Unifying Theories of Objects

Michael Anthony Smith[1,2] and Jeremy Gibbons[1]

[1] Oxford University, UK.
[2] Systems Assurance Group, QinetiQ, UK.
Michael.Smith@kellogg.ox.ac.uk
Jeremy.Gibbons@comlab.ox.ac.uk

Abstract. We present an approach to modelling Abadi–Cardelli-style object calculi as Unifying Theories of Programming (UTP) *designs*. Here we provide a core object calculus with an operational *small-step evaluation rule* semantics, and a corresponding UTP model with a denotational *relational predicate* semantics. For clarity, the UTP model is defined in terms of an operand stack, which is used to store the results of subprograms. Models of a less operational nature are briefly discussed. The consistency of the UTP model is demonstrated by a structural induction proof over the operations of the core object calculus. Overall, our UTP model is intended to provide facilities for encoding both object-based and class-based languages.

1 Introduction

Hoare and He's Unifying Theories of Programming (UTP) [6] can be used to formally define how results produced in one formal model can be translated as assumptions to another formal model. Essentially, programs are considered to be predicates that relate the values of their observable input and output variables (their alphabet). For example, the increment program $x := x + 1$ is typically defined by the relational predicate $x' = x + 1$, where predicate variables x and x' denote the input and output values of the program variable x. In general, the alphabet of a program P is denoted by αP; it is the disjoint union of P's input and output sets ($in\alpha P$ and $out\alpha P$ respectively), which in the case of the example is the set $\{x, x'\}$.

This basic relational model has been specialised to reflect the semantics of various programming paradigms and languages, such as: imperative programs without subroutines; reactive systems for simple message-based concurrency; and class-based object orientation [4, 2, 11]. Here, we consider a variant of the Abadi–Cardelli-style untyped object calculus (ς-calculus) [1]. We hope that, by providing an encoding of the ς-calculus in the UTP, we can provide facilities for modelling and relating a wide range of object-oriented (OO) languages. In particular, we take an object-based rather than class-based approach, following Abadi and Cardelli, providing both object values and references through the use of a heap. We do not discuss delegation, inheritance, or other mechanisms for sharing methods, since these can be implemented in terms of our primitives; nor

again, following Abadi and Cardelli, do we support evolution of object interfaces, although it would be trivial to remove this restriction.

In the remainder of this section we introduce the notion of a UTP design and some notation that is too cumbersome to introduce when it is required. The paper then presents our variant of the object calculus, its stack-based UTP model, the consistency of this model, and some concluding remarks.

1.1 Designs in Unifying Theories of Programming

A *UTP design* specialises the general model of programming within the UTP by adding the notion of program termination. It introduces two special model variables Π_{OK} and $\Pi_{\mathrm{OK}}{}'$ to denote when a program is ready to start and when a program has terminated, respectively.

Definition 1 (UTP design). *A design predicate $p \vdash P$ states that the program represented by the relational predicate P will successfully terminate whenever it has been started in an input state that satisfies input assumption (precondition) predicate p.*

$$p \vdash P \;\widehat{=}\; (\Pi_{\mathrm{OK}} \wedge p) \Rightarrow (\Pi_{\mathrm{OK}}{}' \wedge P)$$
$$\alpha(p \vdash P) \;\widehat{=}\; \alpha P \cup \{\Pi_{\mathrm{OK}}, \Pi_{\mathrm{OK}}{}'\}$$

where $\alpha p \subseteq in\alpha P$ *and* $\Pi_{\mathrm{OK}}, \Pi_{\mathrm{OK}}{}' \notin \alpha P$.

This definition of a UTP design is taken from [4]. It updates the original definition in [6] by ensuring that "the assumption is a precondition, containing only undashed [input] variables [... which ...] corresponds exactly to the third healthiness condition" of [6, page 84]. The remainder of the UTP design language is now summarised as follows:

skip	to represent the program whose outputs are unchanged;
chaos	to represent the program whose outputs are arbitrary;
miracle	to represent the program whose outputs are always correct;
var x	to introduce variable x (i.e. add it to the alphabet);
end x	to complete variable x (i.e. remove it from the alphabet);
$x^k_{i=1} := e^k_{i=1}$	to assign the evaluation of each e_i to x_i simultaneously;
$P \fatsemi Q$	to compose subprograms P and Q sequentially;
$P \lhd b \rhd Q$	to execute P when b is true, and Q otherwise;
$P \sqcap Q$	to choose non-deterministically between P and Q;
$b * P$	to iterate subprogram P whilst b is true;
$\mu\, z \bullet P[z]$	to establish the weakest fixed point of recursive program P.

where the meta variables

b, e	denote a boolean value and a general expression respectively;
P, Q	denote UTP relational predicates (e.g. designs);
x, y	denote variables (in this case program variables);
z	denotes the special fixed point variable;
$P[z]$	denotes a relational predicate P that may contain the variable z.

The miracle program is not implementable; it and chaos are useful for reasoning about program, as they are the bottom and top of the design refinement lattice respectively. For further information on UTP designs refer to [4, 6, 12].

1.2 Design frame and compilation notation

We now introduce two utility notations. First, design frames are introduced to simplify the definitions of relational predicates that affect only some of the variables within an alphabet.

Definition 2 (Design Frame). *Let V be a set of program variables. A design with frame V has the form $V : (p \vdash P)$, denoting the predicate $p \vdash P \wedge w' = w$, where the vector w contains the logical and program variables within the input alphabet of $p \vdash P$ except those in the set V (i.e. $\{x \mid x \in w\} = in\alpha(p \vdash P) \setminus V$).*

Second, the compilation of a source language term t with subterms $t_1, \ldots, t_k$ is denoted by $(\!|\, t\{t_{i=1}^{k}\} \,|\!)^{\mathrm{M}}$, where M represents an optional compilation mode. We use this notation for compiling the heap extended core ς-calculus ($\mathcal{O}_{\mathrm{CH}}$-calculus) into the UTP operand stack model ($\mathcal{U}_{\mathrm{SH}}$) below.

1.3 Operational reduction rule notation

The semantics of object calculus operations is defined in terms of a collection of small-step evaluation-rules. These rules are similar to those in [10] except that they include a notion of a general context (Γ), which is essentially used to denote those *specific* contexts that are irrelevant to a given rule. A specific context, such as the heap in the $\mathcal{O}_{\mathrm{CH}}$-calculus (Section 2.3), can be selected and set as follows:

$\{\textsc{heap} \mapsto H\}$	Let H denote the HEAP context.
$\{\textsc{heap} \mapsto H\}$	Set the HEAP context to the value of H.

The objective of the evaluation rules is to provide the circumstances under which a term t in a context Γ can evaluate in one step to a term t' in context Γ'; such one-step evaluations are denoted by $\Gamma \bullet t \longrightarrow \Gamma' \bullet t'$, where $\_ \bullet \_$ denotes the context-term pair binder and $\_ \longrightarrow \_$ denotes an individual evaluation step. It is now possible to define the rule representation as follows:

$$\frac{\langle \mathit{condition}_1 \rangle \quad \ldots \quad \langle \mathit{condition}_n \rangle}{\langle \mathit{concluded\ term\ evaluation\ step} \rangle}\ \textsc{RuleName} \qquad \boxed{\langle \mathit{optional\ side\ condition} \rangle}$$

where $\mathit{condition}_i$ may be either a logical constraint or an evaluation step.

2 The Object Calculi

The $\mathcal{O}_{\mathrm{CH}}$-calculus we consider in this paper is an extension of the ς-calculus presented in Chapter 6 of Abadi and Cardelli's book on objects [1]. We now provide a brief summary of the ς-calculus (Section 2.1), which is followed by our arithmetic and heap extensions (Sections 2.2 and 2.3 respectively).

