THE SPECIFICATION OF DATA TYPES

C.B. Jones

Rapporteurs: Mr. S.B. Jones
Dr. P. Henderson

Abstract:

Work on proving programs correct has reached a fairly sophisticated stage for programs of a numerical nature. This work has also been carried into rigorous program development. Many people are interested in extending this work into the data processing area, as an advanced software development tool, and I consider that the key to doing this is a proper formulation of the notion of abstract data types. In these lectures I shall concentrate on the formulation of specifications of the software which is to be built. I wish also to show how to construct a body of knowledge about data types which will be useful for many applications.

Lecture 1: Method of specifying data types

Figure 1 shows the "factorial" of abstract data types; this is the stack example. Here the method of description is algebraic, or axiomatic. It is an implicit, or property oriented, method. The first five lines of the figure are the syntactic part of the definition, showing the operations which are available, and giving the domains and ranges of each of the operations. The remaining six lines of the figure are the semantic part which defines a last-in first-out stack discipline by interrelating the available operations. The use of the UNDEFINED element has made INSPECT a total function.

```
NEWSTACK: → Stack
PUSH: Stack El → Stack
INSPECT: Stack → (El|UNDEFINED)
REMOVE: Stack → Stack
ISEMPTY: Stack → Bool
          ~
REMOVE(NEWSTACK()) = NEWSTACK()
REMOVE(PUSH(st,el)) = st
INSPECT(NEWSTACK()) = UNDEFINED
INSPECT(PUSH(st,el)) = el
ISEMPTY(NEWSTACK()) = TRUE
ISEMPTY(PUSH(st,el)) = FALSE
```

Figure 1 **Implicit specification of stack**

This set of axioms is consistent and complete, but how do we know that this is so? Guttag has done some very good work in this area, and has given a method for generating the left hand sides of the axioms in such a way as to ensure a complete set.

There is no implication in this specification of a model for stack implementation, such as a linear list. In contrast, however, we can give a constructive, or model oriented, specification, as in Figure 2, where the notation is similar to that Bjorner (1979) has used. The stack is defined as a sequence of its elements, concatenating and removing elements from the left hand end of the sequence. It is easy to see that this models the last-in first-out stack discipline. But is there a danger in introducing such a model? Has this forced a particular implementation, and hence restricted your choice as an implementer of stacks?

```
Stack = El*
          ~

NEWSTACK() ≜ <>
PUSH(st,el) ≜ <el>⌢st
INSPECT(st) ≜ if st ≠ <> then hd st
                         else UNDEFINED
REMOVE(st) ≜ if st ≠ <> then tl st
                        else <>
ISEMPTY(st) ≜ st = <>
```

Figure 2   Constructive specification of stack

The constructive method has the advantage that extending specifications is simpler. Adding a new operation is more straightforward than in the axiomatic case, or changing the stack specification to that of a queue is much simpler in the constructive case.

We can, of course, prove that the constructive model satisfies the axioms. However, taking one particular implementation as your specification has its dangers. For example, consider the specification of Figure 3. This is a constructive specification of a stack in which use is made of a counter and the head of the stack is not deleted during a remove operation. The specification displays what I call implementation bias. It makes the proving of one kind of implementation simple, but the proving of other implementations is much more difficult. Thus the text of Figure 3 is not suitable as a spcification, although it would be an adequate implementation.

```
Stack2:: El* Nat
          ~
NEWSTACK2() ≜ <<>,0>
PUSH2
INSPECT2(<st,c>) ≜ st(c)
REMOVE2 (<st,c>) ≜ <st,c-1>
ISEMPTY2(<st,c>) ≜ c=0
```

Figure 3 A constructive specification with implementation bias

How can we ensure that we do not construct biased specifications? One way is to consider possible implementations and to define retrieve functions. These are related to Hoare's abstraction function and similar functions used by Milner. The retrieve functions allow us to map between implementation models, and in particular between each implementation model and the alleged specification. Here we can construct

$$\text{retr-stack: Stack2} \longrightarrow \text{Stack}$$

but we can not define

$$\text{retr-stack2: Stack} \longrightarrow \text{Stack2}$$

because the implementation of Stack has discarded elements which Stack2 retains.

