricker, since we must allow for the fact that constructor variables may occur non-linearly (at least, we want to be able to allow this). Since assumption projection for value patterns results in a value-level assumption, we introduce the syntax of these conglomerate associations here (we will also need them in the description of formation for expressions):

$A_V ::= \langle\{x{:}\tau, \ldots\}, \{K{:}\varphi \mapsto s, \ldots\}\rangle \equiv \langle A_{VV}, A_{VK}\rangle$

A value-level assumption thus comprises both an association of types for value variables and an association of type-variables and type constructors for value-constructor variables. As usual, we use multiple subscript labels to indicate successive selection of the components of conglomerates; we also use the following notation for the empty value-level assumption:

$\varnothing \;::=\; \langle\{\}, \{\}\rangle$

As regards the definition of assumption projection itself, the first three clauses are just the obvious extensions of the corresponding clauses for simply-typed value patterns:

$\mathcal{A}(c) = \varnothing$
$\mathcal{A}(x{:}\tau) = \langle\{x{:}\tau\}, \{\}\rangle$
$\mathcal{A}(k{\downarrow}\theta\ p) = \mathcal{A}(p)$

For the constructor-variable and pairing cases, we use again a simple variation on the merge operation which merges two assumptions component-wise and weakly in the second argument, in order to allow for possible non-linearity of constructor variables (once again we use the notation $\oplus'$ for this variation on merge):

$\mathcal{A}((K{:}\varphi \mapsto s){\downarrow}\theta\ p) = \mathcal{A}(p) \oplus' \langle\{\}, \{K{:}\varphi \mapsto s\}\rangle$
$\mathcal{A}(p,q) = \mathcal{A}(p) \oplus' \mathcal{A}(q)$

The basic form of the rule for pattern formation remains much as it was in the simply-typed case, due in large part to our consistent use of concepts, notations and operations concerning patterns in the two languages. The essential idea is that a pattern should be considered well-formed just when its erasure (an expression) is well-typed with respect to its projected assumptions. In the simply-typed case, however, we had no need for a global set of assumptions: here we need to include the upper half of the global assumptions in order to provide context for any type variables which might appear in the value-level pattern. And, since we now have higher-order type functionals whose syntax is governed by kinds, we must check that the type terms used in constraints are actually proper types, as opposed to having some higher kind.

Given these revisions, the formation rule for value patterns in the higher-order calculus is as follows (note that we anticipate here the form of conglomerate assumptions used in formation judgements at the value level, as well as the formation rules for ground-typed expressions themselves):

$$\frac{A_T \vdash \sigma : T \qquad (\sigma \in \mathcal{C}(p)) \qquad \langle A_T, \mathcal{A}(p)\rangle \vdash \mathcal{E}(p) : \tau}{A_T \vdash p : \tau}$$

Here the set of type constraints contained within a value pattern is easily defined in terms of its projected assumptions:

$\mathcal{C}(p) = \mathrm{rng}(\mathcal{A}(p))$

The revisions we need for the subsumption operation fall out as a consequence of revisions to the matching operation. We need to account for several changes to matching: first, we must modify the overall sort of the matching function in order to allow for the substitution of all sorts of variables in the results of a match, namely type variables, type-constructor variables, value variables and value-constructor variables. To this end we define value-level substitutions as follows, with the usual notation for an empty conglomerate association:

$S \;::=\; \langle S_T, \langle [e/x, \ldots], \{[(k \mid K)/K, \ldots]\}\rangle\rangle$
$\quad \equiv \langle S_T, \langle S_{VV}, S_{VK}\rangle\rangle$
$\varnothing \;::=\; \langle \varnothing_T, \langle\{\}, \{\}\rangle\rangle$

Regarding the matching operation itself, in the case of concrete value constructors we simply ignore the type instantiations with which they are now decorated: this is sound because well-formed terms are already well-typed, and so the instantiations will match in any case.

$\mathcal{M}(k{\downarrow}\theta\ p, k{\downarrow}\omega\ e) = \mathcal{M}(p,e)$ *(NB: same k)*

In the cases involving constructor variables, we must attend to the possibility that constructor variables may be non-linear, and that multiple occurrences of the same variable should match only in situations where the same concrete constructor is used in each of the corresponding positions of the matched expression. We account for this by carrying the results of matching on constructor variables forward to subsequent constituents in the applications and pairing forms.

$\mathcal{M}((K{:}\alpha \mapsto s){\downarrow}\theta\ p, k{\downarrow}\omega\ e) = [k/K,\ v/\alpha] + \mathcal{M}(s,S) + \mathcal{M}(p,e)$
where $C_{TS}(k) = v \mapsto S$
$\mathcal{M}((K{:}\alpha \mapsto s){\downarrow}\theta\ p, H{\downarrow}\omega\ e) = [H/K,\ v/\alpha] + \mathcal{M}(s,t) + \mathcal{M}(p,e)$
where $A_{TS}(H) = v \mapsto t$

Here we use two separate cases for the match of constructor variables against either specific concrete constructors or else another constructor variable. Note that we require, in general, to have access to the current set of assumptions in order to appropriately resolve the skolem variable α representing the domain type of the constructor variable in the pattern. While this is reasonable in the context of specifying formation, it does not seem so reasonable in the context of reduction (i.e., at run-time). We will describe an alternative approach to handling constructors and constructor variables in the next chapter which will alleviate these problems: we have allowed ourselves to be a bit vague about how the current contexts are provided here, since we will be changing basic aspects of constructors anyway.

The revisions required to extend the overlap operation to the new forms of patterns are quite simple: we need only observe that concrete value constructions overlap only when their constructors coincide exactly and their corresponding constituent patterns overlap: all the other cases of constructions (i.e., those involving value-constructor variables) overlap whenever their corresponding constituents do. We express the restriction on the concrete constructor case in the usual way, using a non-linear meta-variable and a "default" case; thus we must add the following clauses:

$O(k{\downarrow}\theta\ p,\ k{\downarrow}\omega\ q) = O(p,q)$
$O((K{:}\varphi \mapsto s){\downarrow}\theta\ p,\ h{\downarrow}\omega\ q) = O(p,q)$
$O((K{:}\varphi \mapsto s){\downarrow}\theta\ p,\ (H{:}\psi \mapsto r){\downarrow}\omega\ q) = O(p,q)$
$O(k{\downarrow}\theta\ p,\ (H{:}\psi \mapsto r){\downarrow}\omega\ q) = O(p,q)$
$O(p,q) = \mathrm{false}$, otherwise

In addition to the overlap predicate, which is used to resolve compatibility issues, we must also extend our notion of coverage, via revisions to the definitions of the depth and pattern-set generation operations. Revisions to depth are trivial, since we need only add a clause for type-constructor variables and it is essentially the same as the one for concrete constructors, returning just the successor of the depth of the constituent pattern.

The revisions necessary to accommodate polymorphic types and constructors in patter-set generation are more extensive, but still easy to motivate. We must first allow for the generation of patterns to cover type variables and quantified types: note that type variables may occur in the functional position in a type application (we assume that the

type term argument has been reduced to normal form, so that we need not account for lambda-abstraction forms in functional positions). In all of these cases, we have no refined patterns which will observe any deeper structure in these types, so we can just return a fresh variable of the appropriate type as a singleton pattern set:

$\mathcal{G}^{n+1}(\alpha) = \{x{:}\alpha\}$ (fresh x)
$\mathcal{G}^{n+1}(\alpha\ \theta) = \{x{:}\alpha\ \theta\}$ (fresh x)
$\mathcal{G}^{n+1}(\forall\pi.\theta) = \{x{:}\forall\pi.\theta\}$ (fresh x)

We also need to revise the clauses for basic types to handle type constructions, with either a concrete or variable type constructor. In the case of concrete constructors, we should return value-constructed patterns instantiated to the appropriate component type; the value constructor should be drawn from the set of all such constructors associated with the given type constructor, and its pattern argument should be drawn from the set of all compatibly-typed patterns by way of a recursive call (the information needed here is contained in the constructor setting). The case for type-constructor variables is similar, but here we must also allow for all possible type constructors, since a type-constructor variable might be bound to any available concrete type constructor.

$\mathcal{G}^{n+1}(S\ \theta) = \{k{\downarrow}\theta\ p \mid C_{VK}(k){=}v{\mapsto}S,\ v\ \theta \twoheadrightarrow \omega,\ p{\in}\mathcal{G}^{n}(\omega)\}$
$\mathcal{G}^{n+1}(\Sigma\ \theta) = \{k{\downarrow}\theta\ p \mid S{\in}dom(C_{TS}),\ C_{VK}(k){=}v{\mapsto}S,\ v\ \theta \twoheadrightarrow \omega,\ p{\in}\mathcal{G}^{n}(\omega)\}$

Given the revisions to various pattern operations described above, we can define the related notions of subsumption, overlap, independence, coherence and coverage just as we did in the simply-typed case. The fact that we can so easily accommodate changes in the underlying pattern language (i.e., for type patterns or for patterns using polymorphic constructors) is a sign of the generality and robustness of our approach to patterns: we have only had to change the lower-level operations which specifically depend on the new syntax in each of these cases, while leaving the top-level definitions intact.

We can now proceed to a description of the rules governing the formation of expressions, i.e., the value-level terms which denote the functions and data structures and which are the primary objects of our computations. The judgement form for well-typing of expressions is similar to the form used in the simply-typed case, though here we must allow for both type-level and value-level assumptions, since the explicit type constraints of the polymorphic calculus must be well-formed with respect to some context for type variables (we will normally just use a single meta-variable A for the conglomerate assumptions that we have written out below in full).

$\langle A_T, A_V\rangle \vdash e : \tau$

Our use of the meta-variable τ here can be justified by virtue of the fact that we will only be able to form such judgements for type terms τ which have kind T, i.e., for proper types. This property of the inference system is a consequence of explicit design: for example, in the rule for patter formation given above, we ensured that type constraints used in patterns and thus introduced into assumptions were properly formed with a kind T. While some published type inference systems require explicit formation judgements for assumptions as antecedents to most rules, we are able to simply preserve formation for assumptions at the point of their extension in pattern-formation judgements. More formally, we will have the following property for well-typing judgements in our system:

Proposition (formation coherence for expressions):

For all well-formed assumptions A, expressions e and type functionals θ,

$$A \vdash e : \theta \text{ implies that } A_T \vdash \theta : T$$

Furthermore, all assumptions A' appearing in the derivation of $A \vdash e : \theta$ will be well-formed.

As regards the formation rules themselves, the cases for variables, pairing and applications remain largely the same, needing only to be revised for the extended notions of terms for expressions, types and assumptions involved in the new form of typing judgement. With regard to constructors and constructor variables, we have already anticipated (in Section 2.2) that they will have polymorphic, functional types of a particular form, with specific information on domain type functionals and co-domain type constructors provided by the constructor setting or type assumptions, as appropriate. We use separate rules for the type-constructor variables with concrete and variable type constructor co-domains in order to allow for projection of the kind information from the appropriate context association.

$$\frac{C_{VK}(k) = \theta{\mapsto}S \qquad C_{TS}(S) = \kappa{\to}B \qquad \theta\ \alpha = \tau}{A \vdash k : \forall\alpha{:}\kappa.\ \tau{\to}\ S\ \alpha}$$

$$\frac{A_{VK}(K) = \varphi{\mapsto}S \qquad C_{TS}(S) = \kappa{\to}B}{A \vdash K : \forall\alpha{:}\kappa.\ \varphi\ \alpha{\to}S\ \alpha}$$

$$\frac{A_{VK}(K) = \varphi{\mapsto}\Sigma \qquad A_{TS}(\Sigma) = \kappa{\to}B}{A \vdash K : \forall\alpha{:}\kappa.\ \varphi\ \alpha{\to}\Sigma\ \alpha}$$

For the recursion construct, we must enforce that the explicitly-declared type constraint given to the recursion variable is in fact a proper type (this restriction is needed for the proof of the formation coherence result stated above). Note that we need only the upper half of the conglomerate assumptions in showing that the type constraint is well-kinded, and that we explicitly extend the appropriate component of the whole assumption with the recursion variable and its constraint.

$$\frac{A_T \vdash \tau : T \qquad A{+}_{VV}\{x{:}\tau\} \vdash e : \tau}{A \vdash (\mu x{:}\tau.e) : \tau}$$

The rule for value-pattern abstractions is again just a minor variation on the rule for the simply-typed case: we need only include the upper half of the assumption in the antecedents assuring well-typing of the constituent patterns:

$$\frac{\begin{array}{c}\{p_1, \ldots, p_n\}\ \text{compatible} \\ \{p_1, \ldots, p_n\} \succ \tau \\ A_T \vdash p_i : \tau \qquad (1 \leqslant i \leqslant n) \\ A + \mathcal{A}(p_i) \vdash e_i : \sigma \qquad (1 \leqslant i \leqslant n)\end{array}}{A \vdash (p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) : \tau \to \sigma}$$

The first significant extension to the formation rules of the simply-typed language is the rule for type-pattern abstractions. We take our inspiration for this rule from the similar rule for value-pattern abstractions; specifically, we require compatibility and coverage for the type patterns constituting the left-hand sides of the abstraction and also proper formation for both the patterns (individually) and the expressions, with respect to the assumptions from the corresponding patterns.

