Introducing Reference Semantics via Refinement

Graeme Smith

Software Verification Research Centre, University of Queensland, Australia
smith@svrc.uq.edu.au

Abstract. Two types of semantics have been given to object-oriented formal specification languages. *Value semantics* denote a class by a set of values representing its objects. *Reference semantics* denote a class by a set of references, or pointers, to values representing its objects. While adopting the former facilitates formal reasoning, adopting the latter facilitates transformation to object-oriented code. In this paper, we propose a combined approach using value semantics for abstract specification and reasoning, and then refining to a reference semantics before transforming specification to code.

1 Introduction

Research on object-oriented formal specification languages has gone through two main phases of development.

The first phase focussed on extending existing formal specification languages with object-oriented constructs in order to enhance modularity and reusability. These constructs included classes, objects, inheritance and polymorphism. A number of new formal languages were developed, notable among which are those which extend VDM or Z [21, 14]. The goal was to make formal methods more applicable to larger-scale systems and industrial problems [6, 15].

The languages developed in this phase of research, including MooZ [20], ZEST [23] and early versions of VDM++ [8] and Object-Z [2, 4], have a *value semantics*, i.e., a semantics in which a class is denoted by a set of values. Each value in such a set corresponds to an object of the class at some stage of its evolution. Such a semantics is conservatively based on that of the language being extended and hence introduces no additional complexity to the semantic basis.

The second phase of research saw the inclusion of object references in object-oriented formal specification languages. Object references work in the same manner as pointers in programming languages. They introduce the possibility of object sharing (through aliasing) and non-trivial, recursively defined structures (since objects may reference objects which also reference them). Existing object-oriented formal specification languages such as VDM++ and Object-Z were extended with object references [13, 7, 5, 17]. The goal was to make the transition from formal specification to code easier [9].

To incorporate object references, the new versions of the languages developed in this phase of research have a *reference semantics*, i.e., a semantics in which a class is denoted by a set of references to values (denoting objects). Such a

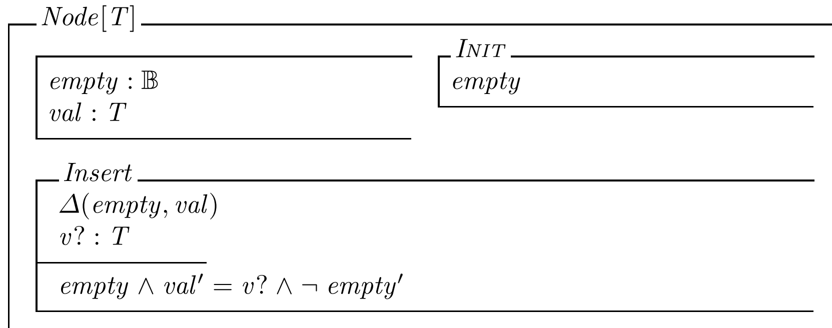
semantics is a major departure from that of the language being extended and hence much effort has subsequently gone into developing suitable semantics [10].

The additional complexity of reference semantics has had a large impact on developing methods for reasoning about specifications [11], refining specifications [3] and encoding languages in tools such as theorem provers [19]. Although object references enable an easy transition from specification to object-oriented code, they hinder the abstract representation of systems and hence unnecessarily complicate refinement and reasoning.

In this paper, we propose a step *backward* to the languages with value semantics for abstractly specifying systems as part of a step *forward* to a new approach to formal object-oriented development. This approach involves refining value-semantics specifications to specifications with reference semantics. This allows reasoning to be carried out in the absence of object references, but allows the addition of references through refinement in order to ease the transition to code. To illustrate our approach, we use the Object-Z specification language for which both value and reference semantics exist. This distinguishes our work from similar work in the refinement calculus where explicit stores are introduced during refinement to model mappings from references to values [22, 1]. In Section 2, we introduce the value-semantics version of Object-Z and in Section 3, the reference-semantics version. In Section 4, we illustrate through a simple case study how a value-semantics specification can be refined to a one with a reference semantics.