Hence a possible bias test is to examine the feasibility of retrieve functions for the implementations under consideration. A more mathematical test is described in Jones (1980).

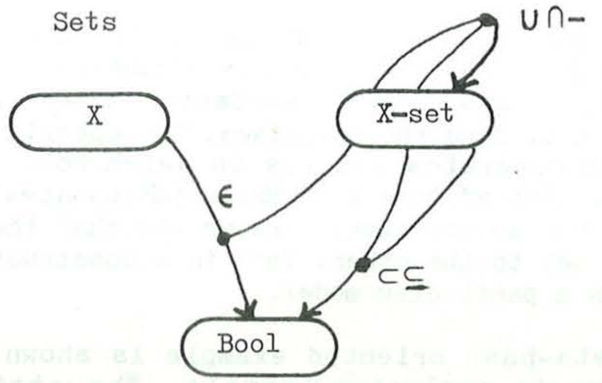
Figure 4 compares the two kinds of specifications which we have used. I do not wish to suggest that constructive methods are always better than implicit methods. In Figure 4 I have compared the corresponding kinds of specification of programming language semantics. The implicit specification method is good for proving the properties of programs written in a language, while the constructive method is good for proving correct an implementation of the language. I think, therefore, that both kinds of specification have an important place.

Most of the literature refers to implicit specification of abstract data types, so let me redress the balance a little in favour of constructive definitions.

	Implicit or Property oriented specification	Constructive or Model oriented specification
Data types	No model - advantage	"over specification" - disadvantage
	Operations together -disadvantage	Operations separate - advantage
	?	consistent
	?	complete
	Good for using objects e.g. Stacks	Good for implementing objects
Programming languages	cf. Hoare axioms	Denotational/Operational Semantics
	Predicate Transformers	
	For proofs of programs written in L	For proof of compilers of L

Figure 4 Comparison of implicit and constructive methods of specification

Figure 5 lists the operations which I will allow on objects of types set, list, mapping and tree. In the case of sets I have given a diagram (based on the work of Thatcher, Goguen, et al) which shows the domains and ranges of each of the operations.



Lists

hd tl len (i)

Mappings

dom rng ∪ + (d)

Trees

mk- s- < > ::

Figure 5   Basic objects and operations

I will introduce the notion of a state, and state updating operations. Figure 6 shows the general form of specifications for such an operation and an example of the definition of POP as a state updating operation. I give the name of the operation, the states on which it is defined, the types of its other arguments and results, and a precondition and postcondition on the use of this operation.

General form:

```
OP
states:S
type: D→R
pre-OP: S D→Bool
post-OP: S D S R→Bool
```

Example:

```
POP
states: Stack
type:→El
pre-POP (st) ≜ st ≠ <>
post-POP (st,st',el) ≜ st' = tl st ∧
                        el  = hd st
```

Figure 6   Specification of a state updating operation

Consider the definition of Figure 7. This defines a data type which keeps track of which students have done certain exercises. Here the state is just two sets of students' names: those who have, and those who have not, done the exercises. The operation INIT is always applicable and generates a state in which both sets are empty. The other operation defined here is COMPL which updates the state when a student completes an exercise. Here we see that the student's name is moved from one set to the other. This is a constructive specification because it uses a particular model.

A more data-base oriented example is shown in Figure 8. It is the familiar parts explosion example. The state is a "bill of materials" (Bom) which is defined as a mapping. The EXP operation returns the set of parts which go to make up a given part, either directly or indirectly. I have used a recursively defined function expl to simplify this definition. The constructive definition is trivial for a non-trivial problem. The trick in finding simple specifications is to choose the "correct" data types. This is the thesis of my forthcoming book (JONES (1980)).

