A New Algebra System

May 29th, 1984

James H. Davenport

CONTENTS

LIST OF ILLUSTRATIONS

THE ALIST MODEL

The thesis of this note is that, in order to achieve uniform semantics between compiled and interpreted code, and to avoid exposing all sorts of internal hacks to the user, we require a model for the semantics of the new algebra system. Among the requirements of this model are that it be

1. Simple

2. Powerful

3. Related to a user's perception of the system.

It is not necessary that it be efficient.

I propose, as I believe others have done, the Alist model for computer algebra systems, in which a domain is conceived of as a set of values (about which little more will be said), a set of attributes (I do not fully understand these, but believe that they will follow much the same lines as operations) and a set of operations. The set of operations lists all the operations that can be performed on elements of the domain, and the user is invited to view the system as searching this list for an operation of the appropriate signature, and then applying it.

Implications for operator resolution

When the system, in parsing, meets an occurrence of an operator, it has to resolve it, i.e. it has to decide how to implement the operation. The model tells us that we can discover what operations are admissible by looking on the Alists which correspond to domains.

This model is a little vague in some respects, so let us spend some time clarifying it. The first problem that we have is nullary operations. The new algebra system supports several nullary operations, which can be regarded as being basically of two types:

1. Domain-specific constants, such as zero and one. Since the zero and one (or any other distinguished elements) of a domain have to be computed specially for each domain, the algebra system treats them as nullary functions, whose values lie in the domain. These functions can be found from the Alist of the domain.

2. Properties of domains. The system supports functions that will tell you about domains, for example the function characteristic defined by:

   ```
   Ring:Category == Join(Rng,Monoid) with
       [operations] characteristic: -> Integer
   ```

which maps into the integers. Clearly it would be possible for all such functions to be found on the Integer Alist, but this is undesirable for many reasons:

- every time a new Ring was instantiated, the list of functions available in Integer would change, which has some baroque implications for modularity;
- there would be a great number of characteristic functions on the Alist of Integer, which might make searching for them hard;
- the user views 'characteristic' as a property of the domain, not of the integers, anyway.

For all these reasons, we state that operations like characteristic are associated with their natural domain, and we indicate this by syntax like

```
<u,lffel>:=SqFr(DivExpts(p,P.characteristic):P)
```

where 'P.characteristic' means "the characteristic of the domain P".

The remark made above "searching this list" is unambiguous for totally homogenous operations, but for heterogeneous operations we can ask "which list?". Heterogeneous operations come in several flavours.

1. On the one hand we have the heterogeneous operations that have the general signature

   ```
   ($,$) -> well known object
   ```

   (or any similar such signature) such as EQUAL, > and so on. These are no problem; they can be looked up on the Alist corresponding to $.

2. A closely related set are those that take a well-defined type among the operands, such as

   ```
   coef: ($,Integer) -> R
   **: ($,Integer) -> $
   ```

   These too are no real problem. We do not want to enter them on the Alist for the "well-known" domains, for the same reasons as those quoted above when we were discussing nullary functions. Therefore we insist that they are only on the Alist of their other operands, which means that, in general, it may be necessary to search the Alists of all the operands to discover a particular function.

3. A more interesting problem are the truly heterogeneous operations, such as the multiplication of non-square matrices. Here we can only say that the operation will be found in the Alist of at least one of the argument domains.

In this model, asserting that a particular domain belongs to a particular category means that you are guaranteed to find, in that domain, all the operations mentioned in the category declaration. You may well find more.

CATEGORIES

There are various categories in any system built in the manner we envisage, and it will be useful to make certain distinctions. For this purpose, we will consider the code in Figure 1 on page 3, which is laid out diagrammatically in Figure 2 on page 4. We have omitted all consideration of attributes, since:

- o They follow the same general principles of inclusion and inheritance as operation definitions;
- o We know more about operations than we do about the semantic meaning of attributes.

```
  Set:Category == with
      [operations] "=": ($,$) -> Boolean
                   format: $ -> PrintBox
  SemiGroup:Category == Set with
      [operations] "*": ($,$) -> $
                   "**": ($,PositiveInteger) -> $
  Monoid:Category == SemiGroup with
      [operations] 1: -> $
                   "**": ($,NonNegativeInteger) -> $
  Group:Category == Monoid with
      [operations] inv: $ -> $
                   "**": ($,Integer) -> $
  AbelianGroup:Category == Set with
      [operations] 0: -> $
                   "+": ($,$) -> $
                   "-": $ -> $
                   "-": ($,$) -> $
  Rng:Category == Join(AbelianGroup,SemiGroup)
  Ring:Category == Join(Rng,Monoid) with
      [operations] characteristic: -> Integer
                   recip: $ -> Union($,"failed")
```

Figure 1. Some code for categories: with the operations listed, corresponding to the tree shown in Figure 2 on page 4.

In the code presented in Figure 1 we see two important keywords, viz. Join and with.

Join	denotes that the category being defined (on the left hand side of the == function) is composed of all the operations (and attributes) of the categories being joined together.
with	indicates that the category being defined consists of the category before the 'with' together with all the operations (and attributes) described after the 'with'. It is possible (e.g. the case of 'Set') for there to be no category before the 'with'. This is then an im-

```
  (Empty Category)          [=, format]
        ]                       ]
        ]_______________________]
                   ]
  [0,+,-,-]       Set               [*, **(PI)]
       ]          ] ]                   ]
       ]__________] ]___________________]
            ]                 ]
        AbelianGroup       SemiGroup         [1, **(NNI)]
                ]           ]    ]                ]
                ]___________]    ]________________]
                      ]                 ]
                     Rng             Monoid    [inv, **(I)]
                      ]              ]   ]          ]
                      ]______________]   ]__________]
                         ]                    ]
  [characteristic,       ]                  Group
       recip]            ]
         ]               ]
         ]_______________]
                 ]
               Ring
```

Figure 2. Corresponding diagram: for the code shown in Figure 1.

plicit reference to the empty category(1). The statement A with B can also be regarded as Join(A,B'), where B' is the category defined by B. Hence we can regard B as defining a category, which we shall term a nonce category, since it is only in effect for the duration of the 'with' statement. 'with' is in fact defined(2) that way, so we can regard all 'with' statements as replaced by equivalent 'Join' statements. Nonce categories are often ignored when drawing diagrams, as we can see by comparing Figure 2 with Figure 4 on page 9 .

'Join' is an n-ary operator, and is forced so by the system, so that the following are all equivalent(3):

----

(1) There is a curiosity in the terminology here. The empty category is that with no operations defined on it, and therefore it is the one to which all domains belong. Perhaps it ought to be renamed the "universal category", but this also has unfortunate connotations.

(2) The reader may then ask why we have both. The answer is that, in practice, it proves easier to have both, since there are many instances where a nonce category is required, and it would be tedious to have to create a named one.

(3) C' being the nonce category corresponding to C.

```
  Join(A,B) with C
  Join(A,B with C)
  Join(A,B,C')
```

We do not assume that Join is commutative, and, in the discussion on principal arrows (page 9) the reason for this should become apparent.

It is also possible to have anonymous categories, such as are formed by Join (or with) constructs that occur in other contexts than the instantiation of named categories, such as:

```
  Integer: Join(OrderedSet,DifferentialRing,EuclideanDomain) with
    [operations] oddp: $ -> Boolean
                 abs: $ ->  $
                 random: PositiveInteger ->  $
                 numberOfDigits: ($,$) -> $
```

Conditional Categories.

So far we have assumed that the relationship between categories is fixed. This is normally(4) true if we are considering pure category definitions, but may well not be true when we come to functors. Consider the following code:

```
  PolRing(R:Ring,E:OrderedAbelianGroup): T == C
   where
    T == Algebra(R) with
      [operations] if R has IntegralDomain then "//": ($,R) -> Union($, "failed")
      [assertions] if R has unitsKnown then unitsKnown
                   if R has commutative("*") then commutative("*")
                   if R has IntegralDomain then IntegralDomain
                   if R has IntegralDomain and R has canonicalUnitNormal
                     then canonicalUnitNormal
```

One might assume from it that $ (i.e. T) was always an Algebra(R), but, in fact, if R is an IntegralDomain, $ is an Algebra(R) with the // operation, and it is also an IntegralDomain. Hence (in this case) its category is, in fact, Join(Algebra(R),IntegralDomain) with // ... . Since the category to which it belongs depends on other information (in this case the category of R, but it could depend on attributes, or on things like the primality of a number), we call this the problem of conditional categories.

This clearly is a problem, since we do not know how to handle the placement of the operators that may or may not be defined in the domain, depending on the conditional information. The solution adopted is to create a maximal representation for $, i.e. if $ could belong to one of several categories (a, b ,c ...), we in fact instantiate it in Join(a, b, c ...).

----

(4) In the versions of this paper of 82/07/19 and before it was always true.

Conditional declarations.

When is it legitimate to make a conditional declaration? The answer to this is that there are really two kinds of conditional declaration that can be made.

The first is when we are declaring confitional information about an existing (input) domain, as in

```
  PolRing(R:Ring,E:OrderedAbelianGroup): T == C
   where
    T == Algebra(R) with
            ...
         if R has IntegralDomain then
           unitNormal(p) ==
              p = () or p.first.c = 1 => UnitCorrAssoc(1,1,p)
                                    ...
```

Here the requirements are quite straight-forward: either the declaration must already be true (i.e. the new type must be an ancestor of the existing type), or it must be possible to add the new type consistently with existing information (i.e. the new type must be a principal descendant of the existing type. If neither of these conditions is the case, then we have some form of type mis-match.

The other possibility is that we have a conditional declaration for an output domain. This is the example above, under "conditional categories". Here the declaration must be made initially, since space must be reserved for additional operators (a sordid implementation reason, but this corresponds to the absence of any syntax for declaring new operators on the fly, so we shall leave it for now).

A genuinely conditional category.

