Towards a Theory of Packages

Snorri Agnarsson
M. S. Krishnamoorthy

Rensselaer Polytechnic Institute
Department of Computer Science
Troy, New York 12181, U.S.A.

ABSTRACT

A model for packages is introduced, along with operations for their manipulation. The model is based on the unifying principle that programs should be represented by trees, and packages by substitutions on trees. Operations are defined on packages, that allow the construction of any package from a collection of basic packages. A programming environment, based on this model, would allow manipulations and operations that are not possible in current languages. Information hiding and encapsulation are automatically supported by the model. A typing mechanism is presented, which allows polymorphic types. The typing does not affect the typeless aspect of the model.

1. Introduction

Emphasis on software maintainability and reusability has grown in recent years. It has been realized that the programmer and software designer needs more precise and powerful tools. The notion of packages or modules has developed as an answer to this need.

A package or module is a collection of items, that are closely related, ideally in such a way that its implementation can be changed without regard to other packages, if functionality is preserved.

In the following sections, a model for packages will be introduced. This model is based on viewing programs as infinite trees. A common everyday example of an infinite tree is a flowchart (a flowchart with backward references is infinite in the sense that its "expansion" is infinite).

A regular tree is an infinite tree with a finite number of subtrees (a flowchart has a finite number of boxes). An excellent introduction to regular trees, and matters relating to infinite trees in general can be found in [Courcelle-83].

The function of a package is to define previously undefined symbols. Therefore we can view packages as a form of *substitution*.

By viewing programs as regular trees, and packages as substitutions on regular trees, we gain mathematical precision. The following advantages are achieved:

1. We can define *operations* on packages, that allow us to construct new packages, and *"do algebra"* with packages.
2. We can formulate and prove theorems about the *algebraic properties* of packages.

The authors were partly supported by NSF grant MCS 83-14600.

3. We can more easily think about packages in a *language independent* fashion.
4. Using the model as an intermediary step between a programming language and its implementation, we are better able to *separate our concerns* in language design and implementation. We can use a similar mathematical model for different languages, and different implementations of the same language.
5. The model can also serve as a step between the programming language and its *semantics*, by defining the semantics of the resulting trees, rather than directly defining the semantics of the language. Thus, the same semantic model can be used for different languages, if they both use the same infinite tree model.
6. *Generic* programs and packages are natural consequences of the model.
7. The model can be augmented with *typing* in a formal fashion, and still allow the same operations as before.
8. The operations defined on packages will often allow the programmer to manipulate packages in ways that would otherwise require extensive editing of source code. A programming environment using those package operations would give the programmer powerful tools, and encourage the construction and use of *reusable* modules or packages. The operations on packages allow the programmer to build his programs from independently written packages.
9. The package operations define the scope of each name (function, subroutine, external variable) explicitly, in an understandable way.
10. The package operations can be used to give a theoretical basis for linkage editing. For example, we can prove that incremental linking yields the same result as linking everything at once. Since the proper operation of linkage editors is essential to good implementations of packages, it is useful to have a formal groundwork for their functions.

2. Original Contributions

This paper defines a model, in terms of packages, for the binding process. Different aspects of the binding process are identified, and embodied as operations on packages. The iteration operation on substitutions, which is used to model mutually recursive binding, is an original contribution, as is the idea of using substitutions to model packages.

3. Infinite and Regular Trees

In the next two sections, we use substitutions to define our model of untyped packages. Section 5 shows some concrete examples of using a prototype package system for Franz Lisp. Section 6 introduces typed packages, with strong polymorphic typing.

Every programmer uses regular trees, though some will not know them by that name. Given a program made up of diverse procedures, each procedure can be viewed as a node in a graph, and each call from procedure A to procedure B as a directed edge from node A to node B. Using this metaphor, the linking of non-recursive procedures results in a finite tree, with the main procedure at the root. When recursion is allowed, the result is a more general directed graph, which can be viewed as an infinite regular tree. Thus a linkage-editor is a constructor of regular trees. Similarly, control structures such as repeat-until and while-do, can be viewed as constructors of regular trees.

The notation, used in the following sections, is similar to the one used in [Courcelle-83].

Definition: A ranked alphabet is a pair $E = <A, \rho>$, where A is a set of symbols, and ρ is a function from A to the non-negative integers, $\rho: A \rightarrow \mathbf{N}$. Given $a \in A$, with $\rho(a) = n$, we say that a is a member of E, with *arity* n.

Definition: A finite tree over a ranked alphabet E is defined by the following:

a) If e is a member of E, with arity n, and $t_1, \ldots, t_n \in M(E)$, then $e(t_1, \ldots, t_n) \in M(E)$

b) All elements of $M(E)$ can be constructed by applying the above rule a finite number of times. □

Notation: If $a \in E$ has arity zero, then we use 'a' interchangeably with '$a()$' to denote the corresponding tree. □

Notation: Given an alphabet E, $M(E)$ is used to denote the set of finite trees over E. □

Starting from the above definition of finite trees, infinite trees can be defined, either by using an order-theoretic approach, or a metric-theory approach [Courcelle-83].

Notation: $M^{\infty}(E)$ is used to denote the set of infinite trees over an alphabet E. □

Definition: Given trees t and t', t is a subtree of t' if and only if one of the following holds:

a) $t = t'$

b) $t = e(t_1, \ldots, t_n)$, and t' is a subtree of t_i, for some i. □

Not all infinite trees are regular trees. The following definition characterizes regular trees.

Definition: A tree $t \in M^{\infty}(E)$ is regular if and only if the set

$$\{t' \in M^{\infty}(E) \mid t' \textit{ is a subtree of } t\}$$

is finite. □

Thus the infinite tree $f\ (f\ (f\ (...)))$ is regular, since it has only one subtree (namely itself).

Notation: We use $R\,(E)$ to denote the set of regular trees over an alphabet E. □

Regular trees are an important subclass of infinite trees because they are relatively easy to work with, and they correspond directly to many of our notions in programming.