2 Value Semantics

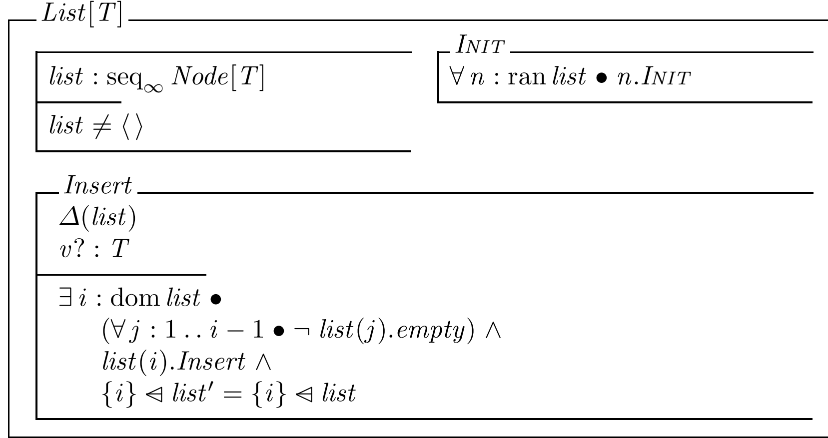
The early work on Object-Z [2, 4] adopts a value semantics [16]. The main extension to Z is syntactic: the introduction of a class schema. A class schema encapsulates a single state schema with its associated initial state schema and all the operations which can change its variables. For example, the following specifies a generic node which has two state variables, *empty* denoting whether or not a value has been inserted into the node and *val* denoting the value.



Initially, the node is empty and a value $v?$ can be inserted into it via the operation *Insert*. The Δ-list of this operation indicates that it is able to change

the variables *empty* and *val*. The operation can occur when *empty* is true, and results in *empty* being false and *val* taking the value $v?$.

Like schemas in Z, a class schema can be used as a type: its instances are values denoting possible objects of the class. For example, a generic list could be defined as a non-empty sequence of node objects as follows ($\mathrm{seq}_\infty X$ extends the Z definition $\mathrm{seq}\, X$ of finite sequences of type X, i.e., finite functions whose domain is a contiguous set of natural numbers with least element 1 and whose range is X, to possibly infinite sequences, i.e., $\mathrm{seq}_\infty X == \mathrm{seq}\, X \cup \mathbb{N} \rightarrow X$).



The class *List* denotes the functionality of a (possibly bounded) list abstractly by defining a possibly infinite sequence of nodes, the non-empty nodes of which denote the actual list. A finite sequence models a bounded list and an infinite sequence, an unbounded list.

Initially, each node n in the list satisfies the initial state of the class *Node*, i.e., it is empty. The operation *Insert* chooses a node such that all other nodes before it in the sequence are not empty, and inserts a value $v?$ into that node. The fact that a node must be empty for an insertion to take place (as defined in class *Node*) ensures that the selected node is the first empty one in the sequence. The final line of the operation ensures that all other nodes are unchanged ($\ntriangleleft$ is domain subtraction). It is needed since the inclusion of *list* in the Δ-list allows *list* to change arbitrarily unless otherwise constrained.

3 Reference Semantics

More recent work on Object-Z [7, 5, 17] adopts a reference semantics [10]. This enables a style of specification which more closely reflects implementation in an object-oriented programming language. In particular, it allows object sharing and non-trivial, recursive structures to be defined. For example, a list could be specified recursively by the variables associated with the node at the head of the list together with a pointer to the tail of the list (as in Fig. 1).

A reference-semantics specification in Object-Z is

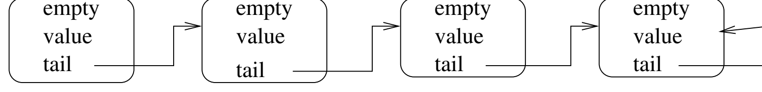
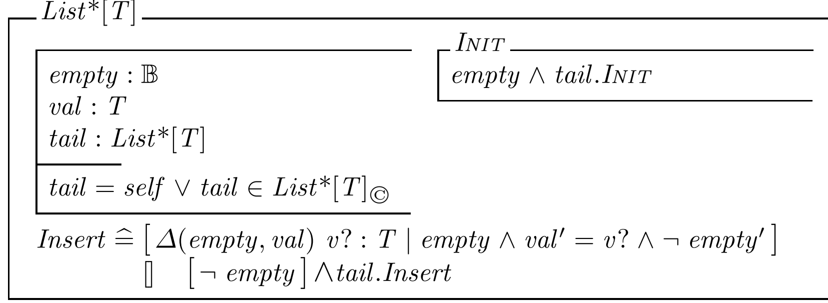


