Refinement in a Type Extension Context

Peter Bancroft
Queensland University of Technology
Brisbane 4001, Australia
e-mail: bancroft@fit.qut.edu.au

Ian Hayes
The University of Queensland
Brisbane 4072, Australia
e-mail: ianh@cs.uq.oz.au

Abstract

This paper extends the methods of the refinement calculi to allow the derivation of an Oberon implementation. The use of records, pointers, opaque types and type extension distinguishes this work from previous examples. A case study for a stack abstract data type illustrates the method, which involves three data refinement steps. Firstly, a sequence of integers is refined to a sequence of integer records with pointers. Secondly, the sequence of records is refined to a linked list accessed through a start pointer. Finally, this is refined to a generic implementation that is close to Oberon code. Pointers are facilitated by declaring an explicit local data store for each stack variable. The advantage of Oberon over Modula-2 is the ability to separate the final code into two modules — a generic stack that maintains the linked list property and an integer instantiation of the stack, using type extension. This separation of concerns (isolation of the linked data structure properties) is the main benefit of the approach.

1 Introduction

The refinement calculi [Bac88, Mor89, Mor94] provide formal methods for the derivation of code from abstract specifications written in the style of the Z notation [Hay93, Spi92]. Fundamental to the approach is the inclusion of a wide spectrum language which allows non-executable statements and code to be intermixed. In the process of refinement, all non-executable statements are removed. The target language is usually an extended version of the guarded command language [Dij76] which may be easily transliterated into Modula-2. Recent work [Ban93] has demonstrated the use of data refinement techniques to produce pointer implementations which are also close to Modula-2.

This paper combines the earlier work on pointers with a recently rediscovered ([BDMN73]) language feature, type extension [Wir88a]. A "generic" implementation for an abstract data type module is calculated. The genericity is in the sense of [RW92], where an imported module and type extension are used to instantiate a specific application. The advantage of the resulting code is that the complexity of maintaining the linked data structure is separated from any concern over the actual contents in its nodes. In the derived program, the target language uses the refinement calculus notation of [Mor94] with type declarations, records and pointers to produce code close to Oberon (Section 2).

The inclusion of pointer variables raises the usual problem of aliasing. This is dealt with by

- the use of "local stores" [Utt95], where each pointer variable of an abstract data type has its own separate data store. It is this feature that makes it easier to reason about pointers.
- non-overlapping procedure parameters as in the languages Euclid [Hor79] and Turing [HH84].
- strict use of opaque types, avoiding direct pointer manipulation in client programs, as in Ada 95 [Bar95] and Java [Rit95].

Calculation of an implementation module from an abstract specification for an opaque type is facilitated using the technique described in [BH93a, BH93b]. This is a modification of that in [MG90] to deal with modules in which there is no local state and the coupling invariant is parameterized for each occurrence of the opaque type in the formal parameters of a procedure. The method also allows for the data refinement of a module with several different opaque types as demonstrated in the case study below.

Modelling the explicit local data store is critical for obtaining an Oberon-like implementation. It is necessary to be able to dereference pointers to obtain the dynamic value of objects; but more importantly the model must provide access to the dynamic type of objects. Oberon does not rely on type inferencing systems such as in [Wan87, Wan93]. Type extension is explicit and *name* based so the type is recorded with the value — the dynamic store is represented as a partial function from locations to type-labelled values (as in [BH95]).

Section 2 contains a brief description of type extension. For a detailed formal semantics refer to [BH95]. A small case study is introduced in Section 3. In Section 4, a context is defined for subsequent refinement — types and their values, assignment compatibility and a model for the explicit data store are introduced. This context allows the use of type extension to derive a generic implementation for the given example, in Section 5.

2 Type Extension

Type extension [Wir88b] is the creation of new record types from existing record types by the addition of extra fields. In the object-oriented languages

Oberon [RW92, Wir88a] and Oberon-2 [MW91], type extension and a relaxation of the usual assignment compatibility rules provide the mechanism by which single inheritance is facilitated. This approach as suggested in [Sei91], has also been adopted for Ada 95 [Bar95].

A record extension is considered to be compatible with the original record type in that the values of the extension can be assigned to variables of that type. The extra fields cannot be stored in such a variable and are removed. This is known as the *projection* of the value into the original type. Other than the creation of a hierarchy, record extension is of little interest on its own, given a projection semantics for assignment.

More importantly, pointer types may also be extended. A pointer variable is declared with reference to a particular base record type (giving its *static* type) but can be assigned a value that refers to extensions of that type. The difference between this pointer assignment and the corresponding record assignment is that the extra fields of the extended base type remain accessible through the pointer (providing a possibly different *dynamic* type).

The following is a typical declaration block in an Oberon-like language:

$$
\begin{array}{l}
\textbf{type}\ rec1 = \textbf{record}\ f1 : \textbf{integer};\ f2 : \mathit{Boolean}\ \textbf{end};\\
\textbf{var}\ r1 : rec1;\\
\textbf{type}\ ptr1 = \textbf{pointer to}\ rec1;\\
\textbf{var}\ p1 : ptr1;\\
\textbf{type}\ rec2 = \textbf{record}(rec1)\ f3 : \textbf{integer end};\\
\textbf{var}\ r2 : rec2;\\
\textbf{type}\ ptr2 = \textbf{pointer to}\ rec2;\\
\textbf{var}\ p2 : ptr2;
\end{array}
$$

Variable *r2* has type *rec2* which is an extension of type *rec1*. Variable *r1* can be assigned *r2* but the field *f3* will be dropped off. Pointer variable *p1* is of type *ptr1* but can be dynamically assigned a value which refers to *r2*. The extra field *f3* remains accessible through *p1* which now has static type *ptr1* but dynamic type *ptr2*.

