Encoding Object-Z in Isabelle/HOL

Graeme Smith[1], Florian Kammüller[2], and Thomas Santen[2]

[1] Software Verification Research Centre
University of Queensland 4072, Australia
smith@svrc.uq.edu.au
[2] Technische Universität Berlin, Softwaretechnik, FR 5-6
Franklinstr. 28/29, 10587 Berlin, Germany
{flokam,santen}@cs.tu-berlin.de

Abstract. In this paper, we present a formalisation of the reference semantics of Object-Z in the higher-order logic (HOL) instantiation of the generic theorem prover Isabelle, Isabelle/HOL. This formalisation has the effect of both clarifying the semantics and providing the basis for a theorem prover for Object-Z. The work builds on an earlier encoding of a value semantics for object-oriented Z in Isabelle/HOL and a denotational semantics of Object-Z based on separating the internal and external effects of class methods.

Keywords: Object-Z, reference semantics, higher-order logic, Isabelle

1 Introduction

Isabelle/HOL is an instantiation of the generic theorem prover Isabelle [8] with a classical higher-order logic based on that of the HOL System [3]. It supports a large library of definitions and rules including those of set theory, and advanced facilities for constructing recursive datatypes and inductive and co-inductive definitions[1]. It has proven to be an ideal basis for theorem prover support for formal specification languages. Existing encodings of formal specification languages include those of Z [7] and CSP [15].

The Z encoding in Isabelle/HOL, referred to as *HOL-Z*, has been extended by Santen [9,12] to support notions of classes and objects similar to those in Object-Z [13]. The main difference between the language encoded by Santen's approach and Object-Z, however, is that the former adopts a *value semantics*: values representing objects are used directly in definitions; in particular, in the definitions of classes of other objects. Object-Z, on the other hand, has a *reference semantics*: values of objects are only referenced from definitions, not used directly. The inclusion of object references in Object-Z facilitates the refinement of specifications to code in object-oriented programming languages, which also have reference semantics.

[1] An inductive definition specifies the smallest set consistent with a given set of rules. A co-inductive definition specifies the greatest set.

D. Bert et al. (Eds.): ZB 2002, LNCS 2272, pp. 82–99, 2002.

Object references also have a profound influence on the structuring of specifications. When an object is merely referenced by another object, it is not encapsulated in any way by the referencing object. This enables the possibility of self and mutually recursive structures. While these can be very useful in specifications, reasoning about them is not always straightforward. For example, when an object calls a method (i.e., invokes an operation) of another object, the called object may in turn call a method on an object, and so on. In specifications involving recursive structures, such sequences of method calls may repeat (when one of the calls is identical to an earlier call in the sequence). The semantics of method calls needs to account for this possibility and hence is most easily defined in terms of fixed points [14].

In this paper, we build on the work of Santen, modifying and extending it to support Object-Z's reference semantics. Our approach is based on a notion of "messages" which define an object's interaction with other objects in the specification. This approach, inspired by the denotational semantics of Object-Z defined by Griffiths [5,4], supports a modular approach to reasoning about encoded specifications. It also enables us to utilise Isabelle/HOL's inductive definition facility in order to avoid the need for explicitly calculating fixed points of recursively defined method calls.

In Section 2, we outline the general approach and discuss how object references and recursion are handled. In Section 3, we discuss the Isabelle/HOL encoding of classes and objects and, in Section 4, we show how collections of these are used to define specifications and (object) environments respectively. Section 6 sketches some technicalities of the encoding that, for the sake of readability and conciseness, we ignore throughout the rest of the paper. In Section 7, we conclude with a brief discussion of future work.

2 References and Recursion: A Message-Based Approach

An Object-Z class typically includes a state schema, initial state schema and one or more methods (i.e., operations) defined by schemas or operation expressions (which are similar to schema expressions in Z). An example is class C shown in Fig. 1.

The state schema of class C declares a variable n of type naturals and a reference a to an object of class A. It constrains the variable x of the object referenced by a to be less than or equal to n. Initially, n is zero and the object referenced by a satisfies the initial condition of class A. The class has two methods: *Inc_n* increases the state variable n of an object of class C by an amount input as $n?$; *Inc_n_and_x* increases n in the same way and simultaneously applies the method *Inc_x* of class A to the object referenced by a.

The method *Inc_n* affects the state of an object of class C only. The method *Inc_n_and_x*, on the other hand, may have a wider effect: the method *Inc_x* of A may affect the state of the object referenced by a, or may call further methods. Since methods in other classes may have references to objects of class C, there is potential for recursion.

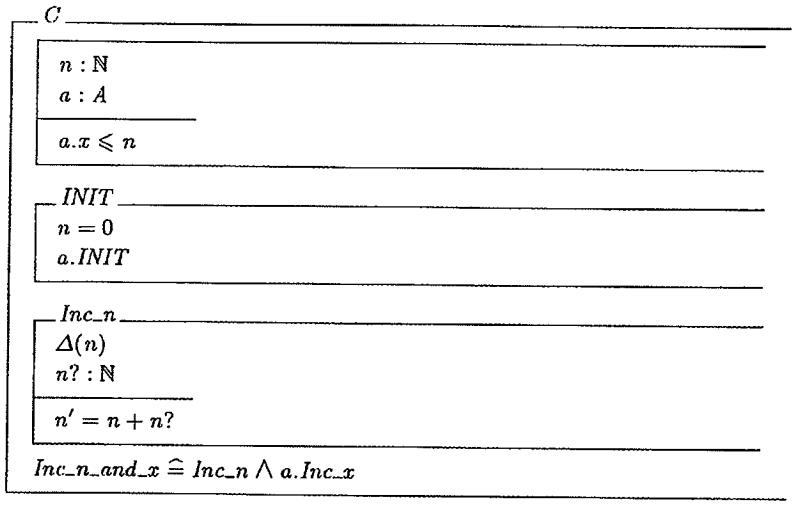

Fig. 1. Example class specification.