On the other hand, there are also differences from the value-pattern abstraction rule: first, we also need to require that free type variables in the individual type patterns and in the quantified parameter pattern not occur free in the global assumptions. For the parameter pattern, this is just a generalization of the usual free-variable constraint on the formation of quantified types (the universal quantifier is only justified if in fact there is no dependence in the assumptions on the particular type variables used); similar reasoning applies for the pattern-bound type variables (note that although any occurrences of the type variables themselves in the domain of the assumption would be shadowed when we extend the assumption , there might still be free occurrences in the types of some value-level variables).

Another difference from the value-pattern rule is that we must project out the kind of the quantification explicitly from its parameter pattern for the purposes of the coverage and type-pattern formation antecedents. Note that we do not need any assumption context for the formation of type patterns, since there are no constraints relevant to kind variables. In the antecedents for expression formation, we explicitly substitute in the body of the quantified type with a match between the overall parameter type and the type pattern corresponding to the particular expression: this …

The complete rule governing formation of type-pattern abstractions is thus as follows:

$$\frac{\begin{array}{c}\{\rho_1, \ldots, \rho_n\}\ \text{compatible} \\ \{\rho_1, \ldots, \rho_n\} \succ \mathcal{K}(\pi) \\ \left(FV(\pi)_T \cup \bigcup_i FV(\rho_i)_T\right) \cong FV(A)_T \\ \vdash \rho_i : \mathcal{K}(\pi) \qquad (1 \leqslant i \leqslant n) \\ A +_T \mathcal{A}(\rho_i) \vdash e_i : \tau\,[\mathcal{M}(\pi, \rho_i)] (1 \leqslant i \leqslant n)\end{array}}{A \vdash (\rho_1 \Rightarrow\!\!\Rightarrow e_1 \mid \cdots \mid \rho_n \Rightarrow\!\!\Rightarrow e_n) : \forall \pi.\tau}$$

The final rule governs the formation of instantiation expressions:

$$\frac{A \vdash f : \forall \pi.\tau \qquad A_T \vdash \theta : \mathcal{K}(\pi)}{A \vdash (f{\downarrow}\theta) : \tau\,[\mathcal{M}(\pi, \theta)]}$$

Note here that the kinds of the quantification and of the actual instantiation argument must match, and that the resulting type is determined by pattern-matching the quantification parameter pattern and the actual argument.

In the next chapter we will discuss reduction and conversion for expressions in the higher-order calculus: such operations will only be considered for terms which are well-formed according to the above rules.

Chapter 4
Operational and Semantic Considerations

In this chapter we develop the operational semantics of the higher-order pattern calculus described in the last chapter, according to the usual syntactic notion of reduction, and also remark on some issues which must be handled in any standard, model-theoretic semantics. For operational purposes, it is convenient to modify slightly our approach to constructor variables, at both of the levels of types and values, to use numeric indices rather than the arbitrary symbolic names we used in the last chapter. This change allows us to more easily track the connection between constructor variables and their associated kind constraints during reduction, without recourse to the set of assumptions under which a particular subterm is well-formed

Symbolic names are of course preferable in programming practice, and in the context of the last chapter they made for an easier introduction to the basic concepts, but the dependence of reduction on type assumptions would be a high price to pay for this convenience. Conversion of the symbolic names to numeric indices can be seen as a simple source-to-source transliteration, similar to the elimination of variables in favor of combinators which is often used in the implementation of standard functional programming languages.

The change to from symbolic names to numeric indices is not semantically significant: Section 4.1 below is thus something of a technical digression which might be skipped on a first reading or by those who are not interested in the details of reduction.

The main point of Section 4.2 is to demonstrate that an important property of reduction in the standard typed calculi is lost in our higher-order pattern calculus, namely the static typing property. More specifically, we must make reference to types during reduction (i.e., at "run-time") in our calculus since we need to perform matching on types to determine the continued course of of the reduction. Although this makes our types "dynamic" in a sense, the situation is not so problematic as in other contexts where the terminology dynamic typing is often used; see, e.g., [DD85, Car*88] for examples. In the cited systems, types are dynamic in the stronger sense that values of dynamic type must carry their types with them at run-time: in our calculus we need only match type instantiations at run-time, and so arbitrary values may participate in analytically-polymorphic computations without paying the penalty of carrying run-time types. Given our paradigm of compatibility and coverage checks for patterns and a suitable choice of pattern-matching regimen, our calculus need never produce run-time errors (though of course non-termination is always possible in the presence of general recursion).

We discuss briefly in Section 4.3 some of the unusual properties which any standard, denotational model of the analytically-polymorphic calculus must satisfy. In particular, types in any such model must be rather "syntactic", in that it must be possible to analyze them for their constituent parts, in order to drive evaluation of terms in a manner commensurate with syntactic pattern-matching on types. We do not know of any model which embodies these properties (this is an important open problem for us), though we suspect that investigation of Leivant's "structural semantics" might prove fruitful (see [Lei83]).

4.1 Reduction and skolem indices

As we mentioned above, before we describe reduction rules for our calculus it will be useful to make some minor changes to the way that constructor variables and the associated skolem variables are treated. In particular, consider the form of the structural clause of a typical analytically-polymorphic function: it is constituted of a type pattern (the left-hand side of the clause), a value pattern and a value expression (these two comprising the right-hand side of the clause). In the structural clause, the type pattern will consist of an application of a type-constructor variable (constrained with a variable kind in its domain) to a type variable bound by the pattern; the value pattern will include the application a value constructor variable, similarly constrained, to a pattern-bound value variable (in the case of the value-level pattern there will typically be some surrounding context):

$$(\Sigma{:}X\to D)\ (\chi{:}X) \Rightarrow\!\!> (\ \ldots\ (K{:}\varphi\mapsto\Sigma)\downarrow\chi\ (x{:}\varphi\ \chi)\ \ldots\ \Rightarrow e[\Sigma, \chi, \varphi, K, x]\ )$$

When these patterns are successively matched, we would expect to first bind the constructor variable Σ to some specific constructor (say List) and then bind the value constructor variable K to some specific value constructor (say cons); in so doing, we should also bind the corresponding skolem variables against the actual kind and type of the specific constructors (List and cons), since these instantiations may be needed during further reduction of the term. Our matching rules use the syntactically-available constraints as a way of specifying how the skolem variables are instantiated, by referring to the corresponding kind and type information for the specific constructors from the appropriate part of the setting:

$$\mathcal{M}((\Sigma{:}V\to D)\ \rho,\ S\ \upsilon) = [S/\Sigma,\ \kappa/V]+\mathcal{M}(\rho,\upsilon) \text{ where } C_T(S)=\kappa\to D$$

$$\mathcal{M}((K{:}\varphi\mapsto\Sigma)\downarrow\theta\ p,\ k\ q) = [k/K,\ \theta/\varphi,\ S/\Sigma]+\mathcal{M}(p,q) \text{ where } C_V(k)=\theta\mapsto S$$

But consider now the case when the terms to which the type and value patterns are bound are not specific constructors, but themselves constructor variables. We cannot simply mimic the above rule, because while the occurrences of constructor variables in patterns are required to have explicit kind and type constraints, occurrences in expression terms are not. In this case, we do not have access to the information we need for the binding, since this information is not contained in the global constructor setting, but rather in the local assumption information, to which we have no access to during reduction, at least in traditional conceptions of typed calculi. That is, we would have to use a rules like the following

$$\mathcal{M}((\Sigma{:}V\to D)\ \rho,\ \Gamma\ \upsilon) = [\Gamma/\Sigma,\ \kappa/V]+\mathcal{M}(\rho,\upsilon) \text{ where } A_{TC}(\Gamma)=\kappa\to D$$

$$\mathcal{M}((K{:}\varphi\mapsto\Sigma)\downarrow\theta\ p,\ H\ q) = [H/K,\ \theta/\varphi,\ \Gamma/\Sigma]+\mathcal{M}(p,q) \text{ where } A_{VC}(H)=\theta\mapsto\Gamma$$

where the assumptions A_{TC} and A_{VC} are not normally available during reduction (we could define a notion of reduction based on formation derivations, as is done for semantics, for example, but this would of course eliminate any chance of static typing right from the outset).

The sort of variable/variable matching which causes the problem above only occurs in the context of situations like η-reduction, where we are reducing a term *within* an abstraction

(in this case, within two abstractions, one for types and one for values). Although these circumstances might seem rare enough, and η-reduction enough a matter of taste (not all people consider extensionality an important property of calculi), we prefer to consider a means of avoiding this problem.

Our approach, as mentioned above, is to modify the syntax of our language slightly so as to drop constructor variables at both levels of terms in favor of slightly different forms involving numeric indices. More specifically, we will express each sort of constructor variable with a special constant (which acts merely as a placeholder), indexed by a numeric expression which may be either a manifest constant (i.e., a specific numeral) or a numeric variable drawn from a new sort of atomic symbols; we also replace the skolem variables used in the constraints on constructor variables with similar indexed forms.

Formally, we need first a grammar for indices which allows for both numerals and variables; letting n range over numerals and ν over numeric variables, we have:

$$n ::= 0 \mid 1 \mid \cdots$$
$$i ::= n \mid \nu$$

We can then introduce an indexed constant form into the languages of proper kinds and data kinds:

$$\kappa ::= \cdots \mid \mathbf{V}_i \mid \cdots$$
$$\delta ::= \cdots \mid \mathbf{V}_i \mid \cdots$$

Finally, we introduce similar indexed forms into the languages of type, type patterns, expressions and patterns:

$$\tau ::= \cdots \mid \mathbf{S}_i \mid \mathbf{T}_{ij} \mid \cdots$$
$$\rho ::= \cdots \mid \mathbf{S}_i\ \rho \mid \cdots$$
$$e ::= \cdots \mid \mathbf{K}_{ij} \mid \cdots$$
$$p ::= \cdots \mid \mathbf{K}_{ij} \downarrow\theta\ p \mid \cdots$$

We will describe the significance of these new forms below; note that in the case of the last four grammars, we can now do without with the old forms for constructors and constructor variables, since these will now be expressed using the numeral and numeric-variable versions of the indexed forms, respectively.

The indices can be interpreted as referring to a revised form of constructor setting, one which is not based on associations of kinds and type to symbolic names, but rather on finite tuples of kinds and types. That is, constructor settings now take the forms described by the following grammars:

$$C ::= \langle C_T,\ C_V \rangle$$
$$C_T ::= \langle \kappa_1 \to B_1,\ \ldots,\ \kappa_n \to B_n \rangle$$
$$C_V ::= \langle \langle \theta_{11},\ \ldots,\ \theta_{1m} \rangle,\ \langle \theta_{n1},\ \ldots,\ \theta_{nm} \rangle \rangle$$

Thus type-level settings consist simply of a tuple of functional kinds with basic range, and value-level settings consist of a tuple of tuples, each of whose ultimate components is a type functional. The interpretation is essentially the same as that for the associations with symbolic names, except that here the type constructors will be referred to by a single numeric index and the value constructors, which are grouped together according to the common type constructor in their range, will be referred to by double numeric indices, the first corresponding to their associated type constructor and the second to the domain functional of the given value constructor.

We note in passing that the change to numeric indices allows us to eliminate the constructor- variable component of both type- and value-level assumptions; we do not need to track numeric variables in the assumptions since they are not subject to any sort of syntactic constraints.

