Chapter 6
First-Order Structural Polymorphism

In the previous chapters we have developed the notions of type pattern abstraction and analytical polymorphism in the context of an explicitly-typed language with higher-order polymorphism. Most functional programming languages, however, are based not on explicit types but rather on implicit typing systems in which types are inferred for type-free terms automatically by the language compiler. The advantage of this approach is that it frees the programmer from the toil and clutter of explicit type declarations and instantiations: this is particularly important for the exploratory paradigm of program development common in functional programming, where programs are written interactively in an interpreted environment.

The standard approach to type inference[11] for functional programming is embodied in Milner's first-order polymorphic type system [Mil78]. In this system, types are automatically inferred by the language processor at compile-time, thus verifying that programs are well-typed. At run-time, all type information can be ignored, since the types serve only as a formation constraints on terms and not as objects of computation themselves. This is essentially the same phenomenon of static typing that is found in the simply-typed and explicitly-typed lambda calculi.

In this chapter and the following two, we develop a language and two type systems
which will accommodate a restricted form of analytical polymorphism called *structural polymorphism* (the particular restrictions and the motivations for making them them are the subject of Section 6.2). In so doing, we would hope to preserve all the good properties enjoyed by the Milner-style system, namely principal typing of the inference system, decidability and completeness of type inference and static typing for reduction. Unfortunately, the expressive power of type-pattern abstraction does not seem amenable to automated inference: even with reasonable restrictions, the purely implicit system described in Chapter 7 does not admit principal types and seems to present an undecidable type inference problem. Faced with these difficulties, we present in Chapter 8 a slightly modified version of the system which requires minimal user-supplied type constraints and which is consequently better behaved, though not so well as Milner's system.

We should point out that, just as for the explicitly-typed calculus with analytical polymorphism, our use of type patterns here will require that a certain amount of type information be available at run-time. Because of the restrictions we make in the next section, our need for run-time information will be limited to just the argument type functions associated with value constructors; in particular, we will not require the tagging of values with types at run-time which is typical of many systems of dynamic types (this point will be brought out later in the chapter).

Although neither of the systems presented in the last two chapters has all of the pleasant theoretical properties of the standard Milner-style type system, between them they define a range of possible engineering compromises which trade off between flexibility, expressiveness and potential for automated analysis. Our implementation experiences suggest that either system would be both usable and useful in the context of functional languages, depending on the extent to which we are willing to accommodate lack of decidability or completeness in a type system. Of the two, the system of Chapter 8, with its minimal user-supplied constraints, represents a more familiar and stable typing environment: although type inference in this system is probably incomplete, the process always terminates and is sufficiently general to handle all of our motivating examples, including the fairly complex transpose function from the end of Section 5.1.

To summarize the remainder of this chapter: in Section 6.1 we review the usual Milner-style polymorphic type system, concentrating only on the essential core features (realistic programming languages extend this core with various constructs that provide additional convenience). In Section 6.2 we sketch our overall approach to incorporating analytical polymorphism into an implicitly-typed language: in doing so, we introduce several simplifications which are necessary to make the analysis of type patterns even feasible (these restrictions distinguish *structural* polymorphism from the more general notion of *analytical* polymorphism). We describe the syntax and formation rules for the fully implicit language in Section 6.3; this language then forms the basis for the two different type systems explored in Chapters 7 and 8 (properly speaking, the language of Chapter 8 is a slight extension, involving an explicit type constraint at one point in the syntax).

6.1 Standard first-order polymorphic type inference

The standard or Milner-style system of parametric polymorphism enjoys several important properties in addition to its support of automated type inference. First, under a natural partial-ordering of types induced by the substitution of types for type variables, the system admits *principal types:* that is, the set of types which are valid for any given expression are comparable in the ordering and there is always a minimal type among these. Furthermore, the type inference algorithm always terminates and finds this most-general, or principal, type (up to the renaming of type variables). These properties are essentially inherited from the first-order unification over types which lies at the heart of the inference algorithm.

The fact that inferred types are principal constitutes a sort of completeness of the algorithm with respect to the inference system (often called *syntactic completeness*); the algorithm also enjoys *syntactic soundness* with respect to the rules, in that any type found for a term by the algorithm is guaranteed to be a valid type for that term under the inference system (by contrast, traditional or *semantic* soundness and completeness refer to properties of the inference system with respect to a semantics for the languages of expressions and types; semantic soundness is expected, but semantic completeness is impossible for any expressive language). With respect to syntactic completeness, we note that the inference rules are in some sense tailored to the type inference algorithm, although we must admit that the rules are quite natural in their own right, and thus constitute a sort of independent verification of the (common) concepts.

Finally, the Milner-style type system supports the paradigm

[11] We use the term "type inference" to refer to the process of automatically finding appropriate typings for implicitly-typed terms. Several other terminologies for this same process are in general use, including type derivation, type assignment, type discovery and type reconstruction.

of *static typing*, in that types can be erased from terms prior to reduction without effect on the final results of computation: thus no representation or manipulation of types is required at run-time. Static typing can be a source of efficiency in implementations, relative to systems which must make dynamic checks against types that are represented overtly at run-time (e.g., under most circumstances in most versions of LISP). The failure of dynamic type checks at run-time can also produce unrecoverable errors during end-user processing: by contrast, statically-typed code need never fail after the point of compilation, i.e., while it is still in the hands of the programmer (we should note, however, that there *are* circumstances in which, contrary to popular belief, dynamic types can provide an extra measure of safety: McDonald relates anecdotal evidence which demonstrates that dynamic typing may provide more graceful and informative recovery from errors whose source is *outside* the purview of the language implementation, e.g., operating systems bugs [McD92]) .

Although the various properties enjoyed by the Milner system are appealing, they nevertheless exact some price in terms of other aspects of the system. First, in order to take full advantage of the type system, a separate linguistic construct for local constant definition must be introduced. This is not necessarily a fault, however, since such definition mechanisms make for a convenient short-hand in the context of programming, and thus might be desired in their own right..

Second, and more important, not all second-order polymorphic terms can be simulated in the first-order system: only those terms which can be given types in which all quantifiers occur at the outside are accepted. For example, functions cannot be typed to take *necessarily* polymorphic arguments, i.e., with quantifications occurring to the left of the function-space arrow. Functions which cannot be typed under these restrictions are rejected by the system as ill-typed, and thus meaningless, even though they might be used in perfectly sensible ways (see Section 6.1 for examples).

Third, there are difficulties in inferring polymorphic types for recursive functions in full generality. In particular, the traditional Milner-style type system does not allow recursive functions to be used polymorphically within their own definitions. Extended systems which relieve this restriction have been proposed [Myc84], but only recently have these systems been shown to be complete [KTU88]. Currently, no practical algorithm for finding most-general types is known: the result in [KTU88] merely bounds the size of the types, so that the space of all types up to that size must be searched for a suitably correct one.

Regardless of its pleasant theoretical properties and its minor limitations, the Milner-style approach to type inference has proven highly successful in implementations of functional programming languages. This demonstrated success is perhaps more important to our purposes than formal properties which may, after all, prove inadequate or irrelevant in actual practice. Our ultimate goal is to provide practicing programmers with the tools necessary to express abstractions which are currently not expressible in the Milner system, namely those which involve quantification over data structures.

We now review the Milner-style approach to first-order polymorphic type inference in greater detail, with an eye toward fixing terminology and notation. As mentioned above, the polymorphic types of the Milner system are assigned not to arbitrary terms, but only to certain locally-defined constants, using a linguistic construct called the *let-definition;* the following grammar gives the syntax for a minimal language including this new form:

$$e ::= x \mid c \mid \lambda x.e \mid e\ e \mid \textbf{let}\ x{=}e\ \textbf{in}\ e$$

Here e ranges over expressions, x over variables and c over constants; note that the lambda-abstraction form does *not* require an explicit type constraint on the bound variable x, in contrast to the explicitly typed calculi we discussed in Chapters 1 and 2. Note also that occurrences of the variable x are considered bound in the body of the definition form, i.e.,

$$FV(\textbf{let}\ x{=}d\ \textbf{in}\ e) = FV(d) \cup (FV(e) \setminus \{x\})$$

The interpretation of a term with the let-definition form is the same as for a certain combination of a lambda abstraction and an application. Intuitively, we interpret let-definitions as follows:

$$[\![\textbf{let}\ x{=}d\ \textbf{in}\ e]\!] = (\lambda x.\ e)\ d$$

The fact that the abstraction and application are syntactically combined allows us to infer a polymorphic type for the "constant" x within the scope of its definition, where such an inference would not unsound for the corresponding case of an isolated lambda abstraction (see, e.g., [Mil78] for examples).

In addition to this definition mechanism, we must of course add quantification to the language of types in order to express polymorphism. In this first-order system, the language of types is split into two levels, types proper and *type schemes;* the latter are used to capture the restriction that all quantification over type variables must appear at the outermost level (here B ranges over some base types):

$$\begin{aligned} \tau &::= B \mid \alpha \mid \tau \to \tau \\ \sigma &::= \forall \alpha,\ldots,\alpha.\ \tau \\ A &::= \{\} \mid A,\ x{:}\sigma \end{aligned}$$

Note that type assumptions now associate a type scheme, rather than just a type, to free variables.

As an illustration of the significance of type schemes, consider the following contrasting examples: the first is a well-formed type scheme, while the second is not, assuming that Nat and Bool are among the base types of the language (we have added parentheses to emphasize that the quantification occurs outermost in the first term, but not in the second):

$$\forall \alpha.\ (\alpha \to Nat \to Bool)$$
$$(\forall \alpha.\ \alpha \to Nat) \to Bool$$

In order to codify the relationships between various types and type schemes, we define two notions of ordering on type schemes (and by extension, on types) in terms of substitution of types for type variables. The first is called *instantiation* and is defined to operate on the free type variables of a scheme (note the similarity to our notion of subsumption for patterns, and that we use the same symbol):

$$\sigma \sqsubseteq \sigma' \text{ iff } \sigma[S] = \sigma'$$

Here S is a substitution of types for type variables, which we will write with the usual syntax $S=[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$; substitutions also operate on terms in the usual manner, respecting the binding power of the quantifier $\forall$.

The second ordering is called *generic instantiation* and is defined in terms of substitution on bound variables:

$$\forall \alpha_1,\ldots,\alpha_n.\ \tau \preccurlyeq \forall \beta_1,\ldots,\beta_m.\ \tau'$$
$$\text{iff } \tau[S] = \tau' \text{ for } dom(S) \subseteq \{\alpha_1,\ldots,\alpha_n\} \text{ and}$$

$$\{\beta_1,\ldots,\beta_m\} \not\cong FV(\sigma)$$

Here the notion of free variables of a type scheme is defined in the obvious way, i.e., to respect binding by the universal quantifier $\forall$.

We now turn to the axioms and inference rules which allow us to derive that an expression is well-typed and thus well-formed. As we mentioned above, the interpretation of a let-definition term is the same as a combined abstraction and application, but the fact that the two are syntactically combined allows us to give a more flexible type than would otherwise be possible. As usual, we use judgements of the form $A \vdash e : \sigma$, where A is a set of assumptions and e an expression; note, however, that the object σ of the judgement is a scheme rather than just a type. The fact that all types are also (degenerate) schemes will be exploited in the rules: some rules will in fact require an occurrence of just a type, rather than a full scheme, which we will indicate by use of meta-variables τ, τ', etc., rather than σ, σ', etc.

The rules for variables, constants and applications are uninteresting, as they are just the same as for other calculi we have discussed in previous chapters (note that we need, as usual, a notion of global constant signature to give types for constants). For lambda abstractions, we have the following rule:

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

Note the use of types rather than type schemes in this rule, and that the type assumed for the bound variable does not occur in the abstraction term itself.

The rule for let-definitions allows a let-defined constant to be given a type scheme in the assumptions: this is thus the source of polymorphism in the system. Note that the definition form itself must be formed with respect to a proper type, and that, once again, there is no explicit type constraint on the variable x.

$$\frac{A \vdash d : \sigma \qquad A+\{x:\sigma\} \vdash e : \tau}{A \vdash (\textbf{let } x=d \textbf{ in } e) : \tau}$$

The following two rules are not tied to any specific term forms, but rather may be used at arbitrary points in the derivation—between them, they allow polymorphically-typed values to be identified and used.

$$\frac{A \vdash e : \sigma \qquad \sigma \leqslant \sigma'}{A \vdash e : \sigma'}$$

$$\frac{A \vdash e : \sigma \qquad \alpha \notin FV(A)}{A \vdash e : \forall \alpha.\sigma}$$

A different set of inference rules has become popular in recent years and is sometimes preferred because it is closer to an algorithmic interpretation of the system (this alternative system was proposed and proved equivalent to the original one in [Cl*86]). We will use this system as a basis for our own work, since it is more in the style of the rules for explicitly-typed systems we have seen in previous chapters. In particular, it does not use the last two rules of the original system above, and thus avoids the use of rules at arbitrary points in a derivation. Instead, each term form has just one rule associated with it, as is usual in the explicitly-typed case. In this sense, the first system is *non-deterministic,* whereas the second is *deterministic,* since in the second system there is but a single derivation structure possible for any term.

Whereas the objects of judgements in the non-deterministic system were schemes, the objects of the deterministic system are just proper types. The basic insight is that type schemes can be introduced in the above rules only for let-defined constants; this same idea is enforced in the deterministic system by the following rule, which gives associates a type scheme with the defined "constant" for the scope of its definition:

$$\frac{A \vdash d : \tau \qquad A \,\&\, \{x:gen(\tau,A)\} \vdash e : \tau'}{A \vdash (\textbf{let } x=d \textbf{ in } e) : \tau'}$$

This rule uses the following operation of *generalization* to specify an appropriate type scheme; generalization simply quantifies out all type variables in its type argument, excepting those which are free in the assumption argument (this is in keeping with the corresponding restriction in the rule for universal generalization in the original, non-deterministic system).

$$gen(\tau,A) = \forall \alpha_1,\ldots,\alpha_n.\tau \text{ where } \{\alpha_1,\ldots,\alpha_n\} = FV(\tau) \setminus FTV(A)$$

The other change in the deterministic system comes into play at the point when variables occur in a term: the following axiom scheme allows for each use of a variable to be typed with a fresh generic instance of the type scheme associated with it by the assumptions:

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

Here the instantiation operation is defined simply as follows:

$$inst(\forall \alpha_1,\ldots,\alpha_n.\tau) = \tau[\beta_1/\alpha_1,\ldots,\beta_n/\alpha_n], \text{ fresh } \beta_i$$

As usual, we consider any expression well-formed if we can derive a well-typing judgement with the given expression as its subject: only well-formed terms are considered meaningful, e.g., for the purposes of reduction or conversion.

An important property common to both of the type systems presented above is that well-formed expressions have *principal types*, i.e., types which are minimal with respect to the instantiation ordering (we state the following proposition for the deterministic system; generic instantiation is used to state the corresponding property of the nondeterministic system).

Proposition (principal typing):

For all assumptions A and well-typed expressions e of type τ, there exists a type ρ such that $A \vdash e : \rho$ for all τ' such that $A \vdash e : \tau'$, we have $\rho \sqsubseteq \tau$.

The inference rules specify the circumstances under which an implicitly-typed term is well-formed; while the deterministic system is close to an operational interpretation, we describe now an algorithm which will effectively compute a type for any well-formed. implicitly-typed term. Furthermore, the algorithm computes for any given term the most general type possible, in the sense of the principal typing proposition given above. Pragmatically, the type inference algorithm would be part of a compiler and would be used to check that implicitly-typed programs submitted by a

programmer are in fact meaningful. With such a positive determination, and given the static typing property, code can then be generated for the program which makes no use of run-time type checks.

The process of type inference for terms proceeds by a recursive traversal of the term; type variables are used as placeholders for the types of new program variables and more specific information about types is determined by the constraints implicit in the combination of terms. For example, a function applied to an argument must have a function type whose domain type matches that of the argument and whose co-domain type is appropriate for the context in which the application occurs. The imposition of contextually-determined constraints on types is effected through *unification* of types containing variables, which results in substitutions sending type variables to more specific types.

More formally, the type-checking algorithm $\mathcal{W}$ takes as arguments a term to be checked and a set of assumptions about the types of free variables, and returns both a type for the term and a substitution of types for type variables representing information discovered in the course of checking the argument term. Using S to range over substitutions of types for type variables, we have:

$$\mathcal{W}(e,A) = (\tau,S)$$

The definition of the algorithm, due to Milner, is given in Figure 6.1.1: it essentially follows the inference rules given above, in that it proceeds by cases on the outer form of the term argument. For each form, types are found for subterms recursively and then unified where appropriate, i.e., where common meta-variables are used in the rules. Note that the substitutions, which represent intermediate results, must be applied to the local type assumptions before recursive calls are made, since new information about variables free in the assumptions may have been determined in the prior call (substitutions are applied point-wise to the type scheme constituents of an assumption, in a way which respects the variable-binding nature of the universal quantifier).

As an example of the operation of the type inference algorithm, consider the application of a function f to an argument pair (x,3), where we assume that f has type $\alpha\times\alpha\to\mathrm{Bool}$ and x has type β, for some type variables α and β. Our top-level call of $\mathcal{W}$ is:

$$\mathcal{W}(\mathrm{f}\ (\mathrm{x},3),\ \{\mathrm{f}{:}\alpha\times\alpha\to\mathrm{Bool},\ \mathrm{x}{:}\beta\}) = \ldots$$

A recursive call to find the type of f immediately returns, finding the type associated with f in the set of assumptions:

$$\mathcal{W}(\mathrm{f},\ \{\mathrm{f}{:}\alpha\times\alpha\to\mathrm{Bool},\ \mathrm{x}{:}\beta\}) = (\alpha\times\alpha\to\mathrm{Bool},\ [])$$

Another call to find the type of the argument pair proceeds similarly, finding type β for x in the assumptions and type Nat for the natural number 3, then returning the returning the product of these types as the type of the original pair:

$$\begin{aligned}&\mathcal{W}((\mathrm{x},3),\ \{\mathrm{f}{:}\alpha\times\alpha\to\mathrm{Bool},\ \mathrm{x}{:}\beta\}) = \ldots\\ &\quad\mathcal{W}(\mathrm{x},\ \{\mathrm{f}{:}\alpha\times\alpha\to\mathrm{Bool},\ \mathrm{x}{:}\beta\}) = (\beta,\ [])\\ &\quad\mathcal{W}(3,\ \{\mathrm{f}{:}\alpha\times\alpha\to\mathrm{Bool},\ \mathrm{x}{:}\beta\}) = (\mathrm{Nat},\ [])\\ &\ldots = (\beta\times\mathrm{Nat},\ [])\end{aligned}$$

$\mathcal{W}$(x, A) =
  return (inst(A(x)), []) ;
$\mathcal{W}$(λx.e, A) =
  let (τ, S) = $\mathcal{W}$(e, A+{x:α}), fresh α ;
  return ((Sα)$\to\tau$, S) ;
$\mathcal{W}$(f e, A) =
  let (ρ, S) = $\mathcal{W}$(f, A) ;
  let (τ, T) = $\mathcal{W}$(e, SA) ;
  let U =unify(Tρ, $\tau\to\beta$), fresh β ;
  return (Uβ, UTS) ;
$\mathcal{W}$(μx.e, A) =
  let (τ, S) = $\mathcal{W}$(e, A+{x:α}) ;
  let σ = gen(τ, SA) ;
  let U = unify(Sα,τ) ;
  return (Uτ, US) ;
$\mathcal{W}$(**let** x=d **in** e, A) =
  let (ρ, S) = $\mathcal{W}$(d, A) ;
  let (τ, T) = $\mathcal{W}$(e, A+{x:gen(ρ,A)}) ;
  return (τ, TS) ;

Figure 6.1.1: Definition of the traditional algorithm $\mathcal{W}$

So far all the substitutions returned have been empty (i.e., the identity substitution), but at this point we unify the function type found for f with a type based on the product type of the argument pair to discover some new information:

$$\mathcal{U}(\alpha\times\alpha\to\mathrm{Bool},\ \beta\times\mathrm{Nat}\to\rho) = [\mathrm{Nat}/\beta,\ \beta/\alpha,\ \mathrm{Bool}/\rho]$$

In particular, we use a new type variable ρ in the co-domain of the function type to discover that the result type of the application will be Bool; in the process of unification we also discover that

$$\begin{aligned}&\mathcal{W}(\mathrm{f}\ (\mathrm{x},3),\ \{\mathrm{f}{:}\alpha\times\alpha\to\mathrm{Bool},\ \mathrm{x}{:}\beta\}) = (\mathrm{Bool},\ [\mathrm{Nat}/\beta,\ \beta/\alpha])\\ &\quad\{\mathrm{f}{:}\mathrm{Nat}\times\mathrm{Nat}\to\mathrm{Bool},\ \mathrm{x}{:}\mathrm{Nat}\}\end{aligned}$$

As discussed above in the context of the formation rules, polymorphic types are associated to variables in the assumptions in the context of a let-definition form. When the body of the definition has been checked and all type constraints satisfied, any remaining type variables are quantified out using the generalization operation, and the resulting scheme is recorded for the variable in the type assumptions used to check the scope of the definition.