Fig. 1. Recursively defined list



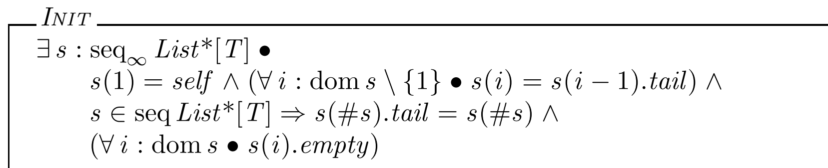
Classes in the reference-semantics version of Object-Z have an implicitly declared constant *self* denoting a reference to the current object. This is used in class *List** to specify that *tail* may point to the current object (when there is no tail). In all other cases, the tail list and all objects referenced either directly or indirectly from it must be "contained" by the current list (denoted by the © symbol decorating the type). Containment is a means of controlling aliasing among references. An object may be directly contained by only one other object and may not contain itself. Hence, circularities in the structure of list are precluded.

The initial state schema and operation *Insert* are defined recursively.

Initially, the list's head is empty and the list's tail is in an initial state. Since the list's tail is a list, this means it's head element is also empty and it's tail is also in an initial state, and so on.

The operation *Insert* is specified by an operation expression (rather than a schema) using the operators for disjoining ($[]$) and conjoining ($\wedge$) operations. It inserts a value $v?$ into the head element of the list (when it is empty), or performs an insert operation on the tail of the list when it's head element is not empty.

The meaning of INIT and *Insert* can be given using fixed point theory as shown by Smith [18]. INIT can be shown to be equivalent to



and *Insert* can be shown to be equivalent to

$$Insert \mathrel{\widehat=} [] \; s : \mathrm{seq}_\infty\, List^*[T];\; i : \mathrm{dom}\, s \mid p \;\bullet\; s(i).NodeInsert$$

where p is the predicate $s(1) = self \wedge (\forall j : \mathrm{dom}\, s \setminus \{1\} \bullet s(j) = s(j-1).tail) \wedge s \in \mathrm{seq}\, List^*[T] \Rightarrow s(\#s).tail = s(\#s) \wedge (\forall j : 1 \,..\, i-1 \bullet \neg\, s(j).empty)$ and $NodeInsert \mathrel{\widehat=} [\, \Delta(empty, val)\; v? : T \mid empty \wedge val' = v? \wedge \neg\, empty' \,]$.

4 Refinement

Refinement in reference-semantics Object-Z is defined in terms of simulation rules by Derrick and Boiten [3]. Downward simulation is defined as follows.

> *An Object-Z class C is a downward simulation of a class A if there is a retrieve relation R such that every abstract operation AOp of A is recast into a concrete operation COP of C and the following hold.*
>
> **DS.1** $\forall\, C.\textsc{Init} \bullet \exists\, A.\textsc{Init} \bullet R$
>
> **DS.2** $\forall\, A.\textsc{State};\; C.\textsc{State} \bullet R \Longrightarrow (\mathrm{pre}\, AOp \Longleftrightarrow \mathrm{pre}\, COp)$
>
> **DS.3** $\forall\, A.\textsc{State};\; C.\textsc{State};\; C.\textsc{State}' \bullet$
> $R \wedge COp \Longrightarrow (\exists\, A.\textsc{State}' \bullet R' \wedge AOp)$

That is, the initial state predicate can be stronger in the concrete class (**DS.1)**), as can operation postconditions (**DS.3**). Operation preconditions (i.e., the predicate pre *Op* for an operation *Op*) can neither be weaker nor stronger (**DS.2**).

This definition is simply that for a *blocking* model of operations, i.e., where an operation *Op* cannot occur unless pre *Op* is true, and is therefore independent of the semantics adopted (see Josephs [12], for example, for a similar definition). Hence, we would like to use it show that *List* of Section 2 is refined by *List** of Section 3. The refinement is done in three phases (see Fig. 2) each comprising one or more refinement steps consistent with downward simulation. These phases represent a general strategy for refining from value semantics to reference semantics.

Phase 1 In the first phase, all object values are changed to references. Hence, the sequence of nodes in *List* becomes a sequence of references to nodes.

Phase 2 In the second phase, references are added between objects where appropriate. Hence, references are added linking each node to the next in the list. A self reference is added to the final node in a finite list.