2.1 Abadi–Cardelli untyped object calculus

The Abadi–Cardelli ς-calculus introduces the notion of an object, as a collection of labelled methods that can be updated and selected as follows.

$[\begin{smallmatrix}k\\i=1\end{smallmatrix}\ l_i = m_i]$	denotes an object value – a partial map from labels to methods, where method m_i is identified by label l_i.
$\varsigma(x)\ e$	denotes a method whose body is defined by the expression e, which may itself contain one or more instances of the self variable (identifier) x.
$o.l \Leftarrow m$	denotes a method update operation, which generates a new object by taking a copy of the object o and replacing the method identified by label l with the method m.
$o.l$	denotes a method selection operation, which evaluates the body of the method with label l in object o, after each instance of the method's *self* variable has been replaced by a copy of the invoking object o.

where the meta-variables

o, l	denote an object and a label value respectively;
m	denotes a method;
e, x	denote an expression and the self identifier (variable/expression).

Note that a ς-calculus expression is either an object value, variable identifier, or an application of the method selection or update operators. In particular, neither a label nor a method is considered to be a value-expression.

Method update The base case for the method update operations can now be defined by the following small-step evaluation rule.

$$\frac{l \Notin o}{\Gamma \bullet o.l \Leftarrow m \longrightarrow \Gamma \bullet \ulcorner o \oplus \{l \mapsto m\} \urcorner} \textsc{UpdM}$$

where

$l \Notin o$	label l is in the domain of object o.
$o_1 \oplus o_2$	object map o_1 is overridden by object map o_2.
$\ulcorner e \urcorner$	the meta-expression e.

There is one other small-step evaluation rule, which ensures that the evaluable argument (i.e. expression argument) of the method update operation is evaluated prior to the operation being applied.

$$\frac{\Gamma \bullet e \longrightarrow \Gamma' \bullet e'}{\Gamma \bullet e.l \Leftarrow m \longrightarrow \Gamma' \bullet e'.l \Leftarrow m} \textsc{UpdM-1}$$

Method invocation (or selection) The base case for the method invocation operations of the ς-calculus can be defined by the following rule.

$$\frac{l \notin o}{\Gamma \bullet o.l \longrightarrow \Gamma \bullet \ulcorner b\{\!\{x \leftarrowtail o\}\!\} \urcorner} \textsc{InvM} \qquad \boxed{\begin{array}{l} m \mathrel{\widehat{=}} o(l) \\ \varsigma(x)\ b \doteq m \end{array}}$$

where

$o(l)$	is the method of object o with label l.
$m \mathrel{\widehat{=}} e$	defines variable m to be the evaluation of meta-expression e.
$\varsigma(x)\ b \doteq m$	binds x and b to the self-variable and body of method m.
$b\{\!\{x \leftarrowtail o\}\!\}$	the substitution of object o for free variable x in term b.

The rule for evaluating an evaluable argument before the base rule can be applied is defined in precisely the same manner as that of the method update operation.

2.2 Core object calculus ($\mathcal{O}_{\text{C}}$-calculus)

The core ς-calculus ($\mathcal{O}_{\text{C}}$-calculus) introduces field assignment, integer literals, and some basic arithmetic operators.

$o.l := e$	denotes the operation that evaluates the expression e to a value v, then applies the method update operation $o.l \Leftarrow \varsigma(\_)\ v$.
i	denotes a (literal) integer value.
$\boxdot(e_{i=1}^{k})$	denotes a k-ary operation on literal values (e.g. binary "+").

The following rules specify the base cases for both of the above operations.

$$\frac{}{\Gamma \bullet \boxdot(v_{i=1}^{k}) \longrightarrow \Gamma \bullet \ulcorner \boxdot(v_{i=1}^{k}) \urcorner} \textsc{LitOp} \qquad \boxed{v_{i=1}^{k} \in \mathrm{dom}(\boxdot)}$$

$$\frac{l \notin o}{\Gamma \bullet o.l := v \longrightarrow \Gamma \bullet \ulcorner o \oplus \{l \mapsto \varsigma(\_)\ v\} \urcorner} \textsc{FldA}$$

The other cases for these operations ensure that their evaluable arguments are processed in a left to right order.

$$\frac{\Gamma \bullet e_n \longrightarrow \Gamma' \bullet e'_n}{\Gamma \bullet \boxdot(v_{i=1}^{n-1}, e_{i=n}^{k}) \longrightarrow \Gamma' \bullet \boxdot(v_{i=1}^{n-1}, e'_n, e_{i=n+1}^{k})} \textsc{LitOp-n}$$

$$\frac{\Gamma \bullet e_1 \longrightarrow \Gamma' \bullet e'_1}{\Gamma \bullet e_1.l := e_2 \longrightarrow \Gamma' \bullet e'_1.l := e_2} \textsc{FldA-1}$$

$$\frac{\Gamma \bullet e \longrightarrow \Gamma' \bullet e'}{\Gamma \bullet o.l := e \longrightarrow \Gamma' \bullet o.l := e'} \textsc{FldA-2}$$

2.3 Heap-extended object calculus ($\mathcal{O}_{\text{CH}}$-calculus)

The $\mathcal{O}_{\text{CH}}$-calculus introduces a copy-based heap storage model [10], where the heap is a partial map from abstract locations to values. Here the contents of an abstract location can be read (dereferenced) or updated (assigned) via atomic operations that take copies of the source values. The new constants and operators introduced by this model now follow.

ℓ_i	denotes an abstract location on the heap and an allocated reference value.
null	denotes the null (i.e. unallocated) reference value.
$\mathsf{¿}$	denotes the unset value.
fresh	denotes the operation that results in the location of a newly allocated heap entry, whose contents are unset.
$*r$	denotes the operation that takes a copy of the contents in heap location r.
$r *= v$	denotes the assignment, by copy, of value v to location r.

where r, v, and i are reference, general, and integer values respectively.

The following rules specify the base cases for the fresh, dereference, and assignment (reference update) operators. The other cases for these operators are defined to follow the usual left to right evaluation order, in a similar manner to those of the $\mathcal{O}_{\text{C}}$-calculus operators.

$$\frac{}{\Gamma\{\text{HEAP} \mapsto H\} \bullet \mathsf{fresh} \longrightarrow \Gamma\{\text{HEAP} \leftarrowtail H'\} \bullet r}\ \textsc{Fresh} \qquad \boxed{\begin{array}{l} r \mathrel{\widehat{=}} \mathsf{fresh}_{\text{LOC}}(\operatorname{dom} H) \\ H' \mathrel{\widehat{=}} H \oplus \{r \mapsto \mathsf{¿}\} \end{array}}$$

$$\frac{r \notin H}{\Gamma\{\text{HEAP} \mapsto H\} \bullet *r \longrightarrow \Gamma \bullet H(r)}\ \textsc{DeRef}$$

$$\frac{r \notin H}{\Gamma\{\text{HEAP} \mapsto H\} \bullet r *= v \longrightarrow \Gamma\{\text{HEAP} \leftarrowtail H \oplus \{r \mapsto v\}\} \bullet r}\ \textsc{UpdL}$$

where $\mathsf{fresh}_{\text{LOC}}$ is a meta-function that takes a set of location values and returns a location that is not within this set.

3 The Operand Stack Model ($\mathcal{U}_{\text{SH}}$) of the $\mathcal{O}_{\text{CH}}$-calculus

The UTP operand stack model ($\mathcal{U}_{\text{SH}}$) extends the notion of a UTP design with an operand stack for storing intermediate results, and a heap map for storing dynamically allocated values in abstract heap locations. Formally this stack and heap are denoted by the UTP context variables Π_{STK}, Π_{HEAP}, $\Pi_{\text{STK}}{}'$, and $\Pi_{\text{HEAP}}{}'$, which represent the input and output states (values) of the operand stack and heap storage respectively.