As a concrete example of the use of the new indexed forms, consider the nested pattern we used as an example above; in the revised version it would be expressed as follows (we also reproduce the original here, for the sake of comparison):

$$\mathbf{S}_i\ (\chi{:}\mathbf{V}_i) \Rrightarrow (\ \ldots\ \mathbf{K}_{ij}{\downarrow}\chi\ (x{:}\mathbf{T}_{ij}\ \chi)\ \ldots\ \Rightarrow e[i, \chi, j, x]\ )$$

versus

$$(\Sigma{:}X{\to}D)\ (\chi{:}X) \Rrightarrow (\ \ldots\ (K{:}\varphi \mapsto \Sigma){\downarrow}\chi\ (x{:}\varphi\ \chi)\ \ldots\ \Rightarrow e[\Sigma, \chi, \varphi, K, x]\ )$$

In the revised term, we use the indexed constants of the forms $\mathbf{S}_i$, $\mathbf{V}_i$, $\mathbf{K}_{ij}$ and $\mathbf{T}_{ij}$ in place of type constructor variables, kind variables, value constructor variables and value constructor domains, respectively (we write all of the new indexed constants in a boldface, sans serif font in order to emphasize their common nature). Note that we have dropped some of the heavy type and kind-constraint syntax from the earlier version: this is possible because the indices on the new sorts of constants allow us to implicitly associate, e.g., a particular domain kind with a given type constructor, simply by the fact that that they share a common index—this implicit but direct connection, as opposed to the indirect but explicit recording of constraints in local assumptions, is the key to the new syntax.

As a more specific and familiar example, compare the following two structural clauses for the count functional from Chapter 3, written in the old style with constructor variables and in the new style with indexed constants:

$$(\lambda\varepsilon{:}T.\ (\Sigma{:}X{\to}D)\ (\chi\ \varepsilon{:}X)) \Rrightarrow$$
$$(\ (K{:}\varphi \mapsto \Sigma){\downarrow}(\chi\ \alpha)\ (x{:}\varphi(\chi\ \alpha)) \Rightarrow C{\downarrow}(\lambda\varepsilon{:}T.\ \eta(\chi\ \varepsilon))\ x)$$

versus

$$(\lambda\varepsilon{:}T.\ \mathbf{S}_\nu\ (\chi\ \varepsilon{:}\mathbf{V}_\nu)) \Rrightarrow$$
$$(\ \mathbf{K}_{\nu\mu}{\downarrow}(\chi\ \alpha)\ (x{:}\mathbf{T}_{\nu\mu}(\chi\ \alpha)) \Rightarrow C{\downarrow}(\lambda\varepsilon{:}T.\ \mathbf{T}_{\nu\mu}(\chi\ \varepsilon))\ x)$$

(recall that C is the recursion variable used in the definition of count and that α is a type parameter from the outer-most type abstraction). Here we see how the use of the numeric variables ν and μ are used to allow for the fact that we wish the forms $\mathbf{S}_\nu$ and $\mathbf{K}_{\nu\mu}$ to match against arbitrary type constructors and value constructors, respectively.

In making these changes to the language, we will of course need to revise the rules for term formation, pattern-matching and reduction; we briefly review these changes below, but refer the reader to Appendix B for a more thorough and cohesive description of the revised calculus with the indexed forms.

At the level of kinds, the only addition we have made is to include the new indexed form $\mathbf{V}_i$, used to represent the domain kinds of type constructors. The new forms are treated in all respects like kind variables (which were in fact introduced for the same purpose), e.g., for the purposes of retraction At the level of types we have introduced two new forms, $\mathbf{S}_i$ and $\mathbf{T}_{ij}$, the first taking the place of type constructor variables and the second used to represent the type functions associated with the domains of value constructors. The second form uses dual indices, where the second index refers to a particular value

constructor from among several associated with the same type constructor

The inference rules required for the new indexed forms are straightforward, since we use the same notation for the projection of the components of a tuples as we do for the projection of range terms from a symbolic association; for example, we use two rules for the $\mathbf{S}_i$ form, one for the case when the index is a constant and one for the case when it is a variable, as follows:

$$A \vdash \mathbf{S}_n : C_{TS}(n)$$

$$A \vdash \mathbf{S}_v : \mathbf{V}_v \rightarrow D$$

The rules for the $\mathbf{T}_{ij}$ form are similar, but note that there are now four cases corresponding to four possible combinations of different sorts of indices. Of these, two cases deserve particular mention (the constant/variable and variable/variable cases are similar to the rules above for $\mathbf{S}_i$). Specifically, we must be concerned with the fact that, in order to be well-formed, a dually-indexed form like $\mathbf{T}_{ij}$ should not be allowed to specify a second index which falls outside the bounds implied by the first index.

That is, if we have only m value constructors associated with the nth type constructor, we should not allow $\mathbf{T}_{nw}$ as well-formed for w>m, since it would otherwise seem to refer to a non-existent value constructor. Note further that if the first index is a variable, then we have no way to check this constraint, and that we should thus consider this combination ill-formed in any case, since later instantiation of the primary index variable (e.g., during reduction) might invalidate the formation of the term.

For these reasons we give no rule for the variable/constant case (it is never well-formed) and use the following rule to effect "bounds-checking" on the secondary index when both indices are constants:

$$\frac{m \leqslant |C_{VK}|}{A \vdash \mathbf{T}_{nm} : C_{TS}(n)}$$

Note that similar concerns about bounds-checking for the constant case of the $\mathbf{S}_i$ rules above do not apply, since the projection operation on the single index there would fail, thus precluding the use of the rule *a priori*. Also note that we must perform bounds-checking on any constant numerals occurring in data kinds as indices in the $\mathbf{V}_i$ construct.(see the data kind formation rule in Appendix B for our approach).

Turning now to the value level, we have introduced a new, doubly-indexed form $\mathbf{K}_{ij}$; the issue of giving context for free numeric variables is already resolved, for expression formation is dependent on value assumptions, which include type- and kind-level variables as components. As for the formation rules for the $\mathbf{K}_{ij}$ form itself, we recall that constructors are construed as polymorphic functions from some domain type function (applied to the polymorphic parameter) to an associated type constructor (applied to the same parameter), where we now use $\mathbf{V}_i$ to represent the domain (kind) of the quantification and $\mathbf{T}_{ij}$ to represent the appropriate domain type function. We use the following rules for the three sensible cases corresponding to possible combinations of constants and variables as indices (the fourth case, variable/constants, is deemed ill-formed for the same reasons as stated above for the $\mathbf{T}_{ij}$ form):

$$\frac{C_{TS}(n) = \kappa \rightarrow B \qquad (C_{VK}(n)(m))\ \alpha = \theta}{A \vdash \mathbf{K}_{nm} : \forall \alpha{:}\kappa.\ \theta \rightarrow \mathbf{S}_n\ \alpha}$$

$$\frac{C_{TS}(n) = \kappa \rightarrow B}{A \vdash \mathbf{K}_{n\mu} : \forall \alpha{:}\kappa.\ \mathbf{T}_{n\mu}\ \alpha \rightarrow \mathbf{S}_n\ \alpha}$$

$$A \vdash \mathbf{K}_{v\mu} : \forall \alpha{:}\mathbf{K}_v.\ \mathbf{T}_{v\mu}\ \alpha \rightarrow \mathbf{S}_v\ \alpha$$

The last case, used for the variable/variable combination is a straightforward formalization of the description above. The middle case, used for the constant/variable combination, is similar but for the use of an appropriate specific domain kind for the quantification, based on the projection of the nth kind from the type-level setting tuple. The first case, used for the constant/constant combination, projects both the domain kind corresponding to the type constructor and the domain type function corresponding to the value constructor, and allows for the conversion of this type function when applied to the quantified type variable. Note that the bounds-checking issue does not arise separately here since, as in the case for the $\mathbf{S}_n$ form, the use of projection from the settings tuples implicitly restricts the use of the rule to cases where the projection succeeds, i.e., where the index is within bounds.

We turn now to the effects on type patterns and value patterns of the introduction of the new indexed forms; these are essentially of two sorts, namely effects on formation, via required changes to the assumption projection operations on patterns, and effects on matching. Consider first the case of type patterns: we have added only a single form to the syntax, namely the $\mathbf{S}_i$ form used to represent both type constructors and their variables; we do not need to explicitly add the form $\mathbf{T}_{ij}$ for value constructor domain functions because we have no need to specifically match instantiated type arguments of this form (in any case, they display no internal structure until they become instantiated to specific types).

With regard to assumption projection from type patterns, we need not include any free numeric variables in the assumption results, since we do not maintain numeric variables in local assumptions (we do not need to because they are not constrained); thus we can simply return the empty type-variable assumption:

$$\mathcal{A}_\Pi(\mathbf{S}_i) = \langle\{\},\{\}\rangle$$

With regard to to the effects of the indexed forms on type-pattern matching, we actually achieve a significant simplification here, since when matching two of the indexed forms we need only match the corresponding indices, using a function newly defined for this purpose (it requires two constants to match exactly, and substitutes for either variable for the other combinations). Thus we have:

$$\mathcal{M}(\mathbf{S}_i,\mathbf{S}_j) = \langle\langle[],\ \mathcal{M}(i,j)\rangle,\ \langle[],\ []\rangle\rangle$$

Recall here that type pattern matching returns a conglomerated substitution as its result; with the extension to include indices and numeric variables, we must allow for components of these substitutions which map numeric variables of the appropriate sort (i.e., primary or secondary) to indices (i.e., numerals or other numeric variables).

While type-pattern matching is driven by the reduction of expressions (i.e., value-level terms), we must also take care to

allow for the behavior of the new indexed forms in the context of type reduction proper. In particular, since the form … is essentially a placeholder for a type functional given in the lower half of the constructor setting, we must allow for the conversion of this form into the associated type functional when its indices are fully instantiated:

$$\mathbf{T}_{nm} = C_{VK}(n)(m)$$

Specifically, this sort of conversion is necessary since we will need to analyze the structure of types when they occur as instantiations to polymorphic values represented by type-pattern abstractions; in order to resolve a type-pattern match against them, we will need to expose the structure of any fully-instantiated $\mathbf{T}_{ij}$ forms. Note that we do not have to be similarly concerned about the behavior of the $\mathbf{S}_i$ or $\mathbf{K}_{ij}$ forms, since when the variable indices of these forms are instantiated their import is fulfilled, in the sense that there is no more refined or complex structure which must then be analyzed (these forms "stand for themselves", like constants, when their indices are instantiated to specific numerals). Also note that the $\mathbf{V}_i$ forms, like other kinds, are never involved in pattern-matching in an analytically significant way: rather, they occur only with parametric significance, and therefore may remain un-reduced when their indices are instantiated.

Turning now to the effects of introducing indexed forms on value-level patterns, we consider again effects related to projection of assumptions and effects related to pattern matching. With regard to the former, we must determine what assumptions are free in a pattern of form $\mathbf{K}_{ij}{\downarrow}\theta\ p$; consider that for such a pattern to be well-formed, if the primary index is a variable then it must already be in scope at the point where the pattern occurs, whereas the secondary index, if a variable, might be a binding occurrence in and of itself. Under these circumstances, it seems clear that that only the secondary index is free in this form, so that we define:

$$\mathcal{A}(\mathbf{K}_{ij}{\downarrow}\theta\ p) = \langle \varnothing_K, \langle\{\}, FV(j)\rangle\rangle +' \mathcal{A}(p)$$

(the +' notation, defined in Appendix B, merely extends its left argument with an empty value-level assumption before merging the associations as usual).

With regard to matching, we derive the same benefits for value patterns in the move to indexed forms as we did for type patterns above; namely, we are able to facilitate matching for the indexed forms by reference to the matching operation on indices, as follows:

$$\mathcal{M}(\mathbf{K}_{ij}{\downarrow}\theta\ p, \mathbf{K}_{hk}{\downarrow}\omega\ e) =$$
$$\quad \textbf{let}\ S = \langle\langle[],\ \mathcal{M}(i,j)\rangle,\ \langle[],\ \mathcal{M}(k,h)\rangle\rangle,\ []\rangle$$
$$\quad \textbf{in}\ \mathcal{M}(p[S],\ e[S])$$

Note that we must be careful to apply the substitutions resulting from the match of the indexed forms to their pattern arguments before recursively proceeding with the match, since the specialization of any numeric variables to specific constants could affect the subsequent matches in the inner pattern (e.g., it could cause a clash between two different constants).