Phase 3 In the third phase, the class describing the system is replaced by the class of an object which is connected via references to all other objects in the system. In the list example, the class refined from *List* is replaced by a class describing the list from the head node, i.e., *List**.

4.1 Phase 1: Replacing object values with references

To accomplish the first phase of refinement in Object-Z, any operation schemas in which operations are applied to objects must be replaced by equivalent operation expressions. This is necessary since the reference-semantics version of Object-Z does not support operation application in schemas [17].

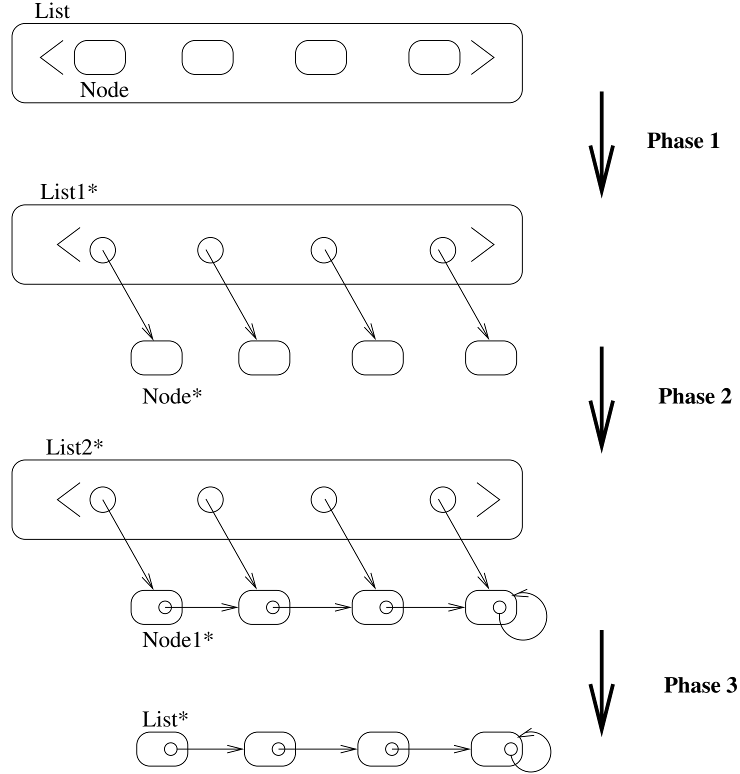


Fig. 2. Refinement of *List* to *List**

The operation *Insert* of *List*, can be replaced by

$$Insert \mathrel{\widehat=} [\,\Delta(list)\,] \wedge (\,[]\ i : \mathrm{dom}\, list \mid p \bullet list(i).Insert)$$

where p is $(\forall j : 1 \mathrel{..} i-1 \bullet \neg\, list(j).empty) \wedge \{i\} \mathbin{⩤} list' = \{i\} \mathbin{⩤} list$.

The main step in this phase of refinement is then to replace all classes C in the specification by a new class with reference semantics. In the case where C has no declared objects, the definition of the new class is syntactically identical to C. The new class is therefore trivially a refinement of C under the identity retrieve relation. In the case where C declares objects, these declarations are changed to refer to the new classes. Furthermore,

- since, in a value semantics, different declarations always refer to different objects, all declared references must be contained by C and an invariant must be added that all declared references are distinct, and
- since, in a value semantics, values are changed by operation application but in a reference semantics, references are not, a schema $[\,a' = a\,]$ must be conjoined with any operation application $a.Op$.

The resulting classes are again trivially a refinement under the retrieve relation which equates the state variables of the corresponding declared objects. Such a retrieve relation is necessary since the semantic representations of the objects in the different semantics are not directly comparable.