Notation: Given ranked alphabets E and V, with all elements of V having arity zero, we use $M\,(E,V)$, $M^{\infty}(E,V)$, and $R\,(E,V)$ to denote $M\,(E \cup V)$, $M^{\infty}(E \cup V)$, and $R\,(E \cup V)$, respectively, with the understanding that the elements of V are variables, subject to substitution. □

Definition: *Given a tree t, in one of $M\,(E,V)$, $M^{\infty}(E,V)$, or $R\,(E,V)$, we define its set of variables, denoted $Vars\,(t)$ by:*

$$Vars\,(t) = \{v \in V \mid v \text{ is a subtree of } t\}$$

□

Definition: A substitution on $M\,(E,V)$, $M^{\infty}(E,V)$, or $R\,(E,V)$, is a function $\sigma: V \to M\,(E,V)$, $\sigma: V \to M^{\infty}(E,V)$, or $\sigma: V \to R\,(E,V)$, respectively, such that its *domain*, $Dom\,(\sigma) = \{v \in V \mid \sigma(v) \neq v\}$ is finite. Substitutions are written on the form $\{v_1 \to t_1, \ldots, v_n \to t_n\}$, where the trees $t_1, \ldots, t_n$ are members of $M\,(E,V)$, $M^{\infty}(E,V)$, or $R\,(E,V)$, respectively, and $v_1, \ldots, v_n \in V$. We use $t\,\sigma$ to denote the result from applying the substitution σ to the tree t. The result is defined by:

$$t\,\sigma = \begin{cases} f\,(t_1\sigma, \ldots, t_n\sigma) & \text{if } t = f\,(t_1, \ldots, t_n),\ f \notin V \\ \sigma(v) & \text{if } t = v,\ v \in V \end{cases}$$

The *range* of σ is denoted by $Ran\,(\sigma)$, and defined by:

$$Ran\,(\sigma) = \bigcup_{v \in Dom\,(\sigma)} Vars\,(v\,\sigma)$$

□

In the following, substitutions are used to define packages, but first, we need to define some operations on substitutions.

The composition of substitutions is a well-known operation: given substitutions σ and γ, $\sigma \circ \gamma$ is a substitution that has the property that for any tree t, $t\,(\sigma \circ \gamma) = (t\,\sigma)\gamma$. We will also need a few other operations:

Definition: Given substitutions σ and γ, with disjoint domains, *join* of σ and γ, denoted by $\sigma + \gamma$, is a substitution defined by its operation on variables:

$$v\,(\sigma + \gamma) = \begin{cases} v\,\sigma & \text{if } v \in Dom\,(\sigma) \\ v\,\gamma & \text{if } v \in Dom\,(\gamma) \\ v & \text{otherwise} \end{cases}$$

If the domains of σ and γ are not disjoint, $\sigma + \gamma$ is not defined. □

Informally, we can write

$$\{x_1 \to t_1, \ldots, x_n \to t_n\} + \{y_1 \to u_1, \ldots, y_m \to u_m\} = \{x_1 \to t_1, \ldots, x_n \to t_n, y_1 \to u_1, \ldots, y_m \to u_m\}$$

Definition: Given a substitution σ and a set N of variables, the *restriction* of σ to N, denoted by $\sigma \mid_N$, is defined by its effect on variables:

$$v\,(\sigma \mid_N) = \begin{cases} v\,\sigma & \text{if } v \in N \\ v & \text{otherwise} \end{cases}$$

□

Informally,

$$\{x_1 \to t_1, \ldots, x_n \to t_n, x_{n+1} \to t_{n+1}, \ldots, x_{n+m} \to t_{n+m}\} \mid_{\{x_1, \ldots, x_n\}} =$$

$$\{x_1 \to t_1, \ldots, x_n \to t_n\}$$

Definition: Non-negative powers of a substitution σ are defined in the obvious way:

a) $\sigma^0 = \{\}$

b) $\sigma^{n+1} = \sigma \circ \sigma^n$ □

Given a metric on trees, a metric on substitutions can be defined, allowing the following definition.

Definition: Given a substitution σ, σ is called *iterable* iff the sequence $\sigma^0,\sigma^1,\sigma^2,\ldots$ is a Cauchy-sequence. If σ is iterable, the iteration of σ, denoted by σ^∞, is defined as the limit of the above sequence. □

Examples of iterations of substitutions are:

$$\{x \rightarrow y\}^\infty = \{x \rightarrow y\}$$

$$\{x \rightarrow f(x)\}^\infty = \{x \rightarrow f(f(f(\cdots)))\}$$

$\{x \rightarrow y, y \rightarrow x\}$ is an example of a non-iterable substitution.

The following theorems relate to iteration of substitutions:

Theorem: If σ is an iterable substitution, and t is any tree, then $t, t\sigma, t\sigma^2,\ldots$ is a Cauchy-sequence, and the limit of the sequence is $t\sigma^\infty$. □

Theorem: A substitution σ is non-iterable iff there exists a variable v, and a positive integer n, such that $v\sigma \neq v$, but $v\sigma^n = v$. □

An important property of regular trees is that if we start out with regular trees, all our trees will remain regular, using the operations we have defined above.

Definition: We call a substitution σ *regular* iff it maps regular trees into regular trees. □

Theorem: The set of regular substitutions is closed under the operations *composition*, *restriction*, *join*, and *iteration*. □

4. Untyped Packages

We now have groundwork needed for formally defining packages. Basically, the packages will be substitutions, but it is convenient to associate a set of variables with each package; that set of variables defines the symbols that the package defines, or exports.

Definition A *package* is a pair $P = \langle N,\sigma\rangle$, where N is a set of variables, and σ is a substitution, such that $\sigma \mid_N = \sigma$. The set N is called the *export* of the package P, denoted by $Exp(P)$. The set of variables in the trees $v\sigma$, where v ranges over N, is called the *import* of P, denoted by $Imp(P)$. $Imp(P) = \bigcup_{v \in N} Vars(v\sigma)$ □

When the export of a package is clear from the context, we will simply use the substitution σ to denote the package $\langle N,\sigma\rangle$.

Definition: Given a tree T, and a package $P = \langle N,\sigma\rangle$, we define the *application* of P to t, denoted tP by:

$$tP = t\sigma$$

□

A package that has imported variables, is similar to a generic package in, e. g. Ada. The imported variables can later be bound in any number of ways, each giving a different instantiation of the same generic package.

A few basic operations are defined on packages:

Definition: The *restriction* of a package $P = \langle N,\sigma\rangle$ to a finite set M of variables is denoted by $P \mid_M$, and is defined by $P \mid_M = \langle N \cap M, \sigma \mid_M\rangle$. □

By restricting a package, one gets rid of some exports, and maybe also some imports.