The obvious approach to handling this recursion in Object-Z is through fixed points. The schemas and expressions modelling the state, initial state and methods of objects in a specification can be defined as the fixed point of characteristic functions based on their syntactic definitions [14]. A specification can be defined as a fixed point on the classes of the specification where classes are ordered according to the orders on their constituent schemas.

This approach, however, ignores the modularity in the specification: the meaning of a schema is given in terms of its effect on the whole system and not just the state of the class to which it belongs. This is not ideal for reasoning about specifications where we would prefer to take advantage of the specification's modularity. Instead, we want an approach where the effect of a schema is separated into its effect on the state of its class and its effect on the rest of the system.

Following a similar argument, Griffiths [5] developed a denotational semantics for Object-Z in which operation schemas are modelled as having two parts: a relation on the state of their class, and a "message" which defines their interaction with the rest of the system. Our approach to encoding Object-Z in Isabelle/HOL adopts this approach and extends it to state and initial state schemas, which may also affect the rest of the system (as in the example above).

A message encodes an Object-Z operation expression. Syntactically, such an expression comprises names (of methods and object references) and various operation operators. We encode a message in Isabelle/HOL using the recursive datatype facility as follows. (α is a polymorphic type representing method names.

ι, ω and υ are polymorphic types representing tuples of inputs, outputs and hidden variables respectively. These polymorphic types appear as parameters to the type definition.)

$$
\begin{array}{lll}
\textit{datatype} & (\alpha, \iota, \omega, \upsilon)\ \textit{Message} = & \\
 & \quad \textit{name}\ \alpha & \\
 & \mid\ \textit{call}\ \alpha\ \alpha & \\
 & \mid\ \textit{and}\ (\alpha, \iota, \omega, \upsilon)\ \textit{Message}\ (\alpha, \iota, \omega, \upsilon)\ \textit{Message} & \\
 & \qquad ([\iota * \omega, \iota * \omega] \to \iota * \omega * \upsilon) & \\
 & \mid\ \textit{choice}\ (\alpha, \iota, \omega, \upsilon)\ \textit{Message}\ (\alpha, \iota, \omega, \upsilon)\ \textit{Message} & \\
 & \mid\ \textit{sequence}\ (\alpha, \iota, \omega, \upsilon)\ \textit{Message}\ (\alpha, \iota, \omega, \upsilon)\ \textit{Message} & \\
 & \qquad ([\iota * \omega, \iota * \omega] \to \iota * \omega * \upsilon) &
\end{array}
$$

Such a datatype is similar to a free-type in Z, but more restrictive, because the parameters to the definition are types, which correspond to maximal sets in Z, whereas the parameters to a free-type definition in Z can be arbitrary sets. The above definition defines five kinds of messages. The constructor *and* serves to encode the Object-Z operation operators $\wedge$ and $\|$. The operators *choice* and *sequence* correspond to the Object-Z operation operators $[]$ and $\fatsemi$ respectively. Each combine two messages. The functions of type $[\iota * \omega, \iota * \omega] \to \iota * \omega * \upsilon$ associated with the *and* and *sequence* messages are isomorphisms [11] used to combine the input and output tuples of component messages to form those of the composite message. For $\|$ and $\fatsemi$, the third result type υ represents those inputs and outputs hidden in the composite message. The use of these isomorphisms is explained in Section 4 and illustrated by the example in Section 5. Other uses of isomorphisms in our embedding, which for reasons of conciseness are largely ignored in this paper, are briefly discussed in Section 6.

The other two kinds of messages, *name* and *call* represent the base cases of the recursive definition. A *name* message is used to refer to a method within the class in which the message occurs. This method may be another message or a schema defining an effect on the state of the class. A *call* message is used to refer to a method of another object. It comprises two names: the first is of the object reference of the object, and the second of the method from that object's class. For example, the message corresponding to the method *Inc_n_and_x* above is *and* (*name Inc_n*) (*call a Inc_x*) *IO*, where the function *IO* determines the inputs and outputs of *Inc_n_and_x* from those of *Inc_n* and *a.Inc_x*.

3 Classes and Objects

Building on the approach of Santen [9,12], we encode an Object-Z class in Isabelle/HOL as a tuple. (γ is a polymorphic type representing *object identifiers* which are used to reference objects. κ and σ are polymorphic types representing tuples of constants and tuples of state variables respectively.)

$$
\begin{aligned}
&\mathit{typedef}\quad (\gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \mathit{classschema}\\
&\quad = \{(C, S, I, \mathit{Mths}, \mathit{Msgs}, \mathit{Refs}) \mid\\
&\qquad (C :: \kappa\ \mathit{constschema})\\
&\qquad (S :: (\kappa, \sigma)\ \mathit{stateschema})\\
&\qquad (I :: (\kappa, \sigma)\ \mathit{initschema})\\
&\qquad (\mathit{Mths} :: (\alpha, (\kappa, \sigma, \iota, \omega, \upsilon)\ \mathit{methodschema})\ \mathit{finmap})\\
&\qquad (\mathit{Msgs} :: (\alpha, (\alpha, \iota, \omega, \upsilon)\ \mathit{Message})\ \mathit{finmap})\\
&\qquad (\mathit{Refs} :: (\alpha, \omega \to \gamma)\ \mathit{finmap}).\\
&\qquad \mathit{dom}_m\ \mathit{Refs} \subseteq \mathit{dom}_m\ \mathit{Mths}\}
\end{aligned}
$$

The first three elements of a class tuple, C, S and I, are $\mathcal{HOL}$-$\mathcal{Z}$ schemas, i.e., functions from tuples to Booleans [7], defining the allowable tuples of constants, state variables and initial state variables respectively[2]. They represent the specified conditions on the state of an object of the class only. That is, for the example class of Section 2, the predicates S and I represent the following schemas.