It is occasionally necessary to declare a (parametrised) category with conditions. The first example of this known to the author is the following (due to Mr. Trager).

```
  BasicPolynomial(R:Ring): Category ==
         GeneralPolynomial(R,NonNegativeSmallInteger) with
                   var: -> Expression
                   varPol: -> $
                   PDerive: $ -> $
                   map: (R -> R,$) -> $
                   if R has Field then EuclideanDomain
                   if R has UniqueFactorizationDomain then content: $ -> R
                   if R has UniqueFactorizationDomain then
                             UniqueFactorizationDomain
```

This category is later used in the following context

```
  SparseMultivariatePolynomial(F,R:Ring,VarSet): C == T
   where
    VarSet: OrderedSet with coerce: $ -> Expression
    S: Ring
    F(S): BasicPolynomial(S) with
    ...
```

where the conditions are required to ensure that the functor F supplies the operations required.

Implicit relationships.

```
|-----------------------------------------------------------------------------|
| (Empty Category)           [=, format]                                      |
|       ]                       ]                                             |
|       ]_______________________]                                             |
|                 ]                                                           |
| [0,+,-,-]      Set               [*, **(PI)]                                |
|      ]          ] ]                   ]                                     |
|      ]__________] ]___________________]                                     |
|           ]                 ]                                               |
|      AbelianGroup       SemiGroup          [1, **(NNI)]                     |
|           ]   ]           ]   ]                 ]                           |
|           ]   ]___________]   ]_________________]                           |
|           ]         ]                  ]                                    |
|           ]        Rng               Monoid   [inv, **(I)]                  |
|           ]                            ]   ]          ]                     |
|           ]____________________________]   ]__________]                     |
|                        ]                        ]                           |
| [characteristic,       ]                      Group                        |
|      recip]            ]                                                    |
|        ]               ]                                                    |
|        ]_______________]                                                    |
|               ]                                                             |
|             Ring                                                            |
|                                                                             |
```

Figure 3. An alternative layout: in which there is an implicit relationship between the categories Rng and Ring.

Consider the relationships of Figure 3 as an alternative to those proposed in Figure 2 on page 4. This differs in that we have

```
  Ring:Category == Join(AbelianGroup,Monoid) with
```

instead of

```
  Ring:Category == Join(Rng,Monoid) with
```

Mathematically, the two should be the same, since a Join(AbelianGroup,Monoid) is automatically a Join(AbelianGroup,SemiGroup), i.e. a Rng. But there seems no obvious way for the system to detect this difficulty, which we call the problem of implicit relationships. So we will content ourselves with advising the user that he should not fall into this trap, with the note that it is an interesting problem for future development.

RELATIONS BETWEEN CATEGORIES

Having defined our model, we must then define how we intend to implement it. For this purpose, we will work by example as much as by formal definition. The scenario of category definitions being considered is that of Figure 4 on page 9 (N.B. I am not stating that all algebra systems, or even that any algebra system, ought to use these category definitions in this way; it is merely a convenient scenario within which we can operate). Throughout this document, I will assume that all categories are based on the fundamental category Set. This does not appear to be a fundamental restriction, but it seems to make some of the presentation easier. This shows a collection of categories. A category that is connected to another one by a downward arrow is declared to be a sub-category of the other one, i.e. by statements such as:

```
  Monoid:Category == SemiGroup with ...
  Group:Category == Monoid with ...
  AbelianGroup:Category == Set with ...
  Ring:Category == Join(AbelianGroup,Monoid) with ...
  OrderedSet:Category == Set with ...
  OrderedRing:Category == Join(OrderedMonoid,Ring) with ...
```

We also use the phrase extension(5) in this context, saying that Monoid is an extension of SemiGroup, or that OrderedRing is an extension of both Ring and OrderedMonoid.

Such structures are generally represented by diagrams such as Figure 4 on page 9. In such diagrams it is conventional to omit the nonce formed by the with statements, and the associated arrows, and draw only the named categories.

We state that a category is a descendant of another category if it is connected(6) to that category by downward arrows. This would imply that all cat-

----

(5) If A is an extension of B, then it has more operations or attributes than B, and therefore fewer domains belong to A than belong to B (in the sense that every domain that belongs to A can be mapped into a domain of B by the forgetful functor, but that not every domain of B is necessarily an image of the forgetful functor). This may seem a somewhat perverse use of the word "extension", but there appears no adequate nomenclature. This problem is the same one as that of the definition of the empty category (page 4).

(6) Note the distinction between this and the concept of extension mentioned above. Every descendant is an extension, but a descendant can be connected by more than one arrow, acting in series. More formally, the re-

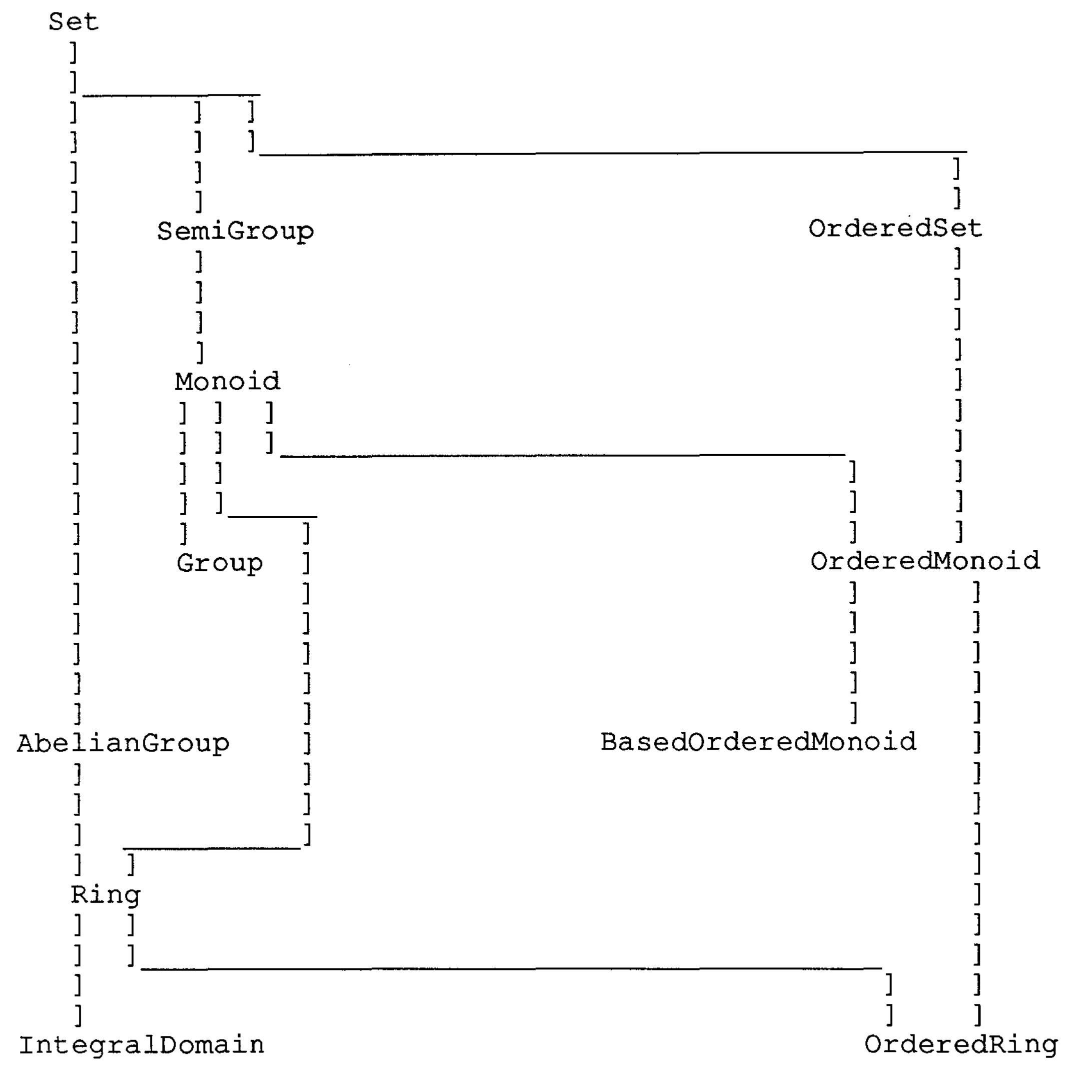

Figure 4.   A Directed Acyclic Graph of Categories

egories are descendants of Set, that OrderedRing is a descendant of OrderedMonoid, Ring, OrderedSet, to name but a few, and many other such descendant relationships. If A is a descendant of B, we can say that B is an ancestor of A.

There can be many downward arrows leading into a category: let us select one and call it the principal arrow. Figure 5 on page 11 shows a choice for all the principal arrows in the situation of Figure 4. Note again that this choice is arbitrary, and the choice we have adopted is that the first category

----

lation is a descendant of is the transitive closure of the relation is an extension of.

mentioned in a Join statement is the one from which the principal arrow comes. This leads to the idea of principal descendants - a category is a principal descendent of another one if it is connected to it by principal arrows.

Having made this, essentially arbitrary, choice of principal arrows (and hence of principal descendants), we can now define the concept of principal ancestor. This follows from the definitions: If A is a principal descendant of B, then B is a principal ancestor of A. Looking at Figure 5 on page 11, we see that, while Ring and OrderedMonoid are both ancestors of OrderedRing, only Ring is a principal ancestor.

This leads to the concept of a fundamental ancestor: we say that the set B1, B2, ... of ancestors of B is a set of fundamental ancestors of B if:

1. Every Bi is an ancestor of B;

2. Every ancestor of B is a principal ancestor of one of the Bi. This property will be used repeatedly in what follows.

A given category may have more than one set of fundamental ancestors, but it can have only one minimal set (e.g. Ring, Monoid, OrderedSet, OrderedMonoid is a set of fundamental ancestors of OrderedRing, but Monoid is redundant, since it is a principal ancestor of OrderedMonoid). The minimal set of fundamental ancestors of a category can be determined by the following algorithm:

```
[1]           Current := [A];  FA := [A]; Processed := [A]
[2]           While Current not equal [Set] do
[2.1]           For each C in Current,
[2.1.1]           For each D such that C is an extension of D
[2.1.1.1]           Current := Current union D
[2.1.1.2]           If C is a principal extension of D
[2.1.1.2.1]           then FA := FA less D
[2.1.1.2.2]           else if D not in Processed then FA := FA union D
[2.2]           Current := Current less C
[2.3]           Processed := Processed union C
```

This algorithm can be justified by considering walking breadth first up the tree, marking (by placing in Processed) every node we come to. We will often use the term "fundamental ancestor of" to mean "member of the minimal set of fundamental ancestors of".