The algorithm $\mathcal{W}$ enjoys two important properties which establish its close correspondence to the inference rules given above, namely *syntactic soundness* and *syntactic completeness*.

Proposition (syntactic soundness):

For all assumptions A, expressions e and types τ, if $\mathcal{W}(e,A)=(\tau,S)$, then $SA \vdash e : \tau$ is derivable according to the formation rules above.

Proposition (syntactic completeness):

For all type substitutions S, assumptions A, expressions e and types τ, if $SA \vdash e : \tau$ is derivable then $\mathcal{W}(e,A)=(\rho,R)$ for some type ρ and type substitution R; furthermore, ρ and R are principal in the sense that there is a type substitution T such that $SA=TRA$ and $T(\mathrm{gen}(\rho,RA))\sqsubseteq\tau$.

Between these properties, we know that any type found for

a term by the assignment algorithm is both a valid one, according to the rules, and also the most general one possible, in the sense of the instantiation ordering.

6.2 Basic approach and syntax

We now proceed to consider extensions to traditional functional languages and to Milner-style parametric type inference that will include a construct expressing recursive analysis of types using type patterns, as in Chapter 2, and which will thereby allow for the definition of functions which manipulate arbitrary data types and structures. Ideally, we would present a language and type system which would preserve all the best properties of traditional first-order polymorphism in this enriched context: implicitly-typed programs, a principally-typed inference system, a sound and complete type assignment algorithm and freedom from types at run-time. Unfortunately, it does not seem possible to preserve all these pleasant properties if we are to capture the essential ideas of analytical polymorphism, even in a restricted first-order context. For this reason, we present two different systems which represent different compromises between convenience and good theoretical properties. The languages of these two systems are very similar, differing only in one expressional form: it is rather their type inference algorithms which differ significantly. We take advantage of the similarity between the two systems by presenting a general approach to first-order polymorphism with type patterns and a common syntax in this section.

Before proceeding to a formal syntax, we first address some basic issues which arise immediately when we attempt to introduce type-pattern abstraction into the context of Milner-style type inference; these issues include the incorporation of type patterns in a language without explicit types; the fact that we will need access to types for matching during reduction (as we saw in the explicitly-typed case in Chapter 4); the inherent ambiguity of determining instantiations for type-functional patterns; and the treatment of recursive uses of structurally polymorphic functions within their own definitions.

Our first problem is to introduce type pattern-abstractions into the otherwise typeless syntax of an implicitly-typed language. Our approach to this problem is to use descriptive keywords to stand for a set of fixed, shallow patterns over type functions; thus we will use *keyword-pattern abstractions* in place of type-pattern abstractions to effect the type-based cases which express structural polymorphism. Although this approach limits the expressiveness of the language, it is sufficient to handle all of the motivational examples and it is much simpler than forcing the user to write type patterns (users who normally need not write types at all would presumably balk at having to write patterns over type functions, and rightly so, in my opinion).

By way of example, consider the following definition (and use) of a structurally polymorphic map functional: this definition will be our "target" in the case of map, i.e., the code we would hope to allow a programmer to write and for which we hope to automatically infer types.

```
fun map f (unit *) = *
  | map f (pair (y,z)) = (map f y, map f z)
  | map f (struc (K x)) = K (map f x)
  | map f (comp#1 a) = f a ;
(map odd alist, map even atree) ;
```

Here we assume that the values alist and atree are data structures such as lists and trees, with components appropriate to the domains of the functions even and odd. This definition is broadly comparable to the usual definition of map for lists, differing mostly in that it includes type-based clauses like the general map from the last chapter (note also the use of the constructor variable K).

The keywords marking the clauses are intended to stand for patterns ranging over type functions whose bodies are formed at top level from the unit type, the product constructor, user-defined data structures, or from the (single) component type, respectively. In general, we would have several component clauses **comp#i**, for i from 1 to n, one for each component type of an n-ary type function. At this point it is probably sufficient to note that the value patterns appearing immediately adjacent to the keywords can be interpreted as having types with the corresponding outermost forms, e.g., the pattern * has unit type, the pattern (y,z) has product type, etc.

As with traditional functional programming languages, we would expect to translate out some of the "syntactic sugar" which makes this definition palatable in favor of a smaller number of simpler language constructs: it is for this simpler "core" language that we will define type inference rules and a type inference algorithm. To continue our example, we would translate the above definition into the following one:

```
let  map = λf. μm.
  (    unit (* ⇒ *)
     | pair ((y,z) ⇒ (m y, m z))
     | struc ((K x) ⇒ K (m x))
     | comp#1 (a ⇒ f a)  )
in  (map odd alist, map even atree)
```

Note in particular that we replace the recursive definition with an explicit recursion form and that we resolve the sequential use of a definition and an expression into a let-definition: both of these transformations are typical of the traditional treatment of parametric type inference.

Less typical is that we have switched the order of abstraction and recursion in the body of the definition: this keeps the recursion and keyword-pattern abstraction forms adjacent, which we require for technical reasons discussed briefly below. The change in the order of abstraction and recursion affects the body of the definition in that the recursive uses of m now need not take the argument f as an actual parameter. We have not thoroughly investigated how to transform the more natural "syntactically sugared" definition into this form, but we expect that it is unproblematic (our formalism will address only the unsugared language).

Regarding the need to keep recursion and keyword-pattern abstraction forms adjacent, recall that all of the functions which comprise our motivational examples are recursive and are used at different type instantiations within their own definitions. The fact that these recursive uses occur at different types is in fact inherent to our approach, since any non-trivial recursively-defined function expressed in terms of type patterns will be applied recursively in different type-pattern clauses to arguments whose type depends on the pattern of that clause. As we stated in the introduction to this chapter, there is no good (i.e., complete and tractable) algorithm known which can assign polymorphic types for such recursive definitions, even in the simpler context of the first-order parametric systems.

This fact, along with other difficulties that arise when

inferring types for recursion forms, motivate us to suggest a form of expression which by its very nature forces the adjacency of recursion and keyword-pattern abstraction forms. More specifically, the form we use to express keyword-pattern abstractions is a combined form with the following syntax:

spoly x **as** $key_1\ e_1 \mid \ldots \mid key_n\ e_n$

We call this form of expression an *spoly definition.* The use of a single form in place of a specific combination of other forms is similar to the use of the let-definition form in standard Milner-style polymorphism: it provides context crucial to the type inference process and thus allows us to infer types which we could not infer for the individual constructs separately.

Having motivated the new form of expressions we will use to express the equivalent of type-pattern abstractions, we now turn to the issue of what forms of types we will use to manage the use of these expressions and to the interaction between the two levels of terms. In particular, recall from the discussion in Chapter 4 the property of static typing failed for the explicitly-typed calculus, due to the need to analyze types via type patterns during reduction (i.e., at run-time). We saw there that the manipulation of types was an inherent part of the expression reduction process: when reducing a type-pattern abstraction which was instantiated with some actual type argument, we needed to match the patterns of the abstraction against the instantiated type, both in order to determine which case of the abstraction to choose for further reduction and in order to obtain a substitution for the variables bound by the chosen type pattern.

In the context of explicitly-typed terms, this observation had the significance that we could not erase all type constituents from expressions prior to reduction (such erasure would not even lead to sensible, well-formed terms). In the context of an implicitly-typed language, the ramifications are even more significant: in the standard conception, we have no types to guide the reduction at all, much less types which we can choose to erase or not.

Our solution to this problem is to modify both the form of the judgements we use to determine that expressions are well-formed and to change our expectations about the process of reduction at run-time. More specifically, we introduce a second sort of *decorated expressions*, which will include forms for explicit instantiation, and specify formation rules for the usual untyped or *raw expressions* in such a way as to associate a decorated expression with each well-typed raw expression. Our notion of reduction will then be defined for decorated expressions only, under the interpretation that we will obtain a decorated expression for any candidate raw expression which turns out to be well-formed, and will then reduce the decorated expression in place of the original, raw one.

We therefore use judgements of the following form, where A is a set of type assumptions (as usual), e a raw expression, d a decorated expression and τ a type:

$$A \vdash e \mapsto d : \tau$$

(there is actually some precedent for this approach even in the context of implicit type systems which do not actually need types at run-time: see [MH88] for an application).

In addition to introducing explicit instantiations, the process of transformation will replace keyword-patterns with the corresponding type patterns and resolve the combined spoly-form into separate recursion and type-pattern abstraction forms. Thus we will be able to define reduction for decorated expressions in much the same way that we did for the expressions of the explicitly-typed languages of previous chapters.

While it is necessary that we allow for some manipulation of types at run-time, one of our goals will be to limit such manipulation as much as possible. In particular, we will produce decorated terms which use explicit instantiation only at the points where type-pattern abstractions are resolved: uses of locally-defined constants which exploit the usual Milner-style parametric polymorphism will require no explicit instantiations in the decorated terms.

Terminologically, we refer to the sort of polymorphism based on type-patterns as *structural polymorphism*, for reasons which will become apparent from further discussion below. Technically, we manage the distinction between standard parametric polymorphism and structural polymorphism by using two different forms of quantification to distinguish the different forms of generic types. More specifically, we use *intersection types* with the following syntax to give types to keyword-pattern abstractions and type-pattern abstractions simultaneously:

$$\tau ::= \cdots \mid \bigcap \alpha{:}\delta.\tau \mid \bigcap \mathbf{S}_\gamma{:}\delta.\tau$$

Here α will range over both individual types and type functions, depending on the data kind δ (in either case, the same data kind restrictions which we motivated in the explicitly-typed case are enforced, i.e., we have no patterns over functional or polymorphic types). We will explain the use of the second intersection form below.

Note that in any case we do not require computations which *produce* types as results, as in some systems (see e.g., [Car86]) nor do we require that values be tagged with their types at run-time as in some approaches to dynamic types (see e.g., [LeMa91]). Rather, we only use the pattern-based analysis of types to drive computations on data at the value level: our language is still statically-typed in the sense that all run-time type errors can be eliminated by compile-time type-checking and values need not be tagged with their types at run-time in order to participate in structurally polymorphic computations.

In the context of explicitly-typed languages, the notion of polymorphism over data structures required type functions and patterns over type functions in order to express the notion of a component of a data structure. Unfortunately, there are fundamental problems with the treatment of type functions in an implicitly-typed language with first-order polymorphism. Recall that the standard Milner-style type inference system is based on the notion of (first-order) unification over types: types which are required by context to be identical are unified in order to determine substitutions for the type variables which occur in them. But unification in a language which contains variables ranging over functions is, in general, undecidable (see e.g., [Gol81]).

In the current context, we need not even appeal to this undecidability to see that problems will arise: just the inherent *ambiguity* of higher-order unification will be sufficient. For the sake of simplicity, let us temporarily express both parametric and structural polymorphisms with just the familiar universal quantifier and assume that all polymorphic expressions must be explicitly instantiated in the transformation to decorated terms. Then consider a context which includes expressions with the following types, where φ is a function from types to types:

foo : $\forall \eta.\ \eta \rightarrow$ Bool
map : $\forall \alpha, \beta, \varphi.\ (\alpha \rightarrow \beta) \rightarrow \varphi\ \alpha \rightarrow \varphi\ \beta$
LTN : List(Tree Nat)

In this context, it is not clear how we should interpret the following raw expression, i.e., it seems that we might choose either of the given decorated expressions as the results of the type inference transformation:

map foo LTN
= map↓(Tree Nat, Bool, λε. List ε) foo LTN
: List Bool

or

= map↓(Nat, Bool, λε. List(Tree ε)) foo LTN
: List (Tree Bool)

In the first case, we take just the data structure List as the function instantiation for φ; in the second case we take the "composition" of List with Tree. Either of these two interpretations is possible and sensible, since each of the following equations holds:

$\varphi\ \alpha\ [(\lambda\varepsilon.\ \text{List}\ \varepsilon)/\varphi,\ (\text{Tree Nat})/\alpha] = \text{List(Tree Nat)}$

and

$\varphi\ \alpha\ [(\lambda\varepsilon.\ \text{List(Tree}\ \varepsilon))/\varphi,\ \text{Nat}/\alpha] = \text{List(Tree Nat)}$

The basic problem is that there is not a unique, minimal solution to unification problems such as these, where we return a lambda abstraction as a substitution for a function variable; rather, there are typically several incomparable solutions, and the general problem is in fact undecidable.

Although we will have more to say about this issue later, for now we suggest that it is reasonable to make some restriction on the forms of functions which may be used to instantiate quantified type function variables. In particular, we make the restriction that structurally polymorphic values be instantiated only with first-order, outermost type constructors, at least in the context of their use, as opposed to the context of their definition (the use of the combined spoly definition form will give us the context necessary to handle more complex type function instantiations within the scope of the definitions).

According to this policy, in the example above we would choose the first of the two solutions, as the instantiation of φ to $(\lambda\varepsilon.\ \text{List}\ \varepsilon)$ corresponds to the identification of List as the outermost data structure in List(Tree Nat), the type of the term LTN.

The restriction to outermost data structures is the origin of the term *structural polymorphism;* it also serves to motivate the distinction between the two intersection forms of types we introduced above: we use the first of the two forms to express the more general type function quantifications allowed within the scope of spoly-definition forms, but use the second form to restrict "external" occurrences of the defined function to instantiations by structures only.

Although the choice to use outermost structures is somewhat arbitrary, it has the advantage that it is specific, i.e., we will always be able to choose an instantiation unambiguously. Furthermore, in many cases we will be able to use multiple applications of structurally polymorphic functions to achieve the affect of an instantiation with a more complex type function. For example, we can express the second interpretation from the example above using two successive applications of map, which will be transformed as follows:

map (map foo) LTN
= map↓(Tree Nat, Tree Bool, λε. List ε)
(map↓(Nat, Bool, λε. Tree ε) foo LTN)
: List (Tree Bool)

We should note here that the restriction to first-order data structures does not constrain expressiveness unduly, as no current functional languages allow the definition of higher-order data structures. It is not clear that it would be impossible to accommodate such definitions in current languages, however—we defer comment on such possibilities until the Conclusions.

This completes the sketch of our basic approach: keyword representatives for type patterns, a definitional form which combines aspects of recursion and type-pattern abstraction, the introduction of limited type decorations during type inference, the use of intersection types to distinguish structural from parametric polymorphism, and the restriction of instantiated type functions to first-order, outermost type constructors.

In order to provide a concrete example of some of the features and restrictions described, we return briefly to the map function: we expect a programmer to define and use map as in the following raw expression (or perhaps some more sugared version, which would be translated into this form during some phase of compilation):

let map = λf. **spoly** m **as**
(**unit** ∗ ⇒ ∗
| **pair** (y,z) ⇒ (m y, m z)
| **struc** (K x) ⇒ K (m x)
| **comp#1** a ⇒ f a)
in (map odd alist, map even atree)

We would then like to generate the following decorated expression as the result of the type inference/decoration phase of compilation; this decorated term would in turn be used as the basis for the run-time execution of the function:

let map = λf. μm.
($(\lambda\varepsilon{:}\text{T}.1) \Rightarrow (*\Rightarrow *)$
| $(\lambda\varepsilon{:}\text{T}.\psi\ \varepsilon\times\zeta\ \varepsilon) \Rightarrow ((y,z) \Rightarrow (m{\downarrow}\psi\ y,\ m{\downarrow}\zeta\ z))$
| $(\lambda\varepsilon{:}\text{T}.\mathbf{S}_{\nu}(\chi\ \varepsilon{:}\mathbf{V}_{\nu})) \Rightarrow$
$((\mathbf{K}_{\nu\mu}\ x) \Rightarrow \mathbf{K}_{\nu\mu}\ (m{\downarrow}(\lambda\varepsilon.\mathbf{T}_{\nu\mu}(\chi\ \varepsilon))\ x))$
| $(\lambda\varepsilon{:}\text{T}.\varepsilon) \Rightarrow (a \Rightarrow f\ a)$)
in (map odd↓List alist, map even↓Tree atree)

Note in particular here that we need instantiate map only with the appropriate type functions, as the other variables occurring in its type will be bound by the usual parametric quantifier (note also that we have taken the liberty of "transliterating" the use of the symbolic constructor-variable K into the indexed form described in Chapter 4).

6.3 Syntax and formation for structural polymorphism

We now turn to the formal description of the language and type system sketched above: our approach is to extend the basic language of the Milner-style type system (i.e., a minimal "core" of Standard ML) to include a the new forms of types

and expressions and to specify formation rules which enforce the restriction to structural polymorphism. The syntax and technical treatment of many of the novel features will follow the lead of the explicitly-typed calculus of previous chapters, so that we essentially have a mix of Milner-style and explicitly-typed terms (this makes sense, given our understanding of reduction on decorated terms). The various definitions we describe here are gathered together in Appendix C in the usual reference-oriented style; since the definitions overlap with those of the explicitly-typed calculus, much of Appendix C simply refers to the material in Appendix B.

The overall architecture of the structurally-polymorphic calculus consists of three levels, namely kinds, types and expressions. The language of kinds is essentially just that of the higher-order calculus, although we will make restrictions on the *order* of kinds appearing in various contexts (see directly below). At the level of types, we will have need for some of the syntax of type functionals, although the restrictions on kinds will limit functions on types to low orders. In addition to types and type functionals, we have type schemes, which are just the usual universally-quantified types of Milner-style type inference, although note that we allow internal quantifications of the intersection forms described above. Finally, at the level of expressions we have two sorts, the raw and the decorated expressions described above; these two expression sorts will participate jointly in the "transformational" formation judgements which relate them to types.

We begin with a notion of order for kinds, which we define over the language of kinds from Chapter 2 as follows:

$$\mathcal{O}(1) = 0 \qquad \mathcal{O}(\mathbf{B}) = 0$$

$$\mathcal{O}(\mathrm{V}) = 0 \qquad \mathcal{O}(\mathbf{V}_i) = 0$$

$$\mathcal{O}(\delta\times\eta) = \max(\mathcal{O}(\delta), \mathcal{O}(\eta))$$

$$\mathcal{O}(\kappa\to\delta) = \max(1+\mathcal{O}(\kappa), \mathcal{O}(\delta))$$

We use this notion of order to restrict the kinds which appear in constant settings, constructor settings and type assumptions to order one; the details are rather uninteresting, so we simply refer the reader to the inference rules of section II.C of Appendix C.

