Using Dependent Types to Express Modular Structure

David MacQueen

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

Writing any large program poses difficult problems of organization. In many modern programming languages these problems are addressed by special linguistic constructs, variously known as modules, packages, or clusters, which provide for partitioning programs into manageable components and for securely combining these components to form complete programs. Some general purpose components are able to take on a life of their own, being separately compiled and stored in libraries of generic, reusable program units. Usually modularity constructs also support some form of information hiding, such as "abstract data types." "Programming in the large" is concerned with using such constructs to impose structure on large programs, in contrast to "programming in the small", which deals with the detailed implementation of algorithms in terms of data structures and control constructs. Our goal here is to examine some of the proposed linguistic notions with respect to how they meet the pragmatic requirements of programming in the large.

Originally, linguistic constructs supporting modularity were introduced as a matter of pragmatic language engineering, in response to a widely perceived need. More recently, the underlying notions have been analyzed in terms of type systems incorporating second-order concepts. Here I use the term "second-order" in the sense of "second-order" logic, which admits quantification over predicate variables [Pra65]. Similarly, the type systems in question introduce variables ranging over types and allow various forms of abstraction or "quantification" over them.

Historically, these type systems are based on fundamental insights in proof theory, particularly the "formulas as types" notion that evolved through the work of Curry and Feys [CF58], Howard [How80], de Bruijn [deB80] and Scott [Sco70]. This notion provided the basis for Martin-Löf's formalizations of constructive logic as Intuitionistic Type Theory (ITT) [M-L71, M-L74, M-L82], and was utilized by Girard [Gir71], who introduced a form of second-order typed lambda calculus as a tool in his proof-theoretic work. The "formulas as types" notion, as developed in de Bruijn's AUTOMATH system and Martin-Löf's ITT, is also central to the "programming logics", PL/CV3 and nu-PRL developed by Constable and his coworkers [CZ84, BC85].

In the programming language area, Reynolds [Rey74] independently invented a language similar to that used by Girard, and his version has come to be called the second-order lambda calculus. An extended form of this language, called SOL, was used by Mitchell and Plotkin [MP85] to give an explanation of abstract data types. The programming languages ML [GMW78, Mil78] and Russell [BDD80, Hoo84, DD85] represent two distinctly different ways of realizing "polymorphism" by abstraction with respect to types. ML is basically a restricted form of second-order lambda calculus, while Russell employs the more general notion of "dependent types" (Martin-Löf's general product and sum, defined in §2). The Pebble language of Burstall and Lampson [BL84, Bur84] also provides dependent types, but in a somewhat purer form. Finally, Huet and Coquand's Calculus of Constructions is another variant of typed lambda calculus using the general product dependent type. It also provides a form of metatype (or type of types), called a "context", that characterizes the structure of second-order types, thus making it possible to abstract not only with respect to types, but also with respect to families of types and type constructors. The Calculus of Constructions is an explicit attempt to combine a logic and a programming language in one system.

Among these languages, Russell and Pebble are distinguished by having "reflexive" type systems, meaning that there is a "type of all types" that is a member of itself (**Type**ϵ*Type*). Martin-Löf's initial version of ITT [M-L71] was also reflexive in this sense, but he abandoned this version in favor of a "ramified"[1] system with a hierarchy of type universes when Girard's Paradox [Gir71] showed that the reflexive system was inconsistent as a constructive logic. In terms of programming languages, the paradox implies at least the existence of divergent expressions, but it is not yet clear whether more serious pathologies might follow from it (see Meyer and Reinhold's paper, this proceedings [MR86]). Since types are simply values belonging to the type **Type**, reflexive type systems tend to obscure the distinction between types and the values they are meant to describe, and this in turn tends to complicate the task of type checking. It is, on the other hand, possible to construct reasonable semantic models for reflexive type systems [McC79, Car85].

The remaining nonreflexive languages distinguish, at least implicitly, between individual types and the universe of types to which they belong and over which type variables range. However, the second order lambda calculus, SOL, and the Calculus of Constructions (despite its "contexts") are "impredicative,"[2] meaning that there is only one type universe and it is closed under type constructions like $\forall t.\sigma(t)$ and $\exists t.\sigma(t)$ that involve quantifiers ranging over itself. The reflexive type systems of Russell and Pebble are also impredicative, in perhaps an even stronger sense since type variables can actually take on Type, the universe of types, as a value. In contrast, the later versions of ITT and Constable's logics are ramified systems in which quantification or abstraction over a type universe at one level produces an element of the next higher level, and they are therefore predicative.

Our purpose here is not to set out the mathematical nuances of these various languages, but to look at some of the pragmatic issues that arise when we actually attempt to use such languages as vehicles for programming in the large. We will begin by discussing some of the consequences of the SOL Type system for modular programming. Then in §2 we briefly sketch a ramified (*i.e.* stratified) system of dependent types from which we derive a small language called DL, which is a generalized and "desugared" version of the extended ML language presented in [Mac85]. The final section uses DL to illustrate some of the stylistic differences between ML and Pebble.