REPRESENTATIONS OF CATEGORIES

We represent every category by a Vector, laid out in the following way:

0   Name of the category (e.g. (IntegralDomain) or (Module R), where R is the argument passed to the LISP function Module. [Is this right? We might have (Module <monstrous Vector representing a Polynomial domain with Matrix coefficients ...>). Perhaps, though, this is better than the alternative, and we could arrange to print this as (Module (Polynomial x (Matrix 2 (UnivariatePolynomial y (ZModN 7)))))]

```
    Set
    ]]
    ]]___________
    ]]      ]]  ]]
    ]]      ]]  ]]
    ]]      ]]  ]]_____________________________________
    ]]      ]]                                        ]]
    ]]      ]]                                        ]]
    ]]     SemiGroup                           OrderedSet
    ]]      ]]                                          ]
    ]]      ]]                                          ]
    ]]      ]]                                          ]
    ]]      ]]                                          ]
    ]]      Monoid                                      ]
    ]]     ]] ] ]]                                      ]
    ]]     ]] ] ]]_______________________________       ]
    ]]     ]] ]                                 ]]      ]
    ]]     ]] ]_____                            ]]      ]
    ]]     ]]       ]                           ]]      ]
    ]]      Group   ]                          OrderedMonoid
    ]]              ]                           ]]       ]
    ]]              ]                           ]]       ]
    ]]              ]                           ]]       ]
    ]]              ]                           ]]       ]
    ]]              ]                           ]]       ]
   AbelianGroup     ]               BasedOrderedMonoid   ]
    ]]              ]                                    ]
    ]]              ]                                    ]
    ]]   ___________]                                    ]
    ]]   ]                                               ]
      Ring                                               ]
    ]]  ]]                                               ]
    ]]  ]]_________________________________________      ]
    ]]                                            ]]     ]
    ]]                                            ]]     ]
   IntegralDomain                                 OrderedRing
```

Figure 5. Same diagram showing principal arrows

1 List of operations, where each operation is stored in the form:

```
( ( name signature) implementation predicate)
```

e.g.

```
( (EQUAL ( (Boolean) $ $)) (ELT $ 6) true)  .
```

A statement like (ELT $ 6) above indicates that, in all domains satisfying the format requirements of this category, the operation EQUAL is to be found in element 6 of the vector. This is also expressed by saying that 6 is the sequence number of EQUAL for this category.

Note that this is a change from the current implementation, in which there is no "implementation" component, but instead EQUAL is represented as

```
( (EQUAL ( (Boolean) $ $)) true)  ,
```

and the fact that it is implemented as element 6 of the vector is deduced from its relative position in the list. The reasons for this change are two-fold: one is that relative positions are dubious tools, especially when one is merging lists (as Join must do), and the other is that we are allowed other forms of implementations, such as macros or

```
( (unit ($ (UnitCorrAssoc $))) CAR true)  .
```

2 List of attributes. Since I do not fully understand attributes at the moment, I will only remark that I think we ought to decide whether we allow operation names, operation signatures or both in the specification of attributes.

3 The LISP data structure '(Category)' (i.e. a list of one element, whose car is the atom Category). This is used to distinguish these vectors from the representation of domains (see below).

4 A list of "associated categories and domains". The CAR of this cell is a list of all the principal ancestors of this category (excluding itself), so that we can find our way back up the category graph efficiently. The CADR of the cell is a list of all the fundamental ancestors of the category (again excluding itself), associated with the conditions that apply to them and their sequence numbers in the vector (see below). The precise format is a list of triples, each triple being a three-list: (fundamental ancestor, condition, sequence number). The CADDR of this cell is a list of all the non-primitive(7) domains that are required by operations defined on this category, such as Boolean (the result type of =) or PrintBox (the result type of format). The format of this list is a list of pairs of domain and slot number.

5 A list of all the parameters of the category, if it has any parameters.

6 et seq. Templates for all the operations, alternative views (corresponding to fundamental ancestors), non-primitive domains and other items that any domain looking like this category will have.

The above description may be more meaningful if it is compared with an example: Figure 6 on page 13.

----

(7) The question of when a domain is primitive, and hence is known by the system without any pointers to it being created, is one of implementation. Currently (19/7/82) the domains Boolean and String are regarded as primitive, but performance reasons may force us to increase this list, adding, say, Integer.

```
|----|---------------------------------------------------|
|    |                                                   |
| 0  |   Name                                            |
|    |   ----                                            |
|    |                                                   |
|    |   (SemiGroup)                                     |
|----|---------------------------------------------------|
|    |                                                   |
| 1  |   List of operations                              |
|    |   ------------------                              |
|    |                                                   |
|    |   (EQUAL ((Boolean) $ $)) true (ELT $ 6)          |
|    |   (format ((PrintBox) $) true (ELT $ 7)           |
|    |   (TIMES ($ $ $)) true (ELT $ 8)                  |
|----|---------------------------------------------------|
|    |                                                   |
| 2  |   List of attributes                              |
|    |   ------------------                              |
|    |                                                   |
|    |     ( ( (associative "*) true))                   |
|----|---------------------------------------------------|
|    |                                                   |
| 3  |   '(Category)'                                    |
|----|---------------------------------------------------|
|    |                                                   |
| 4  |   Associated categories and domains               |
|    |   ---------------------------------               |
|    |                                                   |
|    |   ((Set) NIL ( (PrintBox . 8)) )                  |
|----|---------------------------------------------------|
|    |                                                   |
| 5  |   Parameters                                      |
|    |   ----------                                      |
|    |                                                   |
|    |   NIL                                             |
|----|---------------------------------------------------|
|    |   TEMPLATES                                       |
|    |   ---------                                       |
|----|---------------------------------------------------|
|    |                                                   |
| 6  |   (EQUAL ((Boolean) $ $)) true (ELT $ 6)          |
|----|---------------------------------------------------|
|    |                                                   |
| 7  |   (format ((PrintBox) $) true (ELT $ 7)           |
|----|---------------------------------------------------|
|    |                                                   |
| 8  |   NIL (will be filled in by PrintBox in domains)  |
|----|---------------------------------------------------|
|    |                                                   |
| 9  |   (TIMES ($ $ $)) true (ELT $ 8)                  |
|    |                                                   |
|----|---------------------------------------------------|
```

Figure 6.   Layout of SemiGroup

We also define, for each of the operations defined in that category, a sequence number (as mentioned above, under item 1 of category layout). Sequence numbers start at 6(8) , and are assigned under the following rules:

- o For all operators defined in a category which it inherits from its immediate principal ancestor, the same sequence numbers are assigned as in that ancestor.
- o If this category (A, say) is a non-principal extension of another named category(9) (B, say), then that category (B) is assigned the next available sequence number(10) . These sequence numbers are stored in the CADR of slot 4 of the vector representing the category (vide supra).
- o All other operators(11) are assigned the next available sequence numbers.
- o All non-primitive domains that do not already have slots assigned to them are assigned the next available sequence number. These numbers are stored in the CADDR of slot 4 of the vector representing the category (vide supra).

This has the consequence that, if A is any principal ancestor of B, then all the operations which B inherited from A have the same sequence number in A as they do in B. Note that this is not true for non-principal ancestors. A further consequence is that, if A is any ancestor whatsoever of B, then B has a (minimal) fundamental ancestor C such that all the operations which B inherited from A have the same sequence numbers in A as they do in C.

Category subsumption

There is one exception to the above rule about the assignment of sequence numbers to non-principal ancestors quoted above. This was briefly discussed in the footnote on page 14, and is taken up in more detail here. If B is a prin-

----

(8) These are LISP vector indices, and so agree with the illustration in Figure 6.

(9) In fact, we need only consider the fundamental ancestors of A, by the arguments on fundamental ancestors outlined on page 10.

(10) There is one exception to this rule, when B is a principal extension of a category already appearing in the representation of A. This case is known as category subsumption, and is discussed later in this section, as "Category subsumption".

(11) This begs an interesting question: how do we decide if two operators are the same or not. The obvious answer, that of disambiguating on the signature, does not work for several reasons, and a more complex scheme seems to be necessary. This is discussed at the end of this section, as " Equivalence of operator definitions" on page 16.

cipal extension of a further category (C, say), which already appears in the representation of A, then we can replace the representation of C in A by the representation of B, and we do not need to generate a separate sequence number for B. This case is known as category subsumption, and is illustrated by Figure 7 and Figure 8 on page 16.

```
 Set
(=,format)
]]  ]]
]]  ]]______________________
]]                       ]]
AbelianGroup           SemiGroup
(+,-,0)                ] (*)  ]]
  ]]                   ]      ]]
  ]]  _________________]      Monoid
  ]]  ]                        (1)
  ]]  ]                         ]
   Rng                          ]
    ]]                          ]
    ]]  ________________________]
    ]] ]
    Ring
```

Figure 7. Category Subsumption - the tree: A simple example, showing that the pointer to Monoid can subsume the pointer to SemiGroup in the definition of the category Ring. The vectors are given in Figure 8 on page 16.

	Rng	Ring
6	=	=
7	format	format
8	domain PrintBox	domain PrintBox
9	+	+
10	-	-
11	0	0
12	SemiGroup	Monoid
13	*	*
14		1

Figure 8. Category Subsumption - the vectors: Showing the layout of the vectors for Rng and Ring, especially how Monoid has subsumed the slot of SemiGroup.