Applying these steps to our example specification, results in a class $Node*$ defined syntactically identically to $Node$, and a class $List1*$ defined as follows. (Note that the conjunction of the schema $[\, list'(i) = list(i) \,]$ with $list(i).Insert$ has allowed us to remove $Insert$'s Δ-list and simplify its definition.)

$$
\begin{array}{l}
\textbf{class } List1*[T] \\
\quad list : \mathrm{seq}_\infty\ Node*[T]_{©} \\
\quad \hline \\
\quad list \neq \langle\,\rangle \\
\quad \forall\, i, j : \mathrm{dom}\, list \bullet i \neq j \Rightarrow list(i) \neq list(j) \\[6pt]
\quad \textsc{Init} \\
\quad \forall\, n : \mathrm{ran}\, list \bullet n.\textsc{Init} \\[6pt]
\quad Insert \mathrel{\widehat=} [\!]\ i : \mathrm{dom}\, list \mid \forall\, j : 1 \,..\, i-1 \bullet \neg\, list(j).empty \bullet list(i).Insert
\end{array}
$$

The retrieve relations for these refinements are, respectively, $Node.empty = Node*.empty \wedge Node.val = Node*.val$ and $\mathrm{dom}\, List.list = \mathrm{dom}\, List1*.list \wedge (\forall i : \mathrm{dom}\, List.list \bullet List.list(i).empty = List1*.list(i).empty \wedge List.list(i).val = List1*.list(i).val)$. The latter relates the value-semantics nodes to the reference-semantics nodes by equating the values of their state variables ($empty$ and val).

4.2 Phase 2: Adding references between objects

For the second phase, we need to add references to classes and constraints on references reflecting the desired system structure. We begin by refining the class $Node*$ to a class $Node1*$ which has an added reference $next : Node1*[T]$ denoting the next node in the list, and strengthens the state invariant as follows.

$$
\begin{array}{l}
\textbf{class } Node1*[T] \\
\quad empty : \mathbb{B} \\
\quad val : T \\
\quad next : Node1*[T] \\
\quad \hline \\
\quad next = self \vee next \in Node1*[T]_{©} \\[6pt]
\quad \textsc{Init} \\
\quad empty \\[6pt]
\quad Insert \\
\quad \Delta(empty, val) \\
\quad v? : T \\
\quad \hline \\
\quad empty \wedge \neg\, empty' \wedge val' = v?
\end{array}
$$

$Node1^*$ is a downward simulation of $Node^*$ under the retrieve relation which identifies the variables *empty* and *val*, i.e., $Node^*.empty = Node1^*.empty \wedge Node^*.val = Node1^*.val$.

The state invariant of a class is implicitly conjoined to the initial state predicate and the precondition and postcondition of each operation. To show that the precondition of *Insert* is not strengthened under the retrieve relation, we need to show **DS.2** holds. That is,

$$\begin{array}{l} \forall\, Node^*.\mathrm{S{\small TATE}};\ Node1^*.\mathrm{S{\small TATE}} \bullet \\ \quad Node^*.empty = Node1^*.empty \wedge Node^*.val = Node1^*.val \Rightarrow \\ \qquad (Node^*.empty \Leftrightarrow Node1^*.empty \wedge Node1^*.Inv) \end{array}$$

where $Inv \mathrel{\widehat{=}} [next = self \vee next \in Node1^*[T]_{©}]$. This trivially holds since the declaration of $Node1^*.\mathrm{S{\small TATE}}$ introduces the invariant *Inv* on the variables of $Node1^*$. In general, invariants can be added during refinement provided they do not constrain variables related (by the retrieve relation) to variables that are changed by the abstract operations [3].

To complete this phase, we refine the class $List1^*$ by adding an invariant linking the nodes in the list in the appropriate way. That is, the next node of each node in the list, except the last in the case of a finite list, is that which occurs after it in the list. The next node of the last node of a finite list is itself. Once again, the invariant does not strengthen the precondition under the retrieve relation which in this case is the identity relation, i.e, $List1^*.list = List2^*.list$.

$List2^*[T]$

$$\begin{array}{l} list : \mathrm{seq}_\infty\, Node1^*[T]_{©} \\ \hline list \neq \langle\rangle \\ \forall\, i,j : \mathrm{dom}\, list \bullet i \neq j \Rightarrow list(i) \neq list(j) \\ \forall\, i : \mathrm{dom}\, list \bullet \\ \quad list(i).next = list(i+1) \vee \\ \qquad (list \in \mathrm{seq}\, Node1^*[T] \wedge i = \#list \wedge list(i).next = list(i)) \end{array}$$

$\mathrm{I{\small NIT}}$

$$\forall\, n : \mathrm{ran}\, list \bullet n.\mathrm{I{\small NIT}}$$

$$Insert \mathrel{\widehat{=}} [\!]\ i : \mathrm{dom}\, list \mid \forall\, j : 1 \mathinner{..} i-1 \bullet \neg\, list(j).empty \bullet list(i).Insert$$

4.3 Phase 3: Replacing the system class with an object class

The final phase involves removing the system class needed in a value semantics specification to relate the objects in the specified system. It is replaced by the class of one of the objects from which all others can be referenced. In cases where the value-semantics system class specifies an actual object of the specified system, this final phase may be unnecessary.