The contents of the stack are the semantic entities that represent the operands of the $\mathcal{O}_{\text{CH}}$-calculus operations, i.e. the integers, objects, labels, methods, and

(heap) locations. The idea is that following the execution of a subprogram the top value on the stack represents its result.

The heap storage (map) context is essentially taken from the $\mathcal{O}_{\text{CH}}$-calculus, the main differences being in the changes to its name and the precise representation of its contents (values). In particular, the restriction that the heap can only contain values is kept; thus unlike the stack a heap cannot contain labels or methods. An alternative *trace-based* approach to modelling the $\mathcal{O}_{\text{CH}}$-calculus is the subject of current work as discussed in Section 5.2.

3.1 Literal value programs

The simplest object calculus program is represented by a literal value, which is also the final result value of the program. Therefore, the compilation of such an $\mathcal{O}_{\text{CH}}$-calculus program must result in the $\mathcal{U}_{\text{SH}}$ design $(\mathcal{E}\ sv)$ that pushes a single *stack-value* (sv) – i.e. a label, a method or a value – onto the operand stack.

$$\mathcal{E}\ sv \ \widehat{=}\ \{\Pi_{\text{STK}}\} : (\ \mathsf{true} \vdash \Pi_{\text{STK}}{}' = \langle sv \rangle \frown \Pi_{\text{STK}}\ )$$

The compilation of a literal value lv to the $\mathcal{U}_{\text{SH}}$ is now defined in stages by the following two compilation rules. Here, the first rule compiles the value to a UTP program, whereas the second rule compiles the value to a UTP expression.

$$\langle\!\langle lv \rangle\!\rangle \ \widehat{=}\ \mathcal{E}\langle\!\langle lv \rangle\!\rangle^{\text{E}} \qquad\qquad \langle\!\langle lv \rangle\!\rangle^{\text{E}} \ \widehat{=}\ lv$$

Note that these compilation rules produce $\mathcal{U}_{\text{SH}}$ texts, which can then be converted into a $\mathcal{U}_{\text{SH}}$ program by applying the semantic meaning brackets as follows.

$$[\![\ t\ ]\!] \ \widehat{=}\ t$$

where t is a *valid* output of the program compilation process. This amounts to being in a subset of the available $\mathcal{U}_{\text{SH}}$ operations, whose syntactic forms are amenable to the structural definition of functions involving scope of variables. For example, the free-variable substitution function in Section 3.5 is defined in terms of such a structural definition.

3.2 Modelling object values and method definitions

In the $\mathcal{U}_{\text{SH}}$ an object value is defined as a map from labels to methods. This is denoted by $\{_{i=1}^{k}\ l_i \mapsto m_i\}$, where k represents the number of object methods m_i with distinct labels l_i. The compilation of an object value is similar to that of literal values.

$$\langle\!\langle [_{i=1}^{k} l_i = m_i] \rangle\!\rangle \ \widehat{=}\ \mathcal{E}\langle\!\langle [_{i=1}^{k} l_i = m_i] \rangle\!\rangle^{\text{E}}$$
$$\langle\!\langle [_{i=1}^{k} l_i = m_i] \rangle\!\rangle^{\text{E}} \ \widehat{=}\ \{_{i=1}^{k}\ \langle\!\langle l_i \rangle\!\rangle^{\text{E}} \mapsto \langle\!\langle m_i \rangle\!\rangle^{\text{E}}\}$$

A method is defined as a pair of compiled program texts that represent the method's *self* variable and body. It is denoted by $(\!|\ x, P\ |\!)$, where the scope

of the *self* variable x is the program text P. Methods cannot occur as top level programs as they are not considered to be values, thus they only have an evaluation moded compilation scheme.

$$\langle\!\langle \varsigma(x)\, e \rangle\!\rangle^{\mathrm{E}} \;\widehat{=}\; (\!|\, \langle\!\langle x \rangle\!\rangle^{\mathrm{E}}, \langle\!\langle e \rangle\!\rangle \,|\!)$$

where a variable is represented by itself in both the program and declaration (literal evaluation) contexts.

$$\langle\!\langle x \rangle\!\rangle \;\widehat{=}\; x \qquad\qquad \langle\!\langle x \rangle\!\rangle^{\mathrm{E}} \;\widehat{=}\; x$$

Note that a variable by itself is not a valid program, but it may be the entire contents of a method's body (i.e. a compiled program text). Such variables are substituted by their values, prior to the program text being extracted to a $\mathcal{U}_{\text{SH}}$ subprogram. Details of the program text variable-substitution and extraction processes are presented in the discussion of method invocation (Section 3.5).

3.3 Command expressions

In the $\mathcal{O}_{\text{CH}}$-calculus, almost all the programming operations are expressions. Such expressions are converted into UTP commands by evaluating each of their arguments, whose results are stored in the operand stack (Π_{STK}), and then applying an appropriate stack transformation command.

The $\mathcal{U}_{\text{SH}}$ stack transformation operation $\mathsf{trans}(f, k)$ takes a k-parameter function f, which defines the operation being modelled, and constructs a UTP program that applies this function to the top k contents of the operand stack. Care must be taken to ensure that the parameters are in the order that they are going to appear on the operand stack, as this may not be the same as the left-to-right declaration order.

Given that the meta-variables $x_1, \ldots, x_k$ represent the arguments for function f, then the updated stack can be modelled by $\langle f(x_1, \ldots, x_k)\rangle \frown (\mathsf{tail}^k\ \Pi_{\text{STK}})$, assuming that: it has started ($\Pi_{\text{OK}} = \mathsf{true}$); there are sufficient arguments ($k \leq \#\Pi_{\text{STK}}$); and these arguments are in the domain of the function being modelled ($(x_1, \ldots, x_k) \Elzinf f$).

$$\begin{array}{l}
\mathsf{trans}(f, k) \;\widehat{=} \\
\quad \exists\, x_{i=1}^{k} \bullet (\\
\qquad (k \leq \#\Pi_{\text{STK}}) \;\wedge\; (\forall_{i=1}^{k}\, x_i = \mathsf{head}(\mathsf{tail}^{k-i}\ \Pi_{\text{STK}})) \;\wedge\; (x_{i=1}^{k}) \Elzinf f \\
\qquad \vdash \\
\qquad \Pi_{\text{STK}}{}' = \langle f(x_{i=1}^{k})\rangle \frown (\mathsf{tail}^k\ \Pi_{\text{STK}}) \\
\quad)
\end{array}$$

Having defined the transformation function, the next step is to provide a $\mathcal{U}_{\text{SH}}$ operation that evaluates the arguments for this function and then applies this function to these arguments. Note that these arguments range over acceptable *stack values*, so may include labels and methods, which are considered to be

values for the purpose of the argument evaluation. Therefore, the arguments consist of stack values and general expressions $(se_1, \ldots, se_k)$.

$$\mathsf{cmdExp}(f, (se_{i=1}^{k})) \quad \widehat{=} \quad (\fatsemi_{i=1}^{k} \langle\!\langle se_i \rangle\!\rangle) \fatsemi \mathsf{trans}(f, k)$$

Example 1. The $\mathcal{O}_{\text{CH}}$-calculus addition operation can be modelled in terms of the cmdExp operation as follows:

$$\begin{aligned} \langle\!\langle e_1 + e_2 \rangle\!\rangle \quad &\widehat{=} \quad \mathsf{cmdExp}((\_ + \_), (\langle\!\langle e_1 \rangle\!\rangle, \langle\!\langle e_2 \rangle\!\rangle)) \\ &= \quad \langle\!\langle e_1 \rangle\!\rangle \fatsemi \langle\!\langle e_2 \rangle\!\rangle \fatsemi \mathsf{trans}((\_ + \_), 2) \end{aligned}$$