<u>States:</u>

Studx :: N: Student-set
         Y: Student-set

<u>Operations:</u>

INIT
states: Studx
pre-INIT(s) $\triangleq$ <u>TRUE</u>
post-INIT(s,<n',y'>) $\triangleq n' = \{\ \} \wedge y' = \{\ \}$

ENROL

COMPL
states: Studx
type: Student→
pre-COMPL(<n,y>,nm) $\triangleq$ nm $\in$ n
post-COMPL(<n,y>,nm,<n',y'>) $\triangleq n'=n-\{nm\} \wedge y'=y\cup\{nm\}$

Figure 7   Students doing exercises

```
States:

Bom = Part→Part-set

Operations:

EXP
states: Bom
type: Part→Part-set
post-EXP (bom,p,bom',ps) ≜
      bom' = bom ∧ ps = expl(bom,p)
expl(bom,p) ≜
   {p}∪ union {expl(bom,c)| c ∈ bom(p)}
```

Figure 8   The parts explosion problem

Consider the problem of specifying the greatest common divisor, Figure 9. Here I have defined it as a state updating operation. This is an unusual but adequate definition of GCD, but is it a good specification for building a program? Of course not; we know that the Euclidean algorithm is much better. We would not want to implement directly the specification of Figure 9, so we must be wary of direct implementation of specifications and take this as a warning about the use of abstract data types in programming languages. We must use specifications for proving implementations, and expect professional programmers to design reasonable implementations for us.

```
States:

Sgcd:: X:Nat
       Y:Nat

       ~

Operations:

GCD
states: Sgcd
post-GCD(<x,y>,<x', >) ≜
       x' = maxs(divisors(x)∩divisors(y))

maxs: Int-set → Int
divisors(x) ≜ {d | x mod d = 0}
```

Figure 9   Greatest Common Divisor

Lecture 2: Theories of data types

We have described the process of specification and now we proceed towards an implementation. We use a process of refinement. A more concrete representation, and realisation, of a specification can be related to that specification by retrieve functions (Figure 10). We must consider the consistency of this diagram.

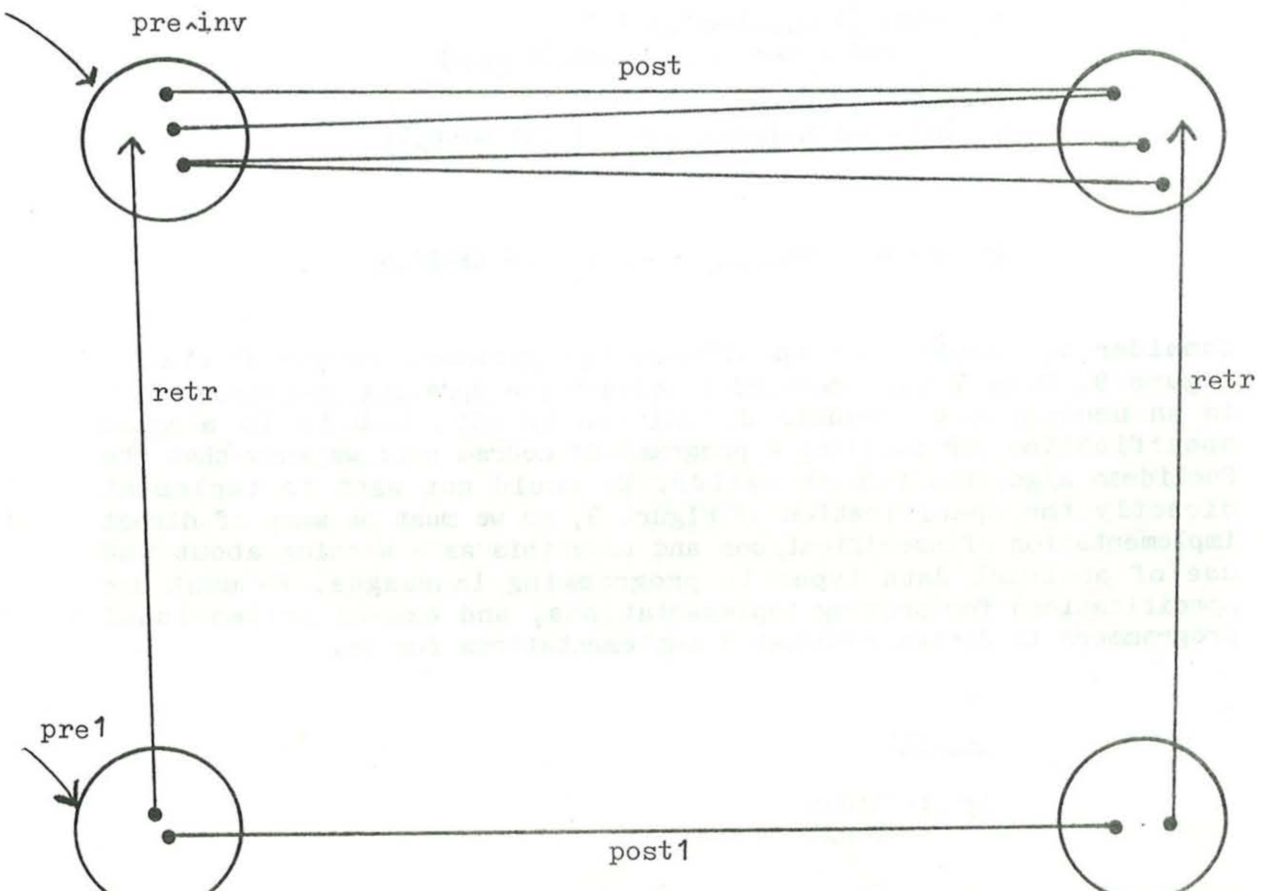
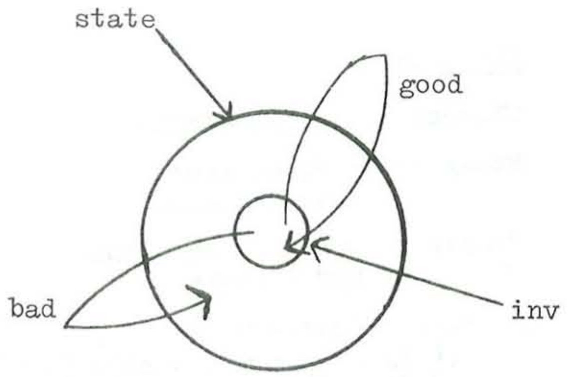