$$
\begin{array}{|l}
n : \mathbb{N} \\
a : A \\
\hline
\exists\, a\_x : \mathbb{N} \bullet a\_x \leqslant n \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\mathit{INIT} \\
\hline
\exists\, a\_\mathit{init} : \mathbb{B} \bullet \\
\quad n = 0 \\
\quad a\_\mathit{init} \\
\hline
\end{array}
$$

These schemas facilitate a modular approach to reasoning. They enable reasoning about objects of the class in isolation from any specified system in which they may occur. For reasoning about complete specifications, however, the global effects implied by a specified class's constant definitions, state schema and initial state schema need also to be captured by the class encoding. For the constants and state schema, the effect is captured by a message *State*. For the example class, this message corresponds to the operation expression $a.X \parallel St$ where St is a method schema

$$
\begin{array}{|l}
St \\
\hline
x? : \mathbb{N} \\
\hline
x? \leqslant n \\
\hline
\end{array}
$$

and X is an *observer*, i.e., a method schema which outputs the value of a state variable,

[2] The actual definitions of the types *constschema*, *stateschema*, *initschema*, as well as that of *methodschema*, can be found in Santen [12, Chapter 5]

X

$x! : \mathbb{N}$

$x! = x$

in class A. In general, the message *State* includes one observer for each state variable of a referenced object referred to in the constant definitions or state schema.

The global effect of the initial state schema is captured in a similar way by a message *Init*. For the example class, this message corresponds to the operation expression $a.INIT \parallel In$ where In is a method schema

In

$init? : \mathbb{B}$

$n = 0$
$init?$

and *INIT* is a method schema in A which outputs $init! : \mathbb{B}$ whose value is true precisely when the initial condition of A is true.

The next two elements of a class tuple, *Mths* and *Msgs*, are finite partial functions (encoded using the type *finmap* of finite mappings [12, Appendix B.1]) between names and method schemas, and names and messages respectively. The method schemas capture all the ways in which the class can affect the state of its objects. They include an observer for each state variable and constant (including the implicit constant *self* which is the identifier of a given object) and the method schemas *St* and *In*. The messages capture the ways in which objects of the class can interact with other objects in a specified system. They include the messages *State* and *Init*.

The final element of a class tuple is a function *Refs* that maps all observers which output an object identifier to the identifier they output. This is necessary as the strong typing of Isabelle/HOL will not allow an output of generic type ω to be identified with an object identifier of generic type γ. The only constraint on a class tuple ensures that the domain of *Refs* is a subset of that of *Mths*, i.e., all observers which output an object identifier are also methods. (This is encoded using the function dom_m which returns the domain of a finite mapping [12, Appendix B.1]).

Given this encoding of a class, we can encode the notion of objects in a way suggested by Santen [12]. Firstly, we encode an object state as a cross product of two tuples, corresponding to the values of the object's constants and variables respectively.

types (κ, σ) *objstate* $= \kappa * \sigma$

Then, the object states belonging to objects of a given class are returned by a function *ObjsS* (*cma* and *sma* return the Boolean-valued functions representing the constant and state schemas of a class respectively).

$$\begin{array}{ll} \textit{constdefs} & \textit{ObjsS} :: (\gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \textit{classschema} \to (\kappa, \sigma)\ \textit{objstate set} \\ & \textit{ObjsS}\ \textit{Cls} == \{(c, s) \mid c\ s.\ (\textit{cma}\ \textit{Cls})\ c \wedge (\textit{sma}\ \textit{Cls})\ c\ s\} \end{array}$$

Given this definition, an object is encoded as an ordered pair where the first element is the class of the object and the second element is the object's state, which must be an object state of its class.

$$\begin{array}{ll} \textit{typedef} & (\gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \textit{object} \\ & = \{(\textit{Cls}, \textit{obj}) \mid \textit{obj} \in \textit{ObjsS}\ \textit{Cls}\} \end{array}$$

Based on the definition in Santen [12, Chapter 6], we define method invocation as follows. ($\underline{obj}$ denotes the ordered pair representing an object obj. $mths$ returns the set of methods associated with a class. $\diamond_c^1$ is the method selector for classes, i.e., $\textit{Cls} \diamond_c^1 n$ returns the method of class $\textit{Cls}$ named n. $\textit{Meth}$ returns the boolean-valued function associated with a method schema. This function takes as arguments a constant tuple c, two variable tuples s and s' denoting the pre- and post-states, an input tuple i and an output tuple o.)

constdefs

$$\begin{array}{l} \xrightarrow{(\_)} :: [(\gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \textit{object} * \iota, \alpha, (\gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \textit{object} * \omega] \to \textit{bool} \\ \textit{oi} \xrightarrow{n} \textit{oo} == (\textbf{let}\ (\textit{obj}, i) = \textit{oi}; \\ \qquad\qquad (\textit{Cls}, \textit{obj}_{st}) = \underline{\textit{obj}}; \\ \qquad\qquad (c, s) = \textit{obj}_{st}; \\ \qquad\qquad (\textit{obj}', o) = \textit{oo}; \\ \qquad\qquad (\textit{Cls}', \textit{obj}'_{st}) = \underline{\textit{obj}'}; \\ \qquad\qquad (c', s') = \textit{obj}'_{st} \\ \qquad \textbf{in}\ c' = c \wedge \textit{Cls}' = \textit{Cls} \wedge n \in \textit{mths}\ \textit{Cls} \wedge \\ \qquad\qquad (\textit{Meth}\ (\textit{Cls} \diamond_c^1 n))\ c\ s\ s'\ i\ o) \end{array}$$

4 Specifications and Environments

An Object-Z specification comprises a set of classes each with a unique name and a set of unique object identifiers. We encode it as a tuple comprising two finite mappings as follows. (β is a polymorphic type representing class names).