Finally, corresponding to the introduction of the new index forms and the new numeric-variable components of assumptions, we will have to make changes to the way that free variable operations are defined; these changes are relatively straightforward (see Appendix B for details).

4.2 Properties of reduction

The standard explicitly-typed lambda calculi, in all of the simply-typed, second- and higher-order polymorphic variants, enjoy several nice properties with respect to reduction; specifically, reduction for all of these calculi is Church-Rosser, preserving of types (often called the *subject reduction* property) and, in the absence of general recursion forms, strongly-normalizing. In addition, all of these calculi enjoy strong, static typing, by which we mean that types can be erased from typed terms prior to reduction, and that when the resultant type-free terms are reduced to normal-form, the result will always always the erasure of the normal form of the corresponding typed term.

With regard to Church-Rosser or confluence properties, we believe that these results will carry over to the higher-order pattern calculus, although we have not pursued any rigorous approach in this area, since the complexity of the task would seem to far outweigh the insights gained. That our calculus preserves types under reduction seems to be a straight-forward result, given the reduction and typing rules we have specified above and in previous chapters. In this case, the task of proving this rigorously does not seem difficult, but again the rewards of the effort seem small enough that we choose to forego it.

More interesting is the property of static typing, which depends in turn on a notion of type erasure. Type erasure is defined for polymorphic calculi so as to eliminate not only explicit type constraints, but also type abstractions and type instantiations; that is, we define the erasure of a second-order polymorphic expression as follows:

$$\mathcal{E}(x) = x$$
$$\mathcal{E}(\lambda x{:}\theta.e) = \lambda x.\ \mathcal{E}(e)$$
$$\mathcal{E}(f{\cdot}e) = \mathcal{E}(f) \cdot \mathcal{E}(e)$$
$$\mathcal{E}(\Lambda\alpha{:}\kappa.e) = \mathcal{E}(e)$$
$$\mathcal{E}(e{\downarrow}\theta) = \mathcal{E}(e)$$

In our higher-order pattern calculus, it is difficult to see how we can even define the erasure function and still preserve the sensibility of terms. For example, it would make sense to define erasure on value-pattern abstractions as follows, given our already-defined notion of erasure on patterns:

$$\mathcal{E}(p_1 \Rightarrow e_1 \mid \cdots \mid p_n \Rightarrow e_n) = \mathcal{E}(p_1) \Rightarrow \mathcal{E}(e_1) \mid \cdots \mid \mathcal{E}(p_n) \Rightarrow \mathcal{E}(e_n)$$

But the extension of this idea to type-pattern abstractions does not seem sensible: if we simply erase, say, kind constraints from the type-patterns on the left-hand sides of clauses, we end up not actually erasing all type information, since the type patterns will still remain:

$$\mathcal{E}(\rho_1 \Rightarrow e_1 \mid \cdots \mid \rho_n \Rightarrow e_n) = \mathcal{E}(\rho_1) \Rightarrow \mathcal{E}(e_1) \mid \cdots \mid \mathcal{E}(\rho_n) \Rightarrow \mathcal{E}(e_n)$$

On the other hand, if we erase the type-patterns completely, we have a set of right-hand sides representing alternative expression bodies, but no way of choosing between them, i.e., no patterns against which to match an actual argument:

$$\mathcal{E}(\rho_1 \Rightarrow e_1 \mid \cdots \mid \rho_n \Rightarrow e_n) = \mathcal{E}(e_1) \mid \cdots \mid \mathcal{E}(e_n)$$

The source of this quandary clearly lies in our intended

interpretation of type-pattern abstractions: we reduce an instantiated type-pattern abstraction by matching its left-hand sides against an actual type instantiation, using the corresponding right-hand side expression as the basis for further reduction. But if we have neither type patterns nor the explicit type instantiation as constituents of the redex, it is clearly impossible to determine a choice of expression from which to proceed (not to mention that we will have no substitution to make into the expression as the result of the reduction).

Our calculus is thus not statically-typed in the usual sense; but neither is it *dynamically-typed* in the usual sense. More particularly, dynamic typing generally refers to systems in which types are maintained along with values at run-time; some systems of dynamic typing restrict this "tagging" of values with types to a limited set of *dynamic values*, whose distinctive types allow them to be thus separated from the remaining statically-typed portion of the language. Explicit type coercions are then used to bring dynamically-typed values under the purview of static typing.

Many systems of dynamic types employ a notion of type-pattern abstraction similar to our own in order to effect the afore-mentioned coercion of dynamic to static values (see [Myc83, DD85, Car*88, LeMa91] for examples). We note, however, that these systems require that any values which participate in such type-cased computations must be dynamic values, and thus be tagged at run-time with their types; as remarked in [LeMa91], this means that these systems are difficult to integrate into existing applications, since most existing code would be statically-typed and thus incompatible with run-time type tags.

By contrast, our system does not depend on explicit type tags at run-time for a distinguished class of values: rather, the process of type-cased computation is driven by explicit type instantiations that are independent of the values whose reduction they drive, at least in the sense that they are not packaged with them as a single entity at run-time. Thus our approach at least shows some promise for a smoother integration with existing type systems for polymorphic functional languages; we will see in Chapter 6 how this promise can be fulfilled by allowing for a very restricted and thus unobtrusive use of run-time type type instantiation.

Regarding the property of strong normalization, we imagine that we could prove a strong-normalization result for the recursion-free fragment of our calculus, but this is not very satisfying, since without recursion we can express only the most trivial of type-pattern abstractions, and certainly not any of our motivating examples. We will remark in Section 5.4 on a possible approach to circumvent this problem: we develop there an a notion of primitive recursion expressed via combinators and generalized away from the usual case of natural numbers to arbitrary higher-order data structures.

Unfortunately, we have been as yet unable to devise an analogous notion of primitive recursion over types themselves (as opposed to specific types), as would naturally correspond to the combination of general recursion and type-pattern abstraction. We will describe in detail the difficulties which arise in defining primitive recursion over types in Section 5.4, but with regard to the issue of strong normalization, we note here that the termination of the computations does not seem to be crucial to the problem.

It is worth pointing out here that the course of recursion realized during analytically-polymorphic computations (i.e., by way of the reduction of recursively-specified type-pattern abstractions) is not determined uniquely by either the types or the values involved. For example, the extent of such a recursion is clearly not determined by the type alone, since a very large data structure, for example, can have an extremely shallow type. On the other hand, the types of the various constructors used to build the data structure are used as the basis for making the choices which guide the course of the recursion through the structure, since they are used in the matches against the type-patterns occurring as the left-hand sides of the abstractions clauses. The course of the recursion is thus *guided* by the types and type patterns, but proceeds over the corresponding values, leading to new and potentially larger type arguments in the process (usually this "growth" in the type occurs as a result of the structural case, where the recursive instantiation is to the argument type functional of a constructor for some recursive data type or structure).

4.3 Some considerations for denotational models

In this section we briefly review some of the considerations which would arise in any attempt to find a traditional denotational or model-theoretic semantics for the higher-order calculus with analytical polymorphism. At the outset it would seem reasonable that we consider a standard model for the traditional higher-order polymorphic lambda calculus and then make appropriate extensions to accommodate the new features we have introduced.

There are several problems with this approach, not the least of which is that there seems to be little agreement on even the form of a model for the polymorphic lambda calculus, much less on a particular standard model (see [See87, Mes89, BMM90] for discussion and for examples of proposed notions of the general structure of a model).

One reason for the lack of agreement in this area may be that there can be no classical set-theoretic model of polymorphism: this result was first obtained by Reynolds [Rey84], although it has since been observed that this result does not hold for an intuitionistic notion of set (see [Pitt87]).

Another possible reason for the lack of agreement about models of polymorphic calculi is that there are in fact many different variations on the basic calculus whose model theory have been studied, variations which require or at least suggest a wide variety of different and perhaps incompatible techniques.

The main problem which must be resolved in any model for the higher-order pattern calculus is the modeling of structural induction on types: in any traditional model, types would be interpreted as some structured set s of values. And yet our calculus requires that types be examined and analyzed for what is essentially their syntactic structure. As pointed out by Scott (see [Sco81]), such a close relationship between the syntax of types and their interpretations will need to be specifically designed in to the model.

Given this consideration, the best current prospect which we see for properly modeling our calculus would be an extension of the Structural Semantics for first-order polymorphic calculi described by Leivant [Lei83]. In particular, Leivant's approach marries a traditional denotational semantics of values with a term-model for types: it seems reasonable to expect that this bipartite approach would allow for the sort of syntactic analysis of types that we require, since types would in fact be modeled just as syntactic entities. (what is not so clear is how we would state in this context some of the usual properties about the relationship

between values and their types, since types would not be interpreted as sets of values in any sense).

Some other features of our language which will require special attention in any model include the following:

- explicit, higher-order types—in fact, dynamic types, so that no "erasure" model will work
- kind polymorphism, at least to the extent that we require kind variables in the constraints on type-constructor variables
- recursion—explicitly at the object level and implicitly at the level of types (since type constructors in a setting may correspond essentially to recursively defined datatypes)
- an appropriate interpretation of the distinction between data types and proper types and of the coercions between the related kinds.

We will pursue the issue of semantics a for the higher-order calculus no further than this this undeveloped stage, partly out of a lack of expertise in this area, but also out of a lack of need for model-theoretic results. In fact, the higher-order calculus is for us more a means to an end, namely to the development of an appropriate framework for implicitly-typed languages which offer features based on type-pattern abstraction, but which are designed more with practical programming applications in mind.

Chapter 5
Examples, Variations and Extensions

In the last three chapters we have developed the higher-order pattern calculus as a formal model of analytical polymorphism and described the formation and reduction rules necessary to give operational meanings to the terms of the calculus. In this chapter we consider some examples and applications of the calculus which should help to motivate the considerable amount of formal specification which was necessary for its definition.

In Section 5.1 we present a number of examples of analytically-polymorphic functions which would be useful in programming contexts. Although these examples are far too cumbersome in their explicit typing to be practical, they are the pre-cursors to more tenable versions which we hope to express in the implicitly-typed languages that are the subject of the last chapters of the dissertation. Our discussion of them here, in the explicitly-typed context, is intended to provide the detailed understanding that will be necessary to make decisions about compromises and trade-offs in the design of the implicitly-typed language.

In section 5.2 we explore a natural extension of the calculus to include abstraction over kind variables and the attendant notion of kind polymorphism. The extension is natural in the sense that we have already included kind variables in the syntax of the calculus in order to allow for arbitrarily-kinded constructor variables in type patterns—the addition of a form of kind abstraction allows us to capture with a single term the essence of schematically-related expressions which are in some sense independent of any particular kind. As a motivating example, we present a polymorphically-kinded map functional whose type arguments are generalized from proper types to higher-order type functionals.

Finally, in section 5.3 we take a longer look at the first-order and polymorphically-kinded generalizations of the map functional, in an attempt to characterize their distinguishing properties. This exercise serves both to increase confidence that our definitions are sensible (by showing that crucial properties of the original map functional actually are preserved in the generalizations) and also to provide a first glimpse of techniques for reasoning about the properties of analytically polymorphic functions. Although we cannot completely prove the characterization we would like for the generalized map, we do find sufficient conditions on the properties of polymorphic functions in our calculus which may suggest how they ought be modeled in a proper semantics. Although our characterization of the properties of map is based on commutative diagrams, we do not explore connections with category theory, mainly out of a lack of expertise in this area.

5.1 Some extended examples

All of the examples of analytically polymorphic functions provided below are written in the language of Chapters 2 and 3, which use symbolic names for type and value constructors and the variables which range over them; since the examples are used here for expositional purposes only, we have no particular need to use the indexed forms of Chapter 4.

We have already provided two extended examples in Section 3.1, namely the definitions there of the size and count functions. As our next example, we consider a function which will implement the equality operation of Standard ML, though here in an explicit;y-typed context.. In Standard ML, the equality function is a primitive, and actually not definable within the language through any other means. In addition, ad-hoc restrictions are imposed on its uses, restricting it so that it cannot be applied to arguments which are not of "equality types".