Figure 10   Refinement of a specification

In Figure 11 we define the notion of an invariant on a state, which restricts a valid state to be a member of a subset of all possible states. Good operations preserve the invariant. Bad operations do not. The invariant for the student-doing-exercises problem is shown in Figure 11. It says that the set of those students who have done exercises does not overlap with the set of those students who have not done exercises.

Invariants form an important part of the documentation of a specification. They can help with proving implementations, and can indicate the motivations for certain decisions.



Example:

Students doing Exercises

Studx :: N: Student-set
         Y: Student-set

inv-studx(<n,y>) $\triangleq$ n $\cap$ y = {}

Figure 11   Data type invariant

Consider an implementation which gives each student a number and then defines the two properties by a sequence of bits. The retrieve function which relates this implementation to its specification is shown in Figure 12. A retrieve function maps the implementation state to the abstract state. The retrieve function should form a part of the documentation of the implementation.

```
Specification: State

Representation: State1

retr-state: State1→State

Example:

Students doing Exercises

Studx :: N: Student-set
         Y: Student-set

Studx1 :: NOS : Student→Nat
          RES : Bool*
```

$$\text{retr-studx}(\langle nm,rl\rangle) \triangleq \langle\{n \mid n \in \underline{\text{dom}}\ nm \wedge \sim rl(nm(n))\}, \{n \mid n \in \underline{\text{dom}}\ nm \wedge rl(nm(n))\}\rangle$$

Figure 12   Retrieve function for the students problem

Another notion we need is that of adequacy (Figure 13). This formulates the notion that for every abstract state there is at least one implementation state.