$$\begin{array}{ll} \textit{typedef} & (\beta, \gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \textit{spec} \\ & = \{(\textit{CMap}, \textit{IdMap}) \mid \\ & \quad (\textit{CMap} :: (\beta, (\gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \textit{classschema})\ \textit{finmap}) \\ & \quad (\textit{IdMap} :: (\beta, \gamma\ \textit{set})\ \textit{finmap}). \\ & \quad \textit{dom}_m\ \textit{CMap} = \textit{dom}_m\ \textit{IdMap} \wedge \\ & \quad (\forall c_1\ c_2. \\ & \qquad c_1 \in \textit{dom}_m\ \textit{CMap} \wedge c_2 \in \textit{dom}_m\ \textit{CMap} \wedge c_1 \neq c_2 \\ & \qquad\quad \Rightarrow \textit{IdMap}\ c_1 \cap \textit{IdMap}\ c_2 = \varnothing)\} \end{array}$$

The first mapping $\textit{CMap}$ relates class names in the specification with classes. The second $\textit{IdMap}$ associates class names with sets of identifiers. The constraint

ensures that each class in the specification is associated with a set of unique identifiers.

To reason about specifications using the message-based approach, we need to introduce the notion of an *environment*. An environment is an instance of a specification (in much the same way that an object is an instance of a class). It associates each object identifier with an object in a way which satisfies the specification, i.e., each identifier belonging to a class in the specification maps to an object of that class.

We encode the notion of an environment as a function from identifiers to objects. The identifier mapping to a given object must be the same as the constant *self* of that object. (*cls_of* is a function which returns the class component of an object. *refs* is a function which returns the finite mapping *Refs* of a class. $\diamond_m$ is the application operator for finite mappings.)

$$\begin{aligned} &\mathit{typedef} \quad (\gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \mathit{Env} \\ &\quad = \{e :: \gamma \to (\gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \mathit{object}. \\ &\qquad \forall\, \mathit{id}.\ (\exists\, \mathit{out}.\ (\mathit{refs}\ (\mathit{cls\_of}\ (e\ \mathit{id})))\diamond_m \mathit{self})\ \mathit{out} = \mathit{id})\} \end{aligned}$$

The environments which are valid for a specification are given by the function *spec_envs*. (*classes_of* is a function which returns the finite mapping *CMap* of a specification. *ids_of* is a function which returns the finite mapping *IdMap* of a specification).

$$\begin{aligned} &\mathit{constdefs} \quad \mathit{spec\_envs} :: (\beta, \gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \mathit{spec} \to (\gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \mathit{Env\ set} \\ &\qquad \mathit{spec\_envs} == (\lambda S. \\ &\qquad\quad \{e.\ (\forall\, \mathit{cn} : \mathit{dom}_m\ (\mathit{classes\_of}\ S).\ (\forall\, \mathit{id} : (\mathit{ids\_of}\ S)\diamond_m \mathit{cn}. \\ &\qquad\qquad \mathit{cls\_of}\ (e\ \mathit{id}) = (\mathit{classes\_of}\ S)\diamond_m \mathit{cn}))\}) \end{aligned}$$

To determine the effect on an environment of a particular event, we introduce an *effect* as a tuple comprising a pre-environment, a post-environment, an object identifier, a message and a tuple of inputs and output parameters.

$$\begin{aligned} &\mathit{types} \quad (\gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \mathit{Effect} \\ &\quad = ((\gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \mathit{Env} * (\gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \mathit{Env} \\ &\quad\ *\gamma * (\alpha, \iota, \omega, \upsilon)\ \mathit{Message} * \iota * \omega) \end{aligned}$$

The set *Effects* denotes all possible effects, i.e., all changes to environments caused by sending a message to one of their constituent objects.

$$\mathit{consts} \quad \mathit{Effects} :: ((\gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \mathit{Effect})\ \mathit{set}$$

It can be defined using Isabelle/HOL's inductive definition facility. It is the smallest set satisfying the following rules. (*msgs* returns the finite mapping *Msgs* of a class. $\underline{e}$ denotes the function representing an environment e. $\diamond_c^2$ is the message selector for classes, i.e., $\mathit{Cls}\ \diamond_c^2\ n$ returns the message of class *Cls* named n.)

inductive Effects

intrs

$$\frac{n \in dom_m\ (mths\ (cls\_of\ (\underline{e}\ id)))\quad ((\underline{e}\ id), i) \xrightarrow{n} ((\underline{e'}\ id), o)}{(e, e', id, name\ n, i, o) \in Effects}\ [\ nameImth\ ]$$

$$\frac{(refs\ (cls\_of\ (\underline{e}\ id))\ \diamond_m\ n)\ o' = id' \quad (e, e', id', name\ m, i, o) \in Effects}{(e, e', id, call\ n\ m, i, o) \in Effects}\ [\ callI\ ]$$

$$\frac{(e, e', id, m_1, i_1, o_1) \in Effects \quad (e, e', id, m_2, i_2, o_2) \in Effects \quad (i, o, h) = IO\ (i_1, o_1)\ (i_2, o_2)}{(e, e', id, and\ m_1\ m_2\ IO, i, o) \in Effects}\ [\ andI\ ]$$

$$\frac{(e, e', id, m_1, i, o) \in Effects}{(e, e', id, choice\ m_1\ m_2, i, o) \in Effects}\ [\ choiceIleft\ ]$$

$$\frac{(e, e', id, m_2, i, o) \in Effects}{(e, e', id, choice\ m_1\ m_2, i, o) \in Effects}\ [\ choiceIright\ ]$$

$$\frac{(e, e'', id, m_1, i_1, o_1) \in Effects \quad (e'', e', id, m_2, i_2, o_2) \in Effects \quad (i, o, h) = IO\ (i_1, o_1)\ (i_2, o_2)}{(e, e', id, sequence\ m_1\ m_2\ IO, i, o) \in Effects}\ [\ sequenceI\ ]$$

$$\frac{n \in dom_m\ (msgs\ (cls\_of\ (\underline{e}\ id))) \quad m = (cls\_of\ (\underline{e}\ id))\ \diamond_c^2\ n \quad (e, e', id, m, i, o) \in Effects}{(e, e', id, name\ n, i, o) \in Effects}\ [\ nameImsg\ ]$$

The rule *nameImth* has two conditions which must be met for an effect $(e, e', id, name\ n, i, o)$ to be in *Effects*. The first condition is that n is the name of a method schema of the object identified by id in environment e. The second is that the invocation of this method schema with inputs i and outputs o transforms the object to that identified by id in environment e'.