Theorem: The categories appearing in the representation of a category C are precisely the fundamental ancestors of C.

Proof: By reductio. Let C be a minimal category for which the theorem asserted is false (i.e. the theorem is true for all principal ancestors of C). Let D be a category occurring in the representation of C, and let D′ be a fundamental ancestor of C which is also a principal descendant of D. Let C′ be the immediate principal ancestor of C. Then there are two cases:

1. D occurs in the representation of C′. Then either D′ occurs in the representation of C′, which contradicts the minimality of C, or it does not, in which case it would subsume the definition of D.

2. D does not occur in the representation of C′. Then there is no point in adding it as we move from C′ to C, since D′ is a principal descendant of D.

Equivalence of operator definitions

Here we consider the question, raised briefly at the bottom of page 14, as to when two operator definitions can be said to be equivalent. This is still a subject(12) deserving of study, and any remarks here will be very preliminary. This is also known as operator subsumption.

Our general model is of two signatures (known as (1) and (2) below: We assume that (1) already exists, and that (2) is a new definition we are comparing with (1)) which may or may not define 'equivalent' operations. There are two type of operator subsumption that we ought to consider:

Output subsumption This occurs when the output of signature (2) is a subset of the output of signature (1). An example of this is:

```
AbelianSemiGroup:Category == Set with
    [operations] "+": ($,$) -> $
                 "-": ($,$) -> Union($,"failed")

AbelianGroup:Category == AbelianSemiGroup with
    [operations] "-": $ -> $
                 "-": ($,$) -> $
                 0: -> $
```

After these definitions, there are two binary '-' operations defined on every AbelianGroup (there is also a unary '-', but this does not concern us). The first one returns either an element of the structure or 'failed', the second one always returns an element of the structure. the first.

Input subsumption This occurs when one of the inputs in signature (1) is a subset of the corresponding input in signature (2). An example of this is:

```
SemiGroup:Category == Set with
    [operations] "*": ($,$) -> $
                 "**": ($,PositiveInteger) -> $

Monoid:Category == SemiGroup with
    [operations] 1: -> $
                 "**": ($,NonNegativeInteger) -> $

Group:Category == Monoid with
    [operations] inv: $ -> $
                 "**": ($,Integer) -> $
```

Here there are three different definitions of '**' on Groups - depending on whether the second argument is a positive integer, a non-negative integer, or an arbitrary integer. But these sets are not independent. Indeed, it is not clear which definition should be invoked for x**2.

----

(12) BMT argues against any operator subsumption, for example, and believes that two operator definitions are equivalent if, and only if, they are identical. This is a plausible point of view, though not, as might be expected, the author's.

It is certainly possible to imagine two signatures which manifested output subsumption and three separate occurrences of input subsumption, but I do not see that this really complicates the problem.

Operator subsumption causes several distinct problems:

1. It is not clear to the compiler which definition should be invoked;

2. The user has to supply three different definitions for '**' whenever he defines a group, and two definitions of '-' whenever he defines an AbelianGroup;

3. The various vectors and other data structures are longer than they need be.

Clearly, what is required is that the system should determine that these definitions really refer to the same conceptual operator, and arrange, in some way, that the above problems are resolved. Determining that operator subsumption has occurred should not be too difficult(13), but it is not clear what the system can do about it.

In order to discuss this, we have to make a further distinction. Let A and B be two types, with A a subset of B (whether for input subsumption or output subsumption). We say that A is a subset of B at the source level in this case. It may further be the case that, in the implementation of the SCRATCHPAD language, the representation of the types A and B are such that the machine representations of A are a subset (with the same meanings attached) of the representations of B. We say that A is a subset of B at the machine level in this case. The definition of subsets at the machine level clearly depends on the implementation of the system, and the illustrations we are about to present therefore depend on the current implementation, and are subject to change without notice.

1. PositiveInteger is a subset of NonNegativeInteger is a subset of Integer at the machine level, because they are all represented as LISP integers.

2. The set Integer is a subset of the set RationalNumber at the source level, but not at the machine level (in LISP370, at least). The integer 1 is a different internal representation from the rational 1, and the LISP function EQUAL will declare that they are not the same.

3. For all types $, $ is a subset of Union($,"failed") at the machine level, since the members of the latter are either a special gensym(14) or the members of $.

----

(13) In principle. We currently do not have a syntax for declaring, for example, that the set PositiveInteger is a subset of the set Integer. It appears that this will be necessary for other reasons, so I do not regard this as a major stumbling block. In the current (81/08/29) system, explicit LISP statements are used to make these declarations.

(14) The reason for a gensym is to hide this "failed" from any other "failed"s

4. Unions where more than branch are not special quoted tags, such as

   ```
   R == Union($,Record(quo:$,rem:$))
   ```

   do not fall into the above paradigm. $ is indeed a subset of R at the source level, but not at the machine level, since the members of R are represented as LISP dotted pairs, whose CAR is a tag stating what branch of the union they belong to, and whose CDR is the actual value.

The reader may ask what this has to do with operator subsumption. The reason is that we must recognise two kinds of operator subsumption:

Machine level subsumption is that subsumption (output, input or both) that takes place when all the subset relationships involved are true at the machine level.

Source level subsumption (by which we mean that source level subsumption that is not also machine level subsumption) is that subsumption that takes place when at least one of the subset relationships is only true at the source level.

Subsumption can arise in several ways.

1. The subsumer can be an existing operator in a category, and the subsumee is being added from a nonce category. This is the easiest case: the user is specifying a redundant operation (and we can promptly forget all about it). The reason we never need to supply a definition (as contrasted with other cases) is that, since the category was a nonce category, the only time the operator can be referenced is in the scope of the current declaration, in which case the subsumer will do equally well.

2. The subsumer can be in the principal category, and the subsumee in a named category being added as a non-principal ancestor. This, regrettably, is not the same case as above, and we must distinguish between the cases of machine-level and source-level subsumption, as is done below.

3. (Almost certainly the most common case). The subsumee exists in the principal ancestor, and some other category is supplying the subsumer. This case also requires us to distinguish the two forms of subsumption, as is done below.

If we have machine-level subsumption, then a trick is available to us: we can re-use the same slot in the vector. The reason that this is possible is best explained if we consider the example of output subsumption on page 17. We are saying that a function satisfying the second signature for binary '-' can be used to satisfy the first as well. And this is clearly possible from the definition of machine-level subsumption - the output types are equivalent as far as the implementation is concerned.

----

that the user may be using as variables in his program. Early implementations do not translate "failed" into a special gensym, for ease of debugging, but this should be regarded as temporary.

Source level subsumption is a major problem, though. One might hope that one could delete the subsumed function from the Alist (which indeed one can) and not bother about that slot any more. Regrettably, that is not the case, and this is illustrated by the following piece of output subsumption (though the principles apply equally well to input subsumption) taken from a purely hypothetical piece of algebra system:

```
  EuclideanDomain:Category == UniqueFactorizationDomain with
      [operations] "/": ($,$) -> Record(q:$,r:$)

  Field:Category == Join(EuclideanDomain,SkewField) with
      [operations] "/": ($,$) -> $
```

While the field is being treated as a field there is no problem, but as soon as we come to treat it as a EuclideanDomain we ask where the '/' function for this domain, regarded as a EuclideanDomain, lives. We can not just place the '/' function from our field there, since that returns $, not Record(q:$,r:$). The solution appears to be that the system will have to invent, from the subsumer, a suitable definition for the subsumee, but I have no idea how that can be done in general.

A related problem

In this section we discuss a question closely related to that of operator subsumption, though the author has not seen sufficiently many instances where this problem occurs that he feels able to characterise the relationship. This problem crops up in the definition of Integer, which we can simplify, for the purpose of illustrating the problem, to:

```
  Ring:Category == Set with
      [operations] "*": ($,$) -> $
                   1: -> $
                   "**": ($,Integer) -> $
                   "+": ($,$) -> $
                   0: -> $
                   "-": $ -> $
                   "-": ($,$) -> $
                   "*": (Integer,$) -> $
```

The problem here is that there are two definitions of *, both of which are, in this particular case, the same, viz. (Integer,Integer) -> Integer. The user clearly should not be expected to supply two definitions, so the system must decide that the definition he supplies fits both templates.

TYPES OF OPERATIONS

A variety of operators can be defined on domains. In the conceptual model, they are all on the same footing, and all live in the Alist. But in practise we need to distinguish the various types of operation, and for the purpose of this paper we make the following distinctions:

Category constant operations are those that have the same definition(15) (by which we mean that the same piece of LISP code is used) for all elements belonging to that category. An example of this might be a 'max' function defined by means of

```
OrderedSet:Category == Set with
  [operations] "<": ($,$) -> Boolean
               max,min: ($,$) -> $ where
                    max(x,y) == if x > y then x else y
                    min(x,y) == if x > y then y else x
```

There are far more operations than one might think that have this species of definition, for example all the UnitCorrAssoc selectors fall into this category.

Category (varying) operations are those operations that exist in a (named) category, but whose definitions depend on the domain in the category. A typical example is the function +, as in

```
AbelianSemiGroup:Category == Set with
    [operations] "+": ($,$) -> $
                 "-": ($,$) -> Union($,"failed")
```

but also all operations that have a default(16) definition, as opposed to a constant definition, also belong to this classification.

Domain-specific operations are those for which the compiler can determine the definition at compile time, since there is no doubt as to which definition is meant. Examples of this are 'oddp' in Integer, 'HornerEval' in UnivariatePolynomial, 'Determinant' in DenseMatrix(17) and so on.

----

(15) This is similar to the idea of unique extensions in the ADJ terminology [Goguen et al., 1978], where we can say that a new operator (and associated axioms) defines a unique extension of a category C if, for every domain D in C, there is a unique domain D' in C' (C with the operation and associated axioms) such that the forgetful functor from C' to C maps D' to D.

The ideas are not quite the same, though, since we may wish to implement a uniquely defined operation (in the ADJ sense) by more than one operation in the LISP sense, e.g. we may wish for more than one implementation of exponentiation, even though there is only one possible definition.