Access to the extra fields of an extended pointer variable is subject to a *type guard*, which is an assertion that the dynamic type of the pointer is an extension of the type named in the guard. In Oberon, the expression,

```
p1(ptr2)^.f3
```

means that variable *p1* has been extended so that it has dynamic type at least *ptr2*, a pointer to a record with a field name *f3*.

In Oberon, the use of guarded expressions gives rise to two new kinds of program statements involving pointer variables. Firstly, there is *guarded assignment* (or reverse assignment) such as,

```
p2 := p1(ptr2)
```

module $spec\_Stacks \triangleq$
opaque type $Stack \triangleq seq_0$**integer**
  initially$(s : Stack) s = \langle\,\rangle$;
procedure $Push$(**value result** $s : Stack$; **value** n : **integer**) $\triangleq$
  $s : [\,\text{true}, s = \langle n \rangle \frown s_0\,]$;
procedure Pop(**value result** $s : Stack$; **result** m : **integer**) $\triangleq$
  $s, m : [\,s \neq \langle\,\rangle, s_0 = \langle m \rangle \frown s\,]$;
procedure $MTStack$(**value** $s : Stack$; **result** $mt : Boolean$) $\triangleq$
  $mt : [\,\text{true}, mt \Leftrightarrow (s = \langle\,\rangle)\,]$
end

Figure 1: Specification Module

Here, *p1* is asserted to have dynamic type at least *ptr2* which must be the same type or an extension of the static type of *p2*. Its value may therefore be assigned to *p2*. The guard is necessary only when the static type of *p1* is not an extension of the static type of *p2*.

Secondly, Boolean expressions, such as

```
p1 IS ptr2
```

may be used, to mean that the dynamic type of pointer variable *p1* is at least *ptr2*. It is usual to make such an assertion in the guards of an *if* or *while* statement whose body relies on the dynamic type of extended variable *p1*. The ensuing dynamic check may avoid a runtime error in the evaluation of the guarded expression.

3 A Small Case Study

The case study in Figure 1 is for a stack type with an abstract representation which is a sequence of integers. An opaque type [BH93b, BH93a] is exported from a module to allow the declaration of any number of variables of that type in a client program. The client program can only operate on such variables by making calls to procedures in the ADT module — no knowledge of the implementation of the opaque type is available. Each instance of the *Stack* opaque type declared in a client program is initialised to empty on entry to the the enclosing scope, according to the **initially** clause. This abstract specification corresponds to a definition module in Modula-2 or a package specification in Ada.

The *spec_Stacks* module is to be refined to a pointer implementation. The remainder of this section details the calculation of two data refinement steps.

Firstly, the abstract sequence is replaced by a sequence of records, consisting of an integer and a pointer field, with the pointers appropriately linked. The second refinement removes the sequence altogether, leaving a linked list of nodes with a start pointer, as might be expected.

3.1 First Data Refinement

For the initial data refinement, several local types are introduced . These are available only within the implementation module. Most important is the definition of a local store, as a partial function from a suitable pointer type to a corresponding record type.

$$\begin{array}{l}
\textbf{local type } \mathit{StackPtr} \triangleq \textbf{pointer to } \mathit{StackRec};\\
\textbf{local type } \mathit{StackRec} \triangleq \textbf{record}\\
\qquad\qquad\qquad\qquad \mathit{next} : \mathit{StackPtr};\\
\qquad\qquad\qquad\qquad \mathit{item} : \textbf{integer}\\
\qquad\qquad\qquad \textbf{end};\\
\textbf{local type } \mathit{Store} \triangleq \mathit{StackPtr} \pfun \mathit{StackRec};
\end{array}$$

Type extension in Oberon is based on named record and named pointer types so the more usual approach of defining a store as a partial function from location to value is avoided, at this stage. However the values of all pointers are locations in the store (see [BH95]) and this will become important in Section 5.

With the appropriate local type definitions, a more concrete stack representation is declared. The inclusion of a *Store* field, makes it clear that each instance of this opaque type is allocated a separate section of the dynamic memory. The explicit data type invariant imposes the usual linked list constraints on the data structure.

$$\begin{array}{l}
\textbf{opaque type } \mathit{Stack} \triangleq \textbf{record}\\
\qquad\qquad\qquad\qquad \mathit{nodes} : \mathrm{seq}_0\, \mathit{StackRec};\\
\qquad\qquad\qquad\qquad \mathit{mem} : \mathit{Store}\\
\qquad\qquad\qquad \textbf{end}\\
\quad \textbf{invariant}(t : \mathit{Stack}) \triangleq (\forall\, i : 0..\#t.\mathit{nodes} - 2 \bullet\\
\qquad\qquad t.\mathit{mem}(t.\mathit{nodes}[i].\mathit{next}) = t.\mathit{nodes}[i+1]) \wedge\\
\qquad\qquad (\mathit{last}\ t.\mathit{nodes}).\mathit{next} = \textbf{nil}\\
\quad \textbf{initially}(t : \mathit{Stack}) \triangleq (t.\mathit{nodes} = \langle\,\rangle);
\end{array}$$