1. Shortcomings of SOL's existential types

The SOL language [MP85] provides existential types of the form

$$\exists t.\sigma(t)$$

where t is a type variable and $\sigma(t)$ is a type expression possibly containing free occurrences of t. Values of such types are introduced by expressions of the form

$$\mathbf{rep}_{\exists t.\sigma(t)}\tau P : \exists t.\sigma(t)$$

where P is an expression of type $\sigma(\tau)$. These values are intended to model abstract data types, and were called *data algebras* in [MP85] and *packages* in [CW85]; we will use the term *structure* to agree with the terminology of [Mac85] that we will be adopting in later sections. The type component τ will be called the *witness* or *representation* type of the structure. Access to the components of a structure is provided by an expression of the form

$$\mathbf{abstype}\ t\ \mathbf{with}\ x\ \mathbf{is}\ M\ \mathbf{in}\ N : \rho$$

which is well-typed assuming $M : \exists t.\sigma(t)$ and $x : \sigma(t) \Rightarrow N : \rho$ with the restriction that t does not appear free in ρ nor in the type of any variable y appearing free in N.

As mentioned in [MP85], and because of the impredicative nature of SOL, these existential types are ordinary types just like *int* and *bool*, and the structures that are their values are just ordinary values. This implies that all the standard value manipulating constructs such as conditionals and functional abstraction apply equally to structures. Thus a parametric module is just an ordinary function of type $\tau \rightarrow \exists t.\sigma(t)$, for example.

There is a tradeoff for this simplicity, however. Let us consider carefully the consequences of the restrictions on the **abstype** expression. Once a structure has been constructed, say

$$A = \mathbf{rep}_{\exists t.\sigma(t)}\tau P$$

[1] Since Bertrand Russell introduced his "ramified type theory," the word "ramified" has been used in logic to mean "stratified into a sequence of levels", normally an infinite ascending sequence of levels.

[2] Roughly speaking, a definition of a set is said to be impredicative if the set contains members defined with reference to the entire set.

the type τ is essentially forgotten. Although we may locally "open" the structure, as in

abstype t **with** x **is** A **in** N

there is absolutely no connection between the bound type variable t and the original representation type τ. Moreover, we cannot even make a connection between the witness type names obtained from two different openings of the *same* structure. For example the types s and t will not agree within the body of

abstype s **with** x **is** A **in**
    abstype t **with** y **is** A **in** $\cdots$

In effect, not only is the form and identity of the representation type hidden, but we are not even allowed to assume that there is a unique witness type associated with the structure A. The witness type has been made not only opaque, but hypothetical! This very strong restriction on our access to an abstraction goes beyond common practice in language design, since we normally have some means of referring to an abstract type as a definite though unrecognizable type within the scope of its definition. This indefiniteness seems to be the price paid for being able to treat the abstract type structure as an ordinary value rather than as a type. (See [CM85], where we use the terms "virtual witness," "abstract witness", and "transparent witness" to describe three possible treatments of the witness type in an existential structure.)

Hierarchies of structures. The consequences of SOL's treatment of the witness type become clearer when we consider building abstractions in terms of other abstractions. Consider the following definition of a structure representing a geometric point abstraction.

$$\begin{aligned} PointWRT(p) = \langle & mk\_point : (real \times real) \rightarrow p, \\ & x\_coord : p \rightarrow real, \\ & y\_coord : p \rightarrow real \ \rangle \end{aligned}$$

$$Point = \exists p . PointWRT(p)$$

$$\begin{aligned} CartesianPoint = \ & \mathbf{rep}_{Point}\,(real \times real) \\ & \langle mk\_point = \lambda(x : real,\ y : real) . (x,y), \\ & \ x\_coord = \lambda p : real \times real . (fst\ p), \\ & \ y\_coord = \lambda p : real \times real . (snd\ p) \ \rangle \end{aligned}$$

Now suppose that we want to define a rectangle abstraction that uses *CartesianPoint*. We must first open *CartesianPoint*, define the rectangle structure, and then close the rectangle structure with respect to the point type.

$$\begin{aligned} RectWRT(p) = \exists rect . \langle & point\_interp : PointWRT(p), \\ & mk\_rect : p \times p \rightarrow rect, \\ & topleft : rect \rightarrow p, \\ & botright : rect \rightarrow p \ \rangle \end{aligned}$$

$$Rect = \exists p .\ RectWRT(p)$$

$$\begin{aligned} CartesianRect = \ & \mathbf{abstype}\ point\ \mathbf{with}\ P\ \mathbf{is}\ CartesianPoint\ \mathbf{in} \\ & \quad \mathbf{rep}_{Rect}\, point \\ & \quad\quad \mathbf{rep}_{RectWRT(point)}\, point \times point \\ & \quad\quad\quad \langle point\_interp = P, \\ & \quad\quad\quad\ \ mk\_rect = \lambda(tl : point,\ br : point) . (br, tl), \\ & \quad\quad\quad\ \ topleft = \lambda r : point \times point . (fst\ r), \\ & \quad\quad\quad\ \ botright = \lambda r : point \times point . (snd\ r) \rangle \end{aligned}$$

If we (doubly) open *CartesianRect* we will get a new virtual point type unrelated to any existing type. We had to incorporate an interpretation of this point type in the *Rect* structure as *point_interp* to provide the means to create elements of that type, which in turn allows us to create rectangles.