Although the language of raw expressions is free from explicit mention of types, we need a language of types in order to give types to constants and constructors, to state formation rules and as constituents in the explicit instantiation form of decorated expressions. The context-free syntax of types is given by the following grammar:

$$\theta ::= \chi \mid \alpha \mid \mathbf{S}_i \mid \mathbf{T}_{ij} \mid \theta,\theta \mid \theta\ \theta \mid \lambda\pi.\theta \mid \bigcap\alpha{:}\delta.\theta \mid \bigcap\mathbf{S}_v{:}\delta.\theta$$

All the meta-variables used here range over the sorts corresponding to those of the explicitly-typed calculus.

In addition to the types, we also need quantified forms corresponding to the usual Milner-style type schemes; using σ to range over type schemes, we have:

$$\sigma ::= \theta \mid \forall\alpha{:}\mathrm{T},\ldots,\alpha{:}\mathrm{T}.\ \theta$$

Here the use of type-formation rules will restrict the bodies of schemes to be just types, rather than type functionals. Note that the type variables quantified by schemes range over proper types, i.e., without any type functions or data type distinctions.

The type formation rules are largely the same as those of the explicitly-typed calculus, but with the following exceptions:

$$\frac{\mathrm{A}_K \vdash \pi : \kappa \qquad \mathrm{A}+_{TV}\,\mathcal{A}(\pi)^- \vdash \theta : \delta}{\mathrm{A} \vdash (\lambda\pi.\theta) : \kappa\to\delta}$$

$$\frac{\delta = \delta^- \qquad \mathcal{O}(\delta)\leqslant 1 \qquad \mathrm{A}+_{TV}\{\alpha{:}\delta\} \vdash \theta : \mathrm{T}}{\mathrm{A} \vdash (\bigcap\alpha{:}\delta.\theta) : \mathrm{T}}$$

$$\frac{\mathrm{A} \vdash \theta : \mathrm{T}}{\mathrm{A} \vdash (\bigcap\mathbf{S}_v{:}\delta.\theta) : \mathrm{T}}$$

The first rule simply enforces the restriction that type functions expressed as lambda abstractions be first-order, i.e., have first-order kinds. The other two rules allow for the formation of intersection types as proper types, given that their bodies are proper types. The rule for the first form of intersection types enforces the restrictions that the quantified variables be data types (i.e., they range over types free from functional and polymorphic forms) and that they be at most first-order (we need to allow for both individual data types and functional data types).

At certain points in the descriptions below we will treat first-order type functions as though they had simply a tuple of type parameters—this is especially convenient for the purposes of referring to the ith component of a type function or data structure. In the syntax above, we have allowed for the same sort of binary tree-structured parameters which we formalized for the explicitly-typed calculus. We allow ourselves the convenience of borrowing from the explicitly-typed formalism while informally treating type parameters rather as lists or tuples of individual type arguments. To be more rigorous, we could restrict type parameters to the usual binary-tree representation of lists, using the unit form as "nil" and restricting the products to always have just the type constant kind on the left, say.

We also need a notion of well-formed type scheme, which is just an extension of the notion of well-formed type. In particular, we use the following rule to derive judgements of the form $\mathrm{A} \vdash \sigma\ \mathrm{scheme}$:

$$\frac{\mathrm{A}+_{TV}\{\alpha_1{:}\mathrm{T},\ \ldots,\ \alpha_n{:}\mathrm{T}\} \vdash \theta : \mathrm{T}}{\mathrm{A} \vdash (\forall\alpha_1{:}\mathrm{T},\ \ldots,\ \alpha_n{:}\mathrm{T}.\theta)\ \mathrm{scheme}}$$

We now turn to the context-free syntax of raw expressions, which is given by the following grammar:

$$e ::= * \mid x \mid \mathbf{K}_{ij} \mid e,e \mid e\ e \mid (p\Rightarrow e)^+ \mid \mu x.e \mid \mathbf{spoly}\ x\ \mathbf{as}\ (\mathrm{key}\ e)^+ \mid \mathbf{let}\ x = e\ \mathbf{in}\ e$$

Here e ranges over raw expressions and key ranges over *keyword patterns* as described below (all the other meta-variables range over the usual sorts). As in previous specifications, we use the postfix $^+$ sign to indicate a non-zero number of constituent clauses separated by vertical bars.

The context-free syntax of decorated expressions is similar to that of raw expressions, except that the spoly-definition form is replaced by type-pattern abstraction and the explicit type instantiation form is revived (both of these forms use the same syntax as in the explicitly-typed calculus). Here we use d (and later also g) to range over decorated expressions:

$$d ::= * \mid x \mid \mathbf{K}_{ij} \mid d,d \mid d\ d \mid (p \Rightarrow d)^{+} \mid \mu x.e$$
$$\mid d{\downarrow}\theta \mid (\rho \Rightarrow\!\!> d)^{+} \mid \textbf{let}\ x = d\ \textbf{in}\ d$$

Both raw and decorated expressions use the same sort of patterns to express pattern-abstraction over values; unlike in the explicitly-typed calculus of the previous chapter, though, occurrences of variables in patterns are not decorated with explicit type constraints (they are nevertheless still binding occurrences). This means that patterns are now just a subsort of expressions, with the following context-free syntax:

$$p ::= c \mid x \mid p\ ,\ p \mid \mathbf{K}_{ij}\ p$$

As patterns are just a subsort of expressions, we will be able to depend on the formation rules for expressions in order to restrict the formation of patterns; as we discuss below, our rule for pattern abstraction formation will necessarily be somewhat different, as we cannot project assumptions out of implicitly-typed patterns in the same way as we did the explicitly-typed case.

We also need sorts for keyword patterns and type patterns: these two stand in roughly the same relationship to each other as do raw and decorated expressions, in which they occur, respectively, as constituents. For type patterns, we use the same syntax as in Chapter 2, noting only that the restrictions to first-order types will be enforced by virtue of our erasure-based notion of formation for patterns.

The syntax of keywords is rather trivial, consisting of three fixed atomic forms and one form with just a natural number constituent:

$$key ::= \textbf{unit} \mid \textbf{pair} \mid \textbf{struc} \mid \textbf{comp\#}i$$

(essentially the i in **comp#**i is a positive integer "manifest constant", since we will not allow the case where n=0).

We will describe the notions of coverage and compatibility for keywords below; we also defer to later a discussion of the correspondence between keyword patterns and type patterns, which is used to effect the transformation of keyword abstractions into type-pattern abstractions.

The syntax of assumptions and settings is similar to that used for the explicitly-typed calculus, but now they associate type schemes (rather than just types) to value variables and constants. The formation rules for these associations must be changed to reflect this change in sorts and also to restrict the data kinds of constructor argument type functionals to be at most first order. Since these changes are straight-forward, we refer the reader to Appendix C for details.

$$A ::= \langle A_T,\ \{x{:}\sigma,\ \ldots\}\rangle \equiv \langle A_T,\ A_V\rangle$$

Next, we turn to the formation rules for expressions: as we discussed above, we use a non-standard judgement form which relates a raw expression and a decorated expression to a type (in the context of a set of type assumptions).:

$$A \vdash e \mapsto d : \tau$$

The rules we give for deriving these judgements are oriented in a sense, so that their application is driven by the forms of the raw expressions. That is, the rules already convey a sense of the operational interpretation we will impose later, under which we translate raw expressions into decorated ones by a process of induction on the forms of raw expressions.

In several cases of atomic expressions, the decorated and raw expression forms will be the same; it is convenient to make use of the following abbreviation in these situations:

$$A \vdash e : \tau \triangleq A \vdash e \mapsto e : \tau$$

where the right-most occurrence of e is actually the corresponding decorated expression

We also need a notion of explicit instantiation to specify the rules for the atomic cases; the operation is essentially the same as for the Milner-style parametric calculus, but note that instantiation does not affect variables quantified by our new forms of intersection types:

$$\text{inst}(\forall \alpha_1{:}T,\ldots,\alpha_n{:}T.\tau) = \tau[\beta_1/\alpha_1,\ldots,\beta_n/\alpha_n],\ \text{fresh}\ \beta_i$$

Given these preliminaries, the rules for the atomic cases of expressions are straight-forward:

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

$$A \vdash x : \text{inst}(A_V(x))$$

$$\frac{C_{TS}(n) = \kappa \rightarrow B \qquad (C_{VK}(n)(m))\ \alpha = \theta}{A \vdash \mathbf{K}_{nm} : \text{inst}(\forall \alpha{:}\kappa.\ \theta \rightarrow \mathbf{S}_n\ \alpha)}$$

$$\frac{C_{TS}(n) = \kappa \rightarrow B}{A \vdash \mathbf{K}_{n\mu} : \text{inst}(\forall \alpha{:}\kappa.\ \mathbf{T}_{n\mu}\ \alpha \rightarrow \mathbf{S}_n\ \alpha)}$$

$$A \vdash \mathbf{K}_{v\mu} : \text{inst}(\forall \alpha{:}\mathbf{V}_v.\ \mathbf{T}_{v\mu}\ \alpha \rightarrow \mathbf{S}_v\ \alpha)$$

Note that we must instantiate the schemes associated to constants and variables by the appropriate contexts, since the judgment forms we use have a type as their object. We do not anticipate any need for structurally-polymorphic constants, but our rules allow for them; intersection-bound type variables must, however, be instantiated according to a separate rule below.

The rules for pairing and function applications are also straight-forward, simply specifying that terms are related according to the relations of their constituents:

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

$$\frac{A \vdash f \mapsto g : \sigma \rightarrow \tau \qquad A \vdash e \mapsto d : \sigma}{A \vdash (f\ e) \mapsto (g\ d) : \tau}$$

The rules for recursion and let-definition forms are not much more interesting, but note the use of a type (i.e., a degenerate form of type scheme) in the assumption on the recursion variable and the use of explicit generalization operation in the assumption on the let-defined "constant":

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

$$\frac{A \vdash e \mapsto d : \tau \qquad A+\{x{:}gen(\tau,A)\} \vdash e' \mapsto d' : \tau'}{A \vdash (\textbf{let } x = e \textbf{ in } e') \mapsto (\textbf{let } x = d \textbf{ in } d') : \tau'}$$

The restriction that a recursion variable be given only a type might be relaxed according to research on inferring polymorphic types for recursively-defined functions (see [Myc84, KTU88]), but we have no special need for these extensions, since our structurally-polymorphic functions will be typed with the explicitly-instantiated intersection types. As in the case of explicit instantiation, we define the generalization operation much the same as for the original Milner-style type system. Note that we must be more careful here to quantify only variables of the proper kind and to project out of conglomerate associations:

$$gen(\tau,A) = \forall \alpha_1{:}T,\ldots,\alpha_n{:}T.\tau \quad \text{where}$$
$$\{\alpha_1,\ldots,\alpha_n\}=\{\alpha \in FV_{TV}(\tau) \setminus FV_{TV}(A_V) \mid A_{TV}(\alpha)=T\}$$

We now turn to the rules governing the two forms of pattern abstraction in the structurally-polymorphic calculus: in the explicitly-typed languages of previous chapters we cast formation rules for pattern abstraction forms for both values and types in terms of compatibility, coverage and the formation of constituent patterns and expressions. In the current calculus, the notions of compatibility and coverage will remain the same for value patterns, while the notion of formation is actually simpler: as we noted above, patterns are just a subsort of expressions and thus are well-formed by virtue of the same rules as for expressions.

Based on these ideas, we can state the inference rule for value-pattern abstractions as follows:

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

Note that we cannot explicitly project assumptions from value patterns in this language since there are no constraints on the variables bound by the patterns. Instead, we simply require that the assumptions used in the formation of each pattern constituent be used to extend the global assumptions for the formation of the corresponding right-hand side expression (in the process of type inference, we will use an operation which asserts a fresh type variable as the type of each pattern variable, maintaining contextual constraints by unification and returning an assumption result).

The rule governing the formation of structurally-polymorphic definitions combines aspects of the rule for recursion with a new notion of keyword-pattern abstraction. Keyword patterns are in some sense translated into the corresponding type patterns, although we must restrict the keyword patterns with resect to coverage and compatibility, since we interpret the raw expression as the "input" term. That is, we need to check that the keyword patterns are properly combined, since they are given arbitrarily by the programmer, but since we have a functional relation between keyword patterns and a fixed set of known type patterns, we need not be concerned with the possibility of conflicts between the type patterns.

More formally, we use the following operation to specify the relationship between keyword patterns and the corresponding type patterns; in order to account for the lambda abstractions needed in the first-order cases, the operation $\mathcal{TP}$ takes also a data kind argument. The zero-order cases are then just:

$$\mathcal{TP}(D, \textbf{unit}) = 1$$
$$\mathcal{TP}(D, \textbf{pair}) = \alpha{:}D\times\beta{:}D \quad (\text{fresh } \alpha, \beta)$$
$$\mathcal{TP}(D, \textbf{unit}) = \mathbf{S}_\nu\ (\chi{:}\mathbf{V}_\nu) \quad (\text{fresh } \nu, \chi)$$

The first-order cases are similar, but return lambda-abstracted type patterns whose pattern-bound variables are appropriately applied to the parameters of the abstraction (this according to the conception of type patterns we described in Chapter 3):

$$\mathcal{TP}(\kappa_1\times\cdots\times\kappa_n\to D, \textbf{unit}) = \lambda\varepsilon_1{:}\kappa_1, \ldots, \varepsilon_n{:}\kappa_n.\ 1$$
$$\mathcal{TP}(\kappa_1\times\cdots\times\kappa_n\to D, \textbf{pair}) =$$
$$\lambda\varepsilon_1{:}\kappa_1, \ldots, \varepsilon_n{:}\kappa_n.\ (\psi\ (\varepsilon_1,\ldots,\varepsilon_n){:}D)\times(\zeta\ (\varepsilon_1,\ldots,\varepsilon_n){:}D)$$
$$(\text{fresh } \psi, \zeta)$$
$$\mathcal{TP}(\kappa_1\times\cdots\times\kappa_n\to D, \textbf{struc}) = \lambda\varepsilon_1{:}\kappa_1, \ldots, \varepsilon_n{:}\kappa_n.\ \mathbf{S}_\nu\ (\chi$$
$$(\varepsilon_1,\ldots,\varepsilon_n){:}\mathbf{V}_\nu) \quad (\text{fresh } \nu, \chi)$$
$$\mathcal{TP}(\kappa_1\times\cdots\times\kappa_n\to D, \textbf{comp\#i}) = \lambda\varepsilon_1{:}\kappa_1, \ldots, \varepsilon_n{:}\kappa_n.\ \varepsilon_i$$

Note that we use indexed tuples of kinds above merely as a device to indicate a specific number of parameters: we actually intend that all of these kinds be just the type constant T (i.e., we should really write something like "$T\times\cdots\times T$, for n occurrences of T" for the kinds on the left). It might seem that we should also specify that the $\mathcal{TP}$ operation fails for data kind arguments of order greater than 1, but in fact the rule below will encompass this restriction by virtue of the formation constraints on the data kind constraints in the intersection type; these two alternative approaches to the restriction seems equally reasonable (our particular choice is thus merely a matter of preference).

With regard to the compatibility of keyword patterns, we need only ensure that we have not used the same keyword form twice in a single abstraction:

$$\{key_1,\ldots,key_n\} \text{ compatible iff for all } 1\leqslant i\neq j\leqslant n,$$
$$key_i \neq key_j$$

Conversely, with regard to the coverage of a data kind by keyword patterns, we need to ensure that for a given n-ary type function, all the relevant "cases" of outermost form are accounted for. Recall that we consider only data kinds of at most first-order and that we (rather informally) coerce the tree-structured parameters engendered by arbitrary products to lists or tuples of type arguments. Thus we must ensure that we have at least one occurrence of each of the three fixed keywords for units, pairs and structures and of the appropriate component keyword for each argument of the n-ary type functions over which the abstraction ranges:

$$keys \succ T^n\to D \text{ iff } \{\textbf{unit}, \textbf{pair}, \textbf{struc}\} \subseteq keys \text{ and for}$$
$$\text{all } i\leqslant n, \textbf{comp\#i} \in keys$$

Given these operations and relations on keyword patterns, we can now present the rules governing the formation of structurally-polymorphic definitions: we actually need two rules, one for the case with an intersection-bound individual type variable of kind D and one for the case of an intersection-bound type-structural variable (of form $\mathbf{S}_\nu$). These two cases differ not so much in the mechanics of the rules themselves, but in that we must restrict the second case to only structural variables, rather than general type-function variables.

The first rule, governing the case for individual type variables, is as follows:

$$\frac{\begin{array}{c}\{key_1, \ldots, key_n\} \text{ compatible} \\ \{key_1, \ldots, key_n\} \succ D \\ \mathcal{TP}(key_i) = \rho_i \qquad (1 \leqslant i \leqslant n) \\ (\{\alpha\} \cup \bigcup_i FV(\rho_i)_T) \ncong FV(A)_T \\ A +_T \mathcal{A}(\rho_i) + \{x{:}\bigcap\alpha{:}D.\tau\} \vdash e_i \\ \mapsto d_i : \tau[\rho_i/\alpha] \qquad (1 \leqslant i \leqslant n)\end{array}}{\begin{array}{c}A \vdash (\textbf{spoly } x \textbf{ as } (\cdots \mid key_i\ e_i \mid \cdots)) \\ \mapsto (\mu x.\cdots \mid \rho_i \Rrightarrow d_i \mid \cdots) : \bigcap\alpha{:}D.\tau\end{array}}$$

Note the restriction that the variable of the quantification and the pattern-bound variables of the type patterns not interfere with the free type variables of the global assumptions: this is just the usual free-variable restriction familiar from the explicitly-typed lambda and pattern calculi (in fact, we have defined the $\mathcal{TP}$ operation to return fresh pattern-bound variables, but we prefer to make the restriction explicit in this rule as a matter of emphasis). The substitution of type patterns for the variable of quantification in the types of corresponding expressions should be familiar from the rule for type patterns from the explicitly-typed calculus, which is in fact quite similar to the rule above.

The rule for the structural-type variable case is similar, but restricts the ultimate type of the term to just a variably-indexed structural type:

$$\frac{\begin{array}{c}\{key_1, \ldots, key_n\} \text{ compatible} \\ \{key_1, \ldots, key_n\} \succ \kappa{\to}D \\ \mathcal{TP}(key_i) = \rho_i \qquad (1 \leqslant i \leqslant n) \\ (\{\alpha\} \cup \bigcup_i FV(\rho_i)_T) \ncong FV(A)_T \\ A +_T \mathcal{A}(\rho_i) + \{x{:}\bigcap\alpha{:}\kappa{\to}D.\tau\} \vdash e_i \\ \mapsto d_i : \tau[\rho_i/\alpha] \qquad (1 \leqslant i \leqslant n)\end{array}}{\begin{array}{c}A \vdash (\textbf{spoly } x \textbf{ as } (\cdots \mid key_i\ e_i \mid \cdots)) \\ \mapsto (\mu x.\cdots \mid \rho_i \Rrightarrow d_i \mid \cdots) : \bigcap\mathbf{S}_\nu{:}\kappa{\to}D.\tau\end{array}}$$

Note that we allow for an arbitrary type-function variable in the type assumed for the recursion variable, rather than restricting to a variably-indexed structure, which we do for the ultimate type of the whole expression: the context provided by the spoly definition form makes this possible.