From the base set of effects generated by *nameImth*, the other rules define the set of all possible effects. For example, the rule *callI* states that the effect $(e, e', id, call\ n\ m, i, o)$ is in *Effects* whenever n is an observer of the class of the object identified by id in e that outputs id', and $(e, e', id', name\ m, i, o)$ is a member of *Effects*. Similarly, the rules *andI*, *choiceIleft* and *choiceIright*, and *sequenceI* state when *and*, *choice* and *sequence* messages, respectively, are in *Effects*.

The parameter functions *IO* of *and* and *sequence* messages map the inputs and outputs of the constituent messages to those of the composite messages and those that are hidden in the composite messages. They model the effect of building the union of schema signatures and subtracting the hidden part of the signatures in the schema calculus. The third component h of the result of applying *IO* is effectively hidden by the rules *andI* and *sequenceI*, because h does not appear in the conclusion of those rules. Similarly, the environment e'' representing the "intermediate state" of a method sequencing is hidden by *sequenceI*.

The final rule *nameImsg* states that the effect $(e, e', id, name\ n, i, o)$ is in *Effects* whenever n is the name of a message of the class of the object identified by id in e, and (e, e', id, m, i, o) is in *Effects* where m is the message associated with name n.

Each Object-Z specification has a distinguished *system class* which specifies the structure of the system and possible interactions between its objects. The effects which are valid for a specification are those that correspond to methods and messages of the system class. They are defined by the function *spec_effects* which takes as parameters a specification and the identifier of its system object (an object of the system class). This function also ensures that the state invariants of all objects are met before and after the effect. This is specified by showing that the effect corresponding to the message *State* (see Section 3), is allowed in the pre- and post-environments of the effect. (Note that *State* is simply a condition on the environment and does not change it.)

constdefs

$$
\begin{aligned}
&spec\_effects :: [(\beta, \gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ spec, \gamma] \to (\gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ Effect\ set \\
&spec\_effects == (\lambda S\ id. \\
&\quad \{(e, e', id, name\ n, i, o).\ e \in spec\_envs\ S \wedge e' \in spec\_envs\ S \wedge \\
&\quad (e, e, id, name\ State, (), ()) \in Effects \wedge \\
&\quad (e', e', id, name\ State, (), ()) \in Effects\})
\end{aligned}
$$

To reason in a modular fashion, we can treat any class in the specification as the system class of a *sub-specification*. The sub-specification corresponding to a given class comprises that class and the classes of all objects which it references.

When reasoning about the effect of a particular message or sequence of messages on an environment, we often wish to limit our attention to those environments which are *initial environments* of the specification. That is, those environments which satisfy the initial state schema of the system class. The set of

such environments for a given specification and system identifier is encoded in terms of the *Init* message (see Section 3) of the system class as follows.

constdefs

$$\begin{array}{l} \mathit{initial\_envs} :: [(\beta, \gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \mathit{spec}, \gamma] \to (\gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \mathit{Env\ set} \\ \mathit{initial\_envs} == (\lambda S\ \mathit{id}. \\ \quad \{e.\ e \in (\mathit{spec\_env}\ S) \wedge (e, e, \mathit{id}, \mathit{name}\ \mathit{Init}, (), ()) \in (\mathit{spec\_effects}\ S\ \mathit{id})\}) \end{array}$$

In other cases, we need to limit our attention to *reachable environments* of the specification. That is, those environments which result from the application of zero or more valid effects to an initial environment. The set of such environments for a given specification and system identifier is encoded as an inductive definition as follows.

consts

$$\mathit{reachable\_envs} :: [(\beta, \gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \mathit{spec}, \gamma] \to (\gamma, \alpha, \kappa, \sigma, \iota, \omega, \upsilon)\ \mathit{Env\ set}$$

inductive reachable_envs S id

intrs

$$\frac{e \in (\mathit{initial\_envs}\ S\ \mathit{id})}{e \in (\mathit{reachable\_envs}\ S\ \mathit{id})}\ [\ \mathit{initI}\ ]$$

$$\frac{\begin{array}{l} e \in (\mathit{reachable\_envs}\ S\ \mathit{id}) \\ (e, e', \mathit{id}, m, i, o) \in (\mathit{spec\_effects}\ S\ \mathit{id}) \end{array}}{e' \in (\mathit{reachable\_envs}\ S\ \mathit{id})}\ [\ \mathit{effectI}\ ]$$

The rule *initI* states that all initial environments are reachable environments. The rule *effectI* states that all environments which result from the application of a valid effect to a reachable environment are also reachable environments.

5 Example Encoding

In this section, we present an example encoding of an Object-Z specification. The specification is of a simple multiplexer based on that of Smith [13, Chapter 1]. The specification comprises two classes modelling a generic queue and a multiplexer comprising two input queues and an output queue of messages.