We will be able to define equality using type patterns, and in the process give a more broadly-motivated and formal account of the corresponding restriction on the types of its argument. The equality function we define has the following type:

$$\text{equal} : \forall \alpha{:}\mathrm{D}.\ \alpha \times \alpha \rightarrow \mathrm{Bool}$$

Thus above-mentioned restriction on equality in Standard ML is expressed in our system as a restriction to arguments whose type is a data type.

The definition of equality itself is must reflect the fact that the intended argument is a pair of commonly-typed values: this is different from most of the examples we have seen so far, in which the domain type of the underlying function was just an occurrence of the type variable appearing in the quantification (or an application thereof in the case of type functions). This is expressed by using pair patterns in the value-pattern abstraction representing the underlying function, as follows:

$$\begin{aligned}
&\text{equal} = \mu \mathrm{E}{:}(\forall \alpha{:}\mathrm{D}.\ \alpha \times \alpha \rightarrow \mathrm{Bool}).\\
&\quad 1 \Rightarrow (*,*) \Rightarrow \text{true}\\
&\quad \mid \beta{:}\mathrm{D} \times \gamma{:}\mathrm{D} \Rightarrow\\
&\qquad ((\mathrm{b}_1{:}\beta,\ \mathrm{c}_1{:}\gamma),(\mathrm{b}_2{:}\beta,\ \mathrm{c}_2{:}\gamma)) \Rightarrow\\
&\qquad\quad \mathrm{E}{\downarrow}\beta\ (\mathrm{b}_1,\ \mathrm{b}_2) \text{ and } \mathrm{E}{\downarrow}\gamma\ (\mathrm{c}_1,\ \mathrm{c}_2)\\
&\quad \mid (\Sigma{:}\mathrm{X} \rightarrow \mathrm{D})\ (\chi{:}\mathrm{X}) \Rightarrow\\
&\qquad (\ ((\mathrm{K}{:}\theta \mapsto \Sigma){\downarrow}\chi\ (\mathrm{x}{:}\theta\ \chi),\ (\mathrm{K}{:}\theta \mapsto \Sigma){\downarrow}\chi\ (\mathrm{y}{:}\theta\ \chi))\\
&\qquad\quad \Rightarrow \mathrm{E}{\downarrow}(\theta\ \chi)\ (\mathrm{x},\ \mathrm{y})\\
&\qquad \mid ((\mathrm{K}{:}\theta \mapsto \Sigma){\downarrow}\chi\ (\mathrm{x}{:}\theta\ \chi),\ (\mathrm{H}{:}\nu \mapsto \Sigma){\downarrow}\chi\ (\mathrm{y}{:}\nu\ \chi))\\
&\qquad\quad \Rightarrow \text{false}\ )
\end{aligned}$$

We presume here the booleans true and false and a binary function and (written infix) with obvious definition. The unit clause is trivial, but note in the pair clause that we must match up corresponding components of the two pair arguments on the right-hand side of the pattern abstraction, since, among other reasons, it is forced by typing considerations. In the structure clause, we are depending on a best-fit matching regimen to choose the first of the two value-pattern clauses when the two value constructors used to construct the two arguments are in fact the same.

As an alternative to this approach in the structural clause, we might attempt to express the same effect by using a single value-pattern clause on the right-hand side of the type clause, with two different value constructor variables, and then comparing these variables for equality using some primitive comparison operator, written say "≡", as follows:

$$\begin{aligned}
&(\Sigma{:}\mathrm{X} \rightarrow \mathrm{D})\ (\chi{:}\mathrm{X}) \Rightarrow\\
&\qquad (\ ((\mathrm{K}{:}\theta \mapsto \Sigma){\downarrow}\chi\ (\mathrm{x}{:}\theta\ \chi),\ (\mathrm{H}{:}\nu \mapsto \Sigma){\downarrow}\chi\ (\mathrm{y}{:}\nu\ \chi))\\
&\qquad\quad \Rightarrow \text{if } \mathrm{K} \equiv \mathrm{H} \text{ then } \mathrm{E}{\downarrow}(\theta\ \chi)\ (\mathrm{x},\ \mathrm{y}) \text{ else false}
\end{aligned}$$

But note that using this approach, the if expression cannot even be well-typed, since the subexpression $\mathrm{E}{\downarrow}(\theta\ \chi)\ (\mathrm{x},\ \mathrm{y})$ will be ill-typed unless $\mathrm{K} \equiv \mathrm{H}$ (if we do not have $\mathrm{K} \equiv \mathrm{H}$, then

the instantiation $E{\downarrow}(\theta\ \chi)$ is not well-typed for the second argument, y, which is of type $\nu\ \chi$ for some $\nu \neq \theta$). We could try to adjust the calculus to allow for ill-typed sub-terms occurring in well-typed terms, but this seems too significant an extension to justify on the basis of some preference for the single-pattern version of the structural clause above.

We can, however, relax the restriction that the two value arguments to equality be of the same type by simply extending the function so as to use a pair of types, possibly different, in the quantified constraint on the recursion variable, and by extending the type patterns to pairs of type patterns in a corresponding way. This approach would give us a slightly more flexible equality function, of type:

Although ostensibly more flexible, this typing is somewhat deceptive in that it allows a well-typed application of equality to be verified statically, even though all instances in which the two types differ must return a result of false, and in such cases this fact can be determined statically by simply comparing the types, as is done automatically for the single type-parameter version given first. This fact might be deemed undesirable from the viewpoint of programming methodology, since it allows an error potentially determinable at compile-time to remain undetected at compile-time.

Nevertheless, the doubly-parameterized version of equality can be written to use only a single clause on the right of the structural type pattern, as shown below, since it the more flexible typing allows us to instantiate the recursive call in the structural clause to both of the argument's types:

```
equal = μE:(∀α:D,β:D. α×β→Bool).
    (1,1) ⇒» (∗,∗) ⇒ true
  | (β:D×γ:D, β:D×γ:D) ⇒»
      ((b1:β1, c1:γ),(b2:β, c2:γ)) ⇒
        E↓β (b1, b2) and E↓γ (c1, c2)
  | ((Σ:X→D) (χ:X), (Σ:X→D) (χ:X)) ⇒»
      ( (((K:θ↦Σ)↓χ (x:θ χ), (H:ν↦Σ)↓χ (y:ν χ))
          ⇒ if K≡H then E↓(θ χ, ν χ) (x, y) else false)
  | (τ:D, σ:D) ⇒» (t:τ, s:σ) ⇒ false
```

Note that in extending the definition in this way, we are again depending on a best-fit matching regimen, this time at the level of type patterns, since only this regimen will allow us to choose the extra clause added at the end of the definition in cases when the actually instantiated types differ (non-linearities in the other type patterns do not allow a match in such cases).

We now turn to the example of the map functional for first-order data structures, the development of which has been our primary motivations in this work. We seek a generalization of the map functional for lists which will apply a function argument "point-wise" to all the components of a data structure, returning a congruent structure of result components. The appropriate type is a simple generalization of the type of map for lists; we repeat it here for reference:

$$map_{FO} : \forall\alpha{:}T,\beta{:}T.\ (\alpha\to\beta)\to\forall\varphi{:}T\to D.\ \varphi\ \alpha\to\varphi\ \beta$$

As usual, the definition consists mainly of a recursively specified type-pattern abstraction:

```
mapFO = Λα:T,β:T. λf:α→β.
  μM:(∀φ:T→D. φ α→φ β).
      (λε:T. 1) ⇒» ∗ ⇒ ∗
    | (λε:T. (ψ ε:D)×(ζ ε:D))
        ⇒» (y:ψ α, z:ζ α) ⇒ (M↓ψ y,M↓ζ z)
    | (λε:T. (Σ:X→D) (χ ε:X))
        ⇒» (K:θ↦Σ)↓(χ α)(x:θ(χ α))
          ⇒ K↓(χ β)(M↓(θ∘χ) x)
    | (λε:T. ε) ⇒» a:α ⇒ f a
```

The definition is straightforward, with one value-pattern clause for each of the corresponding type-pattern clauses. Note that for the component clause we simply apply the function f to the single value argument: by η-congruence we need not even have written a pattern-abstraction here, but we leave it in this form for the sake of symmetry with the other clauses. Note also that the appropriate actual type argument, i.e., α or β, appears in the constraints and instantiations on the right-hand sides of the type-pattern clauses in positions corresponding to occurrences of the lambda-bound type variable ε in the type patterns (in the type patterns it is used to relativize pattern-bound type variables there, e.g., ψ, ζ in the product clause and χ in the structural clause).

As a next example of an analytically polymorphic function, we consider a generalization of the usual function on trees which flattens a tree into a list of its components. In the case of an arbitrary data structure, we will have the following type:

```
flatten :
    ∀α:T. ∀φ:T→D. (φ α)→List α
```

With this typing, we will be able to use the same flatten function on not only trees, but also on doubly- or triply-nested lists, say, by using instantiations as to the type functions λα:T. List(List α) or λα:T. List(List(List α)), respectively, since each of these type functions will allow us to select out the appropriate components for the ultimate (flat) list of results.

The definition of flatten is particularly straight-forward:

```
flatten = α:T ⇒»
  μF:∀φ:T→D. (φ α)→List α.
      (λε:T. 1) ⇒» ∗ ⇒ nil↓α
    | (λε:T. (ψ ε:D)×(ω ε:D)) ⇒»
        (y:ψ α, z:z α) ⇒ (F↓ψ y) @ (F↓ζ z)
    | (λε:T. (Σ:X→D) (χ ε:X)) ⇒»
        (K:θ↦Σ)↓χ (y:θ(χ β)) ⇒ F↓(θ∘χ) x
    | (λε:T. ε) ⇒» a:α ⇒ cons↓α (a, nil↓α)
```

We presume defined the usual append function on lists, which we write here as infix "@", with the following type:

$$@ : \forall\alpha{:}T.\ (\text{List}\ \alpha)\times(\text{List}\ \alpha)\to\text{List}\ \alpha$$

Another simple, but useful function on lists is the function zip: it takes two lists of components, possibly of different types and combines them together into a list of pairs of the respective component types. Generalizing the use of lists here to an arbitrary type function, we get the following type:

```
zip :
```

$$\forall\,\alpha{:}T,\beta{:}T.\ \forall\varphi{:}T\to D.\ (\varphi\ \alpha)\times(\varphi\ \beta)\to\varphi(\alpha\times\beta)$$

In defining the zip function, we must make a decision about how to treat structures of different "shape": although the structures have the same type, they may not use the same value constructors in corresponding positions. In the specific case of lists, for example, one list may not be as long as the other, due to the use of nil in the shorter list corresponding to the use of cons in the longer one. One solution to this problem for lists is to include a "default" component of each type among the arguments to the function, and to fill out the component positions of the resultant list with one of these values when the corresponding list "runs out". This approach has the advantage of preserving all of the "information content" of the two list arguments; unfortunately, it does not generalize well to the case of arbitrary structures, however, since in general we will not have structures with only two value constructors, one with a single component (like cons) and one "degenerate" one with no components or recursive structure (like nil). Rather, we may have an arbitrary number of constructors with a mixture of components and recursive structure of various kinds.