In our example, we want to replace the class $List2^*$ with the class $List^*$ of Section 3 (see Fig. 2). We begin by introducing a new variable *head* as an alias to the head node of the list and redefine INIT and *Insert* in terms of *head*. This is done by adding an existentially quantified variable s to both INIT and *Insert* which denotes a sequence of nodes starting with *head* and such that each other node in the sequence is the *next* node of it predecessor in the sequence.

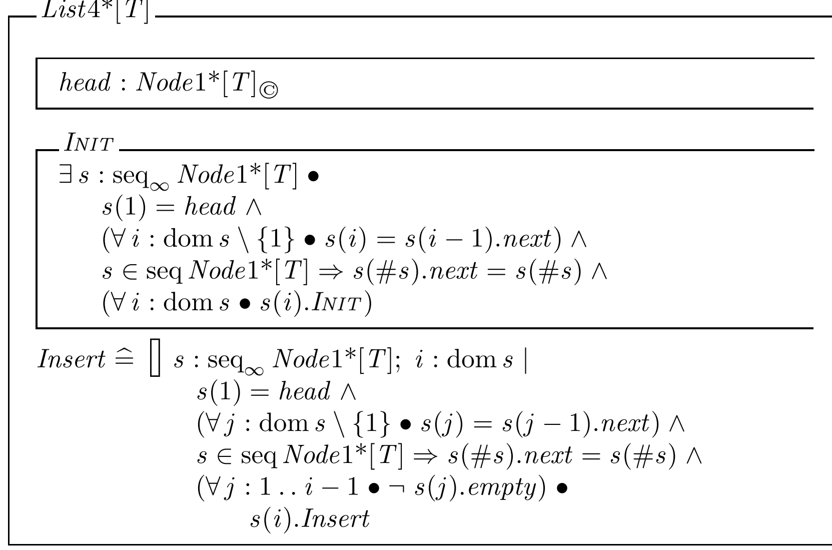
$$
\begin{array}{l}
\textbf{List3*[T]} \\
\quad list : \mathrm{seq}_\infty\, Node1^*[T]_{©} \\
\quad head : Node1^*[T]_{©} \\
\hline
\quad list \neq \langle\,\rangle \\
\quad \forall\, i, j : \mathrm{dom}\, list \bullet i \neq j \Rightarrow list(i) \neq list(j) \\
\quad \forall\, i : \mathrm{dom}\, list \bullet \\
\qquad list(i).next = list(i+1) \vee \\
\qquad\quad (list \in \mathrm{seq}\, Node1^*[T] \wedge i = \#list \wedge list(i).next = list(i)) \\
\quad head = list(1) \\
\\
\quad \textit{INIT} \\
\quad \exists\, s : \mathrm{seq}_\infty\, Node1^*[T] \bullet \\
\qquad s(1) = head \wedge \\
\qquad (\forall\, i : \mathrm{dom}\, s \setminus \{1\} \bullet s(i) = s(i-1).next) \wedge \\
\qquad s \in \mathrm{seq}\, Node1^*[T] \Rightarrow s(\#s).next = s(\#s) \wedge \\
\qquad (\forall\, i : \mathrm{dom}\, s \bullet s(i).\textit{INIT}) \\
\\
\quad Insert \mathrel{\widehat=} [\!]\ s : \mathrm{seq}_\infty\, Node1^*[T];\ i : \mathrm{dom}\, s \mid \\
\qquad\qquad s(1) = head \wedge \\
\qquad\qquad (\forall\, j : \mathrm{dom}\, s \setminus \{1\} \bullet s(j) = s(j-1).next) \wedge \\
\qquad\qquad s \in \mathrm{seq}\, Node1^*[T] \Rightarrow s(\#s).next = s(\#s) \wedge \\
\qquad\qquad (\forall\, j : 1 \mathinner{..} i-1 \bullet \neg\, s(j).empty) \bullet \\
\qquad\qquad\quad s(i).Insert
\end{array}
$$

$List3^*$ is a downward simulation of $List2^*$ under the retrieve relation $List2^*.list = List3^*.list$.