A first implementation module, *impl_Stacks1*, can now be calculated using the opaque type data refinement calculator of [BH93a]. The coupling invariant in this case is

$$\begin{array}{l}
\mathit{CI}(s : \mathit{spec_Stacks.Stack}, t : \mathit{impl_Stacks1.Stack}) \triangleq\\
\qquad\qquad s = \{i : 0..\#t.\mathit{nodes} - 1 \bullet i \mapsto t.\mathit{nodes}[i].\mathit{item}\}
\end{array}$$

$$
\begin{array}{l}
\textbf{module } \mathit{impl\_Stacks1} \triangleq \\
\textbf{local type } \mathit{StackPtr} \triangleq \textbf{pointer to } \mathit{StackRec}; \\
\textbf{local type } \mathit{StackRec} \triangleq \textbf{record} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{next} : \mathit{StackPtr}; \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{item} : \textbf{integer} \\
\qquad\qquad\qquad\qquad\qquad \textbf{end}; \\
\textbf{local type } \mathit{Store} \triangleq \mathit{StackPtr} \nrightarrow \mathit{StackRec}; \\
\textbf{opaque type } \mathit{Stack} \triangleq \textbf{record} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{nodes} : \mathrm{seq}_0\, \mathit{StackRec}; \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{mem} : \mathit{Store} \\
\qquad\qquad\qquad\qquad\qquad \textbf{end} \\
\quad \textbf{invariant}(t : \mathit{Stack}) \triangleq (\forall\, i : 0..\#t.\mathit{nodes} - 2 \mid \\
\qquad\qquad\qquad t.\mathit{mem}(t.\mathit{nodes}[i].\mathit{next}) = t.\mathit{nodes}[i+1]) \wedge \\
\qquad\qquad\qquad (\mathit{last}\; t.\mathit{nodes}).\mathit{next} = \textbf{nil} \\
\quad \textbf{initially}(t : \mathit{Stack}) \triangleq (t.\mathit{nodes} = \langle\,\rangle); \\
\textbf{procedure } \mathit{Push}(\textbf{value result } t : \mathit{Stack};\ \textbf{value } n : \textbf{integer}) \triangleq \\
\quad t : [\mathrm{true}, (\mathit{head}\; t.\mathit{nodes}).\mathit{item} = n \wedge \mathit{tail}\; t.\mathit{nodes} = t_0.\mathit{nodes}]; \\
\textbf{procedure } \mathit{Pop}(\textbf{value result } t : \mathit{Stack};\ \textbf{result } m : \textbf{integer}) \triangleq \\
\quad t, m : [t.\mathit{nodes} \neq \langle\,\rangle, t.\mathit{nodes} = \mathit{tail}\; t_0.\mathit{nodes}]; \\
\textbf{procedure } \mathit{MTStack}(\textbf{value } t : \mathit{Stack};\ \textbf{result } \mathit{mt} : \mathit{Boolean}) \triangleq \\
\quad \mathit{mt} : [\,\mathrm{true}, \mathit{mt} \Leftrightarrow (t.\mathit{nodes} = \langle\,\rangle)\,] \\
\textbf{end}
\end{array}
$$

Figure 2: First Data Refinement

The calculator for a downward simulation of a specification statement is given by

$$
\begin{array}{l}
s, x : [\mathit{preA}, \mathit{postA}] \sqsubseteq \\
\qquad t, x : [(\exists\, s \bullet \mathit{preA} \wedge \mathit{CI}(s,t)), (\exists\, s \bullet \mathit{postA} \wedge \mathit{CI}(s,t))]
\end{array}
$$

where s is an abstract state vector, x is an output vector, t is a concrete state vector and CI is the coupling invariant relating the abstract and concrete states. The implementation module in Figure 2 is the result of the calculation.

3.2 Second Data Refinement

For the second data refinement step, the sequence of nodes is replaced by a concrete data type consisting of a *start* pointer and the local store. The *start* pointer is always in the local store or is equal to the special value **nil**. The data invariant of the previous section is distributed through each procedure, thus

$$
\begin{array}{l}
\textbf{module } \mathit{impl\_Stacks2} \triangleq \\
\textbf{local type } \mathit{StackPtr} \triangleq \textbf{pointer to } \mathit{StackRec}; \\
\textbf{local type } \mathit{StackRec} \triangleq \textbf{record} \\
\qquad\qquad\qquad\qquad\qquad \mathit{next} : \mathit{StackPtr}; \\
\qquad\qquad\qquad\qquad\qquad \mathit{item} : \textbf{integer} \\
\qquad\qquad\qquad\qquad \textbf{end}; \\
\textbf{local type } \mathit{Store} \triangleq \mathit{StackPtr} \twoheadrightarrow \mathit{StackRec}; \\
\textbf{opaque type } \mathit{Stack} \triangleq \textbf{record} \\
\qquad\qquad\qquad\qquad\qquad \mathit{start} : \mathit{StackPtr}; \\
\qquad\qquad\qquad\qquad\qquad \mathit{mem} : \mathit{Store} \\
\qquad\qquad\qquad\qquad \textbf{end} \\
\quad \textbf{invariant}(v : \mathit{Stack}) \triangleq (v.\mathit{start} \in \mathrm{dom}\, v.\mathit{mem} \vee v.\mathit{start} = \textbf{nil}) \\
\quad \textbf{initially}(v : \mathit{Stack}) \triangleq (v.\mathit{start} = \textbf{nil} \wedge v.\mathit{mem} = \{\,\}); \\
\textbf{procedure } \mathit{Push}(\textbf{value result } v : \mathit{Stack};\ \textbf{value } n : \textbf{integer}) \triangleq \\
\quad v : [\mathrm{true}, \\
\qquad v.\mathit{start} \notin \mathrm{dom}\, v_0.\mathit{mem} \wedge \\
\qquad v.\mathit{mem} = v_0.\mathit{mem} \oplus \{v.\mathit{start} \mapsto \mathit{StackRec}(v_0.\mathit{start}, n)\}]; \\
\textbf{procedure } \mathit{Pop}(\textbf{value result } v : \mathit{Stack};\ \textbf{result } m : \textbf{integer}) \triangleq \\
\quad v, m : [v.\mathit{start} \neq \textbf{nil}, \\
\qquad m = (v_0.\mathit{mem}(v_0.\mathit{start})).\mathit{item} \wedge \\
\qquad v.\mathit{mem} = \{v_0.\mathit{start}\} \lhd v_0.\mathit{mem} \wedge \\
\qquad v.\mathit{start} = (v_0.\mathit{mem}(v_0.\mathit{start})).\mathit{next}]; \\
\textbf{procedure } \mathit{MTStack}(\textbf{value } v : \mathit{Stack};\ \textbf{result } \mathit{mt} : \mathit{Boolean}) \triangleq \\
\quad \mathit{mt} : [\,\mathrm{true}, \mathit{mt} \Leftrightarrow (v.\mathit{start} = \textbf{nil})\,] \\
\textbf{end}
\end{array}
$$

Figure 3: Second Data Refinement

preserving the linked list property implicitly. The calculation proceeds as in the previous section, giving Figure 3.