Now suppose we also define a circle abstraction based on the same CartesianPoint structure, and we want to allow interactions between the two abstractions, such as creating a unit circle centered at the top left-hand corner of a given rectangle. This requires that the rectangle structure, the circle structure, and any operations relating them all be defined within the scope of a single opening of the CartesianPoint structure. In general, we must anticipate all abstractions that use the point structure and might possibly interact in terms of points and define them within a single **abstype** expression.

It appears that when building a collection of interrelated abstractions, the lower the level of the abstraction, the wider the scope in which it must be opened. We thus have the traditional disadvantages of block structured languages where low-level facilities must be given the widest visibility. (For further details, see the examples in §6 of Cardelli and Wegner's tutorial [CW85].)

Interpreting known types. The notion of providing operations to interpret a type does not apply only to "abstract" types. It is often useful to impose additional structure on a given type without hiding the identity of that type. For instance, we might want to temporarily view $int \times bool$ as an ordered set with some special ordering. To do this we might define the structure *IntBoolOrd* as follows

$$OrdSet = \exists t.\,\langle le : t \times t \to bool \rangle$$

$$\begin{aligned}
IntBoolOrd = \ & \mathbf{rep}_{OrdSet}\,(int \times bool) \\
& \langle le = \lambda(n_1, b_1), (n_2, b_2).\ \mathbf{if}\ \ b_1 \text{ and } b_2\ \mathbf{then}\ n_1 \leq n_2 \\
& \qquad\qquad\qquad\qquad\quad \mathbf{elseif}\ \neg(b_1\ or\ b_2)\ \mathbf{then}\ n_1 \geq n_2 \\
& \qquad\qquad\qquad\qquad\quad \mathbf{else} b_1 \rangle
\end{aligned}$$

The following related and potentially useful mapping would take an *OrdSet* structure to the corresponding lexicographic ordering on lists

$$\begin{aligned}
& LexOrd : OrdSet \to OrdSet = \\
& \quad \lambda O : OrdSet.\ \mathbf{abstype}\ t\ \mathbf{with}\ L\ \mathbf{is}\ O\ \mathbf{in} \\
& \qquad\qquad \mathbf{rep}_{OrdSet}(list\ t) \\
& \qquad\qquad\quad \langle le = fix\ f.\,\lambda(l,m).\ \mathbf{if}\ \ (null\ l)\ \mathbf{then}\ true\ \ \cdots \rangle
\end{aligned}$$

Under the SOL typing rules, there is no way to make use of *IntBoolOrd* because we could never create any elements to which the ordering operation could be applied. In fact, no structure of type *OrdSet* can ever be used, because of our inability to express values of type t. Of course, this also means that *LexOrd* is useless. However, if we had access to the witness types, then structures like *IntBoolOrd* and mappings like *LexOrd* could be quite useful.

There are various ways of working around these problems within SOL. We can, for instance, delay or avoid entirely the creation of closed structures and instead deal separately with types and their interpreting operations. Thus, *LexOrd* could be rewritten to have the type $\forall t.\,OrdSetWRT(t) \to OrdSetWRT(list\ t)$, with $OrdSetWRT(t) = \langle le : t \times t \to bool \rangle$. However, our preferred solution is to abandon the restrictive SOL rule and view structures as inherently "open" or "transparent." This is suggested by the type rules of ITT, which provide access to both the witness and interpretation components of an existential (*i.e.* general sum) structure. Intuitively, within the scope of the local declaration

$$\mathbf{abstype}\ t\ \mathbf{with}\ x\ \mathbf{is}\ M\ \mathbf{in}\ N$$

we consider t to be simply an abbreviation or local name for the witness type of M. Of course, t itself should not appear in the types of free variables or of the entire expression, because it has only local significance, but its meaning, that is the witness type of M, may. "Abstraction" is then achieved by other means, namely by real or simulated functional abstraction with respect a structure variable (see [Mac85]), which is merely an "uncurried" form of the approach to data abstraction originally proposed by Reynolds in [Rey74]. When structures are transparent, it is clear that they carry a particular type, together with its interpretation; in fact, it is reasonable to think of structures as interpreted types rather than a kind of value. Consequently we also abandon the impredicative two-level system of SOL and move to a ramified system in which quantified types are objects of level 2, while level 1 is occupied by ordinary monomorphic types, structures, and polymorphic functions.