Definition: The composition of two packages $P_1 = \langle M,\sigma\rangle$ and $P_2 = \langle N,\gamma\rangle$ is defined for any two packages P_1 and P_2. It is denoted by $P_1 \circ P_2$, and is defined by:

$$P_1 \circ P_2 = \langle M \cup N, \sigma \circ \gamma\rangle$$

□

Composing package P_1 with P_2 results in a package that has all the exports of both P_1 and P_2, but imports to P_1 are bound to, i.e. substituted by, exported trees from P_2.

Definition: The *importation* into package $P_1 = \langle N,\sigma\rangle$ from $P_2 = \langle M,\gamma\rangle$, is denoted by $P_1 * P_2$, and defined by:

$$P_1 * P_2 = \langle N, (\sigma \circ \gamma) \mid_N\rangle$$

□

Importation can be used to instantiate different versions of a generic package. Importation can also be defined in terms of composition and restriction:

$$P_1 * P_2 = (P_1 \circ P_2) \mid_{Exp(P_1)}$$

The same symbol can be used in a package to denote both an import and an export, and they will not necessarily denote the same object. This is in contrast to e.g. Ada, where it is not possible to have a package that imports an object under name x, and exports an other object under the same name.

Definition: The join of two packages $P_1 = \langle M, \sigma \rangle$ and $P_2 = \langle N, \gamma \rangle$ is defined iff M and N are disjoint. It is denoted by $P_1 + P_2$, and is defined by

$$P_1 + P_2 = \langle M \cup N, \sigma\gamma \rangle$$

□

The join operation is used to merge two packages. The resulting package exports all the symbols exported by either P_1 or P_2, and imports the symbols imported by either P_1 or P_2.

Definition: The *asymmetric join* of two packages P_1 and P_2 is denoted by P_1 / P_2, and defined by:

$$P_1 / P_2 = P_1 + (P_2 \mid_{Exp(P_2) - Exp(P_1)})$$

□

It can be shown that the asymmetric join is well-defined for any two packages.

Definition: The *iteration* of a package $P = \langle M, \sigma \rangle$ is defined iff σ is iterable. It is denoted by P^∞, and is defined by

$$P^\infty = \langle M, \sigma^\infty \rangle$$

□

As with substitutions, the iteration of a package P can be thought of as being equivalent to $P \circ P \circ \ldots$, ad infinitum.

A real-life example of the use of an iteration operation is when a linkage-editor is used to bind external subroutines. If the subroutines are mutually recursive, this may result in the construction of an infinite regular tree.

If $P_1, \ldots, P_n$ are packages with disjoint exports, then $(P_1 + \cdots + P_n)^\infty$ resolves all references between the packages, and this is the operation that most linkage-editors or binders perform.

An other example is the binding of label references inside a subroutine, or the construction of control structures. For example one might define a while-do control structure as the iteration $\{s \rightarrow if(c, seq(ss, s))\}^\infty$, where c is the boolean condition, ss is the subordinate statement, and s is the statement being constructed. 'seq' is a sequencing operator, and 'if' is a conditional operator. Other control structures can be similarly constructed. For example a looping structure with an embedded exit might be defined as $\{s \rightarrow seq(ss_1, if(c, seq(ss_2, s)))\}^\infty$.

In current programming languages and linkage editors, iteration is an implicit operation, and the programmer does not have control over its application. It is better to put iteration under the control of the programmer. This allows greater freedom in resolving references.

For example one may have a generic package, and wish to bind some of the imported symbols to some of the exported symbols. In Ada this can not be done, but using renaming and iteration, this would be possible.

Iteration is also a kind of fixpoint operation; P^∞, if it exists, is one of the fixpoints of the package mapping $m(Q) = P * Q$. The iteration is always unique, however, while the above mapping will not have a unique fixpoint unless the iteration has no exports (this is easily seen, by noting that $P^\infty * R$, for any package R, is a fixpoint of the above mapping).

Having an explicit iteration operation, instead of an implicit one, gives the programmer greater power. With appropriate renaming of imported or exported symbols, iteration can be used to resolve any imported symbol with any exported symbol. In Ada this would be analogous to being able to bind a parameter of a generic package to an exported item. Ada does not allow such an operation, so our model is in this instance more powerful than Ada. In fact we are not aware of any programming language that would allow such a package operation without editing the source code of the package.

Using the operations defined above, it is easy to do some things that may be awkward to do by traditional methods. Operations that might require extensive editing of a source file can be achieved by safer and easier means. A modified version of a package can be constructed without editing a copy of the original. This has obvious advantages in maintaining a software system.

Because of their simplicity, the above package operations do not necessarily require implementations that incur run-time penalties for the use of packages.

Given a basic collection of packages, of the form $\{x \rightarrow f\}$ or $<\{x\},\{x \rightarrow f(y_1, \ldots, y_n)\}>$, for each f in an alphabet E, depending on arity ($x, y_1, \ldots, y_n$ are distinct variables), and also packages of the form $<\{x\},\{x \rightarrow y\}>$, for each pair of variables x, y, then any package can be constructed by a finite number of applications of the above basic operations.

Notation: To save some parentheses in our notation, we prioritize our binary package operations as follows: The unary operations, restriction and iteration, are performed first. Then importation is performed, then composition, then join, then asymmetric join. Also, the binary operations are performed from left to right.