The generic queue in Fig. 2 is modelled with three state variables: *items*, denoting the sequence of items in the queue, *in* denoting the total number of items which have ever joined the queue, and *out* denoting the total number of items which have ever left the queue. Initially, the queue is empty and no items

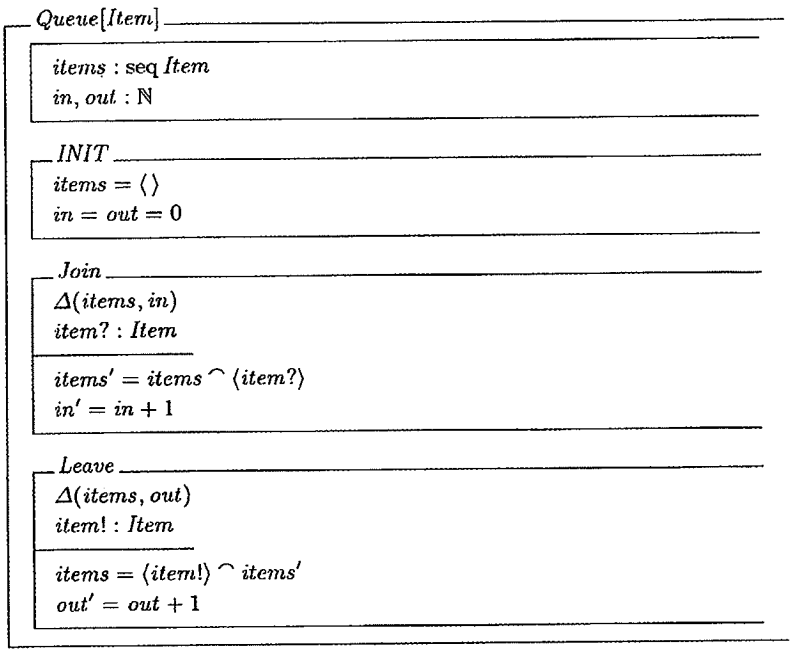

Fig. 2. Class *Queue*.

have joined or left it. Operations *Join* and *Leave* allow items to join and leave the queue respectively.

In Fig. 3, the multiplexer is modelled as having two input queues, $input_1$ and $input_2$, of a given type *Message* and an output queue, *output*, also of the type *Message*. The class's state invariant ensures that the queues are distinct, i.e., not aliases for the same queue object, and that the number of items that have joined the output queue is equal to the sum of the numbers of items that have left the input queues.

Initially, each queue is in its initial state as defined in class *Queue*. Operations $Join_1$ and $Join_2$ allow messages to be joined to queues $input_1$ and $input_2$ respectively. The operation *Transfer* allows a message from one of the input queues to be transferred to the output queue and the operation *Leave* allows a message to leave the output queue.

To illustrate our encoding, we represent this example of an Object-Z specification in HOL. Since the class *Queue* does not contain any messages, its representation is similar to a representation in Santen's earlier encoding [12]. We just show the observer *INIT* of *Queue* below; the new aspects of encoding a class are illustrated by sketching the representation of the class *Multiplexer*.

Multiplexer

$input_1, input_2, output : Queue[Message]$

$input_1 \neq input_2 \wedge input_1 \neq output \wedge input_2 \neq output$
$output.in = input_1.out + input_2.out$

INIT

$input_1.INIT \wedge input_2.INIT \wedge output.INIT$

$Leave_1 \mathrel{\widehat{=}} input_1.Leave$
$Leave_2 \mathrel{\widehat{=}} input_2.Leave$
$Transfer \mathrel{\widehat{=}} (Leave_1 \,[\!]\, Leave_2) \parallel output.Join$
$Leave \mathrel{\widehat{=}} output.Leave$

Fig. 3. Class *Multiplexer*.

Assuming the definition of the methods and observers that are needed, we build the messages *State* and *Init* (see Section 3) by composing the corresponding constituents. For example, the message *Init* is built using the *INIT* observer of class *Queue*. This is a method schema which returns a boolean value $init! : \mathbb{B}$ representing whether the initial condition of *Queue* holds. It is thus defined as:

INIT

$\Delta()$
$init! : \mathbb{B}$

$items = \langle\rangle$
$in = out = 0$

For each of the three aggregated objects, $input_1$, $input_2$, and *output*, a call to this method is issued in the initialisation of *Multiplexer*. For illustration purposes, the output variables *init!* are all made distinct by renaming them accordingly. These init observers are composed in parallel with a method schema *In* of the class *Multiplexer*.

In

$init_1?, init_2?, init_3? : \mathbb{B}$

$init_1? \wedge init_2? \wedge init_3?$

Thus the message *Init* for the class *Multiplexer* can be constructed as

$$input_1.INIT \wedge input_2.INIT \wedge output.INIT \parallel In.$$

In our encoding, this *Init* message is represented as

constdefs

$$\begin{aligned}&\mathit{MultiplexerInit} ==\\ &\quad \mathit{and}\ (\mathit{and}\ (\mathit{and}\ (\mathit{call}\ \text{"}\mathit{input}_1\text{"}\ \text{"}\mathit{INIT}\text{"})(\mathit{call}\ \text{"}\mathit{input}_2\text{"}\ \text{"}\mathit{INIT}\text{"})\ldots)\\ &\qquad (\text{"}\mathit{output}\text{"}\ \text{"}\mathit{INIT}\text{"})\ldots)\\ &\qquad (\mathit{name}\ \text{"}\mathit{In}\text{"})\\ &\qquad \lambda_{iso}((i_1,o_1),(i_2,o_2)).(i_1,o_2,o_1)\ |_{\{((i_1,o_1),(i_2,o_2)).o_1=i_2\}}\ .\end{aligned}$$

Using *and* and hiding, we can model the parallel composition operator $\|$: we build the conjunction of the first three init observers and the method *In* and use the function that is a further argument to the constructor of an *and* message to express the amalgamation and hiding of the parameters. The λ_{iso} term identifies o_1 and i_2 by using a domain restriction, and maps the result to the third position in the image — the hiding position. Evaluating the effects of a specification (see Section 4, rule *andI*) discards the element of an *and* message. For clarity, we omit the functions to compose the inputs and outputs of the first two *and* submessages. They are similar to the last function, but simpler as they neither identify parameters nor hide them.