3.4 Method Updates and Field Assignments

Method update in the $\mathcal{O}_{\text{CH}}$-calculus is compiled in two parts: first, the terms representing the arguments are compiled; and second, they are combined by an appropriate method update transformation function.

$$\langle\!\langle e_1.e_2 \Leftarrow m \rangle\!\rangle \quad \widehat{=} \quad \mathsf{cmdExp}(\mathit{methUpd}, (\langle\!\langle e_1 \rangle\!\rangle, \langle\!\langle e_2 \rangle\!\rangle, \mathcal{E}\langle\!\langle m \rangle\!\rangle))$$

where:

$$\begin{aligned} \mathit{methUpd} = \{ \; & (o, l, m) \mapsto o \oplus \{l \mapsto m\} \mid \\ & (o, l, m) \in \mathit{Object} \times \mathit{Label} \times \mathit{Method} \wedge l \in o \\ \} \end{aligned}$$

A field assignment is compiled in a similar manner.

$$\langle\!\langle e_1.e_2 := e_3 \rangle\!\rangle \quad \widehat{=} \quad \mathsf{cmdExp}(\mathit{fldUpd}, (\langle\!\langle e_1 \rangle\!\rangle, \langle\!\langle e_2 \rangle\!\rangle, \langle\!\langle e_3 \rangle\!\rangle))$$

where:

$$\begin{aligned} \mathit{fldUpd} = \{ \; & (o, l, v) \mapsto o \oplus \{l \mapsto (\!|\, \_, v \,|\!)\} \mid \\ & (o, l, v) \in \mathit{Object} \times \mathit{Label} \times \mathit{Value} \wedge l \in o \\ \} \end{aligned}$$

3.5 Method invocation

Method invocation in the $\mathcal{O}_{\text{CH}}$-calculus is compiled in two parts. First an object-member pair is constructed from the invocation arguments: a pair of expressions (e_1 and e_2) representing an object (o) and a label (l). This is achieved by retrieving the method with label l from object o. The second part performs the actual method invocation, using a generic method call command (call). It executes the body of the method where the method's self variable has been instantiated with the calling object's value.

$$\langle\!\langle e_1.e_2 \rangle\!\rangle \quad \widehat{=} \quad \mathsf{cmdExp}(\mathit{omPair}, (\langle\!\langle e_1 \rangle\!\rangle, \langle\!\langle e_2 \rangle\!\rangle)) \fatsemi \mathsf{call}$$

where:

$$omPair = \{(o, l) \mapsto (o, o(l)) \mid (o, l) \in Object \times Label\}$$

Before formally defining the generic call command, it is worth presenting two helper functions, for method extraction and self variable substitution. Both these functions are defined by cases, where the first case that matches is taken. The $\mathsf{ext}(t, z)$ constructs a program that can be represented by program text t once the fixed point variable z has been instantiated.

$$\begin{array}{lcl}
\mathsf{ext}(t, z) & \widehat{=} & [\![\ extInner(t, z)\ ]\!] \\
extInner(\mathsf{call}, z) & \widehat{=} & z \\
extInner(t\{\!\{_{i=1}^{k}\ t_i\}\!\}, z) & \widehat{=} & t\{\!\{_{i=1}^{k}\ extInner(t_i, z)\}\!\} \\
extInner(t, z) & \widehat{=} & t
\end{array}$$

The following $\mathcal{U}_{\text{SH}}$ substitution function ($t\{\!|x \rightarrowtail sv|\!\}$) performs the same role as that of its $\mathcal{O}_{\text{CH}}$-calculus counterpart, in that it replaces all free occurrences of the program variable x with the stack-value sv in the program text t.

$$\begin{array}{lcl}
x\{\!|x \rightarrowtail sv|\!\} & \widehat{=} & sv \\
(\!|\ x, t\ |\!)\{\!|x \rightarrowtail sv|\!\} & \widehat{=} & (\!|\ x, t\ |\!) \\
t\{\!\{_{i=1}^{k}\ t_i\}\!\}\{\!|x \rightarrowtail sv|\!\} & \widehat{=} & t\{\!\{_{i=1}^{k}\ \ t_i\{\!|x \rightarrowtail sv|\!\}\ \ \}\!\} \\
t\{\!|x \rightarrowtail sv|\!\} & \widehat{=} & t
\end{array}$$

Note that this definition assumes that both the variable x and the stack-value sv have a textual representation. Both variables and literal values are their own texts. This leaves methods and object stack values. As a method is modelled by its text and an object is a partial map from labels to method texts, it is possible to define a straightforward function (text) for taking these values to an equivalent program text.

We are now in a position to define the call command. It is defined as the least fixed point of the apply function, which substitutes the self object o in the method text t for its self variable x.

$$\mathsf{call} \ \widehat{=} \ \mu\, z \bullet \mathsf{apply}(z)$$

where

$$\begin{array}{lcl}
\mathsf{apply}(z) & \widehat{=} & (\ \exists\, o, x, t \mid (o, (\!|\ x, t\ |\!)) = \mathsf{head}\ \Pi_{\text{STK}} \bullet \\
& & \quad \mathsf{pop} \fatsemi \mathsf{ext}(t\{\!|x \rightarrowtail \mathsf{text}(o)|\!\}, z) \\
& &) \\
& & \lhd \#\Pi_{\text{STK}} > 0 \wedge (\mathsf{head}\ \Pi_{\text{STK}}) \in Object \times Method \rhd \\
& & \mathsf{chaos}
\end{array}$$

3.6 Modelling the Heap Operations

The $\mathcal{U}_{\text{SH}}$ model of a heap mirrors that of the $\mathcal{O}_{\text{CH}}$-calculus, where the location, unset and null values are shared semantic entities between the models.

Fresh Operator The $\mathcal{O}_{\text{CH}}$-calculus fresh operation is compiled to its $\mathcal{U}_{\text{SH}}$ mirror.

$$\langle\!\langle \mathsf{fresh} \rangle\!\rangle \mathrel{\widehat{=}} \mathsf{fresh}$$

The fresh command creates a new location on the heap, which is initialised to the explicit unset value; it then pushes the value of this new location onto the operand stack.

$$\begin{aligned} \mathsf{fresh} \mathrel{\widehat{=}} \; & \exists\, r \mid r = \mathsf{fresh}_{\text{LOC}}(\mathrm{dom}\, \Pi_{\text{HEAP}}) \bullet \\ & \Pi_{\text{STK}}, \Pi_{\text{HEAP}} := \langle r \rangle \frown \Pi_{\text{STK}}, \Pi_{\text{HEAP}} \oplus \{r \mapsto \text{¿}\} \end{aligned}$$

Note that this operation deliberately uses the same fresh location generation function as in the $\mathcal{O}_{\text{CH}}$-calculus, as it simplifies the consistency proof between the models. Without this we would have to have a notion of heap equivalence.