The above rules allow us to introduce intersection types for structurally-polymorphic definitions: in order to eliminate these intersection types, we need rules which provide an explicit type instantiation in the decorated expression constituent of the judgement:

$$\frac{A \vdash f \mapsto g : \bigcap\alpha{:}D.\tau \qquad A_T \vdash \theta : D}{A \vdash f \mapsto (g{\downarrow}\theta) : \tau\,[\theta/\alpha]}$$

$$\frac{A \vdash f \mapsto g : \bigcap\mathbf{S}_\nu{:}\kappa{\to}D.\tau \qquad A_T \vdash \mathbf{S}_i : \kappa{\to}D}{A \vdash f \mapsto (g{\downarrow}\mathbf{S}_i) : \tau\,[i/\nu]}$$

According to these rules, we always explicitly instantiate a raw expression of intersection-bound type during the transformation to decorated form; although this is reasonable and sound, there is one technical change we can make to the system which allows for some greater flexibility: in particular, we are concerned with let-definitions which include "incompletely resolved" instantiations of intersection-bound variables in their definition bodies. Consider the following (admittedly artificial) example of a let definition which employs applies the map function in its body:

let me = map even **in** e

By the rules above, this raw expression can be well-typed in relation to the following decorated expression:

let me = map even$\downarrow\mathbf{S}_\nu$ **in** d

Occurrences of me will be given the type $\mathbf{S}_\nu\text{Nat}{\to}\mathbf{S}_\nu\text{Bool}$ for the scope of the definition (i.e., within d), assuming a type of $\text{Nat}{\to}\text{Bool}$ for the function even.

Unfortunately, this is a bit too restrictive, since there is nothing to prevent the function me from being used at different types within the scope of the definition, which will not be possible under the current typing. The problem is essentially that we have "prematurely" instantiated the application of map to the type $\mathbf{S}_\nu$, only to immediately bind the instantiation to a local constant (here me) which would otherwise be treated polymorphically.

One way to rectify the situation is to modify the definition of the generalization operation which is used give type schemes to let-defined constants. In particular, we need to allow for the fact that any free occurrences within the type of the constant which have the form $\mathbf{S}_\nu$ (or any free variables with kind D) should be generalized. These variables should not, however, be generalized in the usual way with universal quantifications via type schemes, but rather *internally* with intersection bindings, since we will need to propagate their dependence on explicit instantiation through the let-definition to their ultimate uses.

The following definition of of generalization suffices, though it is not especially elegant to state:

$$\begin{array}{l}\text{gen}(\tau, A_V) = \\ \quad \forall\,\alpha_1{:}T,\ldots,\alpha_n{:}T. \\ \quad \bigcap\mathbf{S}_{\nu 1}{:}\kappa_1,\ldots,\mathbf{S}_{\nu m}{:}\kappa_m.\bigcap\beta_1{:}D,\ldots,\beta_k{:}D.\tau \\ \text{where} \\ \quad \{\alpha_1,\ldots,\alpha_n\} = \{\alpha \in FV_{TV}(\tau) \setminus FV_{TV}(A_V) \mid A_{TV}(\alpha) = T\}, \\ \quad \{\nu_1,\ldots,\nu_m\} = FV_{TI}(\tau) \setminus FV_{TI}(A_V), \\ \quad \kappa_i = A_{TS}(\mathbf{S}_{\nu i}), \text{ for } 1 \leqslant i \leqslant m, \text{ and} \\ \quad \{\beta_1,\ldots,\beta_k\} = \{\beta \in FV_{TV}(\tau) \setminus FV_{TV}(A_V) \mid A_{TV}(\beta) = D\}\end{array}$$

With this new definition of generalization, we will allow for occurrences of the let-defined constant me to have the type scheme $\bigcap\mathbf{S}_\nu{:}\kappa{\to}D.\ \mathbf{S}_\nu\text{Nat}{\to}\mathbf{S}_\nu\text{Bool}$, for appropriate κ,

within the scope of its definition.

We complete the current section and chapter by highlighting one of the major differences between the Milner-style type system and the structurally-polymorphic system just described: in contrast to the traditional parametric system, the natural inference rule for our language does not admit principle types. That is, a single term may have several incompatible types under the usual instantiation order. This problem stems both from the non-parametric nature of the analytic notion of polymorphism (which is thus incompatible with an ordering based on parametric substitution), and from the ambiguity that can result when user-specified patterns do not contain enough information to precisely determine the intended rôle of the arguments.

Note that this problem of ambiguity appears already at the stage of the type inference system for the language: it is not merely an artifact of the type inference algorithm, but an inherent aspect of the type system. Barring a new inference rule, any type inference algorithm which is complete with respect to the inference system will necessarily return multiple, incomparable results in some cases.

The "failure" of principal typing for our new style of polymorphism is actually not all that surprising. In Milner's system, the notions of most general type and of polymorphism are both based on the idea of a type parameter, thus yielding a natural correspondence between the two. In our system, the notion of polymorphism is *analytic* rather than *parametric*, even though we use the same parametric form for expressing of polymorphism.

Furthermore, as the following example should demonstrate, the "failure" of principle typing for our system is not particularly important in practice: the problems result from a specialized set of circumstances in which the patterns used in definitions are particularly unstructured.

let F = **unit** (u,∗) ⇒ 0
| **pair** ((p,q),(b,c)) ⇒ (size p)+(size b)
| **struc** (s,K x) ⇒ size x) ;

F : $\bigcap\gamma{:}D.\ \gamma\times\gamma\to Nat$

and F : $\forall\alpha{:}D,\beta{:}D.\ \bigcap\gamma{:}D(\alpha\times\beta)\times\gamma\to Nat$

Note that the type variables α and β in the second typing arise in the usual parametric fashion by virtue of the definitional form "**let** F = …"; we have emphasized this fact by giving the explicitly-quantified type schemes which would be assumed for these defined constants within the scope of the definition.

These two type schemes are incomparable in that there are instantiations of either which are not instantiations of the other:

$(\bigcap\gamma{:}D.\ \gamma\times\gamma\to Nat) \preccurlyeq (\mathbf{S}_i\ \tau\times\mathbf{S}_i\ \tau\to Nat)$ for any $\tau{:}D$, $\mathbf{S}_i{:}T\to D$

$(\forall\alpha{:}D,\beta{:}D.\ \bigcap\gamma{:}D(\alpha\times\beta)\times\gamma\to Nat) \preccurlyeq$ $((\sigma\times\rho)\times Bool\to Nat)$ for any $\sigma,\ \rho{:}D$

Essentially the problem stems from the lack of structure in the patterns u and s: there is no way to statically determine whether these patterns participate in the overall type in a "clause-specific" way or in a uniform way. That is, the patterns u and s might correspond to an analyzed type (as the right parts of the pairs clearly do), represented by the analytically inferred variable γ, or they might correspond to an arbitrary but uniform type, represented by the parametrically inferred variable α. In any case, no single type can generalize both of these interpretations since in the latter case the types of each half of the pair are identified, whereas in the former they are not.

In the context of a practical implementation, these problems might be handled by issuing a warning or, more conservatively, a type error. If a warning were deemed appropriate, a clear choice could be made for a preferred type among the finite set of possible types. If an error were issued, information about the ambiguity could be used to suggest how the programmer might rewrite the offending code to use more specific patterns (the rewriting involved would generally be trivial).

In the next chapter we attempt to make the best of the system as it stands, i.e., essentially ignoring the problem of ambiguous typing. As we shall see, there are further difficulties associated with type automated inference in this context; these difficulties, along with the problem of ambiguous typing, will motivate an alternative formulation using (minimal) explicit type constraints, which is explored in Chapter 8.

Chapter 7
Type Inference and Functional-Variant Unification

In this chapter we explore one possible approach to the problem of type inference for the language of structural polymorphism described in Chapter 6. In particular, we define an algorithm which finds correct types for the fully-implicit raw expressions as they were originally defined. For various reasons, it happens that we cannot give a very satisfactory solution to this problem: this situation motivates our investigation in Chapter 8 of a variation on the basic language which requires that minimal explicit type constraints be supplied by the programmer.

One of the problems inherent in any approach to type inference for structural polymorphism is the presence of type-function variables in the language of types. As we described in the last chapter, this is problematic because it means that we must generalize from the first-order unification on which the Milner-style inference algorithm is based to a notion of higher-order unification. Unfortunately, there is no effective algorithm which will compute unifying substitutions for terms which contain arbitrary function variables, so we must be satisfied with either a certain amount of incompleteness in the results of unification or with the possibility of non-termination in the type inference algorithm.

Our approach to this problem is to prefer incompleteness to potential non-termination: although there is some precedent for accepting potentially non-terminating algorithms as compile-time type-checkers [Car86], such an arrangement breaks with traditional presumptions about the behavior of language processors. We therefore use a heuristic unification algorithm which does not find all possible unifiers or most general ones; rather, our extended notion of unification always terminates with failure or a single solution (it may, however, attempt several solutions before failing).

Although vexing, the problem of extended unification is at least familiar: any number of implementations of type-theoretic formalisms employ partial implementations of higher-order unification to practical effect: see, for example [Pau90, NaMi88]. A less familiar problem, and one which is more specific to our type system, arises from our need to determine an appropriate type function for structurally-polymorphic definitions. Essentially, we must unify the types inferred for the various clauses of the definition, not just in the usual sense of finding a unifying substitution, but in the sense of finding both a substitution *and* a certain generalizing type function. More specifically, the type function must be such that, under the substitution returned, it specializes to each clause type when instantiated with the type pattern corresponding to that clause.

We know already from the last chapter that it is possible to derive incomparable types for a single structurally-polymorphic definition: this fact should already suggest that the type inference problem will not admit unique solutions. We would nevertheless prefer a better understanding of the underlying unification problem; to this end we propose a natural generalization of the problem, called *functional-variant unification* (FVU), and investigate it separately from the context of the type inference problem which motivates it.

Unfortunately, our investigation shows that functional-variant unification is ambiguous, potentially infinitary and probably undecidable. We have developed an algorithm which finds complete sets of solutions for instances of functional-variant unification, but have been so far unable to solve the problem of guaranteeing termination of this algorithm (the difficulty lies in the need for a more sophisticated form of occurs check). On the other hand, we have also been unable to show that the problem is undecidable, so there is still some hope that a complete, terminating algorithm might be found. If such an algorithm can be found, however, we would still have to decide on a strategy for choosing one from among several possible results that it would return, or, alternatively, to reject ambiguous cases as ill-formed.

One final positive prospect arises from recognizing that the type inference problem for structural polymorphism may be easier to solve than the general problem of functional-variant unification. That is, it may be that the special circumstances which surround the type inference problem may limit the FVU problem to a decidable subset of instances. In particular, we note that in the special context of type inference we always have a fixed set of type patterns (modulo the use of several patterns for several component types), and that these patterns have certain special properties, e.g., they are non-overlapping. We have not fully explored whether these features make the restricted problem more feasible than the general one.

Given our current understanding of functional-variant unification (or rather, our current lack of understanding), we expect that a more practical approach to structural polymorphism is to require that some explicit constraints be supplied by the programmer, an alternative which we explore more fully in Chapter 8. Thus the material in this chapter can be seen either as an end in itself, if one is willing to accept some of the more problematic aspects of fully-implicit type inference, or a as a motivation for the approach of Chapter 8, if one considers these problems to be unacceptable in a programming language application.

In Section 7.1 we sketch the automated type inference algorithm for the implicitly-typed language introduced in the last chapter; the development is based largely on Milner's algorithm, differing only in the extensions needed for pattern abstractions of various sorts. In Section 7.2 we give a description of the more general problem of functional-variant unification, i.e., for arbitrary terms rather than just for types. And finally, in Section 7.3 we present the algorithm which solves instances of the general problem and discuss the difficulties which must be resolved in developing a termination criterion.

7.1 Inferring types for fully-implicit terms

Recall from the last chapter that our formation rules for the language of structural polymorphism specify a relation between raw expressions, provided by the programmer, and decorated expressions, upon which reduction, compilation or other processing would be based. This represents a change from the usual Milner-style system in which the same sort of untyped expressions are used throughout; consequently, we need to change the general form of the type inference algorithm to allow for decorated expressions as part of the results. Thus calls to our algorithm $\mathcal{W}$ have the following general form (we retain Milner's notation $\mathcal{W}$):

$$\mathcal{W}(e, A) = (d, \tau, S)$$

By comparison with the properties of Milner's algorithm, we would like to devise our algorithm so that the following hold:

Target property (syntactic soundness):

For all raw expressions e, assumptions A, decorated expressions d and types τ, if $\mathcal{W}(e, A) = (d', \tau', S)$, then we can derive $SA \vdash e \mapsto d : \tau$ by the inference rules in Chapter 6.

Target property (syntactic completeness):

For all type substitutions S, raw expressions e, assumptions A, decorated expressions d and types τ, if $SA \vdash e \mapsto d : \tau$ is derivable from the inference rules in Chapter 6, then for some decorated expression d' and type τ', $\mathcal{W}(e, A) = (d', \tau', S')$; furthermore, d', τ' and S' are principal in the sense that there is a type substitution R such that $SA = RS'A$, $R(gen(\tau', S'A)) \sqsubseteq \tau$ and $Rd' \sqsubseteq d$.

Note that in stating completeness we must allow for the fact that type variables in the decorated term might need to be specialized by the substitution: this is in contrast to the situation with Milner-style polymorphism, where expressions never contain types or type variables as constituents (we have used the notation $d' \sqsubseteq d$ here in what we hope is an obvious sense). We will have to account for this aspect of the problem in our definition of $\mathcal{W}$ by applying intermediate substitutions to prior term results when computing types for terms with multiple constituents (for example, pairs and function applications).

From the results of last chapter on ambiguous typings and from the discussion above, it is clear that we must give up any hope of completeness as stated here (we still insist on a sound algorithm, of course: it may be acceptable that some meaningful programs are rejected by a compiler, but we would not want meaningless programs to pass compilation). Ignoring for the moment the problems associated with higher-order and functional-variant unification, we could approach the problem of ambiguity by allowing $\mathcal{W}$ to return multiple results; that is, we might use the following general form instead:

$$\mathcal{W}(e, A) = \{(d_1, \tau_1, S_1), \ldots, (d_n, \tau_n, S_n)\}$$

In this case we could re-state completeness as follows:

Target property (revised syntactic completeness):

For all type substitutions S, raw expressions e, assumptions A, decorated expressions d and types τ, if $A \vdash e \mapsto d : \tau$ is derivable from the inference rules in Chapter 6, then $\mathcal{W}(e, A)$ succeeds and, furthermore, for some $(d', \tau', S') \in \mathcal{W}(e, A)$, there is a type substitution R such that $SA = RS'A$, $R(gen(\tau', S'A)) \sqsubseteq \tau$ and $Rd' \sqsubseteq d$.

As we described in the last chapter, we must ensure that expressions with intersection types are decorated with explicit instantiations in the output. Although the inference rules left the particular point of instantiation undetermined, for the operational purposes of the type inference algorithm, we choose to instantiate intersection-typed expressions immediately upon encountering them. That is, we will examine the types to be returned from any call and, upon finding an intersection type outermost, we will use the following operation to force instantiations until the outermost form is no longer an intersection:

$\mathcal{D}(d, \tau, A) =$
  if $\tau = \bigcap \alpha{:}\delta.\tau'$ then $\mathcal{D}(d{\downarrow}\beta, \tau'[\beta/\alpha], A)$, fresh β
  else (d, τ, A)

Essentially, we will have no specific cases of the algorithm corresponding to the inference rules involving type instantiation, since these rules were not dependent on any specific form of expression; rather, we will achieve the same effect by dispersing this operation of decoration across all cases of the algorithm, thus accounting for any possible need for explicit type instantiation.

We begin our presentation of our algorithm $\mathcal{W}$ with the simple cases corresponding to the trivial constant (of unit type), variables and constructors; these cases are straightforward, involving no recursive calls and only empty substitutions in the results:

$\mathcal{W}(c, A) = (c, inst(C_{VC}(c)), [])$;
$\mathcal{W}(x, A) = \mathcal{D}(x, inst(A(x)), [])$;
$\mathcal{W}(\mathbf{K}_{nm}, A) =$
  let $\theta = nf(C_{VK}(n)(m)\ \alpha)$, fresh α
  return $(\mathbf{K}_{ij}, \theta \rightarrow Sn\ \alpha, [])$;
$\mathcal{W}(\mathbf{K}_{i\mu}, A) = (\mathbf{K}_{ij}, \mathbf{K}_{i\mu}\ \alpha \rightarrow \mathbf{S}_i\ \alpha, [])$, fresh α ;

The next simplest cases are those involving pairing and function application; note that here we must apply intermediate substitutions to both expressions and types returned in the results, since decorated expressions may also have occurrences of type variables.

$\mathcal{W}((e, e'), A) =$
  let $(d, \rho, S) = \mathcal{W}(e, A)$
  let $(d', \tau, T) = \mathcal{W}(e', SA)$
  return $\mathcal{D}((Td, d'), T\rho \times \tau, TS)$;
$\mathcal{W}(f\ e, A) =$
  let $(g, \rho, S) = \mathcal{W}(f, A)$
  let $(d, \tau, T) = \mathcal{W}(e, SA)$
  let $U = unify(T\rho, \tau \rightarrow \beta)$, fresh β
  return $\mathcal{D}(UTg\ Ud, U\beta, UTS)$;

We use here the meta-variables S, T and U to range over the usual type substitutions; we also break with the notation of previous chapters and write the application of a substitution to a term without brackets, and with the substitution to the left of the term; finally, we use juxtaposition of substitutions to represent their composition, which is defined otherwise as usual (these notational changes allow us to more closely follow traditional usage in this area of research).

The case for the recursion form is also straightforward, requiring only that we assume a fresh type variable for the recursion variable, just as in the original algorithm for Milner's system:

$\mathcal{W}(\mu x.e, A) =$
  let $(d, \tau, S) = \mathcal{W}(e, A + \{x{:}\alpha\})$, fresh α
  let $U = unify(S\alpha, \tau)$
  return $\mathcal{D}(Ud, U\tau, US)$;

For the case of let definitions, we depend on the operation of generalization to quantify out type variables in order to find an appropriate type scheme for the assumption on the let-defined "constant", just as in the original algorithm $\mathcal{W}$. Note, however, that we use here the more complex notion of

generalization introduced at the end of Section 6.3, which accounts for the fact that there may be unresolved variables in the type which should be intersection-bound rather than parametrically-quantified.

$\mathcal{W}$(**let** x=e **in** e′, A) =
  let (d, ρ, S) = $\mathcal{W}$(e, A)
  let (d′, τ, T) = $\mathcal{W}$(e′, SA+{x:gen(ρ,SA)})
  return $\mathcal{D}$(**let** x=Td **in** d′, τ, TS) ;

Value-pattern abstractions require a somewhat more complex and less familiar treatment than the previous forms, since we did not present Milner's original algorithm in the context of a pattern-based language. One complication that arises is the need for an auxiliary operation to find types and assumptions for patterns: we give a definition for this operation (notation: $\mathcal{V}(p)=(\tau,A)$) immediately following the specification of the pattern-abstraction case.

Another complication that arises for pattern-abstractions is more of a notational problem: it is convenient to loop through the various clauses of the abstraction, but we need some notation for maintaining lists of intermediate clauses. We use the notation "·" to represent the empty list of clauses and " ‖ " to concatenate a single clause onto the end of a list of clauses.

Apart from these complications, the basic ideas behind this case are the same as those for, e.g., the pair case above, although they are somewhat obscured by the machinery needed for looping through the clauses and for iteratively building up the intermediate substitutions:

$\mathcal{W}$($p_1 \Rightarrow e_1 \mid \ldots \mid p_n \Rightarrow e_n$), A) =
  $g_0 = \cdot$; $\sigma_0 = \beta$ (fresh β) ; $R_0 = Id$; $A_0 = A$
  for i = 1 to n:
    let $A_i = R_{i-1}A_{i-1}$
      $(q_i,\rho_i,B_i) = \mathcal{V}(p_i)$
      $(d_i,\tau_i,S_i) = \mathcal{W}(e_i,A_i+B_i)$
      $U_i = unify(S_i\rho_i \rightarrow \tau_i,\ S_i\sigma_{i-1})$
      $T_i = U_iS_i$
      $g_i = (T_ig_{i-1} \parallel T_iq_i \Rightarrow U_id_i)$
      $\sigma_i = T_i\sigma_{i-1}$
      $R_i = T_iR_{i-1}$
  return $\mathcal{D}(g_n, \sigma_n, R_n)$;

Note here that we use g to range over lists of clauses and that we implicitly extend the notion of substitution point-wise across the components of the list.