The *State* message that comprises observers and methods for the internal and external effects of the state schema of the class *Multiplexer* is built in a very similar fashion. The messages for Leave_1, Leave_2, and *Leave* are left out here as they are simple *call* messages to the respective objects.

The message *Transfer* is constructed as

constdefs

$$\begin{aligned}&\mathit{MultiplexerTransfer} ==\\ &\quad \mathit{and}\ (\mathit{choice}\ (\mathit{name}\ \text{"}\mathit{Leave}_1\text{"})(\mathit{name}\ \text{"}\mathit{Leave}_2\text{"}))\\ &\qquad (\mathit{call}\ \text{"}\mathit{output}\text{"}\ \text{"}\mathit{Join}\text{"})\\ &\qquad (\lambda_{iso}((i_1,o_1),(i_2,o_2)).(i_1,o_2,o_1))\ |_{\{((i_1,o_1),(i_2,o_2))|o_1=i_2\}}\end{aligned}$$

Similar to the *Init* message, the parallel composition is achieved by a corresponding function that identifies the inputs and outputs of the submessages and maps elements that have to be hidden to the third position of the resulting triple.

After these preparations, we can build the representation of the class *Multiplexer* by adding all the constructed observers, methods and messages to a starting element — *basicclass*. The operators $\boxdot_r$ and $\boxdot$ add observers (the former, observers which return an object reference), $\boxplus$ adds methods, and $\boxplus_m$ adds messages to a class.

constdefs

$$
\begin{array}{l}
\mathit{MultiplexerCls} == \\
\quad ((\mathit{basicclass}\ \mathit{MultiplexerConstSchema} \\
\qquad\qquad\quad \mathit{MultiplexerStateSchema} \\
\qquad\qquad\quad \mathit{MultiplexerInitSchema}) \\
\quad \boxdot_r\ (\text{"}\mathit{self}\text{"}, \mathit{MultiplexerselfObs}) \\
\quad \boxdot_r\ (\text{"}\mathit{input}_1\text{"}, \mathit{Multiplexerinput}_1\,\mathit{Obs}) \\
\quad \boxdot_r\ (\text{"}\mathit{input}_2\text{"}, \mathit{Multiplexerinput}_2\,\mathit{Obs}) \\
\quad \boxdot\ \ (\text{"}\mathit{INIT}\text{"}, \mathit{MultiplexerinitObs}) \\
\quad \boxplus\ \ (\text{"}\mathit{State}\text{"}, \mathit{MultiplexerStateOp}) \\
\quad \boxplus\ \ (\text{"}\mathit{In}\text{"}, \mathit{MultiplexerInitOp}) \\
\quad \boxplus_m\ (\text{"}\mathit{Leave1}\text{"}, \mathit{MultiplexerLeave1}) \\
\quad \boxplus_m\ (\text{"}\mathit{Leave2}\text{"}, \mathit{MultiplexerLeave2}) \\
\quad \boxplus_m\ (\text{"}\mathit{Leave}\text{"}, \mathit{MultiplexerLeave}) \\
\quad \boxplus_m\ (\text{"}\mathit{Init}\text{"}, \mathit{MultiplexerInit}) \\
\quad \boxplus_m\ (\text{"}\mathit{Transfer}\text{"}, \mathit{MultiplexerTransfer}))
\end{array}
$$

In a final step, the representations for the classes *Queue* and *Multiplexer* are used to form a specification which is accessible as a single HOL object.

6 A Note on HOL Technicalities

In order to concentrate on the semantic issues that form the core of the encoding of the reference semantics of Object-Z outlined in this paper, we left out some technical details concerning the types of methods and messages. However, having made the essentials clear, we will now discuss a few specialities of the encoding that are particularly interesting from the theorem proving perspective of this work. Naturally, they are relevant for the value of our approach as well, as they determine the applicability of the encoding.

As a major design decision for the mechanical support of Object-Z in Isabelle, we have adopted the approach to formalisation that has previously been taken by Santen [12]: our encoding uses a so-called *shallow embedding* of Z and a *deep embedding* of Object-Z on top of it. In this final section, we will briefly describe these technical terms in order to lead on to an informal discussion of the problems we have encountered with the way we chose to encode Object-Z.

6.1 Shallow versus Deep Embedding

The terms shallow and deep [2] refer to the style chosen for the formalisation of a formal language, say Object-Z, in the logic of a theorem prover. The depth of an embedding describes to what extent the encoded language is made a first class citizen of the logic. That is, are the objects of the language all explicitly described as terms in the logic — then we have deep embedding — or are some concepts of the language identified with concepts of the logic — a shallow embedding? For example, in the shallow embedding of Z the input and output types of

operations are identified with types of higher-order logic (HOL). This is sound as the semantics of Z and HOL are very similar in particular with respect to the type system [10].

In general, for reasoning about a language it is advisable to have a deep embedding when the aim is to reason *about* a language rather than *in* the language. In particular, in cases where the encoded language has abstract structures, like Object-Z classes, it has been illustrated that in order to support modular reasoning in a general fashion one needs to represent structures as first class citizens [6]. When it comes to reasoning about a concrete sentence of the embedded language, like the encoding of the class *Multiplexer*, a deep embedding is not very efficient. Therefore, the art of designing a feasible and practical embedding lies in deciding how deep to embed each feature of the language.