The resulting module is now a concrete specification for a pointer implementation. Application of the laws of algorithm refinement, as in [Mor94], takes the specification to code that may be transliterated into Modula-2 by replacing the local stores with a single global store [Ban93, Utt95]; however, the aim of this paper is to derive Oberon-like code. In the next section, a context for the subsequent development of this specification is described, taking into consideration the effects of type extension.

4 The Programming Context

This section contains a brief description of a type extension context — definitions that are needed to derive an Oberon-like implementation. [BH95] gives full details of an abstract syntax for a language with type extension, as well as a static and dynamic semantics in a denotational style. Only the sections relevant to this paper are discussed here.

4.1 Static Considerations

A basic type $[Id]$ provides a set of identifiers for the abstract language. The possible types of the language are given by a disjoint union.

$$\begin{aligned} Type ::= {} & \ldots \mid record\langle\langle Id \mathrel{+\!\!\!\!\!\rightarrow} Type\rangle\rangle \mid ptr\langle\langle Type\rangle\rangle \mid \\ & extend\langle\langle Id \times (Id \mathrel{+\!\!\!\!\!\rightarrow} Type)\rangle\rangle \mid \ldots \end{aligned}$$

Included in this free type definition are record, pointer and extension types. The function *record* takes a partial function from field names to their types and constructs a record type. The function *ptr* is used to construct a pointer type to a record or extend base type. An extend type consists of an *Id*, which is constrained to be the name of a record or another extend type, together with some extra fields.

Assuming a set of locations, $[Loc]$, in a store, the values that may be attributed to variables and expressions of the allowed types are given by

$$Value ::= \ldots \mid recval\langle\langle Id \mathrel{+\!\!\!\!\!\rightarrow} Value\rangle\rangle \mid refval\langle\langle Loc\rangle\rangle \mid \ldots$$

Record and extension values are recvals and all pointer values are refvals. For convenience, the set of all pointer values is given by

$$PtrVal == refval(\!|\ Loc\ |\!)$$

The semantic function for *Type* in a given static environment is

$$\mid\ Val : Type \rightarrow \mathbb{P}\ Value$$

Given any type t, $Val(t)$ returns the allowable values of the type.

In [BH95] a monolithic structure for the store is given. In this paper the dynamic store is treated as separate local stores (collections) for each instance of an opaque type. Nonetheless, the same model is applicable for each collection. A stored value consists of a type and a value.

$$\begin{array}{|l} \hline \textit{StoredVal} \\ dyntype : Type \\ dynval : Value \\ \hline \end{array}$$

A *store* is a partial function from locations to stored values. This means that each location in the store is "tagged" with the type that it holds.

$$store == Loc \rightarrow\!\!\!\!\!+\, StoredVal$$

There are subsequent constraints for a store to be well-formed. The *dynval* at any location must be a value of the *dyntype* at that location; and the dynamic type of any pointer variable can be any extension of the static type.

Contrary to the simple model for the store in [BH95], a more appropriate definition in this context is given by,

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Store}} \\
\hline
stat : store \\
dyn : Id \twoheadrightarrow\!\!\!\!\!+\, store \\
\hline
\forall\, s, t : \{stat\} \cup \operatorname{ran} dyn \bullet \operatorname{dom} s \cap \operatorname{dom} t = \{\,\} \\
\hline
\end{array}$$

The data store consists of a static store and dynamic store (collections), all of which are pairwise disjoint.

Variable references are elements in the following disjoint union.

$$\begin{aligned} Varref ::= {} & vref\langle\!\langle Id \rangle\!\rangle \mid fieldref\langle\!\langle Varref \times Id \rangle\!\rangle \mid deref\langle\!\langle Varref \times Id \rangle\!\rangle \mid \\ & typeref\langle\!\langle Varref \times Id \rangle\!\rangle \end{aligned}$$

The function *vref* takes a name $id \in Id$ and produces a variable $vref(id)$. The function *fieldref* is used to select a field from a record or extend variable, while *deref* is used to dereference a pointer variable with respect to a collection (Id). The function *typeref* is used to create type-guarded variable references (to deal with extended pointer variables).

Type extension allows the creation of a type hierarchy through single inheritance. In a given static environment, the relation, *ext* is specified as

$$ext : Type \leftrightarrow Type$$

so that $t1 \mapsto t2 \in ext$ is interpretted as "t1 is extended by t2".

An important feature of a language that allows type extension is the relaxation of assignment compatibility rules, as described in the introduction — more formally, in a given static environment,

$$\begin{array}{|l}
TypeCompat : Type \leftrightarrow Type \\
\hline
TypeCompat = ext \cup \operatorname{id} Type
\end{array}$$

This means that types are compatible if they are the same or if the second is an extension of the first. When this is applied to assignment, it is clear that static type checking allows more generality than type strict languages such as Modula-2.