As we mentioned above, we have used here an auxiliary operation which will take as an argument a pattern, and return as a result a pair consisting of a type for the pattern and a set of assumptions for the pattern-bound variables which support this type. We define this operation, written $\mathcal{V}(p)$, as follows:

$\mathcal{V}(c) = (c, inst(C_{VC}(c)), \{\})$;
$\mathcal{V}(x) = (x, \alpha, \{x:\alpha\})$;
$\mathcal{V}(\mathbf{K}_{nm}\ p)$ =
  let $(p',\tau,A) = \mathcal{V}(p)$
  let $\theta = nf(C_{VK}(n)(m)\ \alpha)$, fresh α
  let $\upsilon = \theta \rightarrow \mathbf{S}_n\ \alpha$
  let $U = unify(\upsilon,\tau \rightarrow \beta)$, fresh β
  return $(\mathbf{K}_{nm}\ p', U\beta, UA)$;
$\mathcal{V}(\mathbf{K}_{i\mu}\ p)$ =
  let $(p',\tau,A) = \mathcal{V}(p)$
  let $U = unify(\mathbf{K}_{i\mu}\ \alpha \rightarrow \mathbf{S}_i\ \alpha, \tau \rightarrow \beta)$, fresh α, β
  return $(U(\mathbf{K}_{i\mu})\ p', U\beta, UA)$;
$\mathcal{V}(p,q)$ =
  let $(p',\tau,A) = \mathcal{V}(p)$
  let $(q',\upsilon,B) = \mathcal{V}(q)$
  return $((p',q'), (\tau\times\upsilon), A+B)$;

The specifics of this definition follow clearly from the definition of W for the corresponding forms of ground expressions.

The final case in the definition of algorithm $\mathcal{W}$ is the most crucial one, namely the one governing the structurally-polymorphic definition form. The overall structure of this case is similar to that for value-pattern abstractions, since both of these forms consist of multiple clauses composed of patterns and expressions. Unlike the value-pattern case, however, the patterns here are keywords, which must be translated into the corresponding type patterns: for this purpose we use the operation $\mathcal{TP}$ defined in Chapter 6. Note also that we must enforce here the usual free-variable conditions associated with polymorphic quantifications, although in this context the enforcement becomes a check for type variables free in substitutions.

$\mathcal{W}$((**spoly** x **as** $key_1\ e_1 \mid \cdots \mid key_n \Rightarrow e_n$), A)=
  check that $\{key_1,\ldots,key_n\}$ cover some kind δ, $\mathcal{O}(\delta)\leqslant 1$
  check that $\{key_1,\ldots,key_n\}$ are compatible
  $d_0 = \cdot$; $R_0 = Id$; $A_0 = A+\{x:\cap\alpha:\delta.\beta\}$, fresh β
  for i = 1 to n:
    let $A_i = S_{i-1}A_{i-1}$
      $(e_i',\tau_i,S_i) = \mathcal{W}(e_i,A_i)$
      $\rho_i = \mathcal{TP}(\delta, key_i)$
      $d_i = (S_id_{i-1} \mid \rho_i \Rrightarrow e_i')$
      $R_i = S_iR_{i-1}$
  let $(\varphi,U) = FVU\langle(\mathcal{E}(\rho_1),\tau_1), \ldots, (\mathcal{E}(\rho_n),\tau_n)\rangle$
  if $FV(\rho_1,\ldots,\rho_n)\subseteq dom(UR_n)$ then **fail**
  return
    either $(\mu x.\ d_n, \cap\alpha:D.\varphi(UR_n\beta), UR_n)$, for δ=D
      or $(\mu x.\ d_n, \cap\mathbf{S}_v:\kappa\rightarrow D.\varphi(UR_n\beta[\mathbf{S}_v/\alpha]), UR_n)$,
        for $\delta=\kappa\rightarrow D$;

Note that we use a fresh variable as the body of the type assumed for the recursion variable, expecting that this variable will be specialized by subsequent substitution.

The definition of the FVU function used in this last case of the W algorithm is presented in Section 7.3; we describe the general FVU problem (i.e., for arbitrary terms) in the next section.

7.2 The general functional-variant unification problem

In order to find types for type-pattern abstractions which will work according to the inference rules given in Chapter 6, we must essentially find a type functional τ which will unify

all of the types of the individual clauses, *up to the* parameterization of this functional by the type (function) variable φ. Consider, for example, that we are trying to define functions like the following:

$$\text{equal} : \bigcap\alpha{:}\mathbf{D}.\ \alpha\times\alpha\to\text{Bool}$$

$$\text{(core of) map} : \bigcap\mathbf{S}_\nu{:}\mathbf{T}\to\mathbf{D}.\ \mathbf{S}_\nu\ \alpha\to\mathbf{S}_\nu\ \beta$$

Then the relevant type (functionals) we must discover are, for equal,

$$\tau\ =\ \lambda\alpha.\ \alpha\times\alpha\to\text{Bool}$$

given clause types:

$$\begin{array}{ccccc} 1 & \times & 1 & \to & \text{Bool} \\ (\beta\times\gamma) & \times & (\beta\times\gamma) & \to & \text{Bool} \\ \Sigma\ \delta & \times & \Sigma\ \delta & \to & \text{Bool} \end{array}$$

and for the core of map,

$$\tau\ =\ \lambda\varphi.\ \varphi\ \alpha\to\varphi\ \beta$$

given clause types:

$$\begin{array}{ccc} 1 & \to & 1 \\ (\psi\ \alpha\ \times\ \zeta\ \alpha) & \to & (\psi\ \beta\ \times\ \zeta\ \beta) \\ \Sigma\ (\chi\ \alpha) & \to & \Sigma\ (\chi\ \beta) \\ \alpha & \to & \beta \end{array}$$

This would be difficult or impossible if we did not know the values of the type function φ for each individual clause, but in fact we do have access to this information, as it is supplied through the $\mathcal{TP}$ operation. So the problem reduces to the following form: given a list of pairs, each consisting of a type and a type function, find some type *functional* which "unifies" all the pairs simultaneously, in the sense that for each pair, the type half is equal to the application of the result functional to the corresponding type-function half.

Of course, based on our discussion of ambiguity above, we know that there may be more than one correct result. Furthermore, in the process of finding a type-functional result, we may discover information about type variables through the usual means of unification, so that we actually will need to return a set of pairs, each pair consisting of a substitution and a type functional; this sketch leads to the general statement of the problem of functional-variant unification given below.

In particular, we propose that the specific problem of finding pairs of type functionals and substitutions from pairs of types and type functions is a special case of a more general unification problem which can be stated for arbitrary terms and functions on terms: in this section we study the general problem more thoroughly, in order to give convincing evidence that it does not provide a useful basis for automated type inference.

Let us first fix some notation and terminology regarding arbitrary terms and term functions: we consider terms consisting of variables and of function symbols of fixed arity, applied to a correct numbers of sub-terms; we often use infix syntax for binary function symbols to improve readability. So, for example, $1*x$ and $y+2$ are terms, where $*$ and $+$ are binary function symbols, 1 and 2 are nullary function symbols and x and y are variables.

We also consider term functions, which introduce a notion of bound variable. For our purposes these functions are syntactic entities, i.e., just a bound variable and a term containing instances of that bound variable—we will write, e.g., $f = (\lambda x.1*x)$ for a term function which, when applied to the term $y+2$ yields the term $1*(y+2)$. In general we will allow functions of several term arguments with the usual syntax. Application of a function to term arguments results in another term via beta reduction. We also consider second-order term functions, which may require a couple of reductions to resolve, but everything we work with is typed, so there are no non-terminating reductions, and we always end up with "zero-order" (ground) terms for the purposes of, e.g., equality comparisons. An example of a second-order term function is $h = (\lambda f.f(y+2))$; given this we have:

$$\begin{aligned} h\ (\lambda x.1*x) &= (\lambda f.f(y+2))(\lambda x.1*x) \\ &= (\lambda x.1*x)(y+2) \\ &= 1*(y+2) \end{aligned}$$

Given this notation, we can state the following:

Definition (FVU problem instance):

An instance of the (first-order) FVU problem is a finite set of pairs $(t_1,f_1),\ldots,(t_n,f_n)$; the first part of each pair is a term, the second part a p-ary function on terms for some integer $p>0$ common to all the functions.

Definition (FVU solution):

A solution to an FVU problem instance $(t_1,f_1),\ldots,(t_n,f_n)$ is a pair (S,h) where S is a substitution of terms for term variables and h is a second-order function on terms (appropriate to take an f_i as argument), such that the following holds for each i between 1 and n:

$$S\ t_i\ =\ h\ f_i$$

In general there is more than one solution for a given FVU problem instance, even if we take a standard ordering on solutions and consider only minimal ones. FVU problems of other orders can be considered, but we are only interested in this (first-order) case and the zero-order case where the f_i are not even functions but just plain terms (then h is a first-order function on terms).

A brief aside: if the second-order nature of h causes distress, the form of the solution can be re-cast to consist of a first-order function and a "matrix" of terms (plus the substitution S, as usual). Consider that h must be of the following form:

$$h = \lambda f.\ \text{—}\ (f\ (s_{11},\ldots,s_{1p}))\ \text{—}\ (f\ (s_{m1},\ldots,s_{mp}))\ \text{—}$$

i.e., there must be some number of occurrences, say m, of the bound variable f, each applied to p argument terms. Now instead of the second-order term h, we just have to find the "outer" context:

$$h' = \lambda t.\ \text{—}\ t\ \text{—}\ t\ \text{—}$$

and the argument "matrix" $((s_{11},\ldots,s_{1p}),\ldots,(s_{m1},\ldots,s_{mp}))$ so that

$$h\ f = h'\ (f\ (s_{11},\ldots,s_{1p}),\ldots,f\ (s_{m1},\ldots,s_{mp}))$$

This completes the aside.

The basic idea behind an FVU solution is this: to the extent that h does not "use" its (function) argument we really have just unification. That is, if h is a constant function of the form $h=(\lambda f.t)$ where f does not occur free in t, then we have

simply:

$$S\ t_i = h\ f_i = (\lambda f.t)\ f_i = t$$
$$\text{i.e.,}\ S\ t_i = t$$

so that the f_i do not come into play at all and S just unifies all the t_i to the single term t (or fails if this is not possible).

If, on the other hand, h *does* make use of its argument, then we seek to find the structure of each f_i in the corresponding t_i. For example, if f_1 is the constant function $(\lambda x.3)$ then t_1 should have the form 3, and if f_2 is $(\lambda x.x*2)$, t_2 should have the form of a product $(t*2)$ for some term t.

Thus we are trying to discover the body of a common term function h, with parameter f, so that each term t_i matches the constructed h when it is applied to the corresponding function f_i. We have two possibilities:

(1) if the t_i are all compatible, sharing some common outer form, then we can express this by putting this common outer form into the body of h, then "recursing" to find inner-more form.

(2) we also can try to match the outer form of each t_i with that of the corresponding f_i; if they *all* match (pair-wise), we put an occurrence of the function variable f into the body of h, then "recurse" to find the arguments of f in the body of h.

In a typical solution these two possibilities will be mixed together at different levels: the outer form of all terms might be common (as in plain unification), the next "level" might differ *between* terms, but still match the corresponding functions, etc. In addition, the substitution S may be used to give some necessary structure to a variable subterm of some of the t_i's in order to make this work out. The potential multiplicity of solutions (even up to the natural ordering) alluded to above arises because of the difference between these two possible aspects of a solution: in certain instances which lack enough structure (i.e., have enough variables), we can either use the substitution S to force common form on the terms for a unification-like solution or to force similarity "component-wise" with the functions f_i. If the functions f_i are different enough, these two forms of solution will be incomparable. This aspect of the problem is then reflected in the fact that the type inference rules for structural polymorphism do not admit standard principal types, but rather only "minimal sets" of types.

7.3 A semi-algorithm for functional-variant unification

In this section we develop and describe an algorithm for finding solutions to FVU problem instances. As we mentioned above, the algorithm seems to be complete (by a rather simple argument about the possible forms of solutions), but we have not been able to guarantee termination. More specifically, we have need for an enhanced form of occurs check, one which will allow to pass some patterns of variable occurrences that the usual occurs check would not, and yet which will fail for other, more ill-behaved patterns of occurrences.

As a first step toward the development of the algorithm, we review in the next few paragraphs some of the complications which arise immediately as a result of various aspects of the problem already discussed above.

As described in the previous section, a given problem instance may admit two basic forms of solution. Our approach is to look for both kinds of solutions, i.e., those based on unifying the terms in the usual sense, and those based on unifying the terms through their functional counterparts. Since we can have both kinds of solutions intermixed at different levels, we actually need to perform a "one-step", outermost unification for each of the two kinds of solutions, followed by recursive calls to solve sub-terms, and then combine together the results of each kind of solution. The base case in this recursion comes when we all of the terms considered have been reduced to just variables.

Recall also from the problem specification above that we need to return both a term function and a substitution for each solution (the substitution represents the usual information about instantiation of type variables that would be gathered in the course of standard unification). We also know, because of the ambiguous nature of the example in Section 6.4, that we must return sets of solutions in general. Note that a given substitution will be "local", in the sense that it is applicable only to the term function with which it is paired.

Another complication is that we will need to make recursive calls for all the argument terms of an n-ary function symbol, after making a top-level call for the function symbol itself. Since these recursive calls return sets of solutions, and since each solution also includes a substitution, we have to stitch the results for the argument positions together in a fairly complicated way. On the one hand, we have to return all possible combinations of results, i.e., one from each set for the corresponding argument positions; on the other hand, the substitutions from each position must be propagated forward to the succeeding argument positions. It is not even immediately clear that the solutions arrived at in this way are independent of the order in which we examine the individual argument positions, so we must argue this point separately.

Another potential complication is that we only want to return minimal solutions (in fact, it will turn out that in some cases, infinite sets of solutions will be related as a chain by the ordering; typically these cases involve n-fold applications of the function parameter for any n). We want our algorithm to be complete, but we would also prefer to avoid any redundancy.

By way of some examples, consider the following FVU problem instances, represented as calls to an FVU algorithm, where we assume constant function symbols s, g, 1, 2, + and * of appropriate rank:

$$FVU\langle\ (g(x,b+x),\ \lambda a.a+1),\ (g(y,2*y),\ \lambda a.2*a)\ \rangle$$
$$= (\lambda f.g(1,f\ 1),\ [1/b,\ 1/x,\ 1/y])$$
$$FVU\langle\ (x,\ \lambda a.s\ a),\ (y,\ \lambda a.s(s\ a))\ \rangle$$
$$= (\lambda f.z,\ [z/x,\ z/y]);\ (\lambda f.f\ w,\ [s\ w/x,\ s(s\ w)/y])$$

We now proceed to a more formal description of our algorithm for functional-variant unification. It is convenient to define first a form of call for the top-level which eliminates the parameterization of each recursive call by the same tuple of type functions, since these will not change through-out the course of the computation; we define:

$$FVU((t_1,f_1),\ \ldots,\ (t_n,f_n)) = \{(\lambda f.t,S)\ |$$
$$(t,S)\in FVU'(t_1,\ldots,t_n)\}$$

We can then define the recursive core of the algorithm, FVU', according to cases as follows:

$$FVU'(t_1,\ldots,t_n) =$$

①. *[base case: all variables]*

if each $t_i \equiv v_i$ for some variables v_i
(not necessarily distinct),

return:
{(x, [...,v_i:=x,...])}
∪ (**if** no clash **then** {(f y, [...,v_i:=f_i y,...])} **else** {})
where x, y are fresh and there is a clash if,
for some i≠j, $v_i \equiv v_j$ and f_i y≠f_j y.

If not all $t_i \equiv v_i$, we return the union of ② and ③ below:

②. *[common form]*
if occurCheck(t_1,...,t_n) then return {} else
let (g,m,S) = stripConst(t_1,...,t_n)
(if this fails, return {} for part ②) ;
return
multiFVU({(g(),S)}; (t_1/1,...,t_n/1),...,(t_1/m,...,t_n/m))

③. *[variant form]*
if loopCheck??(t_1,...,t_n) then return {} else
let (S; (x_{11},...,x_{1n}),...,(x_{p1},...,x_{pn})) =
stripVariant(t_1,...,t_n)
(if this fails, return {} for part ③) ;
return
multiFVU({(f(),S)}; (x_{11},...,x_{1n}),...,(x_{p1},...,x_{pn}))

The use of the notation loopCheck?? … in the final clause above is not intended to refer to an actual function, but is rather a placeholder which will be useful in discussing the termination problems associated with the algorithm (see below).

The various other auxiliary functions used above are defined as follows (assuming global f and f_i as above):

multiFVU($\mathcal{R}$, (t_{11},...,t_{1n}),...,(t_{q1},...,t_{qn})) =
[where $\mathcal{R}$ is a solution set]
let $\mathcal{R}_0$ = $\mathcal{R}$;
for i=1 to q:
for each (r_k,R_k) ∈ $\mathcal{R}_{i-1}$:
$\mathcal{R}_{ik}$ = {extend (r_k,R_k) (u,U) |
(u,U) ∈ FVU′(R_k t_{i1}, ..., R_k t_{in})} ;
$\mathcal{R}_i$ = $\bigcup_k \mathcal{R}_{ik}$;
return $\mathcal{R}_n$;

extend(g(r_1,...,r_m),R) (u,U) = (g(Ur_1,...,Ur_m,u),U∘R))

stripConst(t_1,...,t_n) =
if all t_i are of the form v_i or g(t_{i1},...,t_{im}) then
then return (g,m,S)
where S = [..., v_i:=g(x_{i1},...,x_{im}), ...],
for fresh x_{ij} and relevant v_i ;
else **fail** ;

stripVariant(t_1,...,t_n) =
set S_0 = Id and
choose fresh {x_{11},...,x_{1n},...,x_{p1},...,x_{pn}} ;
for i=1 to n:
R_i = unify($S_{i-1}t_i$, f_i (x_{i1},...,x_{in})),
where the free variables in f_i are frozen
S_i = $R_i \circ S_{i-1}$;
return (S_n; (x_{11},...,x_{1n}),...,(x_{p1},...,x_{pn})) ;

We now sketch an argument for completeness of the FVU algorithm described above relative to the specification from the previous section. Consider a solution to an FVU instance as given above: the function h is of the form λf. u[f] for some term f with possible occurrences of f. Given the possibility of occurrences of f, u[f] might have one of three forms:

- it might be a variable x;
- it might be an m-ary function symbol g applied to m sub-terms;
- it might be the function variable f applied to p sub-terms.

We consider each of these cases separately below.

Say that u[f] = x for some variable x. Then we have that h f_i = (λf. x) f_i = x for all i. Since the application of the substitution S (or any substitution) can only make terms more specific, and since the variable x is minimal in this ordering, we must have that all the terms t_i are of the form v_i for some variables v_1,...,v_n (not necessarily distinct) and S v_i = x for all i. Thus either v_i = x or S⊇{v_i:=x} for all i such that $1 \leqslant i \leqslant n$.

Say that u[f] = g(u_1[f],...,u_m[f]) for some m-ary function symbol g and m subterms u_1,...,u_m, each with possibly having occurrences of f. Then we have that h f_i = (λf. g(u_1[f],...,u_m[f])) f_i = g(u_1[f_i],...,u_m[f_i]) for all i. Thus we must have for each i that St_i = g(u_1[f_i],...,u_m[f_i]); this will be the case if either $t_i \equiv g(t_1,...,t_m)$ and St_j = u_j[f_i] or if $t_i \equiv v_i$ and S⊇{v_i:=g(u_1[f_i],...,u_m[f_i])}.