Thus, we get, for example, $P*Q \circ R \mid_N = (P*Q) \circ (R \mid_N)$, $P*Q+R = (P*Q)+R$, $P+Q / R = (P+Q)/ R$, $P*Q*R = (P*Q)*R$ □

The following theorem describes some algebraic properties of packages:

Theorem: Let P, P_1, P_2, and P_3 be packages. Let I be the empty package, $I = <\emptyset,\{\}>$. Then the following equations hold for all instances where one side of the equations are defined, and if one side of an equation is defined, then the other side is defined:

$$(P_1+P_2)^\infty = (P_1*P_2^\infty)^\infty+(P_2*P_1^\infty)^\infty \quad [1]$$

$$(P_1+P_2)*P_3 = P_1*P_3+P_2*P_3 \quad [2]$$

$$P*P^\infty = P^\infty = P \circ P^\infty \quad [3]$$

$$P^\infty*P = P^\infty = P^\infty \circ P \quad [4]$$

$$(P_1+P_2)^\infty = (P_1^\infty+P_2)^\infty \quad [5]$$

$$P_1+P_2 = P_2+P_1 \quad [6]$$

$$P_1+(P_2+P_3) = (P_1+P_2)+P_3 \quad [7]$$

$$P*I = P = P \circ I = I \circ P \quad [8]$$

$$P+I = P \quad [9]$$

$$I*P = I \quad [10]$$

$$P_1*(P_2 \circ P_3) = (P_1*P_2)*P_3 \quad [11]$$

$$P_1 \circ P_2 = (P_1*P_2)/ P_2 \quad [12]$$

$$(P_1/ P_2)*P_3 = P_1*P_3/ P_2*P_3 \quad [13]$$

□

As an example of the usefulness of the above formulas, formula [1] above implies that iteration can be implemented with a "divide and conquer" algorithm, while formula [5] states that incremental linking yields the same result as linking everything at once.

The above operations have been implemented in a LISP system, along with some extensions to allow more powerful operations. The implementation allows the use of second order substitutions (see [Courcelle-83] for a definition of second order substitutions). Implementation details of this extended version can be found in [Agnarsson-85a].

5. Examples of Using Package Operations

In this section, we will show some examples of the use of untyped packages in Lisp. The examples have been tested in a prototype package implementation, written in Franz Lisp. In the following, variables will be distinguished by having names starting with '&'.

Consider the following two Franz Lisp packages:

```
UnionOrderPackage =
  { &CompareAB →
    (lambda (x y)
        (cond ((&MemberA x) (cond ((&MemberA y) (&CompareA x y))
                                  (t -1)))
              ((&MemberA y) 1)
              (t (&CompareB x y)))))}

PairOrderPackage =
  { &ComparePair →
    (lambda (x y)
        (prog (temp)
            (setq temp (&CompareCar (car x) (car y)))
            (cond ((not (zerop temp)) (return temp)))
            (return (&CompareCdr (cdr x) (cdr y)))))}
```

The package UnionComparePackage imports the predicate &MemberA, and the two functions &CompareA and &CompareB. The predicate &MemberA is intended to be a discriminator between some two sets, A and B. It returns "true" or non-nil if its argument is in A, "false" or nil otherwise. The functions &CompareA and &CompareB compare elements of A and B respectively, and return a numeric value indicating the result. Both sets are assumed to be well-ordered. The package then exports a function &CompareAB, which is a well-ordering on the set $A \cup B$, where all members of A are assumed to be less that all members of B.

The package PairOrderPackage imports two comparators, &CompareCar and &CompareCdr. It then exports a comparator for "cons-cells", implementing a lexical ordering, with the car's of each cell being compared with &CompareCar, and the cdr's with &CompareCdr. Both car's and cdr's are assumed to be well-ordered.

Now let's see what we can do with those two packages, using our package operations, and some simple renaming packages. Let

```
P1 = UnionOrderPackage *
                          {&MemberA → atom,
                           &CompareA → &Compare,
                           &CompareB → &ComparePair}
```

And let

```
P2 = PairOrderPackage *
                         {&CompareCar → &CompareAB,
                          &CompareCdr → &CompareAB}
```

Then define TreeOrderPackage = {&Compare → &CompareAB} * $(P_1 * P_2)^{\infty}$. Then TreeOrderPackage exports a function &Compare, which compares binary trees. It imports a function &Compare, which is used to compare the atoms in the trees. When an atom is compared to a non-atom, the non-atom is considered greater.

Now assume a new package

```
AtomOrderPackage =
  { &Compare →
    (lambda (x y)
        (cond ((eq x y) 0)
              ((alphalessp x y) -1)
              (t 1))))}
```

Now, we can define ATOPackage = TreeOrderPackage * AtomOrderPackage. Then ATOPackage has no imports, but exports a function &Compare, which compares binary trees.

The package operations are particularly easy to implement for Lisp, since Lisp uses a tree representation for programs. However, there is no reason to think that similar operations cannot be implemented to manipulate object files of programs written in other languages, such as Modula-2, CLU, C, Ada, or even FORTRAN.

6. Typed Packages

In this section, we describe a technique for adding strong polymorphic typing to the untyped packages defined above. The typing technique presented is not directly related to any scheme for defining the semantics of our packages, but is intended as a heuristic for finding programming errors. It should not be viewed as the only method of adding typing to packages (or even the best), but merely as one possible way to do it. Before we start defining typed packages, here is an example of what we would like to be able to do:

$$if\ (\mathscr{B}c:Boolean, \mathscr{B}e_1:\mathscr{B}T, \mathscr{B}e_2:\mathscr{B}T):\mathscr{B}T$$

$$\{\mathscr{B}c:Boolean \rightarrow gt\ (\mathscr{B}x:Integer, \mathscr{B}y:Integer),$$

$$\mathscr{B}e_1:Integer \rightarrow \mathscr{B}x:Integer,$$

$$\mathscr{B}e_2:Integer \rightarrow \mathscr{B}y:Integer\} =$$

$$if\ (gt\ (\mathscr{B}x:Integer, \mathscr{B}y:Integer), \mathscr{B}x:Integer, \mathscr{B}y:Integer):Integer$$

The example shows a typed package being applied to a typed tree. $\mathscr{B}T$ is a type variable, or a *polymorphic* type. *Integer* and *Boolean* are type constants. The if-expression is polymorphic, since it can assume many types, depending on what is eventually substituted for the type variable $\mathscr{B}T$. When the package is applied to the tree, the variables $\mathscr{B}e_1$ and $\mathscr{B}e_2$ assume *Integer* type, which causes the resulting if-expression to also assume *Integer* type (since the type of the if-expression is constrained to have the same type as the variables $\mathscr{B}e_1$ and $\mathscr{B}e_2$).

Before we define the general case of typed polymorphic packages, however, we will look at the special case of *monomorphic* packages, which are typed packages, where there are no type variables, and hence no way of defining "multityped" trees.