4.2 Dynamic Considerations

The meaning of variables is given with respect to a store. Because we allow record variables and therefore field selectors, it is not sufficient to give the meaning of a variable as a location in a store. In [BH95], a field selector sequence is also given. An appropriate schema type is declared.

$$\begin{array}{|l}
\hline \textit{DynLoc} \\
\textit{dynloc} : \textit{Loc} \\
\textit{dynflds} : \mathrm{seq}\, \textit{Id} \\
\hline
\end{array}$$

The signature for the semantic function of a variable reference is (in a given static and dynamic environment)

$$\mid\ \textit{MVarref} : \textit{Varref} \rightarrow (\textit{Store} \nrightarrow \textit{DynLoc})$$

In particular, the meaning of a pointer variable is a location in the store, together with an empty field selector sequence. The meaning of a dereferenced pointer is the value in the location determined by the meaning of the pointer itself. This is a *refval* which corresponds to a location in a particular collection in the dynamic store, again with an empty field selector sequence.

With this model for a store, *dyntype* can be used to find the dynamic type of a pointer variable. Given a static and dynamic environment, the following function is declared.

$$\mid\ \textit{dtype} : (\textit{Varref} \times \textit{Id}) \rightarrow (\textit{Store} \nrightarrow \textit{Type}))$$

The *dtype* of a pointer variable is a pointer type to the *dyntype* in the location of the dereferenced variable in a given collection in a given store.

This paper uses a concrete syntax very similar to the abstract syntax herein. The reason for the inclusion of this material is to allow the use of some of the definitions. For example, *deref* and *dtype* are used in the derivation to follow and these depend on the preceding definitions, in particular that for a store. As an example, consider the following declarations using concrete syntax.

local type *StackPtr* ≙ **pointer to** *StackRec*;
local type *StackRec* ≙ **record**
    next : *StackPtr*;
    item : **integer**
  end;
opaque type *Stack* ≙ **record**
    start : *StackPtr*;
    mem : *Store*
  end;

var *st* : *Stack*;

With the definition of a *store* as above, the dereferenced value and dynamic type of $st.start$ are given respectively by

$$st.mem(st.start)$$
$$\mathbf{pointer\ to}\ st.mem(st.start).dyntype$$

These are more naturally expressed as,

$$deref(st.start, st.mem)$$
$$dtype(st.start, st.mem)$$

in keeping with the definitions for *deref* and *dyntype*. Abbreviations such as this are used in later sections.

5 An Oberon Implementation

In the context described in Section 4, the *impl_Stacks2* module is refined to an Oberon-like implementation. It is necessary to introduce the new definition for a store to allow easy access to the dynamic type and dereferenced value of each pointer variable. At the same time concrete data types are introduced to provide the mechanism for genericity. As in Section 3, opaque type data refinement calculators are used; but this time two opaque types are refined simultaneously.

5.1 Third Data Refinement

The following local types are introduced to facilitate a *generic* opaque type for the stack.

$$
\begin{array}{l}
\mathbf{local\ type}\ GenPtr \triangleq \mathbf{pointer\ to}\ GenRec;\\
\mathbf{local\ type}\ GenRec \triangleq \mathbf{record}\\
\qquad\qquad\qquad\qquad next : GenPtr\\
\qquad\qquad\qquad\quad \mathbf{end};
\end{array}
$$

With these declarations, integer pointer and record types are introduced, using type extension. A *IntRec* is a *GenRec* extended with an *item* field of type **integer**. This allows a *IntPtr* type which is an extension of *GenPtr* to be declared.

$$
\begin{array}{l}
\mathbf{local\ type}\ IntPtr \triangleq \mathbf{pointer\ to}\ IntRec;\\
\mathbf{local\ type}\ IntRec \triangleq \mathbf{record}(GenRec)\\
\qquad\qquad\qquad\qquad item : \mathbf{integer}\\
\qquad\qquad\qquad\quad \mathbf{end};
\end{array}
$$

The new store is given by

$$
\begin{array}{l}
\textbf{local type}\ \mathit{StoredVal} \triangleq \textbf{record} \\
\qquad\qquad\qquad\qquad\qquad \mathit{dyntype} : \mathit{Type}; \\
\qquad\qquad\qquad\qquad\qquad \mathit{dynval} : \mathit{Value} \\
\qquad\qquad\qquad\qquad \textbf{end}; \\
\textbf{local type}\ \mathit{Store} \triangleq \mathit{Loc} \nrightarrow \mathit{StoredVal};
\end{array}
$$

The opaque type declaration for a generic stack includes a *start* pointer and a local store, as before. This time, however, *start* is a *GenPtr* which refers to a *GenRec* — a node that would not normally be expected to contain an integer value. In addition, the local store is a refinement of the original store.

$$
\begin{array}{l}
\textbf{opaque type}\ \mathit{Stack} \triangleq \textbf{record} \\
\qquad\qquad\qquad\qquad\qquad \mathit{start} : \mathit{GenPtr}; \\
\qquad\qquad\qquad\qquad\qquad \mathit{mem} : \mathit{Store} \\
\qquad\qquad\qquad\qquad \textbf{end} \\
\quad \textbf{invariant}(z : \mathit{Stack}) \triangleq \\
\qquad\qquad (\mathit{refval}^{\sim} z.\mathit{start} \in \operatorname{dom} z.\mathit{mem} \vee z.\mathit{start} = \textbf{nil}) \wedge \\
\qquad\qquad (\forall\, \mathit{sv} : \operatorname{ran} z.\mathit{mem} \bullet \mathit{sv}.\mathit{dyntype} = \mathit{IntRec}) \\
\quad \textbf{initially}(z : \mathit{Stack}) \triangleq (z.\mathit{start} = \textbf{nil} \wedge z.\mathit{mem} = \{\,\});
\end{array}
$$

The second part of the invariant in this declaration is essential if the original specification is to be implemented. The generic stack here is constrained to contain only nodes of type *IntRec*. This is where type extension of pointer variables is effected.