Say that u[f] = f(u_1[f],...,u_p[f]) for p subterms u_1,...,u_p, each with possibly having occurrences of f. Then we have that h f_i = (λf. f(u_1[f],...,u_p[f])) f_i = f_i(u_1[f_i],...,u_p[f_i]) for all i. Thus we must have for each i that St_i = f(u_1[f_i],...,u_p[f_i]); this will be the case if either $t_i \equiv f_i(t_1,...,t_m)$ and St_j = u_j[f_i] or if $t_i \equiv v_i$ and S⊇{v_i:=f_i(u_1[f_i],...,u_m[f_i])}.

Despite this argument for completeness, our algorithm for finding solutions to FVU problems will fail to terminate in some instances; consider for example a problem instance corresponding to the following call:

FVU((x, λa.s a), (s x, λa.a)) = (no finite solutions)

(it may be helpful here to think of s here as the successor function). It is tempting to suggest that a standard occurs check would allow us to halt upon noticing that the left-hand term argument, which is just an occurrence of the variable x, occurs in a "deeper" position in the right-hand term argument.

But the use of a standard occurs check here would result in incompleteness, as evidenced by the following very similar call:

FVU((x, λa.a), (s x, λa.s a)) = { (λf.f x,[]) }

The problem instance represented by the second call has a finite solution *and* has the very same term arguments as the instance represented by the first call. Furthermore, the form of the solution corresponds to the case labeled ③ in the definition of the algorithm we have given above. We require a more sophisticated occurs check than just the traditional one, since we would need to reject the first instance but accept the second, so we cannot simply use a call of occurCheck in place of the placeholder loopCheck?? above.

Some perspective on the first instance can be provided by

recognizing that that there actually *is* an "infinite solution", in some sense, of the form $(\lambda f.f\ (s^{\omega}\ z), [s^{\omega}\ z/x])$. That is,

$$x\ [s^{\omega}\ z/x] = (\lambda f.f\ (s^{\omega}\ z))\ (\lambda a.s\ a) = s^{\omega}\ z$$

$$s\ x\ [s^{\omega}\ z/x] = (\lambda f.f\ (s^{\omega}\ z))\ (\lambda a.a) = s^{\omega}\ z$$

since $s^{\omega}\ z = s^{\omega+1}\ z$.

We have as yet found no appropriate predicate to fill the place of the call to loopCheck?? used in the statement of the algorithm above. From the example we have just given it seems clear that something like an occurs check is needed, but also that the standard form of occurs check will not suffice.

On the other hand, we have also as yet been unable to prove that the general FVU problem is undecidable: we have made some headway using an approach by which arithmetic problems are encoded as FVU instances, but have been unable so far to provide a powerful enough encoding.

It is worth noting, as we did earlier in the chapter, that the particular context of the type inference problem may allow us to find complete solutions effectively, even if the general FVU problem turns out to be undecidable. Consider for example that the type-function patterns corresponding to the right-hand parts of an FVU instance have a very special form: they are always non-overlapping and jointly cover the possible forms of data types. Just the fact that they constitute a fixed set of functions, up to the variable arity of the number of their type components, may provide us with enough structure to find solutions when we could not do so in all generality.

Chapter 8
Type Inference with Minimal Explicit Constraints

In this chapter we describe a variation of structural polymorphism which requires that minimal explicit type constraints be given by the programmer as part of the code. These constraints are "minimal" in the sense that they are only required for uses of the new structurally-polymorphic definition form: none of the traditional expression forms require any explicit constraints.

The language used in this chapter is otherwise the same as the one defined in Chapter 6 and used in Chapter 7 for the system with full type inference. The primary advantage of and motivation for the current system is that it avoids the need for the ill-behaved functional-variant unification algorithm. That is, with an explicit constraint provided by the user, we can check types for structurally polymorphic definitions without the need to simultaneously infer a type function which generalizes the types of the various clauses involved.

The inference algorithm described in this chapter nevertheless still depends on some extensions to first-order unification: these are required since our language of types still includes variables ranging over type functions. Despite the potential problems associated with higher-order unification, our implementation of the system described here suggests that the extended unification we use poses no problems in practical programming situations.

In section 8.1 below we describe the addition of explicit user-supplied type constraints for structurally-polymorphic definitions and the modifications to the type inference algorithm which take advantage of this extra information. We report experimental results of an implementation of this type system and inference algorithm in Section 8.2: the bulk of the section consists of actual output from the algorithm, showing user-input raw expressions and the corresponding decorated expressions and types which are output by the algorithm.

We do not discuss the details of reduction for decorated terms separately, since they are essentially the same as for a fragment of the explicitly-typed calculus of Chapters 2 through 5. One concern we should address in this regard is the fact that we must include a representation for types and perform pattern-matching on this representation at run-time. Fortunately, the restrictions we imposed on structural polymorphism in Chapter 6 allow us to minimize the need for run-time types, so that we require only information about value-constructor arguments, and not about the types of arbitrary values, even those which participate in structurally-polymorphic computations. All other type information is provided explicitly in the decorations introduced explicitly during the course of type inference.

8.1 Inferring types from minimal explicit constraints

In the last chapter we examined in detail the problems inherent in inferring types for fully-implicit structural polymorphism: a lack of principal types, the need for heuristic extensions which approximate higher-order unification, and the apparent need for an effective algorithm for functional-variant unification. Although we might propose various restrictions and hedges which would make such a system feasible, the combination of them would amount to a significantly change relative to the standard context of implicit type inference systems for first-order polymorphism. In order to offer some reasonable alternative, we describe in this section a variation of the system from the last chapter which requires marginally greater effort on the part of programmers, while at the same time making the problem of type inference significantly easier and better-behaved from the point of view of the compiler.

More specifically, we add a single explicit type-constraint to the raw expression form used to define structurally-polymorphic values, as follows:

$$e ::= \cdots \mid \textbf{spoly}\ x{:}\bigcap\alpha{:}\delta.\tau\ \textbf{as}\ (\text{key}\ e)^+$$

Note that none of the other raw or decorated expression forms change: in fact, the type inference rules and type inference algorithm will remain the same for all but this case.

The idea behind the extension is that it will allow us to more easily infer types for spoly definition forms, since we have what amounts to an appropriate type function provided by the user: we thus need not infer such a function using functional-variant unification.

With regard to formation, we use two new rules to govern the typing of the modified form of definitions, just as we used two rules in the fully-implicit version, one for the base case of a data type and one for the case of first-order functional kind. The rule for the data type case is as follows:

$$\frac{\begin{array}{c}\{\text{key}_1, \ldots, \text{key}_n\}\ \text{compatible}\\ \{\text{key}_1, \ldots, \text{key}_n\} \succ \delta\\ (\{\alpha\} \cup \bigcup_i \text{FV}(\rho_i)_T) \cong \text{FV}(A)_T\\ \mathcal{TP}(D, \text{key}_i) = \rho_i \quad (1 \leqslant i \leqslant n)\\ A + \mathcal{A}(\rho_i) + \{x{:}\bigcap\alpha{:}D.\tau\} \vdash e_i\\ \mapsto d_i{:}\tau[\rho_i/\alpha] \quad (1 \leqslant i \leqslant n)\end{array}}{\begin{array}{c}A \vdash (\textbf{spoly}\ x{:}\bigcap\alpha{:}D.\tau\ \textbf{as}\ (\cdots \mid \text{key}_i\ e_i \mid \cdots))\\ \mapsto (\mu x.\cdots \mid \rho_i \Rightarrow\!\!\!> d_i \mid \cdots) : \bigcap\alpha{:}D.\tau\end{array}}$$

The rule for the first-order functional case is:

$$\frac{\begin{array}{c}\{\text{key}_1, \ldots, \text{key}_n\}\ \text{compatible}\\ \{\text{key}_1, \ldots, \text{key}_n\} \succ \kappa \to D\\ (\{\alpha\} \cup \bigcup_i \text{FV}(\rho_i)_T) \cong \text{FV}(A)_T\\ \mathcal{TP}(\kappa \to D, \text{key}_i) = \rho_i \quad (1 \leqslant i \leqslant n)\\ A + \mathcal{A}(\rho_i) + \{x{:}\bigcap\alpha{:}\kappa \to D.\tau\} \vdash e_i\\ \mapsto d_i{:}\tau[\rho_i/\alpha] \quad (1 \leqslant i \leqslant n)\end{array}}{\begin{array}{c}A \vdash (\textbf{spoly}\ x{:}\bigcap\varphi{:}\kappa \to D.\tau\ \textbf{as}\ (\cdots \mid \text{key}_i\ e_i \mid \cdots))\\ \mapsto (\mu x.\cdots \mid \rho_i \Rightarrow\!\!\!> d_i \mid \cdots) : \bigcap\mathbf{S}_v{:}\kappa \to D.\tau\end{array}}$$

These rules are almost the same as the corresponding ones for the fully implicit version, but with the change that the type used in the assumption on the recursion variable x is part of the term input by the programmer; that is, during type inference we will no longer have to simultaneously derive an appropriate type function while also checking the types of the individual clauses.

Turning now to the problem of type inference, we modify the two corresponding cases of algorithm $\mathcal{W}$ as follows:

$\mathcal{W}$((**spoly** x : $\bigcap\alpha{:}D.\tau$ **as** $key_1\ e_1 \mid \cdots \mid key_n \Rightarrow e_n$), A) =
  check that $\{key_1,\ldots,key_n\}$ cover D
  check that $\{key_1,\ldots,key_n\}$ are compatible
  $d_0 = \cdot$; $R_0 = Id$; $A_0 = A+\{x{:}\bigcap\alpha{:}D.\tau\}$
  for i = 1 to n:
    let $A_i = S_{i-1}A_{i-1}$
      $(e_i',\tau_i,S_i) = \mathcal{W}(e_i,A_i)$
      $\rho_i = \mathcal{TP}(D, key_i)$
      $U_i = unify(\tau_i, S_i\tau\ [\rho_i/\alpha])$
      if $FV(\rho_i)\subseteq dom(U_i)$ then **fail**
      $T_i = U_iS_i$
      $d_i = (T_id_{i-1} \mid \rho_i \Rrightarrow U_ie_i')$
      $R_i = T_iR_{i-1}$
  return $(\mu x.\ d_n, \bigcap\alpha{:}D.R_n\tau, R_n)$;

$\mathcal{W}$((**spoly** x : $\bigcap\varphi{:}\kappa\rightarrow D.\tau$ **as** $key_1\ e_1 \mid \cdots \mid key_n \Rightarrow e_n$), A) =
  check that $\{key_1,\ldots,key_n\}$ cover $\kappa\rightarrow D$
  check that $\{key_1,\ldots,key_n\}$ are compatible
  $d_0 = \cdot$; $R_0 = Id$; $A_0 = A+\{x{:}\bigcap\varphi{:}\kappa\rightarrow D.\tau\}$
  for i = 1 to n:
    let $A_i = S_{i-1}A_{i-1}$
      $(e_i',\tau_i,S_i) = \mathcal{W}(e_i,A_i)$
      $\rho_i = \mathcal{TP}(\kappa\rightarrow D, key_i)$
      $U_i = unify(\tau_i, S_i\tau\ [\rho_i/\varphi])$
      if $FV(\rho_i)\subseteq dom(U_i)$ then **fail**
      $T_i = U_iS_i$
      $d_i = (T_id_{i-1} \mid \rho_i \Rrightarrow U_ie_i')$
      $R_i = T_iR_{i-1}$
  return $(\mu x.\ d_n, \bigcap\mathbf{S}_v{:}\kappa\rightarrow D.R_n\tau, R_n)$;

As for the fully-implicit language, the only significant difference between the two cases is in the forms of the final type returned. The cases are otherwise even simpler than the corresponding ones from the definition of the algorithm for the fully-implicit language, since we have no requirement to infer an appropriate type function to generalize the several clause types.

8.2 Experimental results

The following examples were provided by an implementation of the type-checking algorithm $\mathcal{W}$ described above. The input to the algorithm includes a raw expression and a set of assumptions for free variables; the output echoes the input and also shows the corresponding decorated term and type, if successful, using the following format:

A

$\vdash$ e

$\mapsto$ d

: τ

(A and e here would be input by the user, and d and τ would output by the program, which would also echo the user input).

The output below was generated by the program, complete with formatting and including the use of various typefaces, by way of an RTF specification output facility; the only changes we have made for this document are minor cosmetic changes to line-spacing and indentation, and a change to the symbol "$\mapsto$" for indicating the decoration judgement.

Our current implementation of algorithm $\mathcal{W}$ suffers from a number of minor shortcomings: it unduly restricts data structures to just kind $T\rightarrow D$, i.e., to a single component type parameter; it uses constructor variables in the style of Chapter 3, i.e., without the indexed constants introduced in Chapter 4; and it does not implement run-time reduction (e.g., type-pattern matching), but rather only type inference.

The standard map functional for lists

This is the standard definition of map for lists, using only the usual Milner-style polymorphism. Since there is no use of structural polymorphism, there should be no explicit instantiations in the decorated form of the expression.

⟨ {} , {} , {} , {} , {}
  ⟩

$\vdash$ μm. f ⇒
  (nil ∗ ⇒ nil ∗
    | cons(h,t) ⇒ cons(f h, m f t))

$\mapsto$ μm. f ⇒
  (nil ∗ ⇒ nil ∗
    | cons(h,t) ⇒ cons(f h, m f t))

: $(\varepsilon\rightarrow\delta)\rightarrow List\ \varepsilon\rightarrow List\ \delta$

Note that type variables occur free in the resulting type, as in a Standard ML type report, indicating Milner-style polymorphism through implicit quantification. Note also that nil and other nullary constructors are construed as functions of the trivial type, and so must be applied to an empty argument.

A structurally polymorphic map functional

This is a definition for a structurally polymorphic map functional which works for arbitrary data structures rather than

just for lists, as with the standard map above. We show the function applied to two data structures of different structural types in order to illustrate its generic nature.

```
⟨ {} , {} , {} ,
  { even: Nat→Bool, odd: Nat→Bool, alist: List Nat,
      atree: Tree Nat } ,
  {}⟩

⊢ let map = f ⇒
    spoly m: ⋂φ:T→D.φ α→φ β
    as ( unit ∗ ⇒ ∗
       | pair (y,z) ⇒ (m y, m z)
       | struc K x ⇒ K(m x)
       | comp#1 a ⇒ f a )
  in (map even alist, map odd atree)

↦ let map = f ⇒ μm.
    ( (λε.1) ⇒⇒ ∗ ⇒ ∗
    | (λε.ψ ε×ζ ε) ⇒⇒ (y,z) ⇒ (m↓ψ y, m↓ζ z)
    | (λε.Σ(θ ε))⇒⇒(K:χ↦Σ) x⇒
        K(m↓(λδ.χ(θ δ)) x)
    | (λε.ε) ⇒⇒ a ⇒ f a )
  in (map even↓List alist, map odd↓Tree atree)

: List Bool×Tree Bool
```

We recall that the intersection notation used in the explicit constraint on the variable m in the definition is meant to distinguish structural polymorphism from the usual Milner-style polymorphism.

Count: a generalization of length for lists

This definition yields a structurally polymorphic function called count, which generalizes the length function for lists: given any data structure, count returns the number of components contained in it.

```
⟨ {} , {} , {} ,
  { even: Nat→Bool, odd: Nat→Bool, alist: List Nat,
  atree: Tree Nat } ,
  {}⟩

⊢ let count =
    spoly c: ⋂φ:T→D.φ α→Nat
    as ( unit ∗ ⇒ zero ∗
       | pair (y,z) ⇒ (c y) + (c z)
       | struc K x ⇒ succ(c x)
       | comp#1 a ⇒ succ(zero ∗) )
  in (count alist, count atree)

↦ let count = μc.
    ( (λε.1) ⇒⇒ ∗ ⇒ zero ∗
    | (λε.ψ ε×ζ ε) ⇒⇒ (y,z) ⇒ (c↓ψ y) + (c↓ζ z)
    | (λε.Σ(θ ε))⇒⇒(K:χ↦Σ) x⇒succ(c↓(λδ.χ(θ δ)) x)
    | (λε.ε) ⇒⇒ a ⇒ succ(zero ∗) )
  in (count↓List alist, count↓Tree atree)

: Nat×Nat
```

Note that the type instantiations in the outer let-body are made to the predefined type structures List and Tree, whereas instantiations internal to the definition are made to more general type functions; this reflects a limitation on the type inference process necessary to make the inference of instantiations unambiguous.

A general equality predicate

The natural notion of equality between data values used in Standard ML and other languages can be typed within a Milner-style polymorphic system, but it cannot be defined in those languages. Using the case-based definitions of structural polymorphism, we can give an explicit definition of equality which also provides a more detailed analysis of the (necessary) restriction to non-functional arguments.

```
⟨ {} , {} , {} ,
  { even: Nat→Bool, odd: Nat→Bool, alist: List Nat,
  atree: Tree Nat } ,
  {}⟩

⊢ let equal =
    spoly e: ⋂α:D.(α×α)→Bool
    as ( unit (∗,∗) ⇒ true ∗
       | pair ((y1,z1), (y2,z2)) ⇒ (e(y1,y2)) & (e(z1,z2))
       | struc
         ( (K x, K y) ⇒ e(x,y)
         | (K x, H y) ⇒ false ∗ ) )
  in (equal(alist,alist), equal(atree,atree))

↦ let equal = μe.
    ( 1 ⇒⇒ (∗,∗) ⇒ true ∗
    | ε×δ ⇒⇒ ((y1,z1), (y2,z2)) ⇒
        (e↓ε(y1,y2)) & (e↓δ(z1,z2))
    | Σ γ ⇒⇒
        ( ((K:χ↦Σ) x, (K:χ↦Σ) y) ⇒ e↓χ γ(x,y)
        | ((K:ω↦Σ) x, (H:χ1↦Σ) y) ⇒ false ∗ ) )
  in (equal↓List Nat(alist,alist), equal↓Tree
  Nat(atree,atree))

: Bool×Bool
```

Note here the non-linear use of constructor variables in the structural case: we cannot use the natural expression with a primitive equality predicate on constructors since the enclosing if statement would not be well-typed.

Zipping together two data structures

The zip function merges two values of common structural type into a single structure with pairs of components taken from the corresponding parts of the two structural arguments. Since two structures may differ in shape (e.g., as in lists of different lengths), we must provide some mechanism for representing non-matching structures; we use a failure

component at each distinct type in the problematic component positions of the result, thus allowing us to keep all matching structure and component pairs intact.

```
⟨ {} , {} , {} ,
  { fail: ∀α.α } ,
  {}⟩

⊢ let zip =
    spoly z: ⋂φ:T→D.(φ α×φ β)→φ(α×β)
    as ( unit (∗,∗) ⇒ ∗
       | pair ((p1,q1), (p2,q2)) ⇒ (z(p1,p2), z(q1,q2))
       | struc
         ( (K x, K y) ⇒ K(z(x,y))
         | (K x, H y) ⇒ fail )
       | comp#1 (a,b) ⇒ (a,b) )
  in zip

↦ let zip = μz.
    ( (λε.1) ⇒⇒ (∗,∗) ⇒ ∗
    | (λε.ψ ε×ζ ε) ⇒⇒ ((p1,q1), (p2,q2)) ⇒
        (z↓ψ(p1,p2), z↓ζ(q1,q2))
    | (λε.Σ(θ ε)) ⇒⇒
        ( ((K:χ↦Σ) x, (K:χ↦Σ) y) ⇒
            K(z↓(λδ.χ(θ δ))(x,y))
        | ((K:ω↦Σ) x, (H:χ1↦Σ) y) ⇒ fail )
    | (λε.ε) ⇒⇒ (a,b) ⇒ (a,b) )
  in zip↓Γ

: (Γ γ×Γ η)→Γ(γ×η)
```

Note that the fail value is given a standard Milner-style type scheme in the assumptions, with an explicit universal quantifier.