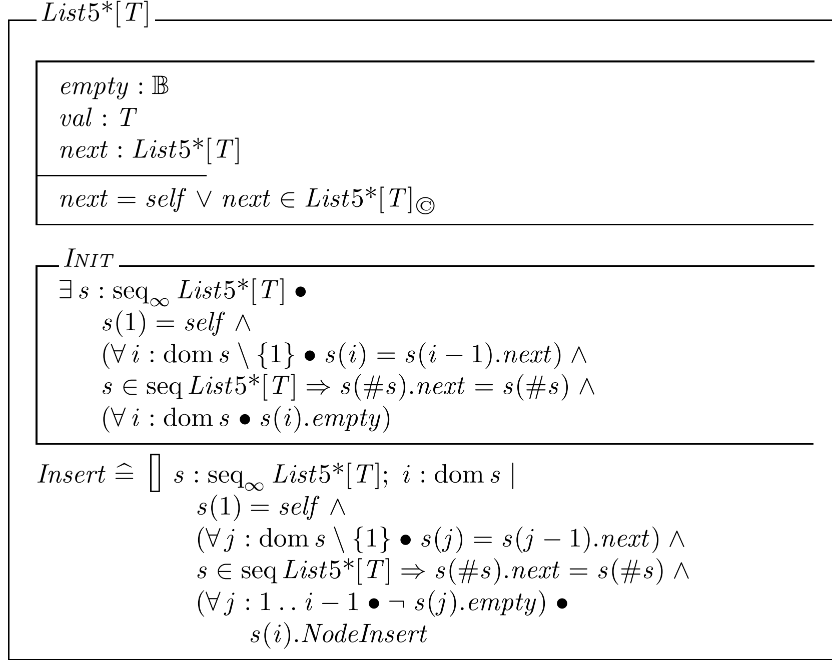
Since *list* is no longer used in the initial state schema or operation *Insert*, the class can be further refined by removing this variable. The invariants in terms of *list* do not constrain *head* and can be removed as well. In particular, the invariant that nodes in the list are distinct is captured by the invariant of $Node1^*$ which precludes circular list structures.

In the case where one or more of the invariants did constrain *head*, they would need to be redefined in terms of *head*.

The following class, $List4^*$, is a downward simulation of $List3^*$ under the retrieve relation $List3^*.head = List4^*.head$.



Our system class $List4^*$ now comprises a single contained object. Such a class can be refined to a class C with the single object declaration $a : A_{©}$ replaced by the state declarations of A, and with all occurrences of a, A, $a.\mathrm{INIT}$ and $a.Op$ replaced by $self$, C, and the definitions of INIT and Op respectively. Hence, class $List4^*$ is refined by



where $NodeInsert \mathrel{\widehat{=}} [\, \Delta(empty, val)\ v? : T \mid empty \wedge val' = v? \wedge \neg\, empty' \,]$.

Given that $self.x = x$ for each state variable x, $List5^*$ is a downward simulation of $List4^*$ under the retrieve relation $List4^*.head = List5^*.self$.

With *next* renamed to *tail*, the initial state schema and operation *Insert* of $List5^*$ are the equivalent schemas of those of $List^*$ derived using fixed point theory in Section 3. Hence, under the retrieve relation $List5^*.empty = List^*.empty \land List5^*.val = List^*.val \land List5^*.next = List^*.tail$, we can refine $List5^*$ to $List^*$.

5 Conclusion

In this paper, we have shown how to refine an object-oriented formal specification with a value semantics to one with a reference semantics. This process allows an abstract specification to be written in a value semantics in order to facilitate reasoning, and then be refined to a concrete specification with a reference semantics in order to facilitate transformation to code.

The general process was illustrated using a simple case study. This case study involved the introduction of recursion. Other refinements to reference semantics could involve the introduction of object sharing. The refinement steps were justified with respect to the definition of downward simulation in Object-Z. A set of rules proved sound with respect to this definition, or that of upward simulation, could be developed to aid the specifier by removing much of the proof burden.

Acknowledgement

Thanks to John Derrick for discussions which led to this work and Ian Hayes for his constructive comments on an earlier draft of this paper. This work was funded by a University of Queensland External Support Enabling Grant.