The required coupling invariant relates the old and new declarations of *Stack* and *Store*, as both of these types are simultaneously data refined.

$$
\begin{array}{l}
\mathit{CI}_{\mathit{Stack}}(v : \mathit{impl\_Stacks2}.\mathit{Stack}, z : \mathit{impl\_Stacks3}.\mathit{Stack}) \triangleq \\
\qquad v.\mathit{start} = \mathit{refval}(z.\mathit{start}) \wedge \mathit{CI}_{\mathit{Store}}(v.\mathit{mem}, z.\mathit{mem})
\end{array}
$$

where

$$
\begin{array}{l}
\mathit{CI}_{\mathit{Store}}(s : \mathit{impl\_Stacks2}.\mathit{Store}, \sigma : \mathit{Stores}.\mathit{Store}) \triangleq \\
\qquad s = \{\mathit{loc} : \operatorname{dom} \sigma \bullet \mathit{refval}(\mathit{loc}) \mapsto \sigma(\mathit{loc}).\mathit{dynval}\}
\end{array}
$$

The data refinement calculation yields the following procedures.

$$
\begin{array}{l}
\textbf{procedure } \mathit{Push}(\textbf{value result } z : \mathit{Stack};\ \textbf{value } n : \textbf{integer}) \triangleq \\
\quad z : [\text{true}, \\
\qquad \mathit{refval}^{\sim} z.\mathit{start} \notin \operatorname{dom} z_0.\mathit{mem} \wedge \\
\qquad z.\mathit{mem} = z_0.\mathit{mem} \oplus \{\mathit{refval}^{\sim} z.\mathit{start} \mapsto \\
\qquad\qquad\qquad\qquad \mathit{StoredVal}(\mathit{IntRec}, \mathit{IntRec}(z_0.\mathit{start}, n))\}]; \\
\textbf{procedure } \mathit{Pop}(\textbf{value result } z : \mathit{Stack};\ \textbf{result } m : \textbf{integer}) \triangleq \\
\quad z, m : [z.\mathit{start} \neq \textbf{nil}, \\
\qquad m = ((z_0.\mathit{mem}(\mathit{refval}^{\sim} z_0.\mathit{start})).\mathit{dynval}).\mathit{item} \wedge \\
\qquad z.\mathit{mem} = \{\mathit{refval}^{\sim} z_0.\mathit{start}\} \lhd z_0.\mathit{mem} \wedge \\
\qquad z.\mathit{start} = ((z_0.\mathit{mem}(\mathit{refval}^{\sim} z_0.\mathit{start})).\mathit{dynval}).\mathit{next}]; \\
\textbf{procedure } \mathit{MTStack}(\textbf{value } z : \mathit{Stack};\ \textbf{result } \mathit{mt} : \mathit{Boolean}) \triangleq \\
\quad \mathit{mt} : [\,\text{true}, \mathit{mt} \Leftrightarrow (z.\mathit{start} = \textbf{nil})\,]
\end{array}
$$

Apart from a few small notational differences, one can be fooled into thinking that this is not much different from *impl_Stacks2*. This is a misconception when it is remembered that there are new definitions for *Store* and *Stack*, the advantages of which are realised in the next part of the derivation.

5.2 A Stores Module

Algorithm refinement at this point allows the identification and extraction of operations on the *Store* (as in [Ban93]). A new module is defined with an opaque type *Store* and the usual procedures, *New*, *Dispose*, ...

$$
\begin{array}{l}
\textbf{module } \mathit{Stores} \triangleq \\
\textbf{opaque type } \mathit{StoredVal} \triangleq \textbf{record} \\
\qquad\qquad\qquad\qquad\qquad \mathit{dyntype} : \mathit{Type}; \\
\qquad\qquad\qquad\qquad\qquad \mathit{dynval} : \mathit{Value} \\
\qquad\qquad\qquad\qquad \textbf{end}; \\
\textbf{opaque type } \mathit{Store} \triangleq \mathit{Loc} \nrightarrow \mathit{StoredVal} \\
\quad \textbf{initially}(s : \mathit{Store}) \triangleq s = \{\,\}; \\
\textbf{procedure } \mathit{New}(\textbf{value result } s : \mathit{Store}; \\
\qquad\qquad\qquad \textbf{value } t : \mathit{Type};\ \textbf{result } p : \mathit{PtrVal}) \triangleq \\
\quad s, p : [\text{true}, (\exists\, r : \mathit{Value};\ \mathit{loc} : \mathit{Loc} \mid r \in \mathit{Val}(t) \bullet \mathit{loc} \notin \operatorname{dom} s_0 \wedge \\
\qquad\qquad\qquad p = \mathit{refval}(\mathit{loc}) \wedge s = s_0 \cup \{\mathit{loc} \mapsto \mathit{StoredVal}(t, r)\})]; \\
\textbf{procedure } \mathit{Dispose}(\textbf{value result } s : \mathit{Store};\ \textbf{value } p : \mathit{PtrVal}) \triangleq \\
\quad s : [\mathit{refval}^{\sim} p \in \operatorname{dom} s, s = \{\mathit{refval}^{\sim} p\} \lhd s_0]; \\
\qquad\qquad\qquad \vdots \\
\textbf{end}
\end{array}
$$

The *Stores* module is more general than that used in [Ban93] because it allows any pointer variable to be passed as an actual parameter to the included procedures. Another important feature is that in any given local store, pointer values are mapped to *StoredVal* — the definition of a well-formed store in the language of [BH95] allows pointers to refer to extended values. (It is of course unrealistic to attempt to implement an infinite sequence — the precondition for *New* should more correctly be $s \neq Loc$.)