Unzipping an arbitrary structure of pairs

The unzip function is dual to the zip function: it takes as argument a structure with paired components and returns a pair of structures. We do not need the failure component here since there is no possibility of mismatched structure.

```
⟨ {} , {} , {} , {} , {}
  ⟩

⊢ let π1 = (p,q) ⇒ p
  in let π2 = (p,q) ⇒ q
    in let unzip =
        spoly u: ⋂φ:T→D.φ(α×β)→φ α×φ β
        as ( unit ∗ ⇒ (∗,∗)
           | pair (y,z) ⇒
               let uy = u y
               in
    let uz = u z

    in ((π1 uy, π1 uz), (π2 uy, π2 uz))
           | struc K x ⇒
               let ux = u x in (K(π1 ux), K(π2 ux))
           | comp#1 (a,b) ⇒ (a,b) )
      in unzip

↦ let π1 = (p,q) ⇒ p
  in let π2 = (p,q) ⇒ q
    in let unzip = μu.
        ( (λε.1) ⇒⇒ ∗ ⇒ (∗,∗)
        | (λε.ψ ε×ζ ε) ⇒⇒ (y,z) ⇒
          let uy = u↓ψ y
          in let uz = u↓ζ z
             in ((π1 uy, π1 uz), (π2 uy, π2 uz))
        | (λε.Σ(θ ε)) ⇒⇒ (K:χ↦Σ) x ⇒
            let ux = u↓(λδ.χ(θ δ)) x
            in (K(π1 ux), K(π2 ux))
        | (λε.ε) ⇒⇒ (a,b) ⇒ (a,b) )
      in unzip↓Γ

: Γ(γ×η)→Γ γ×Γ η
```

A term with ambiguous typing

This example illustrates the fact that several different, incompatible typings may be correct for a single term. This ambiguity is one of the factors which makes full type inference difficult.

In the first typing, the user-supplied constraint for the variable a specifies that both halves of the argument pair should be analyzed for type structure using the supplied keyword type patterns.

```
⟨ {} , {} , {} ,
  { size: ⋂α:D.α→Nat } ,
  {}⟩

⊢ let amb =
    spoly a: ⋂α:D.(α×α)→Nat
    as ( unit (u,∗) ⇒ zero ∗
       | pair ((p,q), (b,c)) ⇒ (size p) + (size b)
       | struc (s, K x) ⇒ size x )
  in amb

↦ let amb = μa.
    ( 1 ⇒⇒ (u,∗) ⇒ zero ∗
    | ε×δ ⇒⇒ ((p,q), (b,c)) ⇒ (size↓ε p) + (size↓ε b)
    | Σ γ ⇒⇒ (s, (K:χ↦Σ) x) ⇒ size↓χ γ x )
  in amb↓η

: (η×η)→Nat
```

Here we use a structurally polymorphic size function from the assumptions to on both halves of pairs and on structural values.

In the next correct typing, the same term is used except that

the user-supplied constraint for a now specifies that only the second of the two arguments should be analyzed via the type patterns.

```
⟨ {} , {} , {} ,
  { size: ⋂α:D.α→Nat } ,
  {}⟩

⊢ let amb =
    spoly a: ⋂α:D.(π×ν×α)→Nat
    as  ( unit (u,∗) ⇒ zero ∗
        | pair ((p,q), (b,c)) ⇒ (size p) + (size b)
        | struc (s, K x) ⇒ size x )
    in amb

↦ let amb = μa.
    ( 1 ⇒» (u,∗) ⇒ zero ∗
      | ε×δ ⇒» ((p,q), (b,c)) ⇒ (size↓π p) + (size↓ε b)
      | Σ γ ⇒» (s, (K:χ↦Σ) x) ⇒ size↓χ γ x )
    in amb↓η

:  (ε1×δ1×η)→Nat
```

The ambiguity in this example arises from the fact that there is no constraint implied by the trivial patterns for the left half of the pair argument in the unit and structural cases, and thus the structure visible in the pair case may be taken either as being general or as being specific to that case.

A structurally polymorphic flatten function

This function is a generalization of the usual flatten function on trees (also called fringe); the append function introduced in the assumptions is the standard function on pairs of lists.

```
⟨ {} , {} , {} ,
  { append: ∀α.(List α×List α)→List α,
    even: Nat→Bool, odd: Nat→Bool,
    alist: List Nat, atree: Tree Nat } ,
  {}⟩

⊢ let flatten =
    spoly f: ⋂φ:T→D.φ α→List α
    as  ( unit ∗ ⇒ nil ∗
        | pair (y,z) ⇒ append(f y, f z)
        | struc K x ⇒ f x
        | comp#1 a ⇒ cons(a, nil ∗) )
  in (flatten alist, flatten atree)

↦ let flatten = μf.
    ( (λε.1) ⇒» ∗ ⇒ nil ∗
      | (λε.ψ ε×ζ ε) ⇒» (y,z) ⇒ append(f↓ψ y, f↓ζ z)
      | (λε.Σ(θ ε)) ⇒» (K:χ↦Σ) x ⇒ f↓(λδ.χ(θ δ)) x
      | (λε.ε) ⇒» a ⇒ cons(a, nil ∗) )
  in (flatten↓List alist, flatten↓Tree atree)

:  List Nat×List Nat
```

A generalized transpose function

This function is similar to the zip and unzip functions above, but generalizes them both to the case of two arbitrary nested structures; the behavior of the function is to invert the two levels of structure. As with zip above, we need to account for the possibility of structures of different shape, for which we use the same failure mechanism.

```
⟨ {} , {} , {} ,
  { map: ∀α,β.⋂Σ:T→D.(α→β)→Σ α→Σ β, zip:
  ∀α,β.⋂Σ:T→D.(Σ α×Σ β)→Σ(α×β),
    fail: ∀α.α, listoftrees: List(Tree Nat),
    π1: ∀α,β.(α×β)→α, π2: ∀α,β.(α×β)→β } ,
  {}⟩

⊢ let transpose =
    let transrec =
      spoly trans:⋂φ:T→D.(φ(Δ α)×Δ α)→Δ(φ α)×Δ
  α
      as  ( unit (∗,D) ⇒ (map(x ⇒ ∗) D,D)
          | pair ((y,z),D) ⇒
            let ty = trans(y,D)
            in  let tz = trans(z, π2 ty)
                in (zip(π1 ty, π1 tz), π2 tz)
          | struc (K x,D) ⇒
              let tx = trans(x,D)
              in (map K(π1 tx), π2 tx)
          | comp#1 (d,D) ⇒ (d,d) )
    in fd ⇒ π1(transrec(fd,fail))
  in transpose listoftrees

↦ let transpose = (λε.Σ ε) ⇒» (λδ.Γ δ) ⇒»
      let transrec = (λγ.Δ γ) ⇒» μtrans.
          ( (λη.1) ⇒» (∗,D) ⇒ (map↓Δ(x ⇒ ∗) D,D)
            | (λη.ψ η×ζ η) ⇒» ((y,z),D) ⇒
                let ty = trans↓ψ(y,D)
                in
      let tz = trans↓ζ(z, π2 ty)

      in (zip↓Δ(π1 ty, π1 tz), π2 tz)
              | (λη.Ω(θ η)) ⇒» ((K:χ↦Ω) x,D) ⇒
                  let tx = trans↓(λε1.χ(θ ε1))(x,D)
                  in (map↓Δ K(π1 tx), π2 tx)
              | (λη.η) ⇒» (d,D) ⇒ (d,d) )
    in fd ⇒ π1(transrec↓Σ↓Γ(fd,fail))
  in transpose↓Tree↓List listoftrees

:  Tree(List Nat)
```

Note the use of a free structural variable in the user-supplied type constraint: this variable remains free and so must be quantified and abstracted out in the decorated term. Note

also the similar treatment of the structural variables which are introduced through instantiation and then abstracted out from the let definition in which they appear.

An example of structure propagation

This example illustrates the necessity of propagating instantiated type structures through let definitions. The use of map within the let definition must be instantiated to a (fresh) type structure variable, but since this variable is not further instantiated during the check of the body of the definition, we must introduce a new abstraction for it. This abstraction is then eliminated through two (different) instantiations when the definition is used.

$\langle$ {} , {} , {} ,
{ even: Nat→Bool, alist: List Nat, atree: Tree Nat,
map: $\forall \alpha,\beta.\bigcap\Sigma{:}T\rightarrow D.(\alpha\rightarrow\beta)\rightarrow\Sigma\ \alpha\rightarrow\Sigma\ \beta$ } ,
{}$\rangle$

$\vdash$ **let** me = map even **in** (me alist, me atree)

$\mapsto$ **let** me = $(\lambda\varepsilon.\Sigma\ \varepsilon) \twoheadrightarrow$ map$\downarrow\Sigma$ even
in (me$\downarrow$List alist, me$\downarrow$Tree atree)

: List Bool×Tree Bool

The effect of the abstractions and instantiations introduced into the decorated term is to propagate the structures List and Tree to the definition of map, where they can be used to guide the recursive traversal of the corresponding data structures.

Conclusions

In this final chapter, we summarize and evaluate the research presented in preceding chapters, compare it to related work and review some areas which show promise for further research. The following paragraphs comprise a broad, summary assessment of the work, whereas the next section covers specific issues on a more detailed basis.

The most valuable aspects of the work are, in my opinion, the robust formalization of patterns and pattern abstraction, the crucial idea of analytical polymorphism and its expression through type-pattern abstraction, and the extension of practical type inference systems to include structural polymorphism. Of these, the formalization of patterns is not especially profound, being more of technical interest than anything else.

I feel that the success of the research is most properly assessed and best appreciated in the context of the pragmatic concerns which originally motivated it. I had moved from programming in APL, where arrays were the only data structure and structural transformation the dominant programming technique, to programming in Standard ML, where data structure definitions provided new expressive power and a more natural representation for most problems, but where structural transformation was hindered by the limitations of polymorphic typing with respect to data structures. I believe that the incorporation of structural polymorphism into the type systems of programming languages such as Standard ML will prove to be of significant value for practicing programmers and that this will justify the work.

Although my proposed systems of type inference for structural polymorphism are incomplete, due to their dependence on extended notions of unification, it is important to recall that even the well-behaved system of Milner is *semantically* incomplete, as indeed any static system must be. The so-called *syntactic* completeness of the Milner type inference algorithm is relative to a formal system which, while natural enough, is in some sense tailored to the limitations of that algorithm.

Furthermore, Milner-style polymorphism strikes a delicate balance between expressive power and amenability to automated analysis, and it has proved difficult to extend this system in any direction without sacrificing some pleasant properties. Witness, for example, the difficulties encountered with respect to recursive definitions [Myc84, KTU88], overloading [Kæ88, WB89] and polymorphic references, as well as various other proposed extensions. In fact, recent trends in programming language design seem to favor more powerful systems at the expense of some incompleteness or even undecidability in the type system: the viability of these approaches is supported by the evidence that they often prove adequate in practice (see, e.g., [Con*86]). It may be more fruitful and realistic to explore systems which rely on user-assistance, resource-bounded type-checking or other "inelegant" methods, in order to support undecidable type systems, than to attempt to preserve good theoretical properties by making only marginal gains in expressive power.

One particularly promising approach, in my opinion, would be to start with an explicitly-typed system and then seek ways to eliminate or mitigate the burden of type declarations and instantiations through the incorporation of type inference techniques. This sort of approach has been quite successful in the context of logic-based type systems (e.g., the tactics of nuPRL [Con*86]), where the burden of explicit typing takes on a dramatic new dimension. In the context of more traditional, computational uses of types, such an approach has been explored in the LEAP system [PL89].

Despite the potential pragmatic benefits of structural polymorphism, I must admit that my theoretical results are generally either negative (e.g., the lack of principal and static typing properties for structural polymorphism) or undeveloped (e.g., a semantics of analytical polymorphism and the issue of decidability for functional-variant unification). Negative results are not very satisfying, but they are still of some value and are in any case unavoidable. With regard to the undeveloped results, I hope that they can at least serve a good purpose by generating future research in the areas of semantics and unification.

Detailed assessment

In Chapter 1 we developed a formal treatment of patterns as they are commonly used in functional languages, including restrictions on coverage and compatibility as well as a set of technical operations for expressing typing restrictions in a natural and succinct manner. Although in the case of simply-typed pattern abstractions this was largely a routine exercise, our approach proved robust enough to maintain uniformity in the face of changes in both pattern-matching regimens and in the underlying syntax of patterns (e.g., polymorphic constructors, the calculus with sums and products in Chapter 1, and higher-order type patterns). This approach to the formalization of patterns may prove useful as the basis for other, more complex notions of patterns as well.

In concert with our technical approach for pattern abstraction, we also developed in Chapter 1 an approach to globally-scoped algebraic data type settings. As with the patterns, this is not a particularly difficult or deep problem for the simply-typed case, but we demonstrated in Chapter 2 how it can be easily generalized to accommodate even higher-order data structures, a concept which has not been previously proposed in the context of programming languages (at least to my knowledge).

Higher-order data structures may prove to be of some independent interest and practical value, even in the context of first-order polymorphic languages: for example, it might be possible to accommodate higher-order data structures in first-order type systems by making the (reasonable) restriction that they take only other data structures as arguments, rather than general higher-order type functions. As currently construed, it seems that we cannot take full advantage of structural polymorphism for higher-order data structures in some cases, such as the map functional, because of the restrictions imposed on quantification by the Milner-style type discipline (we refer here to the necessity to pass polymorphic functional arguments to the higher-order map). These difficulties do not mean that we cannot use either feature independently of the other, or even both in one system, just that there will be limitations on what we can express with the combination of the two.

In Chapter 2 we also introduced the key ideas which underlie all of our research, namely the notion of pattern abstraction over types and the consequent notion of analytical polymorphism. The extension of type abstractions to use patterns rather than individual variables follows an elegant analogy with the usual extension of lambda abstraction to value patterns. While our use of patterns over type functions complicates the technical specification of the language and type system, it is easily accommodated by the approach to

patterns developed in Chapter 1, requiring changes only to the lowest-level operations such as matching and pattern-set generation.

While the basic concepts and formal specification of the higher-order calculus seem successful, we have left its theory and semantic foundations relatively unexplored. We explored even operational semantics only to the point of making those revisions necessary to give a reasonable account of reduction and conversion, i.e., one which made no "dynamic" references to type assumption contexts. The basis of analytical and structural polymorphism requires dynamic reference to some aspects of constructor setting contexts, in that we must know argument the type functionals associated with value constructors in order to drive pattern-matching on structural arguments (though even this dependence can be removed if we are willing to "pre-compile" distinct functions for all possible data structure instantiations).

There are also a few aspects of our account of analytical polymorphism which are unsatisfying: some of these are addressed in the section on further research below, but others are inherent to the basic approach. For example, it is unfortunate but unavoidable that the use of type patterns allows us to define not only the sort of uniform functions which we have given in our examples, but also more ad-hoc functions which behave differently for differently-typed arguments. For example, we can define in the current system a function which returns different integer results for arguments which have different outermost type forms. This problem can be mitigated, by restricting type patterns so that no specific data type constructors are used (i.e., only type-constructor variables are allowed), but some non-uniform examples will be expressible in any system which allows us to distinguish between cases of types. It would be interesting to pursue the notion of parametricity investigated by Reynolds and others (see [Rey83, Mai91]) in the context of analytical polymorphism: such an investigation might also help to complete the characterization of the higher-order map functional we have begun in Section 5.3 and perhaps even to suggest some syntactic means of identifying the more ad-hoc uses of type-pattern abstraction.

The examples presented in Section 5.1 demonstrate that a number of useful functions can be defined using the techniques of analytical polymorphism. It is not clear that there is an endless variety of such functions, however: we will have to depend on practical experience to determine if there is a broad need for these features among applications programmers (it may even be sufficient to offer these features at a restricted level reserved for systems programming). Although the generalized transpose example is more complex than the others we have given, it would be nice to see further examples of more sophisticated functions, i.e., those which go beyond the familiar structural transformations such as map and flatten.

In Chapters 6, 7 and 8 we applied the ideas of analytical polymorphism to the design of type systems for realistic functional programming languages with first-order polymorphic type inference; unfortunately, several problems inherent in the approach lead to incompleteness and possible ambiguity in the type system. Nevertheless, we have demonstrated that these problems can be overcome in practical applications and that the useful examples developed in the explicitly-typed context can be accommodated in realistic programming contexts with a type inference algorithm.

The two systems presented in Chapters 7 and 8 provide contrasting approaches to automated type inference in the context of structural polymorphism: in this sense, they define a spectrum of possible trade-offs which might be struck between expressive power, convenience and automation. The system of Chapter 7, for example, requires no explicit type constraints but is plagued by a severe sort of ambiguity: because structural polymorphism depends on a weak form of dynamic typing, the lack of principal types means that different typings will actually yield different interpretations of the same term. The system of Chapter 8, on the other hand, requires minimal type constraints from the user, and can infer unique types under these circumstances, but still depends on an extended notion of unification which may prevent completeness of the algorithm with respect to the type system.

We have identified the general problem which seems to underlie the difficulties of inferring structurally polymorphic types from unconstrained terms as functional-variant unification. While we have presented an algorithm which solves such problems completely, we suspect that the problem is undecidable, i.e., that the algorithm will fail to terminate in some cases. Unfortunately, we have no solid proof on this matter either way, though we have narrowed the probable causes down to the need for a generalized form of occurs check. Even if functional-variant unification turns out to be undecidable, it is not clear that the full power of the general problem is necessary in the limited context in which we need it, i.e., we may be able to exploit the specific circumstances of the structurally polymorphic inference problem to restrict functional-variant unification to a decidable subset.

Related work

A variety of other research in functional languages and type systems bears consideration in regard to its relation with our work. First and foremost, much research into dynamic types employs the same technical device of patterns over types as we have here, and with similar goals; second, research which addresses overloading in type systems also seeks to solve some of the same problems we do (in fact, our approach can be viewed as a sort of systematic, uniform overloading); and finally, there are other miscellaneous efforts which bring techniques similar to ours to bear on rather different problems.

The earliest work on the use of type patterns to effect dynamic typing seems to be an unpublished manuscript by Alan Mycroft [Myc83]; although the work presented there is of an unfinished and preliminary nature, it suggests a solution to the problem of equality in ML, as do we, along with approaches to printing values of arbitrary type. More recent work on the type systems of the languages Russell, Amber and Modula-3 [DD85, Car*88] and for accommodating dynamic types in ML [LeMa91] has revived the pursuit of dynamic types under the same technique using type patterns. The main differences between these works and our own lie in the significance of the dynamic typing for run-time uses of types and in our notion of analytical polymorphism.

In particular, all of these other systems consider dynamic types which are essentially dependent sums, so that a value of dynamic type is a pair consisting of a type and a value of that type. This type/value pair exists as a conglomerate value at run-time—the type half of the pair is matched against a set of type patterns in order to drive selection of alternative results which may access the value half of the pair. Although this use of pattern-matching on types is similar to our own, note that these other systems identify certain values as dynamic and

require that their type information be carried along with them at run-time. By contrast, our implicitly-typed system of structural polymorphism requires only that the argument type-functions for each value constructor be known at run-time: types will be generated and matched at run-time in our system, but they are not tied to argument values, but rather are associated with the semantics of certain functions. Thus our system avoids the problem (raised in Section 4.2 of [LeMa91]) that all arguments to functions defined by type patterns must be dynamically-typed values; this in turn means that structural polymorphism can be incorporated into current languages without severe impact on current techniques and applications.