2. A language with ramified dependent types

2.1. Dependent types

There are two basic forms of dependent types, which we will call the *general product* and the *general sum*. The general product, written $\Pi x{:}A.B(x)$, is naively interpreted as the ordinary Cartesian product of the family of sets $\{B(x)\}_{x \in A}$ indexed by A, *i.e.*

$$\Pi x{:}A.B(x) = \{f \in A \to \bigcup_{x \in A} B(x) \mid \forall a \in A. f(a) \in B(a)\}$$

It denotes the type of functions that map an element $a \in A$ into $B(a)$, that is functions whose result type depends on the argument, with B specifying the dependence. Elements of $\Pi x{:}A.B(x)$ are introduced by lambda abstraction and eliminated by function application. In the degenerate case where B is a constant function, *e.g.* when $B(x)$ is defined by an expression not containing x free, the general product reduces to the ordinary function space $A \to B$.[3]

[3] General product types are also called "indexed products", "Cartesian products," or "dependent function spaces." Other notations include $x{:}A \to B(x)$ [CZ84], $x{:}A - B(x)$ [BL84], and $\forall x{:}A.B(x)$ (from the formulas as

The general sum, written $\Sigma x:A.B(x)$, is intuitively just the disjoint union of the family $\{B(x)\}_{x\in A}$, *i.e.*

$$\Sigma x:A.B(x) = \{\langle a,b\rangle \in A \times \bigcup_{x\in A} B(x) \mid a\in A \ \&\ b\in B(a)\}$$

Elements of the general sum have the form of pairs, where the first element, called the *witness* or *index* determines the type of the second element. Elements of the general sum are constructed by a primitive injection function

$$\mathbf{inj}: \Pi a:A.(B(a)\to\Sigma x:A.B(x))$$

and they can be analyzed by using the two primitive projection functions

$$\mathbf{witness}: (\Sigma x:A.B(x))\to A$$

$$\mathbf{out}: \Pi p:(\Sigma x:A.B(x)).B(\mathit{witness}\ p)$$

Note that the existence of these projection functions (corresponding roughly to Martin-Löf's E operation) make the general sum an "open" construct, in contrast to the existential type of SOL or the categorical sum (see [MP85], §2.6).[4] In the degenerate case where $B(x)$ is independent of x, the general sum is isomorphic to the ordinary binary Cartesian product *AcrossB*.[5]

In the following sections we will sometimes take the liberty of saying simply "product" or "sum" when we mean "general product" and "general sum."

2.2. Small and large types

The stratified type system we will be working with is basically a simplified version of the type system described in [CZ84]. It has several (in fact infinitely many) levels, though only the first two or three will be mentioned here. At the bottom of the hierarchy are the *small* types, contained in the level 1 type universe $\mathbf{Type}_1$. The small types are generated from the customary primitive types *int, bool,* ... by closing under "first-order" general products and sums (*i.e.* $\Pi x:A.B(x)$ and $\Sigma x:A.B(x)$ where $A:\mathbf{Type}_1$ and $\lambda x:A.B(x):A\to\mathbf{Type}_1$, including their degenerate forms $\to$ and $\times$) and perhaps other constructions such as recursion.[6]

$\mathbf{Type}_1$ serves as a type of all small types, but it is not itself a small type. It resides in the level 2 universe of "large types," $\mathbf{Type}_2$, which in turn is a "very large type" belonging to the next universe $\mathbf{Type}_3$, and so on. The type universes are cumulative, so $\mathbf{Type}_2$ also contains all the small types. $\mathbf{Type}_2$ contains other large types generated from $\mathbf{Type}_1$ using second-order products and sums. For instance, the first-order products and sums can be viewed as operations[7] and as such they have the following large type:

$$\Pi_1,\ \Sigma_1\ :\ \Pi_2 X:\mathbf{Type}_1.(X\to_2\mathbf{Type}_1)\to_2\mathbf{Type}_1\quad :\mathbf{Type}_2$$

where $\to_2$ is the degenerate form of Π_2 (which has an analogous type in $\mathbf{Type}_3$). Note that as elements of a large type in $\mathbf{Type}_2$, Π_1 and Σ_1 are considered level 1 objects even though they do not belong to the basic type universe $\mathbf{Type}_1$.

The existential and universal types of SOL correspond to the following large types:

$$\forall t.\sigma(t) \ \approx\ \Pi_2 t:\mathbf{Type}_1.\sigma(t)\ :\mathbf{Type}_2$$

$$\exists t.\sigma(t) \ \approx\ \Sigma_2 t:\mathbf{Type}_1.\sigma(t)\ :\mathbf{Type}_2$$

The elements of these large types are, respectively, the (first-order) polymorphic functions and the Σ_2-structures, which are the open analogues of SOL's existential structures (we will call them simply "structures" when there is no danger of confusion). Being elements of large types, polymorphic functions and Σ_2-structures are also level 1 objects, *i.e.* they are of the same level as small types. This means that neither polymorphic functions nor Σ_2-structures can be manipulated as ordinary values (which are level 0 objects).

types isomorphism).

[4] A "closed" version of the general sum, analogous to SOL's existential type, can be derived from the general product [Pra65], but the open version used here and in ITT appears to be an independent primitive notion.