In addition, we need to relate the before and after states at both the abstract and implementation levels, as shown in Figure 14.

$$(\forall s \in \text{State})(\exists s1 \in \text{State1})(s=\text{retr-state}(s1))$$

$\sim$

With data type invariants:

aa.

$$(\forall s1)(\text{inv1}(s1) \Rightarrow (\exists s)(\text{retr-state}(s1)=s)) \wedge \text{inv}(\text{retr}(s1)))$$

ab.

$$(\forall s)(\text{inv}(s) \Rightarrow (\exists s1)(\text{inv1}(s1) \wedge s=\text{retr-state}(s1)))$$

Figure 13   Adequacy criteria

$$\text{post-OP1}(s1,s1') \Rightarrow \text{post-OP}(\text{retr}(s1),\text{retr}(s1'))$$

~

da.

$$(\forall s1)(\text{inv1}(s1) \wedge \text{pre-OP}(\text{retr}(s1)) \Rightarrow \text{pre-OP1}(s1))$$

ra.

$$(\forall s1)(\text{inv1}(s1) \wedge \text{pre-OP1}(s1) \wedge \text{post-OP1}(s1,s1') \Rightarrow \text{post-OP}(\text{retr}(s1),\text{retr}(s1')))$$

Figure 14 Relating abstract and implementation levels

Let me summarise what we have achieved so far. We can give two kinds of specification of an abstract data type, either implicit or constructive. I have argued that for some purpose the constructive definition is to be preferred. Such definitions are short, precise and we do have a test to determine whether or not there is any bias which over-constrains the implementer. When refining a constructive specification we find it useful to define an invariant on the state. In terms of this we can define the notion of the adequacy of an implementation.

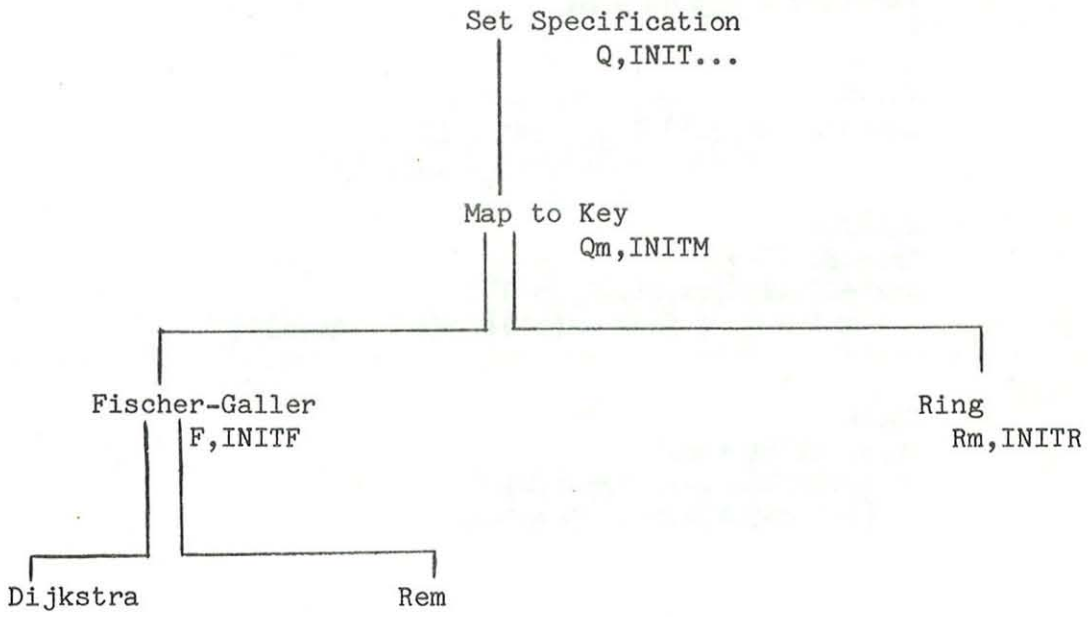


Figure 15 Recording Equivalence relations

I would like to turn now to the problem of avoiding the repetition of proofs required when small changes are made to a specification. Let us take a specific example. Figure 15 shows the relationship between various approaches to the problem of recording equivalence relations. Can we build up a body of knowledge about programming this problem which will be useful for others using the same data structure? In fact we can give constructive specifications for each of the approaches in Figure 15 and relate these to each other using the concepts of refinement.