6.2 Input and Output Types

Although the shallow embedding of Z is very useful for dealing with concrete specifications, the level of detail provided by the concrete input and output types of operations gets in the way when it comes to the level of Object-Z. A class in Object-Z includes a set of methods, that may in general have arbitrary input and output types. Since a type of classes is needed to be able to express concepts like specifications and relations between them, like refinement or behavioral conformance, it is necessary to unify all the different types of inputs and outputs of methods of a class. The unified input and output types are then used in the definition of the class type (cf. Section 3). In other words, at the Object-Z level, we need a deep embedding (in which we explicitly describe the unified input and output types of methods) to be able to achieve the right level of abstraction.

The type system of Isabelle/HOL has been chosen in such a way that type checking and typeability are decidable. In terms of the λ-cube [1], the type system of HOL corresponds to $\lambda{\rightarrow}$-Church. More powerful type systems of this cube, like $\lambda 2$, have universally quantified type variables, e.g. $\forall \alpha.\alpha$, that could be used to express unified types for inputs and outputs of class methods. For example, $(nat * nat) \rightarrow bool$ and $nat \rightarrow nat$ are both instances of $(\forall \alpha.\alpha) \rightarrow (\forall \alpha.\alpha)$.

However, in HOL, i.e. $\lambda{\rightarrow}$-Church, a polymorphic type expression, e.g. $\alpha \rightarrow \alpha$ may refer to an arbitrary α but all occurrences of α in instances have to be the same. In other words, the universal quantification of the type variable α is always at the beginning of a type expression. Therefore, it is necessary to build unified input and output types of all class methods in HOL.

One way to do this is to use the binary sum type constructor + that constructs the sum of two types, i.e., a new type containing two distinguishable copies of both parameter types. Applying this type constructor in an iterated way to all the input and output types of methods, a most general type of all inputs and outputs is created. This provides a means of expressing the type of a class in HOL in which methods also have a most general type.

However, in order to apply the methods unified in this most general type, we need to be able to retrieve the original input and output types of the methods. To that end, we have to administer injections that record the positions where

the concrete input and output types are embedded in the general sum types of the method types.

These injections have been developed in a general way by Santen [11] as so-called *isomorphisms*, as they build bijections on the general sum types that respect the term structure. In our encoding, not only methods but also the messages of Object-Z have input and output types. Therefore, we integrate the existing concept of isomorphisms for methods with the recursive datatype for messages. The concept of isomorphisms is also used for capturing the internal relationship of inputs and outputs of the composite *and* and *sequence* messages.

The handling of the types of inputs and outputs is manageable in our encoding, but it creates a considerable formal overhead. We consider the actual encoding of the input and output types with isomorphisms an implementation detail that may be interesting from the theorem proving perspective, but has to be hidden from the user. We need to implement specially tailored tactics that exploit internal term structure information of the theorem prover representation to hide such implementation detail from the user of our encoding. Fortunately, it is possible to create such additional tactic support in Isabelle. Although we have not implemented it yet, we will do so in the near future.

7 Conclusions

We have presented a new approach of encoding object-oriented specifications with reference semantics in higher-order logic. Central to our encoding is the distinction between the internal state changes and the external effects of method invocations that Griffiths has proposed. This makes the encoding more modular than a straightforward fixed point construction of a global environment of objects would.

Technically, we have built a theory to support our encoding in Isabelle/HOL. It remains to implement tactics that hide the technicalities of the encoding, in particular type conversions by isomorphisms, from the users' view. With those tactics at hand, we will be able provide a proof environment for Object-Z and investigate how the modularity of our encoding helps to modularise reasoning about Object-Z specifications. We feel that only modularisation of proofs will enable us to verify properties of specifications of realistic size and complexity.

Acknowledgements. This research was carried out while Graeme Smith was visiting Berlin on a Research Fellowship generously provided by the Alexander von Humboldt Foundation (AvH), Germany. As well as thanking the AvH, Graeme would like to thank GMD-FIRST, Berlin, for the use of an office and computing facilities during his stay.

References

1. H. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science, Vol. 2.* Oxford University Press, 1992.

2. J. Bowen and M. Gordon. A shallow embedding of Z in HOL. *Information and Software Technology*, 37(5–6):269–276, 1995.
3. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
4. A. Griffiths. *A Formal Semantics to Support Modular Reasoning in Object-Z*. PhD thesis, University of Queensland, 1997.
5. A. Griffiths. Object-oriented operations have two parts. In D.J. Duke and A.S. Evans, editors, *2nd BCS-FACS Northern Formal Methods Workshop*, Electronic Workshops in Computing. Springer-Verlag, 1997.
6. F. Kammüller. *Modular Reasoning in Isabelle*. PhD thesis, Computer Laboratory, University of Cambridge, 1999. Technical Report 470.
7. Kolyang, T. Santen, and B. Wolff. A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs 96)*, volume 1125 of *Lecture Notes in Computer Science*, pages 283–298. Springer-Verlag, 1996.
8. L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
9. T. Santen. A theory of structured model-based specifications in Isabelle/HOL. In E.L. Gunter and A. Felty, editors, *Theorem Proving in Higher-Order Logics (TPHOLs 97)*, volume 1275 of *Lecture Notes in Computer Science*, pages 243–258. Springer-Verlag, 1997.
10. T. Santen. On the semantic relation of Z and HOL. In J. Bowen and A. Fett, editors, *ZUM'98: The Z Formal Specification Notation*, LNCS 1493, pages 96–115. Springer-Verlag, 1998.
11. T. Santen. Isomorphisms – a link between the shallow and the deep. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics*, LNCS 1690, pages 37–54. Springer-Verlag, 1999.
12. T. Santen. *A Mechanized Logical Model of Z and Object-Oriented Specification*. Shaker-Verlag, 2000. Dissertation, Fachbereich Informatik, Technische Universität Berlin, (1999).
13. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.
14. 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.
15. H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C.B. Jones, and P. Lucas, editors, *Formal Methods Europe (FME 97)*, volume 1313 of *Lecture Notes in Computer Science*, pages 318–337. Springer-Verlag, 1997.