[5] General sums have also been called "indexed sums," "disjoint unions," and "dependent products" (an unfortunate clash with the "general product" terminology). Other notations used include $x:A \times\times B(x)$ [BL84] and $\exists x:A.B(x)$ (from the formulas as types isomorphism).

[6] The simpler forms of type language will not admit variables ranging over values and only constant functions B will be definable. Under these circumstances the first-order general product and sum always reduce to their degenerate forms $A\to B$ and $A\times B$.

[7] With, *e.g.*, $\Pi x:A.B(x) = \Pi_1(A)(\lambda x:A.B(x))$.

We will in fact think of Σ-structures as a generalized form of small type.

The level 2 general sum operation Σ_2 and its associated primitive operations actually have very general polymorphic types:

$$\Sigma_2 : \Pi_3 A : \mathbf{Type}_2 \,.\, (A \to_3 \mathbf{Type}_2) \to_3 \mathbf{Type}_2 \quad : \mathbf{Type}_3$$

$$\mathbf{inj}_2 : \Pi_3 A : \mathbf{Type}_2 \,.\, \Pi_3 B : (A \to \mathbf{Type}_2) \,.\, \Pi_2 x : A \,.\, (B(x) \to_2 \Sigma_2(A)(B) \quad : \mathbf{Type}_3$$

The corresponding types for **witness**$_2$ and **out**$_2$ are left as exercises. The basic structure expression

$$\mathbf{rep}_{\exists t.\sigma(t)} \tau P \; : \exists t.\sigma(t)$$

translates into the following

$$\mathbf{inj}_2(\mathbf{Type}_1)(\lambda t : \mathbf{Type}_1 \,.\, \sigma(t))(\tau)(P) : \Sigma_2 t : \mathbf{Type}_1 \,.\, \sigma(t)$$

which we will often abbreviate to $\mathbf{inj}_2\, \tau\, P$ when the polymorphic parameters $\mathbf{Type}_1$ and $\lambda t.\sigma(t)$ clear from the context. Note that because of the generality of Σ_2, we may also create structures with structures rather than types as witnesses (or even with polymorphic functions as witnesses, though we won't pursue this possibility here). We will exploit this generality in the language described in the next section.

The rules for type checking in this system are conventional, consisting of the appropriate generalizations of the usual introduction and elimination rules at each level, together with additional rules to deal with β-conversion and definitional equality.

3. A simple Pebble-like language

We will now describe a fairly simple language which is intended to isolate a useful subset of the ramified type system sketched in the previous section. We will call this language DL, just to have a name for it. DL resembles Pebble in having explicit dependent types, but because of its ramified nature it is closer in spirit to ML and the module facilities of [Mac85].

3.1. Small types

The base type language of DL will be a simplified version of that of ML. For simplicity, we omit recursive types, but add a labeled product to express types of value environments. Type expressions, represented by the metavariable *texp*, have the following abstract syntax:

$$texp ::= bool \mid int \mid real \mid tvar \mid texp \times texp' \mid \langle id_1 : texp_1, \ldots, id_n : texp_n \rangle \mid texp \to texp' \mid witness(svar)$$

where *tvar* ranges over type variables and *svar* over structure variables.[8] The actual small types of DL correspond to the closed (i.e. variable free) type expressions, and this class is denoted simply by **Type** (short for $\mathbf{Type}_1$).

3.2. Signatures

The class of signatures is obtained by starting with $\mathbf{Type}_1$ and closing with respect to the Σ_2 operator. This gives a class of types characterizing the union of small types and "abstraction-free" Σ_2-structures (*i.e.* those that do not contain any second-order lambda abstractions). Rather than use the Σ_2 operator directly, we give a little grammar for signatures that covers the cases of interest:

$$sig ::= \mathbf{Type} \mid \Sigma svar : sig . texp \mid \Sigma svar : sig . sig'$$

where Σ is short for Σ_2. Typically, the *texp* forming the body of a signature is a labeled product type specifying a collection of named functions and other values. Note that if *sig* is **Type** in either of the Σ forms, the structure variable is actually a type variable, so structure variables subsume type variables. Note also that in a signature such as $\Sigma s : A . B(s)$, the structure variable s can appear in B only as a component of a type subexpression. It can appear either directly, if $A = \mathbf{Type}$, or else in a subexpression "**witness**(...s...)," formed by nested application of **witness** and **out** and denoting a small type.

[8] For *witness(svar)* to be proper small type, *svar* should be restricted to range over structures with **Type** witnesses.

3.3. Structures

In DL, the term "structure" is used in a somewhat broader sense than above to match the notion of signature. DL Structures may be either small types or nested Σ_2-structures. As in the case of signatures, we substitute some syntax for the use of the **inj** primitive in its full generality. The syntax of structure expressions naturally follows that of signatures, *viz.*

$$sexp ::= svar \mid texp \mid \mathbf{inj}\ sexp\ exp \mid \mathbf{inj}\ sexp\ sexp'$$

where *svar* ranges over structure variables and *exp* ranges over ordinary value expressions. We will not further specify *exp* other than to say that it includes labeled tuples (called *bindings* in Pebble) expressing elements of labeled products, and expressions of the form "**out**(...*svar*...)," formed by nested application of **witness** and **out** and denoting a value of type depending on the signature of *svar*.