Dereference Operator The dereference $\mathcal{O}_{\text{CH}}$-calculus operation is compiled by evaluating the expression representing the heap location, then applying the $\mathcal{U}_{\text{SH}}$'s command for dereferencing the current result.

$$\langle\!\langle *e \rangle\!\rangle \mathrel{\widehat{=}} \langle\!\langle e \rangle\!\rangle \fatsemi \mathsf{deref}$$

The heap dereference command (deref), takes the heap location on the top of the stack and replaces it with a copy of the associated heap value.

$$\begin{aligned} \mathsf{deref} \mathrel{\widehat{=}} \; & \{\Pi_{\text{STK}}\} : (\\ & \quad \#\Pi_{\text{STK}} > 0 \land (\mathsf{head}\, \Pi_{\text{STK}}) \notin \Pi_{\text{HEAP}} \\ & \quad \vdash \\ & \quad \Pi_{\text{STK}}' = \langle \Pi_{\text{HEAP}}(\mathsf{head}\, \Pi_{\text{STK}}) \rangle \frown (\mathsf{tail}\, \Pi_{\text{STK}}) \\ &) \end{aligned}$$

Heap Update Operator The heap update $\mathcal{O}_{\text{CH}}$-calculus operation is compiled by evaluating its arguments in a left to right order, storing their results into a single location-value pair, and then applying the model of the heap update operation.

$$\langle\!\langle e_1 *= e_2 \rangle\!\rangle \mathrel{\widehat{=}} \mathsf{cmdExp}(\mathit{lvPair}, (\langle\!\langle e_1 \rangle\!\rangle, \langle\!\langle e_2 \rangle\!\rangle)) \fatsemi \mathsf{update}$$

where *lvPair* is a variant of the identity function whose domain elements are defined to be the location-value pairs.

$$\mathit{lvPair} = \{(r, v) \mapsto (r, v) \mid (r, v) \in \mathit{Location} \times \mathit{Value}\}$$

The reason for combining the location and value into a pair, is so that it has the same form as the heap extended result-value and constant-map UTP models of the $\mathcal{O}_{\text{CH}}$-calculus. This helps to highlight the semantic, rather than syntactic, differences between the models. These alternative models are discussed briefly in Section 5.2.

The heap update command consumes the location-value pair on the top of the stack, assigns the new value to the existing heap location, and then pushes the heap location onto the stack.

$$
\begin{array}{lcl}
\mathsf{update} & \widehat{=} & (\ \exists\, r, v \mid (r, v) = \mathsf{head}\, \Pi_{\text{STK}} \bullet \\
& & \quad \Pi_{\text{STK}}, \Pi_{\text{HEAP}} := \langle r \rangle \frown (\mathsf{tail}\, \Pi_{\text{STK}}), \Pi_{\text{HEAP}} \oplus \{r \mapsto v\} \\
& & \quad \lhd\ r \notin \Pi_{\text{HEAP}}\ \rhd \\
& & \quad \mathsf{chaos} \\
& &) \\
& & \lhd\ \#\Pi_{\text{STK}} > 0 \ \ \wedge \ \ (\mathsf{head}\, \Pi_{\text{STK}}) \in \mathit{Location} \times \mathit{Value}\ \rhd \\
& & \mathsf{chaos}
\end{array}
$$

4 Consistency of the Operand Stack Model

We now demonstrate that the $\mathcal{U}_{\text{SH}}$ denotational semantics is consistent with the $\mathcal{O}_{\text{CH}}$-calculus operational semantics via a structural induction over the object calculus' terms — specifically, that the denotational semantics of an $\mathcal{O}_{\text{CH}}$-calculus operation is the same as that of its result. The commuting diagram in Figure 1 illustrates the structure of the proof that the semantic models for an object calculus operation op with k subterms are consistent, where:

se_i	is the i^{th} subexpression of the original operation.
sv_i	is the i^{th} subterm of the operation after its arguments (i.e. stack values) have been evaluated in the correct order.
$[\![t]\!]^{\text{M}}$	is the the combination of the compilation and semantic meaning functions (i.e. $[\![\ \langle\!\langle t \rangle\!\rangle^{\text{M}}\ ]\!]$), where M is the compilation mode.
Γ_i	is the i^{th} object calculus run-time context variable.
$A0$	is the assumption that the subterms can be evaluated in the correct order. Note this assumption also guarantees the consistency of the sub-term mappings (i.e. $\forall_{i=1}^{k}\ \langle\!\langle sv_i \rangle\!\rangle = \mathcal{E}\ sv_i$).
$A1$	is the assumption that the arguments are in the domain of the operation being modelled (i.e. $(sv_{i=1}^{k}) \Subset op$).
$A2$	is the assumption that the result of executing the operation with arguments $sv_{i=1}^{k}$ is the expression e.
$\rightarrow, \twoheadrightarrow$	are the one-step and multi-step $\mathcal{O}_{\text{CH}}$-calculus operations.
$\downarrow$	is a compilation and/or semantic evaluation function.
$=, \parallel$	are two different representations of the equality relation, for horizontal and vertical display contexts respectively.

The left hand square of the commuting diagram in Figure 1 is essentially the same for every operator being checked, as it mirrors the use of the induction hypothesis, that an operation's arguments (subterms) can be evaluated successfully. Therefore, in practice this aspect of the diagram is omitted, as illustrated in Figure 2.

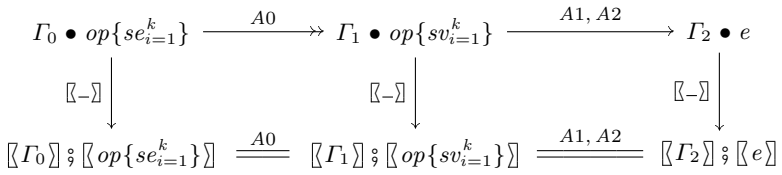

Fig. 1. Commuting diagram principle

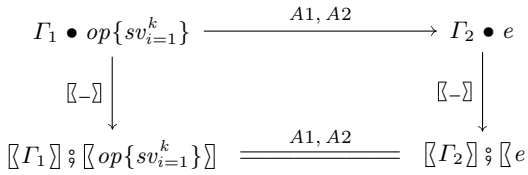

Fig. 2. Commuting diagram practice

The remainder of this section presents a representative sample of the consistency proofs; space limits preclude completeness.

4.1 Scalar value operations

The $\mathcal{O}_{\text{CH}}$-calculus provides a variety of arithmetic operations that take scalar values and return a scalar value. Further, as all of these operations are systematically translated into the $\mathcal{U}_{\text{SH}}$, it is possible to present a generic proof that these operations are consistently modelled.

The commuting diagram in Figure 3 outlines the structure of the proof that a generic infix binary operator $(\_ \odot \_)$, over scalar values in the $\mathcal{O}_{\text{CH}}$-calculus, has a consistent denotational semantics. Here we assume that:

- *A*1 The scalar values sv_1 and sv_2 are in the domain of the infix operator; i.e. $(sv_1, sv_2) \Subset (\_ \odot \_)$.
- *A*2 The scalar value sv_3 is the result of evaluating the binary operator; i.e. $sv_3 = sv_1 \odot sv_2$.

In Figure 3's commuting diagram, *L*1 denotes the first lemma (Lemma 1). It is the key step in this consistency proof, which demonstrates that a command expression has the expected semantics. Essentially, this commuting diagram forms a template for all the $\mathcal{O}_{\text{CH}}$-calculus operations that are defined as command expressions in the $\mathcal{U}_{\text{SH}}$. In particular, field assignment and method update are also covered by this proof template.

Command expression lemma The command expression lemma demonstrates that the effect of applying a command-expression command to a function f, with *pre-evaluated* arguments $sv^k_{i=1}$, is the same as the effect of applying the evaluation command to the result of the function f on its arguments. It assumes that the arguments are in the domain of the function (i.e. $(sv^k_{i=1}) \Subset f$). Note

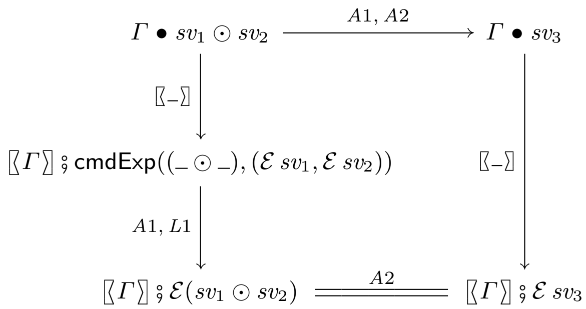

Fig. 3. Generic binary operation commuting diagram

that a *pre-evaluated* argument is an operand-stack value (i.e. a label, method definition or a value).