(16) We currently have no syntax for these definitions, and probably ought to invent one

(17) In one view of the world. It would also be possible to have a category Matrix, into which both DenseMatrix and SparseMatrix were functors. In

Other operations fall into several classes.

1. Category varying operations from unnamed categories. These could be invoked by clauses such as

   ```
   PolRing(R:Ring with unitp: $ -> Boolean,
           Var:OrdAbelianGroup): Algebra(R) with
   ```

   though no such code currently exists in our system, and it is not clear that we would really want to have such code. Such operations have to be supported, since the Alist model clearly states that they exist, but it seems to me that it does not matter if access to such operations is slow.

2. Accidental operations. This category consists of requests for operations that the user has no "right" to expect to exist, e.g. if R is declared to be a Ring, the user may ask(18):

   ```
   if R has gcd: (R,R) -> R
   ```

   There is no inherent reason why this gcd operation should exist, and yet the Alist model demands that, if it exists, the statement should yield 'true', and the user should then be able to access the 'gcd' function of R.

We note that the fourth type, the "other" operations, only make sense when one is looking for an operation. When one is defining an operation, one always knows which of the first three categories it falls into.

STRUCTURES OF DOMAINS

The conceptual model stated that a domain could be regarded, as least as far as its operations are concerned, as an Alist. The true representation of a domain, as adopted in our system, is similar in concept, inasmuch as a domain is essentially represented as an Alist of the categories to which it belongs.

The word 'essentially' was used in the above paragraph because all that is needed is the representations of all the fundamental ancestors of the domain(19) D. So a typical element of the Alist representing the domain D, whose fundamental ancestors are the Di, actually has a list of all the princi-

----

that case the function 'Determinant' would belong to the previous category.

(18) Or some similar syntax

(19) So far, we have not defined what we mean by the fundamental ancestors of a domain. In fact, we are using the term merely as shorthand for "The fundamental ancestors of the category consisting precisely of the domain".

pal ancestors of Di in its CAR, and the vector corresponding to Di in its CDR. Figure 9 on page 24 should make this description clearer.

It may seem that this is a major change from the structure described in previous papers on the new algebra system [Jenks, 1979, Davenport & Jenks, 1980, Jenks & Trager, 1981], but in fact the difference is one of outlook rather than of implementation (and indeed did not occur to the author until this paper was partly written).

The code to look up the vector corresponding to any named category to which the domain belongs then looks like:

```
  (LookupDomain
    (LAMBDA (Category DomainAlist)
      (COND ((NULL DomainAlist) NIL)
            ((MEMBER Category (CAAR DomainAlist))
              (CDAR DomainAlist))
            ("T (LookupDomain Category (CDR DomainAlist))) )))
```

The conceptual Alist for operations

We have described how the conceptual model of a domain is an Alist allowing one to look up operations on it, and then explained how the implementation is really something different, being essentially an Alist of category representations. This poses the question: How do I actually look something up? This is answered by something that we term the conceptual Alist for operations, which is merely a means of answering this question.

The conceptual Alist is never actually constructed(20) but the system performs (when all more subtle means have failed) a search having the same effect. We refer to the various types of operator as defined under "Types of Operations" on page 20.

1. Category constant operations, such as 'max', are to be found on the Alist of the category for which they are defined. They will have an implementation component of the form (%.SUBR.OrderedSet,max,3(21) . $), meaning that the system is to implement max by applying that LISP function to the arguments for max and to $, the vector representing the domain from which max came.

----

(20) , For reasons of efficiency. The system can perform the equivalent search by using the Alist for the domain itself and also the Alist for all the categories to which the domain belongs (though it need only consider the fundamental ancestors of the domain.

(21) This unwieldy object represents a piece of compiled LISP code, and is also called a bpi, standing for "binary program image."

```
|------------|       |--------------|     |------------|
|(Set        |       |(Set          |     |(Set        |
|AbelianGroup|       |SemiGroup     |     |OrderedSet) |
|            |-------|              |-----|            |
|Ring)       |       |Monoid        |     |            |
|            |       |OrderedMonoid)|     |            |
|------------|       |--------------|     |------------|
|The Vector  |       |The Vector    |     |The Vector  |
|in category |       |in category   |     |in category |
|OrderedRing |       |OrderedMonoid |     |OrderedSet  |
|------------|       |--------------|     |------------|
```

Figure 9. A domain in the category OrderedRing

---

2. Category varying operations, such as + in AbelianSemiGroup and all its descendants, can also be found in the Alist of the category in which they were defined. Here the implementation will be of the form (ELT $ 8), indicating that element 8 of the vector representing the domain contains the complete representation of that operator.

3. Domain-specific operations are to be found on the Alist of the domain itself. They will have implementations of the same form as the category constant operations themselves, viz (bpiname . $).

This means that the conceptual Alist is the union of the list of domain-specific operations and the lists of operations for all of the categories to which this domain belongs. We note that it is sufficient to consider the fundamental ancestors of the (possibly unnamed) category to which the domain is declared to belong.

Vectors representing domains

0 Name of the domain (e.g. (Integer) or (Polynomial x R), where x and R are the arguments passed to the LISP function Polynomial that created this domain). [Is this right? We might have (Polynomial x <monstrous Vector representing a Polynomial domain with Matrix coefficients ...>). Perhaps, though, this is better than the alternative, and we could arrange to print this as (Polynomial x (Polynomial z (Matrix 2 (UnivariatePolynomial y (ZModN 7))))))]

1 List of operations, where each operation is stored in the form:

```
( ( name signature) implementation predicate)
```

e.g.

```
( (Evaluate (R $ R)) (Polynomial,Evaluate . $) .
```

A statement like (PolynomialEvaluate . $) above indicates that, in this domain the operation Evaluate with this modemap is to be computed by applying the LISP function Polynomial,Evaluate to the relevant arguments and to the vector representing this domain, which serves as an environment for that function. Note that, following the discussion under "The conceptual Alist for operations" on page 23, this includes only domain-specific operations.

2 List of attributes.

3 The LISP data structure representing the category whose shape this vector has. This would typically be a list of one element, whose car is the atom naming the category, but for a Module, say, it would be the more complicated data structure (Module R), where R was the argument to the category-instantiating function Module.

If, however, the immediate category to which this domain belongs is anonymous, then we will place here the structure representing the closest named principal ancestor of this category. An example of this case arising is:

```
ZModN<n:Integer ] n > 1>:
  Join(QuotientObject(DifferentialRing,Integer),Finite) with
  [assert] if prime?(n) then GaloisField
```

The reason for this is that it is not interesting to know that this domain belongs to an anonymous category, whereas knowing that it is a QuotientObject is more useful. Note that it is an extension of all its principal ancestors (and only its principal ancestors), so that is the data we place here.

4 A pointer to the Alist which "really" represents the domain.

5 A pointer to a vector holding all the local storage required by the domain. Typical examples of such local storage are the cells

```
hi: Integer := n quo 2
lo: Integer := hi - n + 1
```

required by ZModN.

6 et seq. Implementations for all the category varying operations and pointers to vectors corresponding to other categories to which this domain belongs, and other domains (such as PrintBox) required by operations for this domain. 1 These are stored in the correct slots, as defined by their sequence numbers (see page 11).

Items 0, 1, 2, 4 and 5 are EQ in the LISP sense between all vectors representing the same domain. In the case of item 5, the local variables, this is for more than simple reasons of space economy: since multiple such vectors can be being manipulated at the same time, having only one place in which local variables can live, which is shared by all the views, is essential. We note that there are implications for an optimising compiler here - these variables are shared as surely as variables in a FORTRAN COMMON block, and the shared values must be updated every time a function is called.

REFERENCES

Davenport & Jenks,1980 Davenport,J.H. & Jenks,R.D., MODLISP - An Introduction. Proc. 1980 LISP Conference, The LISP Company, Stanford, California, 1980.

Goguen et. al,1978 Goguen,J.A., Thatcher,J.W. and Wagner,E.G., An initial algebra approach to the specification, correctness and implementation of abstract data types. IBM Research Report RC-6487, October 1976. Current trends in Programming Methodology IV: Data Structuring (R.T. Yeh, ed.) Prentice Hall, New Jersey, 1978, pp. 80-149.

Jenks,1979 Jenks,R.D., MODLISP. Proc. EUROSAM 79 (Springer Lecture Notes in Computer Science 72, Springer-Verlag, Berlin-Heidelberg-New York, 1979) pp. 466-480.

Jenks & Trager,1981 Jenks,R.D. & Trager,B.M., A Language for Computational Algebra, Proc. SYMSAC 81 (ACM, New York, 1981) pp. 6-13.

TYPE EQUIVALENCE IN SCRATCHPAD

ABSTRACT This chapter attempts to describe the various issues relating to type equivalence in the SCRATCHPAD language, and finishes with some concrete representations. Throughout this paper, examples will be quoted in a variety of syntactic notations, which should not be taken as part of the SCRATCHPAD language. In particular, no attempt has been made to harmonise quotations.

Definitions For the purpose of this chapter, two types are said to be equivalent if the SCRATCHPAD system treats them as being the same type. In particular, it is possible to assign values from one type to an equivalent type without any explicit coercions being necessary, or any warning messages being generated. Note that this implies, at least naively, that all assignments must be reversible, and so rules out a language in which integers may freely be assigned to reals, but not the converse. To do otherwise seems to place us in the ALGOL68 coercion mess.

This is to be contrasted with the concept of compatibility, by which we mean that it is possible, by explicit coercion, or by compiler-supplied (but warned about) conversion, to convert values from one type to another.

THE ISSUES

In this section, we attempt to give several examples, as they might arise in SCRATCHPAD, of the type equivalence problem, in the hope that consistent answers to these problems will answer the question "when are two SCRATCHPAD types equivalent". We work in two sections: questions of principle and questions of implementation, though the distinction between the two is certainly not rigid.