References

1. P. Bancroft and I.J.Hayes. Type extension and refinement. In L. Groves and S. Reeves, editors, *Formal Methods Pacific (FMP'97)*, pages 23–39. Springer-Verlag, 1997.
2. D. Carrington, D. Duke, R. Duke, P. King, G. Rose, and G. Smith. Object-Z: An object-oriented extension to Z. In S. Voung, editor, *Formal Description Techniques (FORTE'89)*, pages 281–296. North-Holland, 1989.
3. J. Derrick and E. Boiten. *Refinement in Z and Object-Z, Foundations and Advanced Applications*. Springer-Verlag, 2001.
4. R. Duke, P. King, G. Rose, and G. Smith. The Object-Z specification language. In T. Korson, V. Vaishnavi, and B. Meyer, editors, *Technology of Object-0riented Languages and Systems (TOOLS 5)*, pages 465–483. Prentice Hall, 1991.
5. R. Duke and G. Rose. *Formal Object-Oriented Specification using Object-Z*. MacMillan, 2000.
6. R. Duke, G. Rose, and G. Smith. Transferring formal techniques to industry: A case study. In J. Quemada, J. Mañas, and E. Vazquez, editors, *Formal Description Techniques (FORTE'90)*, pages 279–286. North-Holland, 1990.

7. R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, 1995.
8. E.H. Dürr and J. van Katwijk. VDM++ – A formal specification language for object-oriented designs. In B. Meyer, G. Heeg, and B. Magnusson, editors, *Technology of Object-oriented Languages and Systems (TOOLS Europe 92)*, pages 63–78. Prentice-Hall, 1992.
9. A. Griffiths. From Object-Z to Eiffel: a rigorous development method. In C. Mingins, R. Duke, and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems (TOOLS 18)*, pages 293–308. Prentice Hall, 1995.
10. A. Griffiths. An extended semantic foundation for Object-Z. In *1996 Asia-Pacific Software Engineering Conference (APSEC'96)*, pages 194–207. IEEE Computer Society Press, 1996.
11. A. Griffiths. Modular reasoning in Object-Z. In Wai Wong and K. Leung, editors, *Asia-Pacific Software Engineering Conference and International Computer Science Conference (APSEC '97/ICSC '97)*, pages 140–149. IEEE Computer Society Press, 1997.
12. M.B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3:9–18, 1988.
13. K. Lano. *Formal Object-Oriented Development*. Springer-Verlag, 1995.
14. K. Lano and H. Haughton, editors. *Object-Oriented Specification Case Studies*. Object-Oriented Series. Prentice Hall, 1993.
15. K. Rosenberg. The adoption of formal methods within OTC. In K. Parker and G. Rose, editors, *Formal Description Techniques (FORTE'91)*, pages 85–92, 1991.
16. G. Smith. A fully abstract semantics of classes for Object-Z. *Formal Aspects of Computing*, 7(3):289–313, 1995.
17. G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.
18. G. Smith. Recursive schema definitions in Object-Z. In A. Galloway J. Bowen, S. Dunne and S. King, editors, *International Conference of B and Z Users (ZB 2000)*, volume 1878 of *Lecture Notes in Computer Science*, pages 42–58. Springer-Verlag, 2000.
19. G. Smith, F. Kammüller, and T. Santen. Encoding Object-Z in Isabelle/HOL. In D. Bert, J.P. Bowen, M.C. Henson, and K. Robinson, editors, *International Conference of Z and B Users (ZB 2002)*, volume 2272 of *Lecture Notes in Computer Science*, pages 82–99. Springer-Verlag, 2002.
20. S.R.L.Meira and A.L.C. Cavalcanti. Modular object-oriented Z specifications. In *Z User Meeting 1990*, Workshops in Computing, pages 173–192. Springer-Verlag, 1990.
21. S. Stepney, R. Barden, and D. Cooper, editors. *Object-Orientation in Z*. Workshops in Computing. Springer-Verlag, 1992.
22. M. Utting. Reasoning about aliasing. In *Australian Refinement Workshop (ARW 95)*, pages 195–211, School of Computer Science and Engineering, The Univeristy of New South Wales, 1995.
23. H.B. Zadeh and S. Stepney. *ZEST – Z Extended with Structuring: A User's Guide, PROST-Objects, BT.7004.0.20.13, Issue 2*, 1996.