3.4. Functors

We allow (second-order) lambda-abstraction of structure expressions with respect to structure variables to form functions from structures to structures. Following [Mac85], we will call such abstractions *functors*. We will allow nested abstractions, yielding "curried" functors. The type of a functor is a general product that we will call a *functor signature*.

The abstract syntax of functor signatures and functor expressions is

$$msig ::= \Pi svar: sig.sig \mid \Pi svar: sig.msig$$

$$mexp ::= \lambda svar: sig.sexp \mid \lambda svar: sig.mexp$$

where Π represents Π_2. The syntax of structure expressions must be extended to encompass functor applications by adding

$$sexp ::= mexp(sexp)$$

The restrictions embodied in the structure and functor syntax amount to saying that structures cannot have functors as components, nor can functors have functors as arguments. In other words, functors are restricted to be essentially "first-order" mappings over structures. These restrictions are partly a reflection of certain tentative principles for programming with parametric modules, and partly an attempt to simplify implementation of the language. Further experience with parametric modules (functors) and their implementation should help refine our ideas about what restrictions on the full type theory are pragmatically justified.

4. Dependence, abstraction, and signature closure

This section considers some of the interactions that occur as structures and functors are defined in terms of one another. The interactions between Π-abstraction and hierarchical chains of definitions, particularly those involving sharing, are particularly subtle and interesting.

The definition of a new structure will frequently refer to existing ones, setting up various forms of dependency between the new structure and the older structures mentioned in its definition (known as its *antecedents*). For instance, suppose *CPoint* (short for *CartesianPoint*, perhaps) is an existing structure of signature *Point* and we define a new rectangle structure *CRect* in terms of *CPoint* as follows:

```
RectWRT(P:Point) =
    Σrect:Type.⟨mk_rect: |P| × |P| → rect,
                topleft: rect → |P|,
                botright: rect → |p|⟩

CRect: RectWRT(CPoint) =
    inj (|CPoint| × |CPoint|)
        ⟨mk_rect = λ(tl, br).(tl, br),  · · · ⟩
```

Here the dependence of *CRect* on *CPoint* is explicitly indicated by the fact the the name *CPoint* appears free in the signature of *CRect*. In such cases of overt dependency significant use of the dependent structure usually requires access to the referenced structures as auxiliaries. In this instance the manipulation of rectangles using *CRect* is very likely to entail the manipulation of associated points using *CPoint*.

In other cases the dependency between a structure and one of its antecedents may be tacit rather than overt, as when a structure B is defined in terms of a structure expression $str_B(A)$ but A does not appear in the signature of B. This generally occurs when A is used for purely internal purposes in the implementation of B and therefore is not relevant to the *use* of B. The structures on which a structure overtly depends, *i.e.* those referred to in its signature, will be called its *supporting* structures, or more briefly, its *support*.

If we have an overt dependency, such as

$$B = str_B(A) : sig_B(A)$$

where $A : sig_A$, there are two ways of making B self-sufficient relative to A, both of which have the effect of closing the signature $sig_B(A)$ with respect to A. One method is to abstract with respect to A, thus turning B into a functor:

$$B^{\pi} = \lambda A : sig_A \,.\, str_B(A) \ : \Pi A : sig_A \,.\, sig_B(A)$$

whose signature is the Π-closure of sig_B with respect to A.

The other alternative is to incorporate A as the witness component in a Σ-structure with B as the body, yielding the Σ-closure of sig_B as signature:

$$B^{\sigma} = \mathbf{inj}\ A\ str_B(A) \ : \Sigma X : sig_A \,.\, sig_B(X)$$

Note that the B^{π} closure is no longer a structure. In order to get a usable structure we have to apply it to a structure expression, thus recreating the original situation of overt dependency, as in

$$B' = B^{\pi}(F(G(X))) : sig_B(F(G(X))).$$

On the other hand, B^{σ} is truly self-contained, at least so far as A is concerned, and is usable as it stands because it incorporates the necessary supporting structure A within itself. In ML, A is called a *substructure* of B^{σ}.