A more common approach to the problem of zipping differently-shaped structures, again in the case of lists, is to "trim" the longer list to match the shorter one: this is accomplished by returning a nil result whenever either of the two arguments is constructed with nil. Again, we do not have the advantage of presuming a single, "degenerate" constructor (like nil) in the case of arbitrary data structure arguments, but we can simulate this approach by using a polymorphic constant fail of type $\forall\,\alpha{:}T.\,\alpha$ (although a constant of this type is catastrophic in the context of a programming logic" based on the propositions-as-types paradigms, it is perfectly acceptable from a computational viewpoint).

$$\begin{aligned}
&\text{zip} = (\alpha{:}T,\beta{:}T) \Rightarrow\\
&\quad \mu Z{:}\forall\varphi{:}T\to D.\ (\varphi\ \alpha)\times(\varphi\ \beta)\to\varphi(\alpha\times\beta).\\
&\qquad (\lambda\varepsilon{:}T.\ 1) \Rightarrow (*,*) \Rightarrow *\\
&\qquad \mid (\lambda\varepsilon{:}T.\ (\psi\ \varepsilon{:}D)\times(\omega\ \varepsilon{:}D)) \Rightarrow\\
&\qquad\quad ((y_1{:}\psi\ \alpha,\ w_1{:}\omega\ \alpha),\ (y_2{:}\psi\ \beta,\ w_2{:}\omega\ \beta)) \Rightarrow\\
&\qquad\quad\quad (Z{\downarrow}\psi\ (y_1, y_2),\ Z{\downarrow}\omega\ (w_1, w_2))\\
&\qquad \mid (\lambda\varepsilon{:}T.\ (\Sigma{:}X\to D)\ (\chi\ \varepsilon{:}X)) \Rightarrow\\
&\qquad\quad ((\ (K{:}\theta\mapsto\Sigma){\downarrow}\chi\ (x{:}\theta(\chi\ \alpha))\ ,\\
&\qquad\quad\quad (K{:}\theta\mapsto\Sigma){\downarrow}\chi\ (y{:}\theta(\chi\ \beta))\quad)\\
&\qquad\quad\quad\quad \Rightarrow K{\downarrow}\theta(\chi\ (\alpha\times\beta))\ (Z{\downarrow}(\theta\circ\chi)\ (x, y))\\
&\qquad\quad \mid (\ (K{:}\theta\mapsto\Sigma){\downarrow}\chi\ (x{:}\theta(\chi\ \alpha))\ ,\\
&\qquad\quad\quad (H{:}\nu\mapsto\Sigma){\downarrow}\chi\ (y{:}\nu(\chi\ \beta))\quad)\\
&\qquad\quad\quad\quad \Rightarrow \text{fail}{\downarrow}(\Sigma(\chi(\alpha\times\beta)))\\
&\qquad \mid (\lambda\varepsilon{:}T.\ \varepsilon) \Rightarrow (a{:}\alpha,\ b{:}\ \beta) \Rightarrow (a, b)
\end{aligned}$$

We should point out here that in the product case, we have "braided" together the pair of pairs that we receive as arguments into a transposed pair of results, with recursive calls of the zip function interspersed. Also note how the component case is degenerate—it is just the identity function on pairs of type $\alpha\times\beta$—but nevertheless well-typed, due to the fact that the corresponding type function (i.e., the identity function on types) in this degeneration of the function type.

A related function, in fact an inverse of zip in some sense, is the function unzip: in the standard case of lists, this function takes a list of pairs and returns a pair of lists, in the obvious fashion. For the more general function on arbitrary structures of pairs, then, we have this type:

$$\text{zip} : \forall\,\alpha{:}T,\beta{:}T.\ \forall\varphi{:}T\to D.\ \varphi(\alpha\times\beta)\to(\varphi\ \alpha)\times(\varphi\ \beta)$$

The definition is simpler than the definition for unzip, since there a single structure will of course have only one "shape" (thus there is no need for the failure value used in zip); on the other hand, the reverse process of "unbraiding" arguments in the product case is somewhat more complicated to specify in the case of unzip:

$$\begin{aligned}
&\text{unzip} = (\alpha{:}T,\ \beta{:}T) \Rightarrow\\
&\quad \mu U{:}\forall\varphi{:}T\to D.\ \varphi(\alpha\times\beta)\to(\varphi\ \alpha)\times(\varphi\ \beta).\\
&\qquad (\lambda\varepsilon{:}T.\ 1) \Rightarrow * \Rightarrow (*,*)\\
&\qquad \mid (\lambda\varepsilon{:}T.\ (\psi\ \varepsilon{:}D)\times(\omega\ \varepsilon{:}D)) \Rightarrow\\
&\qquad\quad (y{:}\psi(\alpha\times\beta),\ z{:}\zeta(\alpha\times\beta)) \Rightarrow\\
&\qquad\quad\quad (\ (\pi_1\ (U{\downarrow}\psi\ y),\ \pi_1\ (U{\downarrow}\zeta\ z))\ ,\\
&\qquad\quad\quad\quad (\pi_2\ (U{\downarrow}\psi\ y),\ \pi_2\ (U{\downarrow}\zeta\ z))\quad)\\
&\qquad \mid (\lambda\varepsilon{:}T.\ (\Sigma{:}X\to D)\ (\chi\ \varepsilon{:}X)) \Rightarrow\\
&\qquad\quad (K{:}\theta\mapsto\Sigma){\downarrow}\chi\ (x{:}\theta\ (\chi(\alpha\times\beta)))\\
&\qquad\quad\quad \Rightarrow (K{\downarrow}\theta(\chi\,\alpha)\ \pi_1\ (U{\downarrow}(\theta\circ\chi)\ x)\ ,\\
&\qquad\quad\quad\quad K{\downarrow}\theta(\chi\,\beta)\ \pi_2\ (U{\downarrow}(\theta\circ\chi)\ x))\\
&\qquad \mid (\lambda\varepsilon{:}T.\ \varepsilon) \Rightarrow (a{:}\alpha,\ b{:}\ \beta) \Rightarrow (a, b)
\end{aligned}$$

Here we have made use of the following abbreviations for the polymorphic projection functions for pairs with the obvious types:

$$\begin{aligned}
\pi_1 &\equiv (\alpha{:}T,\ \beta{:}T) \Rightarrow (x{:}\alpha,\ y{:}\beta) \Rightarrow x\\
\pi_2 &\equiv (\alpha{:}T,\ \beta{:}T) \Rightarrow (x{:}\alpha,\ y{:}\beta) \Rightarrow y
\end{aligned}$$

We can simultaneously generalize certain cases of both the zip and unzip by functions recognizing that they are both instances of transposing two structures: in the case of zip, we transpose pairs and lists to transform a pair of lists into a list of pairs; in the case of unzip, we transpose lists and pairs for just the opposite affect. Ideally, we would like then to define a general function, transpose and then instantiate the structures Γ and Δ to lists, pairs or other structures in an appropriate order to achieve these same sorts of affects. The definition we give below is adequate only for "diagonal" instances of zip and unzip, i.e., those where the two type parameters are identified; its type is therefore:

$$\text{transpose} : \forall\,\Delta{:}T\to D,\ \Gamma{:}T\to D,\ \alpha{:}T.\ \Delta(\Gamma(\alpha))\to\Gamma(\Delta(\alpha))$$

Thus we seek a definition of transpose, such that for all types τ,

$$\begin{aligned}
\text{zip}{\downarrow}\tau &= \text{transpose}{\downarrow}\tau{\downarrow}((\lambda\alpha{:}T.\ \alpha\times\alpha),\ \text{List})\\
\text{unzip}{\downarrow}\tau &= \text{transpose}{\downarrow}\tau{\downarrow}(\text{List},\ (\lambda\alpha{:}T.\ \alpha\times\alpha))
\end{aligned}$$

We could presumably generalize even the definition of transpose below to a case which allows type functions of two arguments, thus capturing the non-diagonal cases of zip and unzip, but we have not verified this by actually coding it up.

Our strategy in defining transpose is as follows: first we define an auxiliary function (called here transrec), with the type given below, which will serve as a stronger recursive core on which the ultimate transpose function will be based.

$$\forall\,\gamma{:}T\to D,\ \alpha{:}T.\ \forall\varphi{:}T\to D.\ \varphi(\gamma\,\alpha)\times(\gamma\,\alpha)\to(\gamma\,\alpha)\times\varphi(\gamma\,\alpha)$$

Note here that φ and γ take the places of Δ and Γ, respectively, in the type of transpose; note also that we have both an extra argument (of type $(\gamma\,\alpha)$) and an extra result (also of type $(\gamma\,\alpha)$): these are used to carry certain "default" structure templates through the recursive analysis of the structural argument. More specifically, we define transrec as follows:

$$
\begin{array}{l}
\mathrm{transrec} \equiv \\
\quad (\gamma{:}\mathrm{T}\to\mathrm{D},\alpha{:}\mathrm{T})\Rightarrow\!\!> \\
\quad \mu\,\mathrm{T}{:}\forall\,\delta{:}\mathrm{T}\to\mathrm{D}.\,\delta(\gamma\,\alpha)\times\gamma\,\alpha\to\gamma(\delta\,\alpha)\times\gamma\,\alpha. \\
\qquad (\ (\lambda\varepsilon{:}\mathrm{T}.1)\ \Rightarrow\!\!> \\
\qquad\quad (*,g{:}\gamma\,\alpha)\ \Rightarrow\ (\mathrm{map}{\downarrow}(\alpha,1)(a{:}\alpha\Rightarrow *){\downarrow}\gamma\,g,g) \\
\qquad |\ (\lambda\varepsilon{:}\mathrm{T}.(\psi\,\varepsilon){:}\mathrm{D}\times(\zeta\,\varepsilon){:}\mathrm{D})\ \Rightarrow\!\!> \\
\qquad\quad ((y{:}\psi(\gamma\,\alpha),z{:}\zeta(\gamma\,\alpha)),g{:}\gamma\,\alpha)\ \Rightarrow \\
\qquad\qquad ((ty{:}\gamma(\psi\,\alpha),tyg{:}\gamma\,\alpha)\ \Rightarrow\ (tz{:}\gamma(\zeta\,\alpha),tzg{:}\gamma\,\alpha)\ \Rightarrow \\
\qquad\qquad\quad (\mathrm{zip}{\downarrow}(\psi\,\alpha,\zeta\,\alpha){\downarrow}\gamma(ty,tz),tzg)) \\
\qquad\qquad (\mathrm{T}{\downarrow}\zeta(z,tyg))(\mathrm{T}{\downarrow}\psi(y,g)) \\
\qquad |\ (\lambda\varepsilon{:}\mathrm{T}.\Sigma{:}\mathrm{V}\to\mathrm{D}((\chi\,\varepsilon){:}\mathrm{V})\ \Rightarrow\!\!> \\
\qquad\quad ((\mathrm{K}{:}\theta\mapsto\Sigma){\downarrow}(\chi(\gamma\,\alpha))(x{:}\theta(\chi(\gamma\,\alpha)),g{:}\gamma\,\alpha)\ \Rightarrow \\
\qquad\qquad ((tx{:}\gamma(\theta(\chi\,\alpha)),txg{:}\gamma\,\alpha)\ \Rightarrow \\
\qquad\qquad\quad (\mathrm{map}{\downarrow}(\theta(\chi\,\alpha),\Sigma(\chi\,\alpha))(\mathrm{K}{\downarrow}\chi\,\alpha){\downarrow}\gamma\,tx,txg)) \\
\qquad\qquad (\mathrm{T}{\downarrow}(\theta\circ\chi)(x,g)) \\
\qquad |\ (\lambda\varepsilon{:}\mathrm{T}.\varepsilon)\Rightarrow\!\!>(g_1{:}\gamma\,\alpha,g_2{:}\gamma\,\alpha)\Rightarrow\!\!>(g_1,g_1))
\end{array}
$$

The definition of the final transpose function we sought can now be given as follows:

$$
\begin{array}{l}
\mathrm{transpose} \equiv \\
\quad (\Gamma{:}\mathrm{T}\to\mathrm{D},\Delta{:}\mathrm{T}\to\mathrm{D},\alpha{:}\mathrm{T})\ \Rightarrow\!\!>\ \mathrm{DG}{:}\Delta(\Gamma\,\alpha)\ \Rightarrow \\
\qquad ((\mathrm{GD}{:}\Gamma(\Delta\alpha),g{:}\Gamma\,\alpha)\ \Rightarrow\ \mathrm{GD}) \\
\qquad\quad (\mathrm{transrec}{\downarrow}(\Gamma,\alpha){\downarrow}\Delta(\mathrm{DG},\mathrm{fail}{\downarrow}\Gamma\,\alpha))
\end{array}
$$

Here we have simply passed in the (appropriately instantiated) failure value as the original default template for the transrec recursion; we also project out the second argument of the result of transrec, thus throwing away the returned final value of the default template.

5.2 Kind polymorphism

In the development of the higher-order pattern calculus, we were forced to introduce a notion of variables at the level of kinds, in order to be able to account for arbitrarily-kinded data type constructors in type patterns. This level of generality has not usually been incorporated into even higher-order lambda calculi in previous research, but once we have admitted it into the framework of our calculus, it is interesting to pursue the idea further to see if any other applications for it can be found.