Lemma 1 (Command Expression Lemma).

$$(sv_{i=1}^{k}) \in f \quad \Rightarrow \quad \mathsf{cmdExp}(f, (_{i=1}^{k}\ \mathcal{E}\ sv_i)) = \mathcal{E}(f(sv_{i=1}^{k}))$$

Note that within the following proof, the left-hand-side of the initial implication is added as an assumption to the proof context.

$$\mathsf{cmdExp}(f, (_{i=1}^{k}\ \mathcal{E}\ sv_i))$$

= .... Defn. of cmdExp

$$(⨾_{i=1}^{k}\ \mathcal{E}\ sv_i) ⨾ \mathsf{trans}(f, k)$$

= .... Defn. of $\mathcal{E}$

$$(⨾_{i=1}^{k}\ (\mathsf{true} \vdash \Pi_{\mathrm{STK}}' = \langle sv_i \rangle \frown \Pi_{\mathrm{STK}})) ⨾ \mathsf{trans}(f, k)$$

= .... Defn. of ⨾ and predicate logic

$$(\mathsf{true} \vdash \Pi_{\mathrm{STK}}' = \langle_{i=1}^{k}\ sv_{k+1-i} \rangle \frown \Pi_{\mathrm{STK}}) ⨾ \mathsf{trans}(f, k)$$

= .... Defn. of trans

$$\begin{array}{l}(\mathsf{true} \vdash \Pi_{\mathrm{STK}}' = \langle_{i=1}^{k}\ sv_{k+1-i} \rangle \frown \Pi_{\mathrm{STK}}) ⨾ \\ \exists\, x_{i=1}^{k} \bullet \#\Pi_{\mathrm{STK}} \geq k\ \wedge \\ \qquad (\forall_{i=1}^{k}\ x_i = \mathsf{head}(\mathsf{tail}^{k-i}\ \Pi_{\mathrm{STK}}))\ \wedge \\ \qquad (x_{i=1}^{k}) \in f \\ \qquad \vdash \\ \qquad \Pi_{\mathrm{STK}}' = \langle f(x_{i=1}^{k}) \rangle \frown (\mathsf{tail}^{k}\ \Pi_{\mathrm{STK}})\end{array}$$

= .... Defn. of ⨾ and predicate logic

$$\begin{array}{l}\#(\langle_{i=1}^{k}\ sv_{k+1-i} \rangle \frown \Pi_{\mathrm{STK}}) \geq k\ \wedge \\ (\forall_{i=1}^{k}\ sv_i = \mathsf{head}(\mathsf{tail}^{k-i}\ \Pi_{\mathrm{STK}}))\ \wedge \\ (sv_{i=1}^{k}) \in f \\ \vdash \\ \Pi_{\mathrm{STK}}' = \langle f(sv_{i=1}^{k}) \rangle \frown \mathsf{tail}^{k}(\langle_{i=1}^{k} sv_{k+1-i} \rangle \frown \Pi_{\mathrm{STK}})\end{array}$$

= .... Predicate logic

$$(sv_{i=1}^{k}) \in f\ \vdash\ \Pi_{\mathrm{STK}}' = \langle f(sv_{i=1}^{k}) \rangle \frown \Pi_{\mathrm{STK}}$$

$$= \quad \cdots\cdots\cdots\cdots \quad A1\text{, i.e. } (sv_{i=1}^{k}) \in f$$

$$\mathsf{true} \vdash \mathit{\Pi}_{\mathrm{STK}}' = \langle f(sv_{i=1}^{k}) \rangle \frown \mathit{\Pi}_{\mathrm{STK}}$$

$$= \quad \cdots\cdots\cdots\cdots \quad \text{Defn. of } \mathcal{E}$$

$$\mathcal{E}(f(sv_{i=1}^{k}))$$

4.2 Method invocation

The commuting diagram in Figure 4 demonstrates that the $\mathcal{O}_{\mathrm{CH}}$-calculus method invocation is consistent with one unwinding of the fixed-point function (call) that defines method invocation, where:

- $A1$ The label l of object o has the method $\varsigma(x)\ e$;
- $A2$ The compilation of substitution process is defined as:

$$\langle\!\langle\, e\{\!\!\{x \hookleftarrow o\}\!\!\}\,\rangle\!\rangle \ \widehat{=}\ \mathsf{ext}(\langle\!\langle\, e\,\rangle\!\rangle\{\!\!\{x \hookleftarrow \langle\!\langle\, o\,\rangle\!\rangle^{\mathrm{E}}\}\!\!\}, \mathsf{call})$$

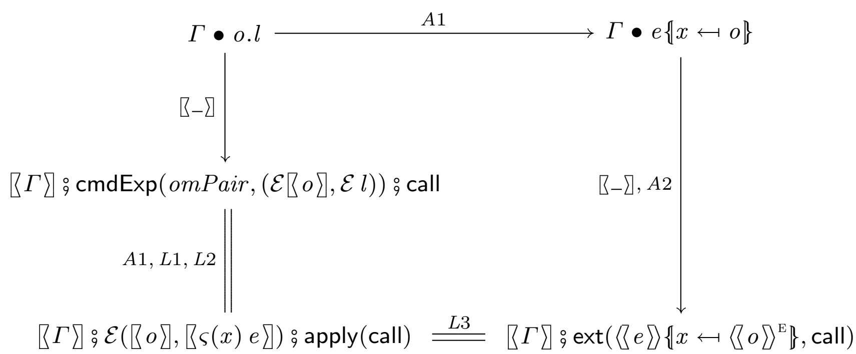

Fig. 4. Method invocation commuting diagram

The consistency diagram in Figure 4 relies on three lemmas: the command expression lemma (Lemma 1); the unwinding lemma (Lemma 2), which uses the fixed-point definition to provide a single unwinding; and the method call lemma (Lemma 3), which demonstrates that this unwinding is correct. We now state and prove the remaining two lemmas.

Recall that the method call operation is defined as the fixed point of an apply function ($\mu\, z \bullet \mathsf{apply}(z)$). The structure of this definition leads to the following unwinding lemma.

Lemma 2 (Unwinding Lemma).

$$\mathsf{call} = \mathsf{apply}(\mathsf{call})$$

Proof

$$\mathsf{call}$$

$$= \quad \cdots\cdots\cdots\cdots \quad \text{Defn. of } \mathsf{call}$$

$$(\mu\, z \bullet \mathsf{apply}(z))$$

$=$ ..... Defn of fixed point μ
$\mathsf{apply}(\mu\, z \bullet \mathsf{apply}(z))$
$=$ ..... Defn. of call
$\mathsf{apply}(\mathsf{call})$

Method call lemma Informally the method call lemma says that the effect of applying the call command is equivalent to the effect of applying one iteration of this command to itself.