Now consider what happens when there is a chain of dependencies such as

$$A = str_A : sig_A$$

$$B = str_B(A) : sig_B(A)$$

$$C = str_C(A, B) : sig_C(A, B)$$

and we wish to abstract C with respect to its supporting structures. There are three different ways to do this: (1) full abstraction with respect to all supporting structures:

$$\begin{aligned} MkC_1 &= \lambda A : sig_A \,.\, \lambda B : sig_B(A) \,.\, str_C(A, B) \\ &\quad : \Pi A : sig_A \,.\, \Pi B : sig_B(A) \,.\, sig_C(A, B) \end{aligned}$$

(2) abstraction with respect to B, with a residual dependence on the fixed A:

$$\begin{aligned} MkC_2 &= \lambda B : sig_B(A) \,.\, str_C(A, B) \\ &\quad : \Pi B : sig_B(A) \,.\, sig_C(A, B) \end{aligned}$$

and (3) abstraction of both B and C with respect to A:

$$MkB = \lambda A : sig_A \,.\, str_B(A) : \Pi A : sig_A \,.\, sig_B(A)$$

$$\begin{aligned} MkC_3 &= \lambda A : sig_A \,.\, str_C(A, MkB(A)) \\ &\quad : \Pi A : sig_A \,.\, sig_C(A, MkB(A)) \end{aligned}$$

Now suppose that we first Σ-close B with respect to A, obtaining

$$B' = \mathbf{inj}\ A\ (str_B(A)) : sig_{B'} = \Sigma X : sig_A \,.\, sig_B(A)$$

Then abstracting C with respect to B' gives

$$\begin{aligned} MkC' &= \lambda B' : sig_{B'} \,.\, \mathbf{inj}\ B'\ str_C(|B'|, \mathbf{out(B')}) \\ &\quad : \Pi B' : sig_{B'} \,.\, sig_C(|B'|, \mathbf{out}(B')) \end{aligned}$$

If we both Σ-close C with respect to B' *and* abstract with respect to B' we get

$$MkC' = \lambda B' : sig_{B'} \,.\, \mathbf{inj}\ B'\ (str_C(|B'|, \mathbf{out}(B'))) \ : \Pi B' : sig_{B'} \,.\, sig_{C''}$$

where $sig_{C''} = \Sigma B' : sig_{B'} \,.\, sig_C(|B'|, \mathbf{out}(B'))$. The rules of type equality will insure that for all structures $S : sig_{B'}$, $|MkC'(S)| = S$, even though the relation between the argument and result of MkC' is not manifest in its signature.

Note that when B was Σ-closed to form B' the support of C was coalesced into a single structure, which made it easier to fully abstract C with respect to its support. When there are many levels of supporting structures this efficiency of abstraction becomes a significant advantage. On the other hand, it became impossible to abstract with respect to B' while leaving A fixed, because A had become a component of B.

The final example illustrates the interplay between sharing and abstraction. Suppose structures A, B, C, and D are related as follows:

$$A = str_A : sig_A$$

$$B = str_B(A) : sig_B(A)$$

$$C = str_C(A) : sig_C(A)$$

$$D = str_D(A, B, C) : sig_D(A, B, C)$$

i.e. D depends on A, B, and C while B and C both depend on A. If we fully abstract D with respect to its support we have

$$MkD = \lambda A : sig_A . \lambda B : sig_B . \lambda C : sig_C . str_D(A, B, C) \;:\; \Pi A : sig_A . \Pi B : sig_B . \Pi C : sig_C . sig_D(A, B, C)$$

If, on the other hand, we first Σ-close B and C with respect to A and then abstract D with respect to its support, we get

$$B' = \mathbf{inj}\ A\ str_B(A) : sig_{B'} = \Sigma X : sig_A . sig_B(A)$$

$$C' = \mathbf{inj}\ A\ str_C(A) : sig_{C'} = \Sigma X : sig_A . sig_C(A)$$

$$\begin{aligned} MkD' = {} & \lambda B' : sig_{B'} . \lambda C' : sig_{C'} . str_D(|B'|, \mathbf{out}(B'), \mathbf{out}(C')) \\ & : \Pi B' : sig_{B'} . \Pi C' : sig_{C'} . sig_D(|B'|, \mathbf{out}(B'), \mathbf{out}(C')) \text{—}\mathbf{sharing}\, |B'| = |C'| \end{aligned}$$

In the type of *MkD'* something new has been added. The way that B and C support the definition of D probably depends on the fact that B and C share the same support A (think of B and C as rectangles and circles, and A as points, for example). For *MkD* this sharing is directly expressed by the signature, but this is not the case for *MkD'*, so a special sharing constraint must be added to the signature.

Two styles of modular programming have been illustrated here. The first, which is favored in Pebble, expresses dependencies by allowing structure names to appear in the signatures of other structures, and tends to abstract directly and individually on each supporting structure. The other style is representative of modules in ML. It involves forming Σ-closures to capture dependencies and coalesce the support of structures into one level. In fact, the ML module language goes so far as to require that all signatures be Σ-closed, even the argument and result signatures of functors. There are several other factors involved which indirectly support this strict closure rule. In particular, ML's "generative" declarations of datatypes and exceptions, and the fact that structures can contain state, make it necessary to maintain fairly rigid relations between structures. In addition, Σ-closed structures appear to be more appropriate units for separate compilation and persistent storage.