None of the dynamic type systems we have mentioned considers patterns over type functions, which is crucial to expressing analytical and structural polymorphism, in that it allows us to express the notion of a component of a structure.

In some of these systems, dynamic values may cause run-time failures if they cannot be properly coerced to a statically-specified type constraint (we expect that this could be easily fixed following our approach to pattern coverage, though).

Interestingly, most of these other efforts consider printing functions as a motivating example: we have not purposely avoided this example, and remark here only that we would need a means of associating printable representations to atomic value constructors in order to give a satisfactory and general print function for arbitrary types.

Although we have not directly addressed the idea of overloading individual symbols with functions of different types, it may seem that our idea of casing based on the forms of types is similar to various research efforts in this area. In fact, some conceptions of type overloading are expressed in terms of type patterns, although none of them seem to consider patterns over type functions. Our notions of analytical and structural polymorphism have been designed around examples which are specifically uniform in their behavior, rather than examples which branch on different types in an ad-hoc manner. We can express the overloading of, e.g., the addition function in our calculus, but not in any satisfying way, for we can return useful results only for a finite and very limited set of type arguments. The proper consideration of the relation between different types based on their (non-initial) algebraic behavior is beyond the scope of our work: such considerations are specifically addressed, for example, in the ScratchPad II system (see [JSW88]).

Although there is a small literature which addresses the compilation of patterns, we are not aware of much research which addresses the formal analysis of patterns in any broad way. The authors of the formal definition of Standard ML use techniques roughly similar to ours in describing patterns: the patterns used in Standard ML are both more complex than ours, in that they proived various pragmatic extensions to the basic forms, and simpler, in that there is no notion of patterns over types or type functions nor any notion of constructor variable.

Hagino makes a (largely implicit) use of a map functional defined at all recursive types positive in their recursion parameter in his work on co-data types in ML [Hag86, Hag89]. In general, it seems likely that techniques of category theory (on which Hagino's work is based) might provide a useful perspective on our research: this seems particularly likely in the case of the characterization of the generalized map, given the inspiration we took in that section from commutative diagrams.

In contrast to our higher-order pattern calculus, most applications which make use of explicitly-typed polymorphism seem to require no more than second-order features, or at most higher-order constants. One exception is the calculus underlying the LEAP language [PL89], which uses higher-order type functions, kind polymorphism and constructors with polymorphic arguments to express the meta-level encoding of programs within the calculus itself. The LEAP approach is based on a generalization of the representation of arbitrary data structures using polymorphic, functional types, which in turn generalizes the usual polymorphic version of Church numerals [BöBe86]—in order to apply our ideas in the LEAP framework, we would need to modify our approach to better accommodate functional and polymorphic types.

The technique of representing data structure types via generalized Church numerals provides an interesting contrast to our approach using explicit constructors; this technique might shed light on ways to transform our conception of constructors to a curried style (see *Further Research* below). There has also been a recent spate of research on inductively-defined types in logic-oriented type theories [CoMo90]; these research efforts might profit from the sort of generalization represented by our higher-order data structures.

The primary goal of the work presented in [MFP91] is to develop an algebraic calculus of functional programming including various specialized recursion operators; within this framework, the authors consider a general notion of map which probably coincides with ours for first-oder data structures. Their remark that "map can be defined both as a catamorphism and as an anamorphism" may have some connection to the duality we observed in our definition of the higher-order map of Chapter 5. Although the algebraic calculus described in [MFP91] is not oriented towards pattern-based specifications, it would be interesting to explore cross-applications between the algebraic calculus and our notions of higher-order data structures and analytical polymorphism.

Further research

In this final section we list some aspects of the current effort which could benefit from further research. Some of these ideas are merely technical variations on various design decisions we have taken here; others address areas which we have left largely untouched, due to lack of either time or expertise, such as the semantics of analytical polymorphism. Others address the removal of some of the restrictions we have made, e.g., we might seek a more complete notion of data kinds, pattern-based computation over types, or dependent data types and structures. Finally, some areas would benefit simply from further elaboration.

With regard to models of analytical polymorphism, we have sketched some ideas about the requirements any such model should have, but we have made no attempt at all to exhibit a specific model. We should further codify our requirements and then pursue the construction of a model, or identify some already-proposed model of polymorphism as meeting our proposed requirements.

In our approach to constructors at both the type and value levels, we have required that all constructors have restricted forms of types, namely polymorphic functional types with just a single argument at each level. While our use of unit and binary product types allows us to represent naturally those constructors which would normally take zero or multiple arguments, it would be interesting to develop an alternative approach which used currying to represent multiple

arguments. Such an approach would allow for a more concise set of forms for types and would be more in keeping with practice in some functional languages (e.g., Miranda, where all constructors are normally curried). Two problems which we would need to resolve for this approach to be successful would be the incorporation of function types into patterns, at least for some limited notion of "constructive" function types (e.g., the curried functional types of constructors) and the expression of some of our standard examples (since it is not clear how they would be carried over to a language which lacked unit types as a base case).

Another interesting variation on our approach to constructors would be to consider a calculus with dependently-typed constructors and pattern abstractions. Although examples such as lists with fixed but arbitrary length have been used to motivate or explain the idea behind dependent types (see [MR86]), they are usually not definable as constructor-based algebraic types in the dependent calculi which have actually been studied. Taking a dependently-typed approach would allow us to merge the type and value levels, thus eliminating some of the overhead associated with the parallel constructs at each of the levels of our calculus (e.g., units, products, constructors, etc.). Dependently-typed calculi are usually not statically-typed, since the types of function results may depend, in general, on the values of their arguments; nevertheless, we think that the exercise could yield interesting insights.

One aspect of our research which begs for further study is the relaxation of some of the restrictions we have imposed on data kinds: in particular, we have foregone several possible generalizations simply because they did not mesh well with our motivating examples. Outside the context of examples like count and equality, it might be feasible to consider patterns over some limited sort of functional types, for example, to the sort of positive parametric occurrences considered by Hagino [Hag86]. An approach to functional types could easily lead to results for polymorphic quantifications as well, based on the idea of representing such quantifications as the application of a constant to a lambda-abstracted function (note that we have already considered the use of kind polymorphism, which would be necessary to allow for quantifications over arbitrary kinds using a single constant form).

One interesting extension of the pattern-based approach, would be to allow functions over types to be expressed using patterns and recursion: it would probably be natural in this context to represent our algebraic data structures directly as recursive types. A further generalization of this same idea would be to use pattern-abstractions to express kind polymorphism; in such a language we might be able to express further generalizations of the map functional which captured our first- and higher-order exapmples as dicstinct instances of a single functional.

Returning to the restrictions imposed by our notion of data kind, we should remark that our approach allows us to incorporate structural polymorphism into languages with type constructs we have not considered, such as dependent types, explicitly dynamic or abstract types, etc., merely by including these types among the "problematic forms" which are excluded from the kind of data types. Although this is a trivial sort of inclusion, it may be useful in that it would allow for the gradual extension of structural polymorphism to other forms of types without delaying its benefits with regard to those forms which we have already considered.

It would be interesting to see if the techniques we used in formalizing indexed constructors could be applied to the more pedestrian end of formalizing polymorphism over arbitrary-length tuples. A preliminary investigation of such features suggests that the choice of a practical and complete set of tuple-manipulation primitives would be crucial. Alternatively, we might seek some sensible way to recursively decompose tuples, although in either case we must prevent the escape of indices from the scope of their product-quantification. It would also be interesting to incorporation of such features into a calculus with labeled tuples and record-style inheritance (see, e.g., [Wan89]).

We have already remarked in the first section above about the potential for including higher-order data structures in first-order, implicitly polymorphic type systems like that of Standard ML. We have not fully investigated the implications such an extension would have for automated type inference based on the usual notion of unification, although our experience with structural polymorphism suggests that at worse we might give up some little-used completeness in exchange for greater expressive power.

And finally, as we remarked in the first section, we should pursue the problem of functional-variant unification, in order to determine finally the question of its decidability and the degree of its impact on the problem of type inference for undecorated structural polymorphism.

References

[ADJ79] James W. Thatcher, Eric G. Wagner and Jesse B. Wright. *Notes on Algebraic Fundamentals for Theoretical Computer Science*. In *Foundations of Computer Science III, Part 2: Languages, Logic, Semantics*, ed. J. W. de Bakker and J. van Leeuwen (Mathematical Centre Tracts #109). 1979.

[Bar84] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics* (Studies in Logic and the Foundations of Mathematics #103). North-Holland, 1984.

[Bar9?] Henk Barendregt. *Typed Lambda Calculi: Their Syntax and Semantics*. North-Holland, 199? (not yet published).

[BH90] Henk Barendregt and Kees Hemerik. *Types in Lambda Calculi and Programming Languages.* In *Proceedings of the European Symposium on Programming* (Lecture Notes in Computer Science #432), Springer-Verlag, 1990.

[BöBe86] C. Böhm and A. Berarducci. *Automatic Synthesis of Typed λ-programs on Term Algebras.* In *Theoretical Computer Science*, vol. 39, pp. 135-154, 1985.

[Bœ85] Hans-J. Bœhm. *Partial Polymorphic Type-checking is Undecidable*. In *Proceedings of the 26th Symposium on Foundations of Computer Science*, ACM, 1985.

[Br88] Val Breazu-Tannen. *Combining Algebras and Higher-order Types*. In *Proceedings of the 3rd Symposium on Logic in Computer Science*, pp. 82-90, IEEE, 1988.

[BrGa89] Val Breazu-Tannen and J. Gallier. *Polymorphic Rewriting Conserves Algebraic Strong Normalization and Confluence*. In *Proceedings of the International Conference on Automata, Languages and Programs 1989*, pp. 137-150. (Lecture Notes in Computer Science #372), Springer-Verlag, 1989.

[BMM90] Kim Bruce, Albert Meyer and John Mitchell. *The Semantics of Second-order Lambda Calculus*. In *Information and Computation*, vol. 85, pp. 76-134, 1990.

[BMS80] Rod M. Burstall, David B. MacQueen and Donald T. Sannella. *HOPE: An Experimental Applicative Language*. In *Report on the 1980 LISP Conference*, pp. 82-90, ACM, 1980.

[Car86] Luca Cardelli. *A Polymorphic Lambda Calculus with Type:Type*. Research report, Digital Systems Research Center, 1986.

[Car*88] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kaslow and Greg Nelson. *Modula-3 Report*. Research report, Digital Systems Research Center, 1988.

[Cl*86] Dominique Clément, Jöelle Desperyoux, Theirry Desperyoux, and Giles Kahn. *A Simple Applicative Language: Mini-ML.* In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, ACM, 1986.

[Con*86] R.L. Constable, et al. *Implementing Mathematics With the Nuprl Proof Development System*, Prentice-Hall, 1986.

[CoMo90] Thierry Coquand and Christine Paulin. *Inductively Defined Types*. In *Proceedings of the 1988 International Conference on Computer Logic* (ed. by P. Martin-Löf and G. Mints), pp. 50-66, Springer-Verlag, 1990.

[DM82] Luis Damas and Robin Milner. *Principal Type-Schemes for Functional Programs*. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, ACM, 1982.

[DD85] James Donahue and Alan Demers. *Data Types are Values*. In *Transactions on Programming Languages and Systems*, vol. 7, pp. 426-445, ACM, 1985.

[FLO83] Steven Fortune, Daniel Leivant and Michael O'Donnell. *The Expressiveness of Simple and Second-order Type Structures*, The Journal of the ACM vol. 30, pp. 151-185. ACM, 1983.

[Gall90] Jean Gallier. *On Girard's Candidats de Reducibilité*. In *Logic in Computer Science* (ed. by P. Odifreddi), Academic Press, 1990.

[Gir72] Jean-Yves Girard. *Interpretation Fonctionelle et Elimination des Coupres dans l'Arithmetique d'Ordre Superieur*. Ph.D. dissertation, Paris, 1972.

[Göd58] Kurt Gödel. *Über Eine bisher Noch Nicht Benütze Erweiterung des Finiten Standpunktes*. In *Dialectica* vol. 12, pp. 280-287, 1958. (English translation in *The Journal of Philosophical Logic* vol. 9, pp 133-142, 1980).

[Gol81] Warren D. Goldfarb. *The Undecidability of the Second-order Unification Problem (note)*, Theoretical Computer Science vol. 13, pp. 225-30, 1981.

[Hag87] Tatsuya Hagino. *A Typed Lambda Calculus with Categorical Type Constructors*. In *Category Theory and Computer Science* (Lecture Notes in Computer Science #283), Springer-Verlag, 1987.

[Hag89] Tatsuya Hagino. *Codatatypes in ML*. In *The Journal of Symbolic Computation* vol. 8, pp 629-650,1989.

[Hin83] Roger J. Hindley. *The Completeness Theorem for Typing λ-terms*. Theoretical Computer Science, vol. 22, pp. 1-17, 1983.

[HS86] Roger J. Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus* (London Mathematical Society Student Texts 1). Cambridge University Press, 1986.

[Ive62] Ken Iverson. *A Programming Language*. Wiley, 1962.

[JSW88] Richard D. Jenks, Robert S. Sutor and Stephen M. Watt. *Scratchpad II: An Abstract Datatype System for Mathematical Computation*. In *Mathematical Aspects of Scientific Software*, pp. 157-182, Springer-Verlag, 1988.

[Kae88] Stefan Kaes. *Parametric Overloading in Polymorphic Languages*. In *Proceedings of the 1988 European Symposium on Programming* (Lecture Notes in Computer Science #300), Springer-Verlag, 1988.

[KTU88] A. J. Kfoury, J. Tiuryn and P. Urzyczyn. *A Proper Extension of **ML** with an Effective Type-assignment*. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, ACM, 1988.

[Lei83] Daniel Leivant. *Structural Semantics for Polymorphic Data Types (preliminary report)*, In *Conference Record of the 10th Annual ACM Symposium on Principles of Programming Languages*, ACM, 1983.

[LeMa91] Xavier Leroy and Michel Mauny. *Dynamics in ML.* In *Functional Programming Languages and Computer Architecture (proceedings)*, ed. J. Hughes (Lecture Notes in Computer Science #523), Springer-Verlag, 1991.

[Mac82] David B. MacQueen and Ravi Sethi. *A Semantic Model of Types for Applicative Languages*. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, ACM, 1982.

[Mai91] H. Mairson. *Outline of a Proof Theory of Parametricity*. In *Functional Programming Languages and Computer Architecture (proceedings)*, ed. J. Hughes (Lecture Notes in Computer Science #523), Springer-Verlag, 1991.

[McC*56] John McCarthy *et al. LISP 1.5 Programming Manual*. MIT Press, 1956.

[McC79] Nancy McCracken. *An Investigation of a Programming Language with a Polymorphic Type Structure*, Ph.D. dissertation, Syracuse University, 1979.

[McD92] James L. McDonald (mcdonald@xenon.stanford.edu). *Re: dynamic typing in SML*. In the Usenet newsgroups comp.lang.functional and comp.lang.scheme, 19 May 92.

[M-L71] Per Martin-Löf. *A Theory of Types*. Unpublished manuscript, 1971.

[MFP91] Erik Meijer, Maarten Fokkinga and Ross Paterson. *Functional Programming with Bananas, Lenses Envelopes and Barbed Wire*. In *Functional Programming Languages and Computer Architecture (proceedings)*, ed. J. Hughes (Lecture Notes in Computer Science #523), Springer-Verlag, 1991.

[Mes89] J. Meseguer. *Relating Models of Polymorphism.* In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, ACM, 1989.

[Mey82] Albert R. Meyer. *What Is a Model of the Lambda Calculus?* Information and Control, vol. 52, pp. 87-122, 1982.

[MR86] Albert R. Meyer and Mark Reinhold. *Type Is Not a Type: Preliminary Report.* In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, ACM, 1986.

[Mil78] Robin Milner. *A Theory of Type Polymorphism in Program-ming*. Journal of Computer and System Sciences, vol. 17, 1978.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Mit84] John Mitchell. *Lambda Calculus Models of Typed Programming Languages*, Ph.D. dissertation, MIT, 1984.

[MH88] John Mitchell and Robert Harper. *The Essence of ML*. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, ACM, 1988.

[Myc83] Alan Mycroft. *Dynamic Types in ML*. Unpublished draft, Department of Computer Science, Edinburgh University, 1983.

[Myc84] Alan Mycroft. *Polymorphic Type Schemes and Recursive Definitions*. In *Proceedings of the International Symposium on Programming*, pp. 217-228, Springer-Verlag,1983.

[NaMi88] G. Nadathur and D. Miller. *An Overview of λProlog*. In *Logic Programming: Proceedings of the 5th International Conference and Symposium)*, ed. by R. Kowalski and K. Bowen, vol. 1, pp. 810-827, MIT Press, 1988.

[NS91] Tobias Nipkow and Gregor Snelting. *Type Classes and Overloading Resolution via Order-Sorted Unification*. In *Functional Programming Languages and Computer Architecture (proceedings)*, ed. J. Hughes (Lecture Notes in Computer Science #523), Springer-Verlag, 1991.

[O'Don85] Michael J. O'Donnel. *Equational Logic as a Programming Language*(Foundations of Computing Series). MIT Press, 1985.

[Pau90] L. C. Paulson. *Isabelle: the Next 700 Theorem Provers*. In *Logic in Computer Science* (ed. by P. Odifreddi), Academic Press, 1990.

[PL89] Frank Pfenning and Peter Lee. *LEAP: A Language with Eval and Polymorphism*. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development* (Lecture Notes in Computer Science #352), Springer-Verlag, 1989.

[Pitt87] A. M. Pitts. *Polymorphism Is Set-theoretic, Constructively*. In *Category Theory and Computer Science* (Lecture Notes in Computer Science #283), pp. 12-39, Springer-Verlag, 1987.

[Rey74] John Reynolds. *Towards a Theory of Type Structure*. In *Proceedings Colloque sur la Programmation* (Lecture Notes in Computer Science #19), Springer-Verlag, 1974.

[Rey83] John Reynolds. *Types, Abstraction and Parametric Polymorphism*. In *Information Processing '83* (ed. by R.E.A. Mason), pp. 513-523, North-Holland, 1983.

[Rey84] John Reynolds. *Polymorphism Is Not Set-theoretic*. In *Semantics of Data Types* (Lecture Notes in Computer Science #173), pp. 145-156, Springer-Verlag, 1984.

[Sco76] Dana Scott. *Data Types as Lattices*. In *The SIAM Journal of Computing*, vol. 5, no. 3, pp. 522-87, 1976.

[Sco81] Dana Scott. *Lectures on a Mathematical Theory of Computation*. Technical monograph PRG-18, Oxford University Computing Laboratory, Programming Research Group, 1981.

[See87] R. A. G. Seely. *Categorical Semantics for Higher-order Polymorphic Lambda Calculus*. In *The Journal of Symbolic Logic*, vol. 52, pp. 969-989, 1987.

[Tha88] Satish Thatte. *Type inference with Partial Types*.In *Proceedings of the 15th International Conference on Automata, Languages and Programs*, pp. 137-150, Springer-Verlag, 1988.

[Tur85] David A. Turner. *Miranda: A Non-strict Functional Language with Polymorphic Types*. In *Functional Languages and Computer Architecture*, ed. J.-P. Jouannaud (Lecture Notes in Computer Science #201), Springer-Verlag, 1985.

[WB89] Philip Wadler and Steven Blott. *How to Make* Ad-hoc *Polymorphism Less* Ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, ACM, 1989.

[Wan89] Mitchell Wand. *Type Inference for Record Concatenation and Multiple Inheritance.* Selections from the *1989 IEEE Symposium on Logic in Computer Science*, in *Information and Computation* vol. 93,pp 1-15. 1989.