Definition: A monomorphic typed tree is a triple $s = <e, I, t>$, where e is a tree with no variables, t is a tree, and I is a package, with $Exp(I) = Vars(t)$, and $Imp(I) = \emptyset$. We use $Vars(s)$ to denote $Vars(t)$. We use $ExpType(s)$ to denote e. We use $ImpType(s, v)$ to denote vI. We use $Untyped(s)$ to denote t. □

Given a typed tree $<e, I, t>$, we think of e as the "export type", t as the untyped version of the tree, and vI as the type of the unresolved symbol v.

An example of a typed tree is

$$s = <Statement, \{\mathscr{B}c \rightarrow Boolean, \mathscr{B}s_1 \rightarrow Statement, \mathscr{B}s_2 \rightarrow Statement\}, if\ (\mathscr{B}c, \mathscr{B}s_1, \mathscr{B}s_2)>,$$

which corresponds to an if-then-else statement with variable condition and substatements.

Definition: A monomorphic package is a triple $Q = <E, I, P>$, where E, I and P are packages, $Exp(E) = Exp(P)$, $Exp(I) = Imp(P)$, and $Imp(E) = Imp(I) = \emptyset$. We use $ExpType(Q, v)$ to denote vE. We use $ImpType(Q, v)$ to denote vI. We use $Untyped(Q)$ to denote P. □

We think of a monomorphic package $Q = <E, I, P>$ as a typed version of the package P, where E specifies the types of the exports, and I specifies the types of the imports.

Definition: Given a monomorphic package $Q = <E, I, P>$, and a set N of variables, we define the restriction of Q to N, by:

$$Q \mid_N = <E \mid_N, I \mid_{Imp(P \mid_N)}, P \mid_N>$$

Definition: Given a monomorphic tree $s = <e, I_1, t>$, and a monomorphic package Q, we define the application of Q to s, denoted by sQ, as follows: Let $Q' = <E, I_2, P> = Q \mid_{Vars(s)}$. If there exists a variable $v \in Exp(Q')$, such that $vI_1 \neq vE$, then sQ is undefined. Also, if there exists a variable $v \in Vars(s) \cap (Imp(Q') - Exp(Q'))$, such that $vI_1 \neq vI_2$, then sQ is undefined. Otherwise, we define

$$sQ = <e, (I_2 / I_1) \mid_{Vars(tP)}, tP>$$

□

The rationale behind this definition is that we allow an unresolved symbol in the monomorphic tree s to be substituted for by an export of the monomorphic package Q, if and only if the types match. We consider it an error, if the types do not match. Also, the types of new occurrences of symbols must match types of old occurrences that remain in the result. If the types do not match, we leave the result undefined.

Example: Let

$$s = <Integer,$$

$$\{\mathscr{B}x \rightarrow Integer, \mathscr{B}y \rightarrow Integer, \mathscr{B}u \rightarrow Integer\},$$

$$\max(\mathcal{E}x, \mathcal{E}y, \mathcal{E}u)>$$

and

$$\begin{aligned} Q = <&\{\mathcal{E}x \to Integer, \mathcal{E}y \to Integer, \mathcal{E}z \to Integer\}, \\ &\{\mathcal{E}x \to List(Character), \mathcal{E}y \to List(Character)\}, \\ &\{\mathcal{E}x \to length(\mathcal{E}x), \mathcal{E}y \to length(\mathcal{E}y), \mathcal{E}z \to 7\}> \end{aligned}$$

Then

$$\begin{aligned} Q' &= Q \mid_{Vars(s)} \\ &= <\{\mathcal{E}x \to Integer, \mathcal{E}y \to Integer\}, \\ &\quad \{\mathcal{E}x \to List(Character), \mathcal{E}y \to List(Character)\}, \\ &\quad \{\mathcal{E}x \to length(\mathcal{E}x), \mathcal{E}y \to length(\mathcal{E}y)\}> \end{aligned}$$

Now we note that $ImpType(s, \mathcal{E}x) = Integer = ExpType(Q', \mathcal{E}x)$ and $ImpType(s, \mathcal{E}y) = Integer = ExpType(Q', \mathcal{E}y)$. Therefore the types of the exports of Q match the types of the variables in s. Also, there are no new occurrences introduced of variables present in s. We can therefore deduce that sQ is defined. When we compute sQ according to the formula given above, we get:

$$\begin{aligned} sQ = <&Integer, \\ &\{\mathcal{E}x \to List(Character), \mathcal{E}y \to List(Character), \mathcal{E}u \to Integer\}, \\ &\max(length(\mathcal{E}x), length(\mathcal{E}y), \mathcal{E}u)> \square \end{aligned}$$

Example: Let

$$\begin{aligned} s = <&Integer, \\ &\{\mathcal{E}x \to Integer, \mathcal{E}y \to Integer, \mathcal{E}u \to Integer\}, \\ &\max(\mathcal{E}x, \mathcal{E}y, \mathcal{E}u)> \end{aligned}$$

and

$$\begin{aligned} Q = <&\{\mathcal{E}x \to Integer, \mathcal{E}y \to Integer, \mathcal{E}z \to Integer\}, \\ &\{\mathcal{E}x \to List(Character), \mathcal{E}u \to List(Character)\}, \\ &\{\mathcal{E}x \to length(\mathcal{E}x), \mathcal{E}y \to length(\mathcal{E}u), \mathcal{E}z \to 7\}> \end{aligned}$$

Then

$$\begin{aligned} Q' &= Q \mid_{Vars(s)} \\ &= <\{\mathcal{E}x \to Integer, \mathcal{E}y \to Integer\}, \\ &\quad \{\mathcal{E}x \to Integer, \mathcal{E}u \to List(Character)\}, \\ &\quad \{\mathcal{E}x \to 5, \mathcal{E}y \to length(\mathcal{E}u)\}> \end{aligned}$$

As in the previous example, above, $ImpType(s, \mathcal{E}x) = Integer = ExpType(Q', \mathcal{E}x)$ and $ImpType(s, \mathcal{E}y) = Integer = ExpType(Q', \mathcal{E}y)$. Therefore the types of the exports of Q match the types of the variables in s. But now a new occurrence of the variable $\mathcal{E}u$ would be introduced by applying Q to s. The variable $\mathcal{E}u$ is a member of $Vars(s) \cap (Imp(Q') - Exp(Q'))$, and $ImpType(s, \mathcal{E}u) = Integer \neq List(Character) = ImpType(Q', \mathcal{E}u)$. Therefore sQ is undefined. If sQ were defined, it would have to look something like this:

$$sQ = <Integer, \{\mathcal{E}u \to \cdots\}, \max(5, length(\mathcal{E}u), \mathcal{E}u)>$$

It is therefore apparent that the variable $\mathcal{E}u$ has occurrences that are incompatible with each other. Since we do not allow overloading of variables, this does not make sense. $\square$

6.1. Operations on Monomorphic Packages

We already defined the restriction operation on monomorphic packages, and used it when defining application, above. The join operation has an intuitively obvious definition.