In fact, as we discuss in this section, we can use the notion of kind variables to support a form of kind abstraction, i.e., a form of lambda abstraction over kind variables with respect to expression bodies. Such a form can in turn be used to express a generalized, polymorphically-kinded version of map, along with other functions that would be useful in manipulating higher-order data structures.

The additional syntax we require amounts to one new form of type and two new forms of expressions; specifically, we add the following clauses to the appropriate grammars:

$$
\begin{array}{lcl}
\theta & ::= & \cdots \ |\ \forall\mathrm{V}.\theta \\
\mathrm{e} & ::= & \cdots \ |\ \Lambda\mathrm{V}.\mathrm{e} \ |\ \mathrm{e}{\downarrow}\kappa
\end{array}
$$

Corresponding to these new forms we add the following formation rules:

$$\frac{\mathrm{A}\vdash\theta:\mathrm{T}}{\mathrm{A}\vdash(\forall\mathrm{V}.\theta):\mathrm{T}}$$

$$\frac{\mathrm{V}\notin\mathrm{FV}_{\mathrm{KV}}(\mathrm{A})\qquad \mathrm{A}\vdash\mathrm{e}:\tau}{\mathrm{A}\vdash(\Lambda\mathrm{V}.\mathrm{e}):\forall\mathrm{V}.\tau}$$

$$\frac{\mathrm{A}\vdash\mathrm{f}:\forall\mathrm{V}.\tau}{\mathrm{A}\vdash(\mathrm{f}{\downarrow}\kappa):\tau\ [\kappa/\mathrm{V}]}$$

These rules are simply analogues of the usual rules for type quantification, type abstraction and type instantiation found in the second-order lambda calculus, but lifted now one level higher so as to concern kinds rather than types.

Given this new syntax, we can express the type of our generalization of map as follows:

$$
\begin{array}{l}
\mathrm{map}_{\mathrm{HO}}: \\
\qquad \forall\,\mathrm{V}.\ \ \forall\gamma{:}\mathrm{V}\to\mathrm{T},\eta{:}\mathrm{V}\to\mathrm{D}. \\
\\
\qquad\quad (\forall\varepsilon{:}\mathrm{V}.\ \gamma\ \varepsilon\to\eta\ \varepsilon)\to\forall\nu{:}(\mathrm{V}\to\mathrm{T})\to\mathrm{D}.\ \ \nu\ \gamma\to\nu\ \eta
\end{array}
$$

Here we have abstracted away from the specific case of two type parameters α and β to allowing two type-ranged functions with arbitrarily-kinded domains.

$$
\begin{array}{l}
\mathrm{map}_{\mathrm{HO}}\ =\ \Lambda\mathrm{V}.\ \Lambda\gamma{:}\mathrm{V}\to\mathrm{T},\eta{:}\mathrm{V}\to\mathrm{D}. \\
\quad \lambda\mathrm{F}{:}(\forall\varepsilon{:}\mathrm{V}.\ \gamma\ \varepsilon\to\eta\ \varepsilon). \\
\qquad \mu\mathrm{M}{:}(\forall\nu{:}(\mathrm{V}\to\mathrm{T})\to\mathrm{D}.\ \nu\ \gamma\to\nu\ \eta). \\
\qquad\quad (\lambda\varphi{:}\mathrm{V}\to\mathrm{T}.\ 1)\ \Rightarrow\!\!>\ *\ \Rightarrow\ * \\
\qquad\quad |\ (\lambda\varphi{:}\mathrm{V}\to\mathrm{T}.\ (\psi\ \varphi{:}\mathrm{D})\times(\zeta\ \varphi{:}\mathrm{D})) \\
\qquad\qquad \Rightarrow\!\!>\ (y{:}\psi\ \gamma,\ z{:}\zeta\ \gamma)\ \Rightarrow\ (\mathrm{M}{\downarrow}\psi\ y,\mathrm{M}{\downarrow}\zeta\ z) \\
\qquad\quad |\ (\lambda\varphi{:}\mathrm{V}\to\mathrm{T}.\ (\Sigma{:}\mathrm{X}\to\mathrm{D})\ (\chi\ \varphi{:}\mathrm{X})) \\
\qquad\qquad \Rightarrow\!\!>\ (\mathrm{K}{:}\theta\mapsto\Sigma){\downarrow}(\chi\ \gamma)(x{:}\theta(\chi\ \gamma)) \\
\qquad\qquad\quad \Rightarrow\ \mathrm{K}{\downarrow}(\chi\ \eta)(\mathrm{M}{\downarrow}(\theta\circ\chi)\ x) \\
\qquad\quad |\ (\lambda\varphi{:}\mathrm{V}\to\mathrm{T}.\ \varphi\ ((\rho\ \varphi){:}\mathrm{V})) \\
\qquad\qquad \Rightarrow\!\!>\ g{:}\gamma(\rho\ \gamma)\ \Rightarrow\ \mathrm{M}{\downarrow}(\eta\circ\rho)(\mathrm{F}{\downarrow}(\rho\ \gamma)\ g)
\end{array}
$$

The typing of the component clause here is a bit tricky, so we will review it here in some detail; we have:

$$
\begin{array}{l}
(\eta{:}\ \mathrm{V}\to\mathrm{D})\circ(\rho{:}(\mathrm{V}\to\mathrm{T})\to\mathrm{V}):(\mathrm{V}\to\mathrm{T})\to\mathrm{D} \\
\mathrm{M}{\downarrow}(\eta\circ\rho):\eta\ (\rho\ \gamma)\to\eta(\rho\ \eta) \\
\mathrm{F}{\downarrow}((\rho\ \gamma){:}\mathrm{V}):\gamma(\rho\ \gamma)\to\eta(\rho\ \gamma) \\
\mathrm{M}{\downarrow}(\eta\circ\rho)(\mathrm{F}{\downarrow}(\rho\ \gamma)\ g):\eta\ (\rho\ \eta)\ =\ \nu\ \eta[\lambda\varphi.\ \varphi(\rho\ \varphi)/\nu]
\end{array}
$$

Note that we require here the fact that the constraint on the type function η above be such that it has a data type range; the corresponding constraint on γ is not needed for the typing above, so we have left it with the more general constraint of a (proper) type range.

Regarding this "broken symmetry", it is interesting to note that we can write another version of the map functional which differs only in the way in which we treat the component clause

case; in particular, we have chosen above to write the application of the recursive map M and the parameter function F in that order; but it seems possible to reverse this order so to write as follows. In so doing, we must reverse the sense of the data type restriction on the ranges of the γ and η type function parameters (we have underlined the modified constraints for emphasis):

$$\mathrm{map}_{HO} = \Lambda V.\ \Lambda\gamma{:}\underline{V \to T}, \eta{:}\underline{V \to D}.$$
$$\ldots$$
$$\Rightarrow\!\!\Rightarrow\ g{:}\gamma(\rho\ \gamma) \Rightarrow M{\downarrow}(\eta \circ \rho)(F{\downarrow}(\rho\ \gamma)\ g)$$

We suspect, based on our characterization of the properties of the map functional in the next section, that these two variations will coincide, modulo the restriction of one of the type function parameters to a data type range.

5.3 Characterizing the generalized map functionals

In the last chapters we have defined three progressively general versions of the map functional: the usual one for lists, one for first-order, single parameter type functions and one for higher-order type functionals. These various functionals have the following types, respectively:

$$\mathrm{map} : \forall\alpha,\beta.\ (\alpha \to \beta) \to \mathrm{List}\ \alpha \to \mathrm{List}\ \beta$$
$$\mathrm{map}_{FO} : \forall\alpha{:}T,\beta{:}T.\ (\alpha \to \beta) \to (\forall\varphi{:}T \to D.\ \varphi\ \alpha \to \varphi\ \beta)$$
$$\mathrm{map}_{HO} : \forall V.\ \forall\gamma{:}V \to D,\eta{:}V \to D.\ (\forall\varepsilon{:}V.\ \gamma\ \varepsilon \to \eta\ \varepsilon) \to (\forall\varphi{:}(V \to T) \to D.\ \varphi\ \gamma \to \varphi\ \eta)$$

The map functional for lists is of course widely known and its behavior well-understood. Unfortunately, we cannot say the same of the latter two: following our process of progressive generalization, it is no longer clear what relation they bear to the original, familiar map for lists. In order to justify the name "map", we now develop a plausible characterization of the essential properties of map based on the original version, and show how this characterization can be naturally extended to the more general ones.

In order to make a convincing argument, we must first decide what crucial properties can be said to characterize the original map functional for lists. To map a function across a list is to apply that function point-wise to each component, without affecting the structure of the whole (i.e. the shape of the list itself). In this sense map is a fundamental operation on structures, for it is in this aspect of its behavior that it realizes the separation between structure and components that is the underlying theme of our entire research effort. We attempt to codify this behavior initially in terms of two classes of functions which might be applied to lists: selector functions and measures.

By *a selector* we mean any of those functions which select components of lists in a uniform way, such as by taking the first or last component (by "uniform" we mean to exclude those functions which select components on the basis of the values of the components themselves). Given any such selector function, we should obtain the same result either by mapping a function g across a list and then selecting a component, or by selecting a component and then applying the function g to it: this describes the property that map spreads functions inside structures to their components. Conversely, by a *measure* we mean any of those functions on lists whose results are independent of the components contained in the list, depending instead only on the list structure itself (i.e., the length function). The result of applying a measure to a list should be the same as the result of applying the same measure to a "mapped" version of the list, regardless of the function we map across the list. This describes the property of map that it does not affect the structural aspects of a list in any way.

Finally, we might also note that it should be the case that map of the identity function for some component type is the identity on lists of that component type.

We now formalize the above remarks before coordinating them into a more general characterization of map for lists. First consider a list l with components of type τ and a function g which maps values of type τ to those of type σ (thus g is appropriate for mapping across l):

$$l : \mathrm{List}\ \tau$$
$$g : \tau \to \sigma$$

A selector function as defined above will have a polymorphic type mapping lists over some component type back to that same type[9]:

$$f : \forall\alpha.\ \mathrm{List}\ \alpha \to \alpha$$

Given these types, our first characterization of map can be stated via the following equation:

$$f{\downarrow}\sigma\ (\mathrm{map}{\downarrow}(\tau,\sigma)\ g\ l) = g\ (f{\downarrow}\tau\ l)$$

An alternative, more graphic, expression of the above can be given in terms of a commutative diagram, thus:

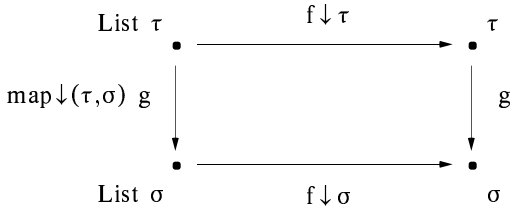

Figure 5.3.1: Commutative property for map and selectors

We turn now to the second characterization above: note that if the results of a measure function are to be truly independent of the values of the argument list's components, then that function must be a polymorphic function on lists, albeit with a "degenerate" result type, i.e., one with no occurrence of the polymorphic parameter. For if a measure function could not be given such a polymorphic type, it could only be due to some dependence it had on a specific component type—being unable to depend on that component type, its result type is necessarily a fixed type, such as Nat in the case of the length function. We have:

$$f : \forall\alpha.\ \mathrm{List}\ \alpha \to \delta \qquad (\alpha \notin \mathrm{FV}(\delta))$$

Now our second characterization of map can be described via the following equation:

$$f{\downarrow}\sigma\ (\mathrm{map}{\downarrow}(\tau,\sigma)\ g\ l) = f{\downarrow}\tau\ l$$

[9] Actually, no *total* function of this type exists, since there is no way to return a value of type α if the list is empty. The argument still holds, however, for the more complex type necessary to avoid this problem.

and once again we can express this graphically with a commutative diagram:

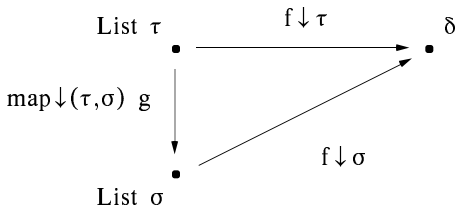

Figure 5.3.2: Commutative property for map and measures

At this point, it may seem that the two particular characterizations we have chosen here are somewhat ad hoc, as there are many other such classes of functions besides selectors and measures to which we might have appealed. For example, rather than a "simple" selector of type $\forall\alpha.\ \mathrm{List}\ \alpha\rightarrow\alpha$, we might have chosen a more complex selector of type $\forall\alpha.\ \mathrm{List}\ \alpha\rightarrow\alpha\times\alpha$; e.g., a function which returns the first and last components of a list, or the first two. Or we might have chosen a function which combines selector-like behavior with measure-like behavior, say one of type $\forall\alpha.\ \mathrm{List}\ \alpha\rightarrow\alpha\times\delta$, for some δ. For each of these examples, we could state a similar equation and draw a similar commutative diagram, though more complex result types would require more complex equations and diagrams.