Questions of Principle

Here we try to list some of the questions that any typed language should answer about its type equivalence algorithm.

Equivalence of named types

A named type is defined to be one to which a specific name has been assigned by means of an == statement. In the following piece of code, is the assignment valid (as it stands)? (Term is assumed to be a pre-defined type, with no complications)

```
t1 == List Term;
t2 == List Term;
x : t1;
y : t2;
y := x;
```

Equivalence of anonymous types

In contrast, an anonymous type is one to which no such explicit name has been attached. This is a particularly important concept in a language in which type expressions, such as "List Term" below, can occur. In the following piece of code, is the assignment valid (as it stands)? (Term is, again, assumed to be a pre-defined type, with no complications)

```
x : List Term;
y : List Term;
y := x;
```

Relation between named and anonymous types

In the following piece of code, is the assignment valid (as it stands)?

```
t == List Term;
x : List Term;
y : t;
x := y;
```

Basic types

In the following piece of code,(22) is the assignment valid (as it stands)?

```
x : Integer;
y : Integer;
y := x;
```

----

(22) I know of no-one who would deny that it was valid, but there are substantial implementation questions arising from the difference between this and "Equivalence of anonymous types".

Questions of implementation

In this section, we look at various questions that arise in the particular context of SCRATCHPAD, which complicate, as well as motivate, the discussion of data types.

Basic Types

In the SCRATCHPAD system, the internal representation of the type "Integer" is the LISP structure (Integer), which causes, when instantiated(23) causes the function Integer to be called, and to return a vector "representing" the type Integer. Similarly, the internal representation of the type List Term is the LISP structure (List Term), which causes the function List to be applied every time List Term is instantiated, and to return a vector representing this type.

Thus, it is hard to see how we can differentiate "Equivalence of anonymous types" on page 28 and "Basic types" on page 28 . It could be argued, of course, that this repeated instantiation is inefficient, but this issue is beyond the scope of this discussion.

Named Types

No decision(24) has yet been made in the SCRATCHPAD system as to the implementation of statements like t1 == List Term. The obvious implementation is as (SETQ t1 (List Term)), but this would make "Equivalence of named types" on page 27 behave exactly like "Equivalence of anonymous types" on page 28 . Perhaps it ought to mark the resulting vector in some way with its "t1"-ness.

Passing types out

This is something that no previous language, that I know of, really implements, but several SCRATCHPAD algorithms are going to want to return a member E of some domain D, and the nature of that domain is determined at run-time,

----

(23) a run-time operation, in the sense that, every time the piece of code

```
i : Integer
j : Integer
i+j
```

is executed, the function Integer is called.

(24) That the author is aware of.

though it will (and, one hopes, can be guaranteed by the type-checker) to lie in some category.

This causes nasty scope problems, as can be seen by considering

```
      t1 == List Term;
      x : t1;
      y := Fred x;
      if x = y then ...
                where Fred ...
                     t1 == List Term;
                     z : t1
                     z;
```

and we ask if the two t1 are the same or not.

A similar problem would arise if we allowed "own" types, or some way of keeping types between invocations of a program.

Uniform semantics

This relates to the problem of compiled and interpreted code. We want the same actions to take place in one, and, in particular, if we decided that "Equivalence of named types" on page 27 was invalid, but "Equivalence of anonymous types" on page 28 was valid, we would not want the compiler to reject "Equivalence of named types" but the interpreter to accept it because, after it had processed all the declarations, all it had associated with x was List Term, and it had forgotten that this came from t1, and was not to be confused with the List Term that came from t2.

Information hiding

It is generally accepted that one of the major principles of an abstract data type language, such as SCRATCHPAD aims to be, is "information hiding" in the following sense: in the body of the code

```
  FreeModule(R:Ring,S:OrderedSet) ==
    add
      [representations]
         Term == Record(k:S,c:R ] c # 0)
         Rep == List Term
         .....
```

an object declared (explicitly or implicitly) as a member of FreeModule is equivalent(25) to one declared as List Term, but outside the scope of this representation no such equivalence applies.

The "ZMod?" problem

BMT has raised the question of a ZMod? type, which would be useful in implementing Hensel's Lemma, and analogies of this could be used for power series domains etc. While this domain does not appear to have been fully thought out, it would have "state", since it would know to what precision it was expected to compute (say modulo 729 or 2143), and there would be a function for changing the state, so that

```
D == ZMod?;
x,y : D;
x := 500;
y := 500;
D.setprec(729);
x + y;     [returns 271]
D.setprec(2143);
x + y;     [returns 1000]
```

would work. This system works fairly well with a name-equivalence view (i.e. one in which "Equivalence of named types" on page 27 to "Relation between named and anonymous types" on page 28 are illegal, but in a structure equivalence view of the word we have problems, because we must ask whether we include the internal state of the domain when comparing structures.

If we do, then the compiler has to know what the internal state is, and this means running the program, and, in general, makes the general type-equivalence problem undecidable.

But, if we ignore the state of the domain when comparing, we arrive at the following situation:

----

(25) This need not be the case. It could be that they were merely compatible, and that explicit (or compiler-supplied) coercions were required. Taking this position solves some problems, but either makes the code look much more ugly or complicates the type question, since, having decided that two types are not equivalent, the compiler must then ask if they are implicitly coercible.

```
      D1,D2 == ZMod?;
      x : D1;
      y : D2;
      D1.setprec(729);
      D2.setprec(81);
      x := 500;      [really is 500]
      y := x;        [becomes 14]
      x := y;        [is still 14]
```

where we have lost precision, and the type-checker can tell us nothing about it.

The top-level interpreter

While the top-level interpreter for SCRATCHPAD is not written, we know that it will follow previous designs in trying to maintain a common domain of computation, using calls to RESOLVE etc. to assist in this. If "Equivalence of anonymous types" on page 28 is invalid, then it will have to maintain some form of caching for domain structures to ensure that it never instantiates the same domain more than once, else the user will see a "type mismatch" error that he did not expect. Not only this, but this caching must apply at all levels of the domain tree, not merely at the top level.

Implicit parametrisation

It is possible for a module to create, and export, a structure which is implicitly parametrised. An example of this is provided by

```
  IntegralDomain:Category == Ring with
      [operations] unitNormal: $ -> UnitCorrAssoc
                   UnitCorrAssoc == Record(unit:$,canonical:$,associate:$)
                   .....
```

where a piece of code that instantiates an IntegralDomain may later call the function unitNormal on an element of it, thus ending up with an object of type UnitCorrAssoc. But this object is really parametrised by the $ of IntegralDomain, and it is certainly the case that the following code is illegal:

```
      x : Integer;
      x := -2
      y : Polynomial<t> Rational;
      y := 3*t+4;
      unitNormal(x) = unitNormal(y)
```

despite the fact that, superficially, we are comparing objects of type UnitCorrAssoc.

The answer to this, it seems, must be to parametrise the type UnitCorrAssoc with the type of the particular IntegralDomain it refers to, so that unitNormal(x) would have type UnitCorrAssoc(Integer), while unitNormal(y) would have type UnitCorrAssoc(Polynomial(t,Rational)). Whether the system does this behind the user's back or whether the user should have written

```
  IntegralDomain:Category == Ring with
      [operations] unitNormal: $ -> UnitCorrAssoc($)
                   UnitCorrAssoc($) == Record(unit:$,canonical:$,associate:$)
                   .....
```

is largely a matter of taste: the present author would vote for the second alternative at the moment.

This does have a disadvantage, though, inasmuch as, for consistency, one would have to write

```
      [representations]
         Term(R,S) == Record(k:S,c:R ] c # 0)
         Rep == List Term(R,S)
```

instead of

```
      [representations]
         Term == Record(k:S,c:R ] c # 0)
         Rep == List Term
```

in the functor FreeModule. The real reason why UnitCorrAssoc needs to be parametrised, while Term does not, is that the type UnitCorrAssoc is (implicitly) exported from IntegralDomain whereas, at least in current implementations, Term is not exported from FreeModule. This question seems to require further study.

PREVIOUS WORK

Here we look at a small amount of the previous work that has been done on questions of type equivalence, noting that there is a substantial literature on the subject, with little standardisation of vocabulary.

The "Russell" camp

Demers et al. [1978] consider the following PASCAL fragment

```
         type t = integer;
         var x : integer;
         var y : t;
         x := 0
         y := x
```

and ask whether or not it is well-formed, i.e. whether or not y := x is permitted. They point out that the PASCAL report [Jensen & Wirth,1975] does not comment on this issue, while EUCLID [Lampson et al.,1977] would consider it valid and Alphard [Wulf et al.1976] would not.

Demers et al. [op. cit.] incline to the view that it is not valid, though the author finds it somewhat hard to follow their reasoning. They argue that the program fragment quoted above should be equivalent to:

```
procedure p( type t with (dcl, :=)
           var x : integer;
           var y : t;
           x := 0
           y := x
           end p;
p(integer)
```

and, in this context, the assignment is clearly illegal.

"Pascal" and sons

Welsh et al. [1977] discuss this matter in the context of clearing up some ambiguities in PASCAL, and consider the following example:

```
type T = array [1..10] of INTEGER
var A,B: array [1..10] of INTEGER
      C: array [1..10] of INTEGER
      D: T;
      E: T;
```

In this context they consider two possible definitions of equivalence, as follows:

Name equivalence. "Two variables are considered to be of the same type only if they are declared together (as A and B) or if they are declared using the same type identifier (as D and E)." "Thus A, C and D all have different types." They add that "primitive types are specified using type identifiers, so two variables will have the same type if they are both declared INTEGER."

Structural equivalence. "Two variables are considered to be of the same type whenever they have components of the same type structured in the same way." All five above have the same type. This gives rise to two sub-definitions of structural equivalence,(26) which I shall call Structural equivalence with selectors and Structural equivalence without selectors The difference arises when one considers the following piece of code:

----

(26) Ichbiah et al. [1979, pp. 4-5,4-6] produce eight different elaborations of structural equivalence.

```
var F: record T,U: REAL end;
    G: record V,W: REAL end;
```

and ask if F and G have the same type. Structural equivalence with selectors would say that they do not, since the names of the selectors are different, while structural equivalence without selectors would regard them as having the same type.