Definition: Given monomorphic packages $Q_1 = <E_1,I_1,P_1>$ and $Q_2 = <E_2,I_2,P_2>$, we define their join Q_1+Q_2, in the following fashion: If $Exp(Q_1)\cap Exp(Q_2) \neq \emptyset$, then Q_1+Q_2 is undefined. If there exists a variable $v \in Imp(Q_1)\cap Imp(Q_2)$, such that $vI_1 \neq vI_2$, then Q_1+Q_2 is undefined. Otherwise Q_1+Q_2 is defined by:

$$Q_1+Q_2 = <E_1+E_2, I_1/\ I_2, P_1+P_2>$$

□

In the above definition, the restriction on the types was to make sure that imports to Q_1+Q_2 would have only one type each. Note that the restrictions on the types ensure that $I_1/\ I_2 = I_2/\ I_1$, so that $Q_1+Q_1 = Q_2+Q_1$.

Example: Let

$$Q_1 = <\{\mathscr{G}x \rightarrow Integer\},$$
$$\{\mathscr{G}p \rightarrow Integer, \mathscr{G}s \rightarrow String\},$$
$$\{\mathscr{G}x \rightarrow \max(\mathscr{G}p, length(\mathscr{G}s))\}>$$

and

$$Q_2 = <\{\mathscr{G}y \rightarrow Integer\},$$
$$\{\mathscr{G}p \rightarrow Integer, \mathscr{G}s \rightarrow Integer\},$$
$$\{\mathscr{G}x \rightarrow \min(\mathscr{G}p, \mathscr{G}s)\}>$$

Then Q_1+Q_2 is undefined, because the import type of $\mathscr{G}s$ in Q_1 does not match the import type of $\mathscr{G}s$ in Q_2. □

Definition: Given monomorphic packages Q_1 and Q_2, we define the composition of Q_1 by Q_2 as follows: Let $Q_1 = <E_1,I_1,P_1>$, and $Q_2 = <E_2,I_2,P_2>$. Let the sets S_1,S_2 be defined by:

$$S_1 = Imp(Q_1)\cap Exp(Q_2)$$
$$S_2 = Imp(Q_1)\cap Imp(Q_2 \mid_{Exp(Q_2)-Exp(Q_1)})$$

If there exists a variable $v \in S_1$, such that $vI_1 \neq vE_2$, then the composition is undefined. If there exists a $v \in S_2$, such that $vE_1 \neq vE_2$, then the composition is undefined. Otherwise, we define the composition by:

$$Q_1 \circ Q_2 = <E_1/\ E_2, (I_2/\ I_2) \mid_{Exp(P_1\ P_2)}, P_1 \circ P_2>$$

□

The definition above ensures that for any monomorphic tree t, $t(Q_1 \circ Q_2) = (tQ_1)Q_2$.

Definition: Given monomorphic packages $Q_1 = <E_1,I_1,P_1>$ and $Q_2 = <E_2,I_2,P_2>$, we define the importation into Q_1 from Q_2 by:

$$Q_1 * Q_2 = (Q_1 \circ (Q_2 \mid_{Imp(Q_1)})) \mid_{Exp(Q_1)}$$

□

Definition: Given a monomorphic package $Q = <E,I,P>$, we define its iteration, Q^∞, as follows: If P is not iterable, then Q^∞ is not defined. If there exists a variable $v \in Exp(Q)\cap Imp(Q)$, such that $vI \neq vE$, then Q^∞ is not defined. Otherwise, we define Q^∞ by:

$$Q^\infty = <E, I \mid_{Imp(P^\infty)}, P^\infty>$$ □

The rationale behind the above definition is best shown by examples.

Example: Let

$$Q = <\{\mathscr{G}s \rightarrow Statement\},$$
$$\{\mathscr{G}s \rightarrow Statement, \mathscr{G}ss \rightarrow Statement, \mathscr{G}c \rightarrow Boolean\},$$
$$\{\mathscr{G}s \rightarrow \text{if}(\mathscr{G}c, seq(\mathscr{G}ss, \mathscr{G}s))\}>$$

Then the typeless part of Q is iterable. Also $Exp(Q)\cap Imp(Q) = \{\mathscr{G}s\}$, and $\mathscr{G}s$ has matching import and export types. Q is therefore iterable, and

$$Q^\infty = <\{\mathscr{G}s \rightarrow Statement\},$$
$$\{\mathscr{G}ss \rightarrow Statement, \mathscr{G}c \rightarrow Boolean\},$$

$$\{\mathscr{s} \rightarrow \text{if}(\mathscr{c}, seq(\mathscr{ss}, \text{if}(\mathscr{c}, seq(\mathscr{ss}, \cdots)))))\} > \square$$

Example: Let

$$Q = <\{\mathscr{x} \rightarrow Integer\},$$
$$\{\mathscr{x} \rightarrow String\},$$
$$\{\mathscr{x} \rightarrow length(\mathscr{x})\}>$$

Then the typeless part of Q is iterable. But the variable $\mathscr{x}$ is both imported and exported, and the import type does not match the export type. Therefore Q is not iterable. □

Definition: Given monomorphic packages Q_1 and Q_2, we define Q_1 / Q_2 by:

$$Q_1 / Q_2 = Q_1 + Q_2 \mid_{Exp(Q_2) - Exp(Q_1)}$$

6.2. Polymorphic Packages

We can build on the definitions given above, for monomorphic packages, and define polymorphic packages. Polymorphic packages differ from monomorphic packages in that they have flexible types.