I will begin at the map-to-key level of specification, because that will allow me to make the necessary points. Figure 16 defines the domains and operations at this level. This specification relates each element to a key. The idea is that equivalent elements map to the same key. The update operation EQUATEM records a new equivalence and the equivalences recorded in the state are examined using the operation TESTM. Figure 17 shows a worked example of adding five relationships and then testing them. You see that the specification properly implements symmetry, reflexivity and transitivity. The specification of Figure 16 would be an adequate implementation if we had a sufficiently high-level language, but it is not by any standard an efficient algorithm. The Fischer-Galler algorithm gives us a very cunning way of recording the key mappings which leads to a more efficient implementation.

$$
\begin{array}{l}
\mathrm{Qm} = \mathrm{El} \to \mathrm{Key} \\
\mathrm{invm}(qm) \triangleq \underline{\mathrm{dom}}\ qm = \mathrm{El} \\
\qquad\quad \sim \\
\\
\mathrm{INITM} \\
\text{post-INITM}(,qm') \triangleq \underline{\mathrm{dom}}\ qm' = \mathrm{El} \wedge \\
\qquad\qquad e1 \neq e2 \Rightarrow qm'(e1) \neq qm'(e2) \\
\\
\mathrm{EQUATEM} \\
\mathrm{type}{:}\mathrm{El}\ \mathrm{El} \to \\
\text{post-EQUATEM}(qm,e1,e2,qm') \triangleq \\
\qquad qm' = qm \dagger [e \mapsto qm(e1) \mid qm(e) = qm(e2)] \\
\\
\mathrm{TESTM} \\
\mathrm{type}{:}\ \mathrm{El}\ \mathrm{El} \to \mathrm{Bool} \\
\text{post-TESTM}(qm,e1,e2,qm',b) \triangleq \\
\qquad (b \Leftrightarrow qm(e1){=}qm(e2)) \wedge qm'{=}qm
\end{array}
$$

Figure 16 Map-to-key specification

```
      El = {a,b,c,d,e,f}              Key = {ka,...}
INITM() = [a→ka,b→kb,....,f→kf]                             =m1
EQUATEM(a,b)(m1) = [a→ka,b→ka,c→kc,...,f→kf]                =m2
EQUATEM(b,a)(m2) = m2
EQUATEM(b,c)(m2) = [a→ka,b→ka,c→ka,d→kd,...,f→kf]           =m3
EQUATEM(e,f)(m3) = [a→ka,b→ka,c→ka,d→kd,e→ke,f→ke]          =m4
EQUATEM(a,f)(m4) = [a→ka,b→ka,c→ka,d→kd,e→ka,f→ka]          =m5

TEST(b,a) = TRUE
TEST(d,d) = TRUE
TEST(f,d) = FALSE
```

Figure 17   Map-to-key example

The idea behind this algorithm is shown in Figure 18. The elements are organised into a tree in such a way that we can regard the root of the element as that element's key. Both updating and retrieving of this representation are very efficient.

m4 represented by:

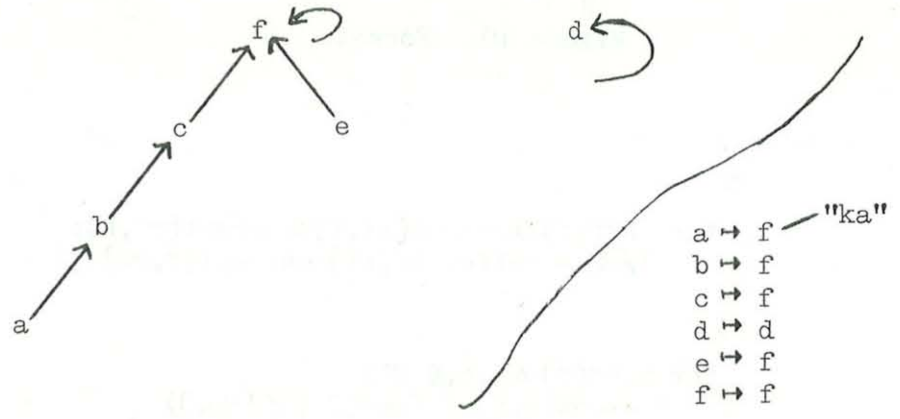


Figure 18   Fischer-Galler algorithm

Before defining the Fisher-Galler algorithm I develop the notion of a forest, Figure 19. A forest is a mapping from elements to elements which satisfies the invariant invf. Four useful operations are defined. Some properties of a forest are shown in Figure 20. The first says that a forest is a disjoint partition of the contained elements. The second concerns grafting two trees in a forest. When a tree is grafted at a root, the property of being a forest is retained. With these properties the proof of the Fischer-Galler algorithm becomes not only simpler but much more intuitive. Furthermore, the collection of properties can be built into an "Engineering Handbook" for use with other problems.

```
F = El → El

invf(f) ≜ dom f = El ∧
            (∀e)(is-root(e,f)∨
                       e ∉ reach(f(e),f))
is-root(e,f) ≜ f(e) = e
reach: El F → El-set

root(e,f) ≜ if is-root(e,f) then e
            else root(f(e),f)

collect(f,r) ≜
   {e|e ∈ dom f ∧ root(e,f)=r}
```

Figure 19   Forests

```
F1

  for invf(f)∧is-root(r1,f)∧is-root(r2,f):
     r1≠r2 ⇒ collect(f,r1)∩collect(f,r2)={}

F3

  for invf(f) ∧d,e,∈ El:
     is-root(e,f) ⇒ invf(f † [d↦e])
```

Figure 20   Forest properties

Mr. Jones did not have time to complete his development of the Fischer-Galler algorithm in the lecture. The definitions and an example are, however, shown in Figures 21 and 22.

```
INITF
states: F
post-INITF(,f')≜f'=[e↦e|e∈El]

EQUATEF
states: F
type: El El →
post-EQUATEF(f,e1,e2,f')≜
     f'=f † [root(e1,f)→root(e2,f)]

TESTF
states: F
type: El El → Bool
post-TESTF(f,e1,e2,f',b) ≜
   (b⇔root(e1,f)=root(e2,f))∧f'=f
```

Figure 21 Definitions for the Fisher-Galler algorithm

```
INITF() =[a→a,b→b,...,f→f]                                  =f1
EQUATEF(a,b)(f1)=[a→b,b→b,c→c,...,f→f]                      =f2
EQUATEF(b,a)(f2)= f2
EQUATEF(b,c)(f2)=[a→b,b→c,c→c,...,f→f]                      =f3
EQUATEF(e,f)(f3)=[a→b,b→c,c→c,...,e→f,f→f]                  =f4

TESTF(b,a)(f4) = TRUE
TESTF(d,d)(f4) = TRUE
TESTF(f,d)(f4) = FALSE
```

Figure 22 Example of the Fischer-Galler algorithm

The concepts of the constructive theories of data types are beginning to give us tools with which we can construct programs and proofs where the proofs can be modified along with the program. They do not have to be thrown away whenever the program is changed.

A collection of the properties of data stuctures enables a higher level of discussion during proofs of algorithms, and from an educational viewpoint it would be beneficial to encourage students to explore and collect such properties.

Discussion

Dr. Scoins pointed out that Mr. Jones appeared to have 'specified ordered trees and not distinguished these from unordered trees', and that this might make certain operations unnecessarily difficult. Mr. Jones agreed that he had probably tackled the definitions wrongly. 'The characterisation of trees I have given in the paper is of ordered trees. I should rather have developed a specification of unordered trees and then specialised them, thus bringing in more useful properties!'

References

Bjorner, D. (1979)	These proceedings.
Jones, C.B. (1980)	Software Development: A rigorous approach. Prentice-Hall International.

Additional material

Bjorner, D. (1978)	The Vienna Development Method: The Meta-Language, Springer-Verlag 'Lecture notes in Computer Science', Vol. 61, 1978.
Jones, C.B. (1979)	Constructing a Theory of a Data Structure as an Aid to Program Development, Acta Informatica, Vol. 11, pp. 119-137, 1979.