5.3 Algorithm Refinement

Further algorithm refinement is calculated, with the *Stores* module now being imported into the *Stacks* implementation module. Instances of *Store* are automatically initialised to the empty set. The context described in Section 4 is used here, in particular, a *deref* variable reference as defined above. This has an advantage over defining a *Dereference* procedure in *Stores* in that it can be conveniently used as an r-value or an l-value in an assignment statement and makes an *Update* procedure unnecessary. The auxiliary function *dtype* improves the readability of the program. The result of the derivation is in Figure 4.

The implementation in Figure 4 is close to Oberon code but there is not yet a generic implementation. Several statements in the program are marked $\mathcal{G}$. These commands are all related to the maintenance of the linked data structure — in this case a simple linked list. They are unaffected by the actual *item* field in a record. For this reason it is decided to separate this "generic" code into a new module.

5.4 A Generic Module

The generic code from Figure 4 is encapsulated in the following module which is then available for import into any client program. The usual parameter-passing rules [Mor94] for procedures and the module distribution approach apply in such cases. The one difference is that in the *Push* procedure, even though p is a *GenPtr*, it may in fact refer to a *IntRec*, as detailed previously. *GenStacks* needs to import the *Stores* module to maintain the linked list that is created. The opaque type data invariant does not constrain nodes to be of type *IntRec* at this level. This is dealt with in the implementation for a stack of integers in the next section.

5.5 An Implementation Using Genericity

Finally, the generic stack is imported to instantiate a stack of integers, using type extension. It is also necessary to import *Stores* so that new nodes containing integers can be created using *New*. That all nodes are of type *IntRec* is assured by the procedures — this invariant is again distributed in the algorithm refinement. The initialisation of *Stack* is implicit from the definition of *GenStacks.Stack*. The other point of interest is the assertion $\{dtype(p, z.mem) = IntPtr\}$. In Oberon, this becomes the type assertion *p IS IntPtr* which is a more general assertion — the type of p is an *extension* of *IntPtr*. The presence of $z.mem$ in the expression

```
module impl_Stacks3 ≙
import Stores;
local type GenPtr ≙ pointer to GenRec;
local type GenRec ≙ record
                        next : GenPtr
                    end;
local type IntPtr ≙ pointer to IntRec;
local type IntRec ≙ record(GenRec)
                        item : integer
                    end;
opaque type Stack ≙ record
                        start : GenPtr;
                        mem : Store
                    end
    invariant(x : Stack) ≙
            (refval~(x.start) ∈ dom x.mem ∨ x.start = nil) ∧
            (∀ sv : ran x.mem • sv.dyntype = IntRec)
    initially(x : Stack) ≙ (x.start = nil);
procedure Push(value result z : Stack; value n : integer) ≙
var p : IntPtr •
    New(z.mem, IntPtr, p);
    deref(p, z.mem).item := n;
    deref(p, z.mem).next := z.start;                          𝒢
    z.start := p                                              𝒢
procedure Pop(value result z : Stack; result m : integer) ≙
var p : GenPtr •
    p := z.start;                                             𝒢
    z.start := deref(z.start, z.mem).next;                    𝒢
    {dtype(p, z.mem) = IntPtr}
    n := deref(p, z.mem).item
    (* Dispose(z.mem, p) − Oberon has garbage collection *)
procedure MTStack(value z : Stack; result mt : Boolean) ≙
    mt := (z.start = nil)                                     𝒢
end
```

Figure 4: Code for Stacks

$$
\begin{array}{l}
\mathbf{module}\ \mathit{GenStacks} \triangleq \\
\mathbf{import}\ \mathit{Stores}; \\
\mathbf{local\ type}\ \mathit{GenPtr} \triangleq \mathbf{pointer\ to}\ \mathit{GenRec}; \\
\mathbf{local\ type}\ \mathit{GenRec} \triangleq \mathbf{record} \\
\qquad\qquad\qquad\qquad\qquad \mathit{next} : \mathit{GenPtr} \\
\qquad\qquad\qquad\qquad \mathbf{end}; \\
\mathbf{opaque\ type}\ \mathit{Stack} \triangleq \mathbf{record} \\
\qquad\qquad\qquad\qquad\qquad \mathit{start} : \mathit{GenPtr}; \\
\qquad\qquad\qquad\qquad\qquad \mathit{mem} : \mathit{Stores.Store} \\
\qquad\qquad\qquad\qquad \mathbf{end} \\
\quad \mathbf{invariant}(x : \mathit{Stack}) \triangleq \\
\qquad\qquad\qquad (\mathit{refval}^{\sim} x.\mathit{start} \in \mathrm{dom}\, x.\mathit{mem} \vee x.\mathit{start} = \mathbf{nil}) \\
\quad \mathbf{initially}(x : \mathit{Stack}) \triangleq (x.\mathit{start} = \mathbf{nil}); \\
\mathbf{procedure}\ \mathit{GenPush}(\mathbf{value\ result}\ z : \mathit{Stack};\ \mathbf{value}\ p : \mathit{GenPtr}) \triangleq \\
\quad \mathit{deref}(p, z.\mathit{mem}).\mathit{next} := z.\mathit{start}; \\
\quad z.\mathit{start} := p \\
\mathbf{procedure}\ \mathit{GenPop}(\mathbf{value\ result}\ z : \mathit{Stack};\ \mathbf{result}\ p : \mathit{GenPtr}) \triangleq \\
\quad p := z.\mathit{start}; \\
\quad z.\mathit{start} := \mathit{deref}(z.\mathit{start}, z.\mathit{mem}).\mathit{next}; \\
\mathbf{procedure}\ \mathit{GenMTStack}(\mathbf{value}\ z : \mathit{Stack};\ \mathbf{result}\ \mathit{mt} : \mathit{Boolean}) \triangleq \\
\quad \mathit{mt} := (z.\mathit{start} = \mathbf{nil}) \\
\mathbf{end}
\end{array}
$$

Figure 5: A Generic Stack Module

is a natural consequence of using separate collections rather than a global store which is implicit in the Oberon equivalent.