Lemma 3 (Method Call Lemma).

$$\mathcal{E}([\![o]\!]^{\mathrm{E}}, [\![\varsigma(x)\ e]\!]) \fatsemi \mathsf{apply}(\mathsf{call}) \quad = \quad \mathsf{ext}(\langle\!\langle e \rangle\!\rangle \{\!| x \leftarrowtail \langle\!\langle o \rangle\!\rangle^{\mathrm{E}} |\!\}, \mathsf{call})$$

Proof

$\mathcal{E}([\![o]\!]^{\mathrm{E}}, [\![\varsigma(x)\ e]\!]) \fatsemi \mathsf{apply}(\mathsf{call})$
$=$ ..... Defn. of $[\![\varsigma(x)\ e]\!]$
$\mathcal{E}([\![o]\!]^{\mathrm{E}}, (\!|\ x, \langle\!\langle e \rangle\!\rangle\ |\!)) \fatsemi \mathsf{apply}(\mathsf{call})$
$=$ ..... Defn. of $\mathcal{E}$
$(\mathsf{true} \vdash \mathit{\Pi}_{\mathrm{STK}}{}' = \langle([\![o]\!]^{\mathrm{E}}, (\!|\ x, \langle\!\langle e \rangle\!\rangle\ |\!))\rangle \frown \mathit{\Pi}_{\mathrm{STK}}) \fatsemi$
$\mathsf{apply}(\mathsf{call})$
$=$ ..... Defn. of apply
$(\mathsf{true} \vdash \mathit{\Pi}_{\mathrm{STK}}{}' = \langle([\![o]\!]^{\mathrm{E}}, (\!|\ x, \langle\!\langle e \rangle\!\rangle\ |\!))\rangle \frown \mathit{\Pi}_{\mathrm{STK}}) \fatsemi$
$($ $($ $\exists\, o_1, x_1, t_1 \mid (o_1, (\!|\ x_1, t_1\ |\!)) = \mathsf{head}\ \mathit{\Pi}_{\mathrm{STK}} \bullet$
$\mathsf{pop} \fatsemi \mathsf{ext}((\!|\ t_1 \{\!| x_1 \leftarrowtail \mathsf{text}(o_1) |\!\}\ |\!), \mathsf{call})$
$)$
$\triangleleft\, \#\mathit{\Pi}_{\mathrm{STK}} > 1 \ \wedge\ (\mathsf{head}\ \mathit{\Pi}_{\mathrm{STK}}) \in \mathit{Object} \times \mathit{Method} \,\triangleright$
skip
$)$
$=$ ..... Defn. of pop
$(\mathsf{true} \vdash \mathit{\Pi}_{\mathrm{STK}}{}' = \langle([\![o]\!]^{\mathrm{E}}, (\!|\ x, \langle\!\langle e \rangle\!\rangle\ |\!))\rangle \frown \mathit{\Pi}_{\mathrm{STK}}) \fatsemi$
$($ $($ $\exists\, o_1, x_1, t_1 \mid (o_1, (\!|\ x_1, t_1\ |\!)) = \mathsf{head}\ \mathit{\Pi}_{\mathrm{STK}} \bullet$
$(\#\mathit{\Pi}_{\mathrm{STK}} > 1 \vdash \mathit{\Pi}_{\mathrm{STK}}{}' = \mathsf{tail}\ \mathit{\Pi}_{\mathrm{STK}}) \fatsemi$
$\mathsf{ext}((\!|\ t_1 \{\!| x_1 \leftarrowtail \mathsf{text}(o_1) |\!\}\ |\!), \mathsf{call})$
$)$
$\triangleleft\, \#\mathit{\Pi}_{\mathrm{STK}} > 1 \ \wedge\ (\mathsf{head}\ \mathit{\Pi}_{\mathrm{STK}}) \in \mathit{Object} \times \mathit{Method} \,\triangleright$
skip
$)$
$=$ ..... Defn. of $\fatsemi$
$($ $\exists\, o_1, x_1, t_1 \mid (o_1, (\!|\ x_1, t_1\ |\!)) = ([\![o]\!]^{\mathrm{E}}, (\!|\ x, \langle\!\langle e \rangle\!\rangle\ |\!)) \bullet$ and predicate logic
$(\mathsf{true} \vdash \mathit{\Pi}_{\mathrm{STK}}{}' = \mathit{\Pi}_{\mathrm{STK}}) \fatsemi$
$\mathsf{ext}((\!|\ t_1 \{\!| x_1 \leftarrowtail \mathsf{text}(o_1) |\!\}\ |\!), \mathsf{call})$
$)$
$\triangleleft\, \mathsf{true} \ \wedge\ ([\![o]\!]^{\mathrm{E}}, (\!|\ x, \langle\!\langle e \rangle\!\rangle\ |\!)) \in \mathit{Object} \times \mathit{Method} \,\triangleright$
skip
$=$ ..... Defn. of $(\_ \triangleleft \_ \triangleright \_)$
$\exists\, o_1, x_1, t_1 \mid (o_1, (\!|\ x_1, t_1\ |\!)) = ([\![o]\!]^{\mathrm{E}}, (\!|\ x, \langle\!\langle e \rangle\!\rangle\ |\!)) \bullet$ and predicate logic
$(\mathsf{true} \vdash \mathit{\Pi}_{\mathrm{STK}}{}' = \mathit{\Pi}_{\mathrm{STK}}) \fatsemi$
$\mathsf{ext}((\!|\ t_1 \{\!| x_1 \leftarrowtail \mathsf{text}(o_1) |\!\}\ |\!), \mathsf{call})$

$=$ .... One point rule and defn of skip

$\mathsf{skip} ;$
$\mathsf{ext}(\llparenthesis\ \langle\!\langle e \rangle\!\rangle \{\!| x \leftarrow \mathsf{text}([\![o]\!]^{\mathrm{E}}) |\!\}\ \rrparenthesis, \mathsf{call})$

$=$ .... skip unit of $(\_ ; \_)$ and defn. of text

$\mathsf{ext}(\llparenthesis\ \langle\!\langle e \rangle\!\rangle \{\!| x \leftarrow \langle\!\langle o \rangle\!\rangle^{\mathrm{E}} |\!\}\ \rrparenthesis, \mathsf{call})$

4.3 Fresh heap locations

The commuting diagram in Figure 5 outlines the structure of the proof that the fresh operator in the $\mathcal{O}_{\mathrm{CH}}$-calculus has a consistent denotational semantics. Here we assume that:

- $A1$ The initial $\mathcal{O}_{\mathrm{CH}}$-calculus context-heap value is H_0.
- $A2$ The expression $\mathsf{fresh}_{\mathrm{LOC}}(\mathrm{dom}\, H_0)$ evaluates to ℓ_j.
- $A3$ The context-heap value H_1 is $H_0 \oplus \{\ell_j \mapsto ¿\}$.

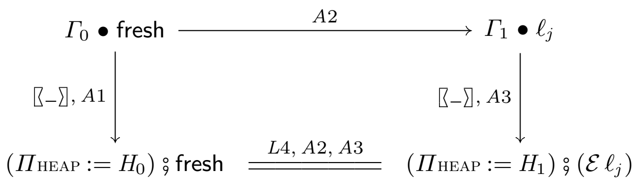

Fig. 5. Fresh operator commuting diagram

Lemma 4 (Fresh location lemma).

$$(\Pi_{\mathrm{HEAP}} := H_0) ; \mathsf{fresh} \quad = \quad (\Pi_{\mathrm{HEAP}} := H_1) ; (\mathcal{E}\, \ell_j)$$

Proof

$(\Pi_{\mathrm{HEAP}} := H_0) ; \mathsf{fresh}$

$=$ .... Defn. of fresh

$(\Pi_{\mathrm{HEAP}}' = H_0) ;$
$(\ \exists\, r \mid r = \mathsf{fresh}_{\mathrm{LOC}}(\mathrm{dom}\, \Pi_{\mathrm{HEAP}}) \bullet$
$\quad \Pi_{\mathrm{STK}}, \Pi_{\mathrm{HEAP}} := \langle r \rangle \frown \Pi_{\mathrm{STK}}, \Pi_{\mathrm{HEAP}} \oplus \{r \mapsto ¿\}$
$)$