5. Conclusions

The main thrust of this work is that a ramified type system with general dependent type constructs is an effective tool for the analysis and design of programming language type systems, particularly those oriented toward programming in the large. We have explored some of the design choices that have been raised by recently proposed languages such as Pebble, SOL, and Standard ML with modules. But many important questions remain to be answered. For instance, we need to have precise characterizations of the relative strengths of predicative vs impredicative type systems, and reflexive vs irreflexive systems. It would be desirable to have a representation independence result analogous to that of Mitchell [Mit86] for the stratified system used here. Finally, it appears that the basic polymorphic type system of ML [Mil78] is in fact a ramified system, and that the system described in §2, rather than the second order lambda calculus, can be viewed as its most natural generalization.

References

[BC85] J. L. Bates and R. L. Constable, *Proofs as Programs*, ACM Trans. on Programming Languages and Systems, 7, 1, January 1985, pp 113-136.

[BDD80] H. Boehm, A. Demers, and J. Donahue, *An informal description of Russell*, Technical Report TR 80-430, Computer Science Dept., Cornell Univ., October 1980.

[BL84] R. M. Burstall and B. Lampson, *A kernel language for abstract data types and modules*, in Semantics of Data Types, G. Kahn, D. B. MacQueen, and G. Plotkin, eds., LNCS, Vol 173, Springer-Verlag, Berlin, 1984.

[Bur84] R. M. Burstall, *Programming with modules as typed functional programming*, Int'l Conf. on 5th Generation Computing Systems, Tokyo, Nov. 1984.

[Car85] L. Cardelli, *The impredicative typed λ-calculus*, unpublished manuscript, 1985.

[CF58] H. B. Curry and R. Feys, **Combinatory Logic I**, North-Holland, 1958.

[CH85] T. Coquand and G. Huet, *A calculus of constructions*, Information and Control, to appear.

[CM85] L. Cardelli and D. B. MacQueen, *Persistence and type abstraction*, Proceedings of the Appin Workshop on Data Types and Persistence, Aug 1985, to appear.

[CW85] L. Cardelli and P. Wegner, *On understanding types, data abstraction, and polymorphism*, Technical Report No. CS-85-14, Brown University, August 1985.

[CZ84] R. L. Constable and D. R. Zlatin, *The type theory of PL/CV3*, ACM Trans. on Programming Languages and Systems, 6, 1, January 1984, pp. 94-117.

[deB80] N. G. de Bruijn, *A survey of project AUTOMATH*, in To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism, Academic Press, 1980, pp. 579-607.

[DD85] J. Donahue and A. Demers, *Data Types are Values*, ACM Trans. on Programming Languages and Systems, 7, 3, July 1985, pp. 426-445.

[Gir71] J.-Y. Girard, *Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types*, in Second Scandinavian Logic Symposium, J. E. Fenstad, Ed., North-Holland, 1971, pp. 63-92.

[Hoo84] J. G. Hook, *Understanding Russell--a first attempt*, in Semantics of Data Types, G. Kahn, D. B. MacQueen, and G. Plotkin, Eds., LNCS Vol 173, Springer-Verlag, 1984, pp. 69-85.

[How80] W. Howard, *The formulas-as-types notion of construction*, in To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism, Academic Press, 1980, pp. 476-490. (written 1969)

[Mac85] D. B. MacQueen, *Modules for Standard ML (Revised)*, Polymorphism Newsletter, II, 2, Oct 1985.

[McC79] N. J. McCracken, *An investigation of a programming language with a polymorphic type structure*, Ph.D. Thesis, Computer and Information Science, Syracuse Univ., June 1979.

[M-L71] P. Martin-Löf, *A theory of types*, unpublished manuscript, October 1971.

[M-L74] P. Martin-Löf, *An intuitionistic theory of types: predicative part*, Logic Colloquium 73, H. Rose and J. Shepherdson, Eds., North-Holland, 1974, pp. 73-118.

[M-L82] P. Martin-Löf, *Constructive mathematics and computer programming*, in **Logic, Methodology and Philosophy of Science, VI**, North-Holland, Amsterdam, 1982, pp. 153-175.

[MR86] A. R. Meyer and M. B. Reinhold, *'Type' is not a type*, 13th Annual ACM POPL Symposium, St. Petersburg, January 1986.

[MIL78] R. Milner, *A theory of type polymorphism in programming*, JCSS, 17,3, Dec 1978, pp. 348-375.

[Mit86] J. C. Mitchell, *Representation independence and data abstraction*, 13th Annual ACM POPL Symposium, St. Petersburg, January 1986.

[MP85] J. C. Mitchell and G. D. Plotkin, *Abstract types have existential types*, 12th ACM Symp. on Principles of Programming Languages, New Orleans, Jan. 1985, pp. 37-51.

[Rey74] J. C. Reynolds, *Towards a theory of type structure*, in **Colloquium sur la Programmation**, Lecture Notes in Computer Science, Vol. 19, Springer Verlag, Berlin, 1974, pp. 408-423.

[Sco70] D. Scott, *Constructive Validity*, in Symposium on Automatic Demonstration, Lecture Notes in Math., Vol 125, Springer-Verlag, 1970, pp. 237-275.