6 Conclusion

The case study in this paper demonstrates a method for calculating Oberon-like code from an abstract specification. The module in Figure 6 may be transliterated into an Oberon implementation of the specification in Figure 1. The approach relies on a type extension context such as that described in Section 4. In practice, type extension is implemented using pointers which are dealt with here by declaring an explicit local store with each instance of an opaque type. The most important aspect of the approach is the definition of a more general model for the store. The use of local stores is simply a device for reasoning about pointers during the derivation. They are refined to a single global store

module *Stacks* $\triangleq$
import *Stores*, *GenStacks*;
local type *IntPtr* $\triangleq$ **pointer to** *IntRec*;
local type *IntRec* $\triangleq$ **record**(*GenStacks.GenRec*)
item : **integer**
end;
opaque type *Stack* $\triangleq$ *GenStacks.Stack*;
procedure *Push*(**value result** z : *Stack*; **value** n : **integer**) $\triangleq$
var p : *IntPtr* $\bullet$
$Stores.New(z.mem, IntPtr, p)$;
$deref(p, z.mem).item := n$;
$GenStacks.GenPush(z, p)$
procedure *Pop*(**value result** z : *Stack*; **result** m : **integer**) $\triangleq$
var p : *GenStacks.GenPtr* $\bullet$
$GenStacks.GenPop(z, p)$;
$\{dtype(p, z.mem) = IntPtr\}$
$n := deref(p, z.mem).item$
$(*\ Stores.Dispose(z.mem, p)\ *)$
procedure *MTStack*(**value** z : *Stack*; **result** mt : *Boolean*) $\triangleq$
$GenStacks.GenMTStack(z, mt)$
end

Figure 6: Code Using a Generic Stack

as a final step and are therefore not a necessary part of the implementation. The main advantage of the Oberon implementation is in the separation of concerns — the generic stack module maintains the invariant for the linked data structure.

7 Acknowledgements

Peter Bancroft would like to thank David Edmond and Ray Nickson for their comments on this paper and Mark Utting for discussions on local stores.

References

[Bac88] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(7):593–624, 1988.

[Ban93] P. G. Bancroft. Pointers in refinement calculus: a case study. In *Proceedings of the 7th Australian Software Engineering Conference*, pages 11–19. IREE, July 1993.

[Bar95] J. G. P. Barnes. *Programming in Ada 95*. Addison-Wesley, 1995.

[BDMN73] G. M. Birtwistle, O-J Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Auerbach, Philadelphia, Pa., 1973.

[BH93a] P. G. Bancroft and I. J. Hayes. Refinement of opaque types. Technical Report 267, University of Queensland, Key Centre for Software Technology, Department of Computer Science, April 1993. An earlier draft was presented at: 2nd Australian Refinement Workshop, 1992.

[BH93b] P. G. Bancroft and I. J. Hayes. Refining a module with opaque types. In Gopal Gupta, George Mohay, and Rodney Topor, editors, *Proceedings of the 16th Australian Computer Science Conference*, pages 615–624. Australian Computer Society, February 1993.

[BH95] P. G. Bancroft and I. J. Hayes. A formal semantics for a language with type extension. In Jonathan P. Bowen and Michael Hinchey, editors, *ZUM'95: The Z Formal Specification Notation*, number 967 in Lecture Notes in Computer Science, pages 299–314. Springer-Verlag, 1995.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[Hay93] I. J. Hayes(ed). *Specification Case Studies*. Prentice Hall, 2nd edition, 1993.

[HH84] R. C. Holt and J. N. P. Hume. *Introduction to Computer Science using the Turing Programming Language*. Reston Pub. Co., 1984.

[Hor79] J. J. Horning. A case study in language design: Euclid. In F. L. Bauer and M. Broy, editors, *Program Construction*, number 69 in Lecture Notes in Computer Science, pages 125–132. Springer-Verlag, 1979.

[MG90] C. C. Morgan and P.H.B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.

[Mor89] J. M. Morris. Laws of data refinement. *Acta Informatica*, 26:287–308, 1989.

[Mor94] C. C. Morgan. *Programming From Specifications*. Prentice-Hall, 2nd edition, 1994.

[MW91] H. Mössenböck and N. Wirth. The programming language Oberon-2. *Structured Programming*, 12(4), 1991.

[Rit95] T. Ritchey. *Java!* New Riders, 1995. ISBN: 1-562-05533-X.

[RW92] M. Reiser and N. Wirth. *Programming in Oberon: Steps beyond Pascal and Modula.* Addison-Wesley, 1992.

[Sei91] E. Seidwitz. Object-oriented programming through type extension in Ada9X. *Ada Letters*, 11(2):86–97, March 1991.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 2nd edition, 1992.

[Utt95] M. Utting. Reasoning about aliasing. In *Proceedings of the Fourth Australian Refinement Workshop (ARW-95)*, pages 195–211. School of Computer Science and Engineering, The University of New South Wales, April 1995.

[Wan87] M. Wand. Complete type inference for simple objects. In *Proc. Symp. on Logic in Computer Science*, pages 37–44, 1987.

[Wan93] M. Wand. Type inference for objects with instance variables and inheritance. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, Foundations of Computing Series, pages 97–120. MIT Press, 1993.

[Wir88a] N. Wirth. The programming language Oberon. *Software Practice and Experience*, 18(7), July 1988.

[Wir88b] N. Wirth. Type extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, April 1988.