Definition: A polymorphic package G is a set of monomorphic packages, such that either G is empty, or there exists a triple $<E, I, P>$ of packages, such that G can be written as follows:

$$G = \{<E*R, I*R, P> \mid R \textit{ is a package such that } Imp(E*R) = \emptyset = Imp(I*R)\} \square$$

Clearly, a non-empty polymorphic package, G, is completely specified by the triple $<E, I, P>$. We therefore use $<E, I, P>$ to represent the set G.

The empty set is included in the definition, because it is useful, for technical reasons, to allow it as a package. A typing error in package operations causes the result to be the empty set.

Definition: Given polymorphic packages G, G_1, and G_2, and a set N of variables, we define package operations as follows:

$$G_1 + G_2 = \{Q_1 + Q_2 \mid Q_1 \in G_1,\ Q_2 \in G_2, \text{ and } Q_1 + Q_2 \textit{ is defined}\}$$
$$G_1 * G_2 = \{Q_1 * Q_2 \mid Q_1 \in G_1,\ Q_2 \in G_2, \text{ and } Q_1 * Q_2 \textit{ is defined}\}$$
$$G_1 \circ G_2 = \{Q_1 \circ Q_2 \mid Q_1 \in G_1,\ Q_2 \in G_2, \text{ and } Q_1 \circ Q_2 \textit{ is defined}\}$$
$$G_1 / G_2 = \{Q_1 / Q_2 \mid Q_1 \in G_1,\ Q_2 \in G_2, \text{ and } Q_1 / Q_2 \textit{ is defined}\}$$
$$G \mid_N = \{Q \mid_N \mid Q \in G\}$$
$$G^\infty = \{Q^\infty \mid Q \in G, \textit{ and } Q^\infty \textit{ is defined}\} \square$$

The operations defined above would not be much use unless the results are somehow computable. Hence the following theorem.

Theorem: If G, G_1, and G_2 are any polymorphic packages, and N is a set of variables, then $G_1 + G_2$, $G_1 * G_2$, $G_1 \circ G_2$, G_1 / G_2, $G \mid_N$, and G^∞ are all polymorphic packages.

proof: The proof is trivial for empty packages, so without loss of generality, we assume that G, G_1, and G_2 are all nonempty. Also, if the resulting sets are empty, they are trivially polymorphic packages, so we assume, again without loss of generality, that the resulting sets are nonempty.

We now proceed to prove the theorem for $G_1 + G_2$. The proof for the other sets is omitted.

Let $G_1 = <E_1, I_1, P_1>$, and $G_2 = <E_2, I_2, P_2>$. Without loss of generality, assume that the sets $Imp(E_1) \cup Imp(I_1)$ and $Imp(E_2) \cup Imp(I_2)$ are disjoint.

Since $G_1 + G_2$ is non-empty, there exist $Q_1 \in G_1$, and $Q_2 \in G_2$, such that $Q_1 + Q_2$ is defined. Let

$$Q_1 = <E_1 * R_1, I_1 * R_1, P_1>$$

and

$$Q_1 = <E_1 * R_1, I_1 * R_1, P_1>$$

Assume, without loss of generality, that $Exp(R_1) \cap Exp(R_2) = \emptyset$. Then

$$Q_1 = <E_1 * (R_1 + R_2), I_1 * (R_1 + R_2), P_1>$$

and

$$Q_1 = \langle E_1*(R_1+R_2), I_1*(R_1+R_2), P_1\rangle$$

For Q_1+Q_2 to be defined, R_1+R_2 has to be a unifier for the set of pairs

$$\{\langle vI_1, vI_2\rangle \mid v \in Exp\,(I_1)\cap Exp\,(I_2)\}$$

Since one unifier exists, a most general unifier exists. Let U be a most general unifier for the set. Without loss of generality, assume that

$$Imp\,(U)\cap(Imp\,(E_1)\cup Imp\,(I_1)\cup Imp\,(E_2)\cup Imp\,(I_2)) = \emptyset$$

Let

$$\begin{aligned} G' &= \langle E_1*U + E_2*U, I_1*U / I_2*U, P_1+P_2\rangle \\ &= \langle (E_1+E_2)*U, (I_1/ I_2)*U, P_1+P_2\rangle \end{aligned}$$

Then G' is a well-defined polymorphic package. We now proceed to prove that

$$G' = G_1+G_2$$

First, we prove that $G_1+G_2 \subseteq G'$. Let $Q_1 \in G_1$, and $Q_2 \in G_2$, such that Q_1+Q_2 is defined. Let $Q_1 = \langle E_1*R_1, I_1*R_1, P_1\rangle$, and $Q_2 = \langle E_2*R_2, I_2*R_2, P_2\rangle$. Then

$$Q_1+Q_2 = \langle E_1*R_1+E_2*R_2, I_1*R_1/ I_2*R_2, P_1+P_2\rangle$$

Without loss of generality, we assume that $Exp\,(R_1)\cap Exp\,(R_2) = \emptyset$, and $Imp\,(E_1) = Exp\,(R_1)$. Hence

$$\begin{aligned} E_1*(R_1+R_2) &= E_1*((R_1+R_2) \mid_{Imp(E_1)}) \\ &= E_1*R_1 \end{aligned}$$

Similarly, we can show that $I_1*(R_1+R_2) = I_1*R_1$, $E_2*(R_1+R_2) = E_2*R_2$, and $I_2*(R_1+R_2) = I_2*R_2$. Hence

$$Q_1 = \langle E_1*(R_1+R_2), I_1*(R_1+R_2), P_1\rangle$$

$$Q_2 = \langle E_2*(R_1+R_2), I_2*(R_1+R_2), P_2\rangle$$

and therefore

$$Q_1+Q_2 = \langle (E_1+E_2)*(R_1+R_2), (I_1/ I_2)*(R_1+R_2), P_1+P_2\rangle$$

Again, since Q_1+Q_2 is defined, R_1+R_2 is a unifier for the set of pairs

$$\{\langle vI_1, vI_2\rangle \mid v \in Exp\,(I_1)\cap Exp\,(I_2)\}$$

Therefore, there exists a package L, such that $R_1+R_2 = U \circ L$. Hence we have:

$$\begin{aligned} Q_1+Q_2 &= \langle (E_1+E_2)*(R_1+R_2), (I_1/ I_2)*(R_1+R_2), P_1+P_2\rangle \\ &= \langle (E_1+E_2)*(U \circ L), (I_1/ I_2)*(U \circ L), P_1+P_2\rangle \\ &= \langle ((E_1+E_2)*U)*L, ((I_1/ I_2)*U)*L, P_1+P_2\rangle \in G' \end{aligned}$$