More to the point, however, it turns out that we can generalize all of these seemingly disparate characterizations by appealing to the expressive power of type functionals and to the generalized map for first-order type functions. In particular, take the function f to be of type $\forall\alpha.\ \mathrm{List}\ \alpha\rightarrow\psi\ \alpha$, for some type functional $\psi:\mathrm{T}\rightarrow\mathrm{D}$. Thus in the previous examples we would have ψ equal to $(\lambda\alpha.\alpha)$, $(\lambda\alpha.\delta)$, $(\lambda\alpha.\alpha\times\alpha)$, or $(\lambda\alpha.\alpha\times\delta)$, respectively. Given this assumption, we can begin to write an equation to characterize map in terms of an arbitrary f of this type: on the left-hand side, we write the same term as occurs on the left-hand sides for the selector and measure cases above, representing the results of f applied to the mapping of g on the list. On the right-hand side we want to apply f directly to the list l and then, somehow, apply g to the result; to be well-typed as a whole, both sides of the equation must have type $\psi\ \sigma$:

$$f{\downarrow}\sigma\ (\mathrm{map}{\downarrow}(\tau,\sigma)\ g\ l) = [\ldots\ g\ \ldots]\ (f{\downarrow}\tau\ l)\ :\ \psi\ \sigma$$

Of course, we cannot apply g directly to the result of f, since the result of f has type $\psi\ \tau$, and g takes an argument of type τ; we can, however, exploit the general first-order map functional $\mathrm{map_{FO}}$ to "spread" g across the τ-typed components of f's result, as follows:

$$f{\downarrow}\sigma\ (\mathrm{map}{\downarrow}(\tau,\sigma)\ g\ l) = \mathrm{map_{FO}}{\downarrow}(\tau,\sigma)\ g\ {\downarrow}\psi\ (f{\downarrow}\tau\ l)$$

(recall here that the type of $\mathrm{map_{FO}}$ is:

$$\forall\alpha{:}\mathrm{T},\beta{:}\mathrm{T}.\ (\alpha\rightarrow\beta)\rightarrow(\forall\varphi{:}\mathrm{T}\rightarrow\mathrm{D}.\ \varphi\ \alpha\rightarrow\varphi\ \beta)\ ).$$

At this point it may seem that we have a problem with circular reasoning: in order to justify the definitions of the generalized versions of map, we are appealing to an equation for the version of map for lists, which equation itself refers to one of the generalizations we originally sought to justify. Recall, however, that we can always expand the term $\mathrm{map_{FO}}{\downarrow}(\tau,\sigma)\ g\ {\downarrow}\psi$ "in-line" to an equivalent term which does not mention $\mathrm{map_{FO}}$ at all; for example, using the selector and measure characterizations from above we can determine the following:

$$\mathrm{map_{FO}}{\downarrow}(\tau,\sigma)\ g\ {\downarrow}(\lambda\alpha.\alpha) = g$$
$$\mathrm{map_{FO}}{\downarrow}(\tau,\sigma)\ g\ {\downarrow}(\lambda\alpha.\delta) = \mathrm{Id}_\sigma\ \ (\text{since}\ \alpha\notin\mathrm{FV}(\delta))$$

and thus recover the specific equations given above for these cases. Furthermore, we claim that we could do essentially the same thing for any particular type function ψ with similar results, thus eliminating the circularity in our argument.

We should perhaps verify independently that the above equations actually hold: indeed, we can give a formal argument by induction on the form of the data structure argument that they do. It is easy to see that they hold *informally*, however, by recalling the definition of $\mathrm{map_{FO}}$: in the first case, the type function argument $(\lambda\alpha.\alpha)$ will cause us to fall through to the component clause of $\mathrm{map_{FO}}$, where g is applied directly, with no recursion or any other action, thus giving the equivalence to g. In the second case, since the the variable α does not occur free in the body δ, we will never use the component clause of $\mathrm{map_{FO}}$, and thus g will never be applied (the other clauses of $\mathrm{map_{FO}}$ all pass through with no effect, under the inductive assumption that the recursively defined map is the identity). We might also note here that the second of the two equations above accounts for the triangular shape of the commutative diagram in the case of measure functions: one "leg" of the diagram has collapsed due to the degenerate nature of $\mathrm{map_{FO}}\ {\downarrow}(\tau,\sigma)\ g\ {\downarrow}(\lambda\alpha.\ \delta)$ being just the identity on σ.

The general equation displayed above, given in terms of map for lists on the left-hand side and $\mathrm{map_{FO}}$ on the right, captures all of the specific instances which we would generate by making such "in-line" expansions; we admittedly take it for granted that the results of such expansions would be intuitively appealing in any specific instance, as they were for the selector and measure cases above. Having made this small leap of faith, however, we can further abstract from the specific dependence on lists to the case of an arbitrary first-order type function (say, χ) in a natural and appealing way, with the following equation (here we have $x{:}\chi\ \alpha$ and $f{:}\forall\alpha.\ \mathrm{List}\ \alpha\rightarrow\psi\ \alpha$, with $g{:}\tau\rightarrow\sigma$ as before):

$$f{\downarrow}\sigma\ (\mathrm{map_{FO}}{\downarrow}(\tau,\sigma)\ g{\downarrow}\chi\ x)=\mathrm{map_{FO}}{\downarrow}(\tau,\sigma)\ g{\downarrow}\psi\ (f{\downarrow}\tau\ x)$$

As an alternative formulation, we can avoid the specific mention of the data structure argument x by stating the equation in terms of functional composition (this is perhaps more in keeping with the categorical bent of the discussion):

$$f{\downarrow}\sigma\ \circ\ (\mathrm{map_{FO}}{\downarrow}(\tau,\sigma)\ g{\downarrow}\chi)=\mathrm{map_{FO}}{\downarrow}(\tau,\sigma)\ g{\downarrow}\psi\ \circ\ (f{\downarrow}\tau)$$

Once again, we can express this equation as a commutative diagram:

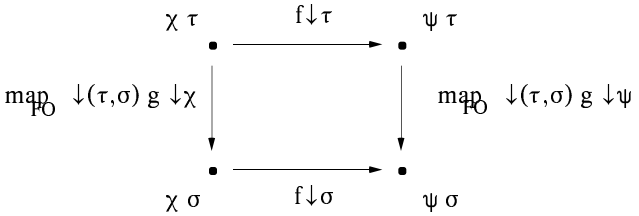

Figure 5.3.3: Commutative property for first-order map

The next step in our progressive generalization is the general map functional for higher-order data structures, with the following type:

$$\mathrm{map}_{HO} : \forall V.\ \forall \gamma{:}V\to D, \eta{:}V\to D.\ (\forall \varepsilon{:}V.\ \gamma\ \varepsilon\to\eta\ \varepsilon)\to(\forall \varphi{:}(V\to T)\to D.\ \varphi\ \gamma\to\varphi\ \eta)$$

Note in particular that this is not a direct generalization from the previous, first-order version, since here we have γ and η as type *functions*, as opposed to previously when the corresponding α and β were just simple types. If we were to take V=1, however, we would see that γ and η must then be degenerate functions and therefore equivalent to the corresponding simple types[10]. Also note that the function argument is become polymorphic, in order to account for the functional nature of γ and η.

Given some specific kind κ, then, we have the following generalized assumptions:

$$\tau,\sigma : \kappa\to D$$
$$g : \forall \varepsilon{:}\kappa.\ \tau\ \varepsilon\to\sigma\ \varepsilon$$
$$\chi,\psi : (\kappa\to T)\to D$$

and, relative to these assumptions, the following generalized equation:

$$(f\downarrow\sigma)\circ(\mathrm{map}_{HO}\downarrow\kappa\downarrow(\tau,\sigma)g\downarrow\chi) = (\mathrm{map}_{HO}\downarrow\kappa\downarrow(\tau,\sigma)g\downarrow\psi)\circ(f\downarrow\tau)$$

Finally, in graphical terms, we have the following commutative diagram:

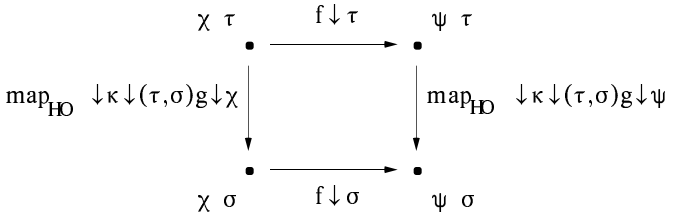

Figure 5.3.4: Commutative property for higher-order map

Note that this is almost exactly the same diagram as for the first-order case, except for the need to instantiate map_{HO} to the specific kind κ (of course, the types of the terms which serve as labels are more complex, but this does not change the diagram itself in any appreciable way).

Having motivated our characterization of the map functionals in a general way, we would like proceed to give a proof of this property for both the first- and higher-order versions of map. Unfortunately, a straightforward proof by simultaneous induction on the forms of the analyzed type parameters fails in the crucial component step: in particular, we seem to need a lemma which restricts the behavior of the polymorphic functional argument to map so that it is, in Reynolds' terms, *parametrically-polymorphic* [Rey83].

As it turns out, the very core of our approach to analytical polymorphism is the culprit here, since our use of type patterns and analytic pattern-matching over types subverts the uniform behaviors associated with the standard parametric polymorphism. We can in our language, for example, define a polymorphic function of type $\forall\alpha{:}T.\ \alpha$, but which behaves non-uniformly on certain types. More specifically, consider the following function, which is nearly the identity, but which is defined as the successor function on natural numbers:

$$\begin{aligned}&\mathrm{notID} \equiv\\ &\quad (\alpha{:}T) \Rightarrow\!\!> (x{:}\alpha \Rightarrow x)\\ &\quad \mid (\mathrm{Nat}{:}T \Rightarrow\!\!> (n{:}\mathrm{Nat} \Rightarrow \mathrm{succ}\ n)\\ &: \forall\alpha{:}T.\ \alpha\end{aligned}$$

One response to the problem described above would be to seek appropriate syntactic restrictions on the forms of analytically-polymorphic function definitions which would ensure that they are uniformly defined; although some recent work has addressed the question of parametericity from a proof-theoretic perspective [Mai91], we are not optimistic that a syntactic restriction is possible without significant changes to the underlying approach. In particular, we would need to implement what amounts to a complex semantic condition on the "similarity" of behavior of functions, namely those determined by the various clauses of an analytically-polymorphic definition.

Another response to the problem would be to alter our fundamental approach to the syntax of types and thus to the issues of coverage and compatibility for type patterns. More specifically, it is clear from the example above that the root of the problem lies in our allowing the expression of different clauses, and thus different semantic behaviors, for the same function at different types. We could eliminate the specific example above by insisting that no concrete type constructors appear in type patterns (i.e., that we always use general type-constructor variables in such contexts), but the same sort of behavior could be expressed for one of our built-in type constants, such as the unit or product.

A true resolution to the problem would thus have to challenge our inclusion of these constants in the language of types. There are other reasons (see the Conclusions chapter) to suggest that we ought consider expressing type functions as completely curried entities, thus avoiding the need for products and units as a mechanism to facilitate passing multiple type parameters. We have been as yet unable to formalize a good notion of type pattern over curried type functionals, but given such a notion it seems at least promising that the remaining distinctions necessary to the pattern-based approach, namely between different "component" types (i.e., lambda-bound type variables in the patterns) could be eliminated, thus removing the possibility of defining non-uniformly-polymorphic functions.

[10] This is not quite true, because of the restriction, discussed in earlier sections, that at least one of γ and η be a datatype (or datatype-valued degenerate function), whereas the simple types α and β could be any types.