There appears to be little support for structural equivalence without selectors in the literature, and we shall not discuss it further; assuming "structural equivalence" to mean "structural equivalence with selectors". selectors in the literature, and we shall not discuss it further; assuming

We should note that ADA [Ichbiah et al., 1979, p.4-2] chooses name equivalence in the following form: "Each type definition introduces a different type".

EUCLID [Lampson et al., 1977, p.31] chooses structural equivalence (with provision for information hiding, as discussed in "Information hiding" on page 30) in the following words: "A type identifier is an abbreviation for its definition. After all such abbreviations have been removed, two types are the same if their definitions look the same. However, a module type, or any type exported from a module, is considered to be different from any other type".

Summary. The literature divides essentially into two: name equivalence (in which "Equivalence of named types" on page 27 to "Relation between named and anonymous types" on page 28 are invalid, but "Basic types" on page 28 is valid) and structural equivalence (in which all four are valid).

POSSIBILITIES FOR SCRATCHPAD

Here we discuss the various proposals that have been suggested, with varying degrees of formality, for the SCRATCHPAD language.

Structural equivalence

This is a very simple proposal, akin to the EUCLID implementation (see the remark quoted above), and suggests that we implement equivalence of data types by asking if the two data types "look" the same. Thus all that is required, in the vector implementation of domains, is to compare the first elements of the vectors, which will be LISP structures of the form (Integer), (List Term), (Polynomial x (Integer)) etc.

It implies that the compiler would have to de-reference structures like

```
t1 == List Term;
x : t1;
```

in order to ensure that x was "really" a List Term, rather than a t1.

It also has slight problems with the information hiding mentioned "Information hiding" on page 30, inasmuch as, if the two lists do not compare, one has to see if one is a valid (i.e. in scope) representation of the other.

A major problem with this approach is that, the following declarations, such as might be made in the course of a factorisation package,

```
type SquareFreeFactorisation: List Polynomial;

type PrimeFactorisation: List Polynomial;

var x : SquareFreeFactorisation;

var y : PrimeFactorisation;
```

provide no assistance from the type-checker, since it thinks that SquareFreeFactorisation and PrimeFactorisation are short-hand for the same objects.

Name equivalence.

This, too, appears a very simple process (akin, this time, to the ADA implementation): we compare the "names" of types, so that, at compile time, the compiler compares t1 and t2 (using EQ), or (List Term) and (List Term) [getting different results in case "Equivalence of anonymous types" on page 28, since they are two different, though EQUAL, lists]. This has the drawback that, at least naively, "Basic types" on page 28 would also appear illegal, since the two instantiations of (Integer) would appear different. It also has slight problems with "The top-level interpreter" on page 32.

A somewhat different interpretation is required in interpreted code, since we must ensure that the interpreter retains the t1-ness of x in code such as "Equivalence of named types" on page 27. Though I do not recall this proposal being made before, it seems that we would need to reserve an extra slot in vectors for the name of the object, which would be t1 in the case of

```
t1 == List Term;
x : t1;
```

and some unique(27) identifier in the case of

```
x : List Term;
```

and possibly in the case of

```
x : Integer;
```

----

(27) This could either be a "gensym" or an integer returned by a function guaranteed never to return the same integer twice. It appears, from discussion with various LISP/370 people, that the integer approach would probably be much more efficient.

"EQ" equivalence

BMT has made this proposal, that equivalence of types be determined by the two vectors that represent them being EQ (in the usual LISP sense), as opposed to EQUAL. In the case of compiled code, we would then check identifiers, and t1 (in the scenario of "Equivalence of named types" on page 27) is not EQ to t2. This would obviate the necessity for the "unique identifier" mentioned above in the case of List Term, but would prevent two Integer instantiations from being the same. This could be by-passed by a system in which functions like Integer were massaged only to compute a result once, and deliver the same result whenever they were called. This would not, however, seem to solve the "information hiding" problem of "Information hiding" on page 30, unless additional checks were made after the EQ failed.

JHD's proposal

This proposal, which has gradually solidified after looking at the other possibilities, is as follows:

1. Two named types are only equivalent if the names are the same (i.e. "Equivalence of named types" on page 27 is illegal);

2. Two anonymous types are equivalent by the rules of structural equality (with selectors), thus making "Equivalence of anonymous types" on page 28 and "Basic types" on page 28 legal;

3. An anonymous type is never equivalent to a named type (i.e. "Relation between named and anonymous types" on page 28 is illegal);

4. If the above rules state that the two types are inequivalent, then a check is made to see if one is the representation (in scope) of the other.

This is implemented by planting a name field in every vector (as discussed for name equivalence), but this field is left NIL for anonymous types, so that (List Term) can be implemented in LISP as (List (Term)), but t1 == List Term is implemented as

```
  (SETQ t1 ((LAMBDA (Z) (PROGN (SETELT Z 2 "t1) Z)) (List Term))).
```

This seems, naively, to the author to solve the problems of "Questions of implementation" on page 29(though not at no cost):

- o "Basic Types" on page 29 is solved since all instantiations of Integer will have the name NIL, and will so be equivalent.

- o "Passing types out" on page 29 is still somewhat of a problem, but we can solve these scope problems the way we solve other ones: by prefixing the SCRATCHPAD name with a description of its level before making it into a LISP name, as in Integer,Plus,1.

- Since named types retain their names at run-time, while anonymous types are EQUAL at compile-time and at run-time, it can be hoped that the semantics of compiled and interpreted code agree (as requested in "Uniform semantics" on page 30).

- As with all these schemes, we still need a mechanism to ensure that, within a module, its data-types can see their representation (see "Information hiding" on page 30). Outside the module there is no problem, since (Polynomial x (Integer)) is not equal to (List (Term)), and this inequality will persist at run-time. Rule (4) above will need to be build into the type-checker, and it is well to recognise that fact.

- "The "ZMod?" problem" on page 31 is solved by our reliance on name equivalence for named types, since, in order for it to make sense to have a ZMod? construct around, it has to be named to allow one to refer to its setprec function.

- Since the top-level interpreter will be constructing anonymous types, all the types it constructs (of the same structure) will be equivalent, which should ensure that the naive user sees what he expects.

- The problem of implicit parametrisation must be dealt with in one of the ways suggested in "Implicit parametrisation" on page 32.

Against this must be balanced the fact that this proposal is a compromise between the two main camps of name and structure equivalence. While it appears to have the advantages of both, it may later turn out also to have the disadvantages of both.

REFERENCES

Demers et al.,1978 Demers,A., Donahue,J. & Skinner,G., Data Types and Values: Polymorphism, Type-checking, Encapsulation. Proc. Fifth ACM Symposium on Principles of Programming Languages, ACM, 1978, pp. 23-30.

Ichbiah et al.,1979 Ichbiah,J.D., Barnes,J.G.P., Heliard,J.C., Kreig-Brueckner,B., Roubine,O. & Wichmann,B.A., Rationale for the Design of the ADA Programming Language. SIGPLAN Notices 14(1979) 6B (entire issue).

Jensen & Wirth,1975 Jensen,K. & Wirth,N., PASCAL User Manual and Report. 2nd. ed., Springer Verlag, Berlin-Heidelberg-New York, 1975.

Lampson et al.,1977 Lampson,B.W., Horning,J.J., London,R.L., Mitchell,J.G. & Popek,G.L., Report on the Programming Language EUCLID. SIGPLAN Notices 12(1977) 2 (entire issue).

Welsh et al.,1977 Welsh,J., Sneeringer,W.J. & Hoare,C.A.R., Ambiguities and Insecurities in Pascal. Software - Practice and Experience 7(1977) pp. 685-696.

Wulf et al.,1976 Wulf,W., London,R.L. and Shaw,M., Abstraction and Verification in Alphard: Introduction to the Language and Methodology. USC/ISI Research Report 76-46, June 1976.

INNARDS OF THE SCRATCHPAD SYSTEM

Introduction: Here we attempt to describe some of the internal workings of the system, especially those parts which the author considers others may have difficulty with. This means that the list is closely related to the list of things the author himself had difficulty with.

THE LISP ITERATORS.

In this section we describe the form in which the iterators are implemented the YKTLISP implementation of SCRATCHPAD, and not those of the LISP/370 implementation. The reason is that the YKTLISP conversion has just been made, and it was an attempt to define this that lead to the present(28) codification. The LISP/370 implementation is supposed to provide the same facilities, but this may not always be the case, especially when it comes to the finer points of what is wrapped round the user's code.

REPEAT and COLLECT

There are two(29) basic iterators provided by the iteration package, COLLECT and REPEAT The difference between them is precisely the difference between MAPCAR and MAPC: COLLECT returns a (useful) value, while REPEAT (in its simplest form) does not. In what follows we will generally concentrate, for simplicity, on REPEAT, and let the reader generalise to COLLECT.

The general form of REPEAT is given by Figure 10 on page 41.

----

(28) This section was written 19th. September, 1981. The author believes that changes may have been made between that date and the date that this footnote was written (15th. July, 1982.

(29) There is also the DO macro, in terms of which the others are defined. This macro is also used for bootstrapping the definitions of REPEAT and COLLECT. However, its specification is obscure, and the macro should not be called directly by users.

```
|--------------------------------------------------------------------------|
| (REPEAT                                                                  |
|    <iterators>                                                           |
|    <body>)                                                               |
|                                                                          |
| such as:                                                                 |
|                                                                          |
| (REPEAT                                                                  |
|   (IN X L1)                                                              |
|   (STEP I 1 1 100)                                                       |
|   (ON Y L2)                                                              |
|   (SETELT X I Y))                                                        |
|                                                                          |
| Figure 10.  The REPEAT Construct                                         |
|--------------------------------------------------------------------------|
```

A wide variety of iterators are supported, and these lead to great variations in the nature of the code produced. However, some common principles apply. All these macros can be regarded as expanding into the following structure:

```
  (PROG(30)
     (variable bindings)
     initialisation statements
  Label
     iterator-related statements
     (SEQ (EXIT body))
     more iterator-related statements
     (GO Label))
```

This has the following consequences:

1. An EXIT(31) within the body will cause that iteration through the loop to terminate, and the next iteration to begin. Note that the value quoted in the EXIT is ignored. This can be compared with the ITERATE statement found in many modern languages (e.g. REX, except that no provision is made for controlling loops other than the innermost one). In the case of a COLLECT, the value is not added to the list being collected.