This shows that $G_1+G_2 \subseteq G'$. Now, we show that $G' \subseteq G_1+G_2$. Let $Q \in G'$. Then there exists a package L, such that

$$\begin{aligned} Q &= \langle ((E_1+E_2)*U)*L, ((I_1/ I_2)*U)*L, P_1+P_2\rangle \\ &= \langle (E_1+E_2)*(U \circ L), (I_1/ I_2)*(U \circ L), P_1+P_2\rangle \end{aligned}$$

Now let

$$Q_1 = \langle E_1*(U \circ L), I_1*(U \circ L), P_1\rangle$$

$$Q_2 = \langle E_2*(U \circ L), I_2*(U \circ L), P_2\rangle$$

Then $Q_1 \in G_1$, and $Q_2 \in G_2$. Since U is a unifier for

$$\{\langle vI_1, vI_2\rangle \mid v \in Exp\,(I_1)\cap Exp\,(I_2)\}$$

$U \circ L$ is also a unifier. Therefore Q_1+Q_2 is defined. By definition, we get $Q = Q_1+Q_2$. Hence $G' \subseteq G_1+G_2$, and therefore $G' = G_1+G_2$. □

7. Conclusion

The main thesis of this paper is that packages can be defined as objects with algebraic properties. We believe that it is important to recognize the basic operations on packages, and allow their use by the programmer.

All the operations above, except iteration, can be simulated in current programming languages (e.g. Ada). Giving iteration under the control of the programmer is a way of giving him more complete control over the binding of symbols and references. For example, he can construct a variety of looping control sequences, such as while-do and repeat-until, using the if-then-else control sequence as a building block. Iteration allows bindings that are not possible using the other, strictly hierarchical, operations.

The operations on packages presented here enable the programmer to work at higher levels of abstraction, thus increasing productivity. A high level view of programming is a necessity when large systems are being built.

A programmer who has at his disposal tools that allow the composition of programs from previously written parts, while maintaining type security, without extra editing, will be more productive, and will be better able to maintain the high level view needed for the success of a large project.

Having the separate operations *iteration* and *join,* instead of the combined operation commonly used, is perhaps the most important new aspect of this model.

8. References

[Ada-80] *Ada Programming Language*, MIL-STD-1815, 10 December 1980.

[Agnarsson-85a] **Agnarsson, S., M. S. Krishnamoorthy and B. D. Saunders**, An Algebraic Implementation of Packages, *to appear in proceedings of EUROCAL-85*, 1985.

[Agnarsson-85b] **Agnarsson, S.**, *Packages as Substitutions*, Forthcoming Ph.D. Thesis, RPI, Troy, NY, April 1985.

[Bloom-80] **Bloom, Stephen L., Calvin C. Elgot and Jesse B. Wright**, Solutions of the Iteration Equation and Extensions of the Scalar Iteration Operation, *SIAM Journal of Computing, 9*, No. 1, February 1980.

[Bloom-83a] **Bloom, Stephen L., James W. Thatcher, Eric G. Wagner and Jesse B. Wright**, Recursion and Iteration in Continuous Theories: The "M-Construction", *Journal of Computer and System Sciences, 27* (1983) 148-164.

[Bloom-83b] **Bloom, Stephen L.**, All Solutions of a System of Recursion Equations in Infinite Trees and Other Contraction Theories *Journal of Computer and System Sciences, 27* (1983) 225-255.

[Burstall-84a] **Burstall, R.**, Programming with Modules as Typed Functional Programming, *Proc. International Conference on Fifth Generation Computing Systems, Tokyo, Nov. 1984*.

[Burstall-84b] **Burstall, R. and B. Lampson**, A Kernel Language for Abstract Data Types and Modules, *Lecture Notes in Computer Science, No. 173*, Springer-Verlag, New York, 1984.

[Courcelle-78] **Courcelle, B.**, A representation of trees by languages, *Theoretical Computer Science, 6* (1978) 225-279 and 7 (1978) 25-55.

[Courcelle-79] **Courcelle, B.**, Infinite Trees in Normal Form and Recursive Equations Having a Unique Solution, *Mathematical Systems Theory, 13* (1979) 131-180.

[Courcelle-83] **Courcelle, B.**, Fundamental Properties of Infinite Trees, *Theoretical Computer Science, 25* (1983) 95-169.

[Demers-80a] **Demers, A. J., and J. E. Donahue**, "Type Completeness" as a Language Principle, *Proceedings Seventh Annual Principles of Programming Languages Symposium*, 1980, pp. 234-244.

[Demers-80b] **Demers, A. J., and J. E. Donahue**, Data Types, Parameters and Type Checking, *Proceedings Seventh Annual Principles of Programming Languages Symposium*, 1980, pp. 12-23.

[Gallier-81] **Gallier, J.** DPDA's in 'atomic normal form' and applications to the equivalence problems, *Theoretical Computer Science, 14* (1981) 155-186.

[Ginali-79] **Ginali, Susanna**, Regular Trees and the Free Iterative Theory, *Journal of Computer and System Sciences, 18* (1979) 228-242.

[Goguen-84] **Goguen, Joseph A.**, Parameterized Programming, *IEEE Transactions on Software Engineering, Vol. SE-10, 5*, September 1984.

[Jenks-84] **Jenks, R. D.**, A Primer: 11 Keys to New Scratchpad, *Lecture Notes in Computer Science, Vol. 174*. Springer-Verlag, New York, 1984.

[Liskov-81] **Liskov, Barbara, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheiffler and Alan Snyder**, *CLU Reference Manual*, Lecture Notes in Computer Science, Vol. 114, Springer-Verlag, New York, 1981.

[Milner-78] **Milner, R.**, A Theory of Type Polymorphism in Programming, *Journal of Computer and System Sciences 17*, 348-375 (1978).

[Reynolds-70] **Reynolds, John C.**, GEDANKEN - A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept, *Communications of the ACM 17*, no. 5, pp. 308-319, May, 1970.

[Wirth-83] **Wirth, Niklaus**, *Programming in MODULA-2*, Springer-Verlag, New York, 1983.