$=$ .... Defns. of $;$ and $:=$

$\exists\, r \mid r = \mathsf{fresh}_{\mathrm{LOC}}(\mathrm{dom}\, H_0) \bullet$
$\quad \Pi_{\mathrm{STK}}, \Pi_{\mathrm{HEAP}} := \langle r \rangle \frown \Pi_{\mathrm{STK}}, H_0 \oplus \{r \mapsto ¿\}$

$=$ .... $A2$

$\exists\, r \mid r = \ell_j \bullet$
$\quad \Pi_{\mathrm{STK}}, \Pi_{\mathrm{HEAP}} := \langle r \rangle \frown \Pi_{\mathrm{STK}}, H_0 \oplus \{r \mapsto ¿\}$

$=$ .... 1-point rule & $A3$

$\Pi_{\mathrm{STK}}, \Pi_{\mathrm{HEAP}} := \langle \ell_j \rangle \frown \Pi_{\mathrm{STK}}, H_1$

$=$ .... Defns. of $;$ and $\mathcal{E}$

$(\Pi_{\mathrm{HEAP}} := H_1) ; (\mathcal{E}\, \ell_j)$

5 Conclusions and Related Work

In this paper we have provided a UTP encoding of an Abadi–Cardelli-style ς-calculus with an explicit heap model, along with a proof of its consistency. It is straightforward to add several other features, such as direct support for eagerly evaluated untyped lambda calculus (λ-calculus) functions, and for treating labels as values. In the former case, this amounts to relaxing the restriction on the definition of the $\mathcal{U}_{\text{SH}}$'s call operator, to accept any value-method pair rather than an object-method pair. The latter case amounts to treating labels as values, and adding operations for conditional execution and for checking whether an object contains a method with a given label.

5.1 Related work

Hoare and He's UTP [6, 12] provides a rich model of programs as relational-predicates. Abadi and Cardelli's ς-calculi [1] provides an alternative model of programs as objects. Our contribution is to model the Abadi–Cardelli notion of an object in the UTP, which provides: a simple untyped object calculus with a relational-predicate denotational semantics; and the UTP with an object-based model of object-orientation. Further, as the UTP already has several models of concurrency, this encoding provides the potential for adding one (or more) of these concurrency models to the ς-calculus.

This is not the first time object-oriented ideas have been added to (or modelled in) the UTP. In particular, there have been several works that model class-based object-orientation, such as [4, 2, 11]. These differ fundamentally from our approach, as each object is considered to be an instance of a class, rather than a class being a special sort of object. In particular, within our approach objects need not be associated with a class. This opens the possibility of considering prototype-based languages, such as Self.

Within the more general field of predicative programming [5], another notion of object-orientation has been modelled [9]. It defines objects as a combination of their attributes and behaviour, where each attribute (field) has a unique address and the details of its behaviour (methods) are defined by its type (e.g. class). This approach is similar to that of Abadi and Cardelli's imperative ς-calculus [1], except that in this case both methods and fields are bound to objects. Further, the $\mathcal{O}_{\text{CH}}$-calculus deliberately separates the heap and object representations, to gain a measure of orthogonality between concerns. An earlier version of this predicative programming model [8] did not use addresses in the definition of an object, but was still essentially class-oriented in its outlook.

Alternative approaches to modelling references (pointers) in the UTP have been provided in [7, 3]. The former of these approaches was the inspiration for the *trace* model that is briefly discussed in Section 5.2. The latter of these approaches uses path-based equivalence classes to identify variables that share the same reference, which are referred to as *entity groups*. Preliminary results of the on-going work in this area suggest that our trace-based model is also essentially an entity group model, which ought to enable us to unify these ideas.

5.2 On-going work

The operand-stack model of the $\mathcal{O}_{\text{CH}}$-calculus is arguably too operational in nature. In order to address this issue three further models have been constructed, the *result-value*, *constant-map*, and *trace* models. For reasons of space we can only provide a brief description of these models.

Result-value model This replaces the stack with a single value that represents the result of executing an $\mathcal{O}_{\text{CH}}$-calculus model, and intermediate results are stored in temporary variables, which are introduced and completed in the usual UTP manner. The one significant complication introduced by this model is the need to manage the scope of its alphabets – specifically the requirement to hide the intermediate result variables from the *execution* of a subprogram. This follows from two observations: first, that a subprogram's execution is independent of the result – but not side-effecting heap updates – of a previous subprogram; and second, that the weakest fixed point semantics of method invocation requires the alphabets both before and after any method invocation to be the same.

Constant-map model This extends the result-value model by updating the representation of a method (and its invocation). Here a method is represented by a triple: a *self* variable; a program-text *body*; and a *map* from the free variables within that body to their values. Such values may be updated during the method invocation process, which recursively updates all free instances of a method's self variable, within both its own and its inner-method variable maps. The idea is that by the time of a method's invocation, all the free-variables within a method's body have a defined value in their associated variable map; and that this variable map is used to introduce read-only (constant) variables for the scope of the method's definition.

Trace model This takes a fundamentally different approach from that presented in this paper, in that it models variables, values, and heap locations, in terms of a directed graph that can be represented by a set of trace sets. This approach was inspired by trace-based pointers in [7] and is also similar to the entity-group work in [3]. Essentially, it came from the motivation of using the ideas presented in these UTP models on the $\mathcal{O}_{\text{CH}}$-calculus. Here, each entity group is represented by an equivalence class, which is the set of traces that defines a node of the directed graph. There are several complicating factors, not least of which is that in the $\mathcal{O}_{\text{CH}}$-calculus not all values have locations (nor should they).

Having said that the trace-based model is fundamentally different from the others, it also has some striking similarities to the constant-map model; specifically, that the layout of the graph essentially mirrors the structure of the variables and the heap of the constant-map model. With a little extra work, we can make use of the constant-map model's variable-maps to provide a named path (trace) to any location within the graph.

References

1. Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.
2. A.L.C. Cavalcanti, A.C.A. Sampaio, and J.C.P. Woodcock. Unifying classes and processes. *Software and System Modelling*, 4(3):277–296, 2005.
3. Ana Cavalcanti, Will Harwood, and Jim Woodcock. Pointers and records in the unifying theories of programming. In Steve Dunne and Bill Stoddart, editors, *First International Symposium on Unifying Theories of Programming*, volume 4010 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.
4. Jifeng He, Zhiming Liu, and Xiaoshan Li. Towards a refinement calculus for object systems. Research Report 251, UNU/IIST, P.O. Box 3058, Macau, May 2002.
5. Eric C.R. Hehner. *A Practical Theory of Programming*. Springer-Verlag, 1993. Electronic edition freely available on line from: www.cs.utoronto.ca/∼hehner/aPToP.
6. C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Computer Science. Prentice Hall, 1998.
7. C.A.R. Hoare and Jifeng He. A trace model for pointers and objects. In 13^{th} *European Conference on Object-Oriented Programming*, pages 1–17, 1999.
8. Ioannis T. Kassios. Objects as predicates. Technical report, Computer Systems Research Group, University of Toronto, 2004.
9. Ioannis T. Kassios. *A Theory of Object Oriented Refinement*. PhD thesis, University of Toronto, 2006.
10. Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
11. Thiago Santos, Ana Cavalcanti, and Augusto Sampaio. Object-orientation in UTP. In Steve Dunne and Bill Stoddart, editors, *First International Symposium on Unifying Theories of Programming*, volume 4010 of *Lecture Notes in Computer Science*, pages 20–38. Springer-Verlag, 2006.
12. J.C.P. Woodcock and A.L.C. Cavalcanti. A tutorial introduction to designs in unifying theories of programming. In *IFM 2004: Integrated Formal Methods*, volume 2999 of *Lecture Notes in Computer Science*, pages 40–66. Springer-Verlag, 2004. Invited tutorial.