2. A RETURN(32) within the body causes the entire loop to terminate, and the value of the entire REPEAT or COLLECT statement is that quoted in the RETURN statement. This can be compared with the LEAVE statement found in many modern languages (e.g. REX, except that no provision is made for controlling loops other than the innermost one).

----

(30) REPEAT in fact generates a LAMBDA, but this is inconsequential, since PROG is, in fact, merely a macro for LAMBDA anyway. COLLECT generates two nested PROGs, but the difference is again immaterial.

(31) Unless masked by an inner SEQ or PROG, of course.

(32) Unless masked by an inner PROG, of course.

3. The bound variables of the iteration are bound by the iteration, and therefore can not be referred to outside the loop, nor can they affect other variables outside the loop.

We show (in Figure 11 on page 43 and Figure 12 on page 44) some actual examples of simple REPEAT and COLLECT statements now, so that the reader can get some feel for the code. These are precisely as generated by the Yorktown LISP system, except that the names of GENSYMmed variables have been changed to (lower-case) mnemonic names.

The various iterators supported by the macro are:

IN (IN A B) causes the variable A to run over the various members of the list B in turn, so that its successive values are (CAR B), (CADR B), (CADDR B) ... . A is bound by the iteration, but the initial value of B is evaluated outside the iteration, and so is not affected by any other bindings the iteration may cause. This means that the following piece of code does the 'right' thing, though the author would not recommend it as an example of clear programming:

```
(SETQ A "(1 2 3 4))
(SETQ N 1)
(REPEAT (IN A (REVERSE A))
        (SETQ N (PLUS N A)))
```

ON (ON A B) works exactly like (IN A B) except that A takes on successive tails of B, so that its values are B, (CDR B) , (CDDR B), ... It stops as soon at it reaches an atomic value, so one can guarantee that A is always a pair. The remarks about variable bindings made under IN also apply to ON.

STEP (STEP I initial increment final) makes I take all the values 'initial', 'initial+increment', 'initial+2*increment' ... that are not greater than 'final', or, in the case that the increment is(33) a negative integer, not less than 'final'.

If 'final' is omitted the loop will run for ever (unless terminated by a RETURN statement, of course). The value of 'initial' is evaluated precisely once, outside the scope of the loop. 'Increment' and 'final' are evaluated every time their values are needed, and these evaluations take place inside the loop, and so see all the bindings of the loop. Hence the following code will step I through the triangular numbers (though, again, it is not recommended as good programming practice):

----

(33) At macro-expansion time. This could be regarded as a deficiency in the macros, and they probably ought to be altered to produce generalised code in the event that they can not determine the sign of the increment at compile time. However the current state is that described.

```
(REPEAT (IN A B) (SETQ V (CONS A V))))
becomes
( (LAMBDA (Blist A)
    (SEQ
  Label
      (COND
        ( (OR (ATOM Blist) (PROGN (SETQ A (CAR Blist)) NIL))
          (SEQ (RETURN NIL)) ) )
      (SEQ (EXIT (SETQ V (CONS A V))))
                  ___________________
      (SETQ Blist (CDR Blist))
      (GO Label) ) )
  B
  NIL )
```

Figure 11. An example of REPEAT: with the actual body of the loop underlined.

```
(REPEAT (STEP J 1 1 100)
        (STEP I 1 (ADD1 J))
        (PRINT (LIST J 'th triangular number is ' I)))
```

UNTIL (UNTIL x) will cause the loop to stop as soon as x (which is evaluated inside the environment of the loop) becomes true.

WHILE (WHILE x) is a synonym for (UNTIL (NOT x))

EXIT (EXIT(34) x) states that the value of REPEAT is to be x, rather than NIL. When used with COLLECT, it causes the collection to be skipped for that iteration.

SUCHTHAT (SUCHTHAT x) says that the body of the loop should only be executed when x is true. It is equivalent to replacing the body with (COND (x body)). A good example of its use is:

```
(COLLECT (IN A L)
         (STEP I 1 1)
         (SUCHTHAT (PRIMEP I))
         A)
```

for selecting those elements of a list that occur in prime-indexed positions.

---

(34) Not to be confused with the LISP statement EXIT. Perhaps this one ought to be renamed VALUE, or some other keyword, especially since this statement does not cause an exit, merely states what to do when an exit is taken.

```
(COLLECT (IN A B) (LIST A)))
becomes
(PROG (Answer)
  (SETQ Answer NIL)
  (RETURN
    ( (LAMBDA (Blist A)
        (SEQ
      Label
          (COND
            ( (OR (ATOM Blist) (PROGN (SETQ A (CAR Blist)) NIL))
              (SEQ (RETURN (NREVERSE0 Answer))) ) )

          (SEQ (EXIT (SETQ Answer (CONS (LIST A) Answer))))

          (SETQ Blist (CDR Blist))
          (GO Label) ) )
      B
      NIL ) ) )
```

Figure 12. An example of COLLECT: with the actual body of the loop and the RETURN statement that yields the result underlined.

The REDUCE statement

The REDUCE statement is closely connected with the iterators described above. The general form of the REDUCE statement is

```
(REDUCE operator axis list)
```

such as

```
(REDUCE PLUS 0 "(1 2 3 4))
```

which will add up all the members of the list. The following reduction allows us to reduce all cases of reduction to the case of zero-axis (which is, almost certainly, the most common case):

```
(REDUCE operator axis list)
is equivalent to
(COLLECT (IN U list)
         (REDUCE operator axis-1 U))
```

In general, REDUCE generates code to iterate down a list, applying the operator to the partial result and the next term. However, the macros are clever in the case of reducing over a collected list, as can be seen in Figure 13 on page 45, and the list is never actually created.

One result of this cleverness is that the semantics of EXIT and SEQ, as defined on page 41, are altered. They now become the following:

```
(REDUCE APPEND 0 (COLLECT (IN A B) (LIST A))))
becomes
(PROG (Answer)
  (SETQ Answer NIL)
  (RETURN
    ( (LAMBDA (Blist A)
        (SEQ
      Label
          (COND
            ( (OR (ATOM Blist) (PROGN (SETQ A (CAR Blist)) NIL))
              (SEQ (RETURN Answer)) ) )
          (SEQ (EXIT (SETQ Answer (APPEND Answer (LIST A)))))
                     ________________________________________
          (SETQ Blist (CDR Blist))
          (GO Label) ) )
      B
      NIL ) ) )
```

Figure 13. An example of REDUCE: With the answer-producing code underlined.

1. An EXIT(35) within the body will cause that iteration through the loop to terminate, and the next iteration to begin. Note that the value quoted in the EXIT is ignored.

2. A RETURN(36) within the body causes the entire loop to terminate, and the value of the entire REDUCE statement is that quoted in the RETURN statement.

THE IMPLEMENTATION LANGUAGE - SOME REMARKS ON THE SYNTAX

This section concentrates on the language in which the system is implemented (known as the BOOT language), and in particular on some of the pitfalls it has. These are, of course, subject to change (rectification?) without notice, but perhaps this list will be useful to those who do not obtain the results they expect.

Functions of no arguments. GENSYM() applies the function GENSYM to no arguments, GENSYM () applies the function GENSYM to one argument - NIL.

"is true" The construct 'x is true' translates into "T at the LISP level.

'=>' The use of the '=>' (thenexit) symbol can be confusing. In the code

----

(35) Unless masked by an inner SEQ or PROG, of course.

(36) Unless masked by an inner PROG, of course.

```
atom x =>
  t:=
    isSymbol x => compSymbol(x,e)
    <x,primitiveType x or return nil,e>
  convert(t TO m)
```

the high-lighted '=>' is subordinate to the ':=' on the previous line, and so the value of a call to compSymbol is assigned to the variable 't'.

"if" and indenting The rules for indenting the "if" statement are that anything subordinate to the predicate of the 'if' must be indented at least as far as the first character of the operand, so that

```
..if a and b
    and c then..
```

is erroneous, while

```
..if a and b
      and c then..
```

has the desired effect.

not The not operator in the boot language has a very low precendence. In particular, its precedence is less than that of '=>', so that

```
not( (CAR sig) = (CAR pattern)) => nil
```

parses as

```
not( ((CAR sig) = (CAR pattern)) => nil )
```

and thus exits under the same circumstances as

```
(CAR sig) = (CAR pattern) => nil
```

which is probably not what the programmer expected.

SOME DEBUGGING AIDS

The TRACEUT system

TRACEUT LISP K offers(37) a convenient tool for debugging. By doing a:

(37) This description was copied from JENKS MAIL 81/09/14 01:44:24

```
          MAKEPROP(fn,"/TRANSFORM,"(tr t1 ... tn))
```

followed by:

```
          (TRACE fn)      or      (/t fn)
```

the n arguments a1,...,an of fn are transformed by t1,...,tn, the result of fn by tr. In all transforms, * denotes the expression, NIL is used if the argument is not to be printed. As an example, TRACEUT contains:

```
     (MAKEPROP "compFormWithModemap "/TRANSFORM "((DROPENV *) * * NIL *))
```

which causes arguments 1,2, and 4 to be printed out on call, argument 3 (the environment) not to be printed, and the result transformed by DROPENV (which simply drops the environment part of the result) before printed.

TRACEUT LISP K contains the triggers which cause the TRACE functions to act specially as well as some suggested MAKEPROPs. Of course, you are free to make your own private copy.

INDEX