Embedding an Object Calculus in the Unifying Theories of Programming

Michael Anthony Smith
Kellogg College

A thesis submitted in partial fulfilment
of the requirements for the degree of
Doctor of Philosophy in Software Engineering
at the University of Oxford,
Hilary Term 2009.

Abstract

Hoare and He's Unifying Theories of Programming (UTP) provides a rich model of programs as relational predicates. This theory is intended to provide a single framework in which any programming paradigms, languages, and features, can be modelled, compared and contrasted. The UTP already has models for several programming formalisms, such as imperative programming, higher-order programming (e.g. programing with procedures), several styles of concurrent programming (or reactive systems), class-based object-orientation, and transaction processing. We believe that the UTP ought to be able to represent all significant computer programming language formalisms, in order for it to be considered a unifying theory.

One gap in the UTP work is that of object-based object-orientation, such as that presented in Abadi and Cardelli's untyped object calculi (ς-calculi). These ς-calculi provide a prominent formalism of object-based object-oriented (OO) programs, which models programs as objects. We address this gap within this dissertation by presenting an embedding of an Abadi–Cardelli-style object calculus in the UTP. More formally, the thesis that this dissertation argues is that it is possible to provide an object-based object orientation to the UTP, with value- and reference-based objects, and a fully abstract model of references.

We have made three contributions to our area of study: first, to extend the UTP with a notion of object-based object orientation, in contrast with the existing class-based models; second, to provide an alternative model of pointers (references) for the UTP that supports both value-based compound values (e.g. objects) and references (pointers), in contrast to existing UTP models with pointers that have reference-based compound values; and third, to model an Abadi–Cardelli notion of an object in the UTP, and thus demonstrate that it can unify this style of object formalism.

Acknowledgements

I have had considerable support and encouragement whilst endeavouring to complete this part-time DPhil, since I started in September 1999. In particular I would like to thank:

- Professor Jim Woodcock (my original supervisor) and Professor Colin O'Halloran (my boss) for their support and encouragement to take on the challenge of a part-time DPhil in Software Engineering.
- Dr. Jeremy Gibbons (my current supervisor) for his patience, direction, and encouragement throughout the eight years I have been under his guidance.
- Nick Moffat my colleague – and fellow part-time DPhil student – for his willingness to provide detailed reviews of my paper submissions and discuss issues concerning my studies.
- Tom McCutcheon for his kind support and advice at a particularly low point in my DPhil. He kept my thoughts in perspective and enabled me to make the decision to go genuinely part-time at work.
- My work colleagues for their consideration and support. In particular, for allowing me to adapt my part-time working hours so that I could regularly study in blocks of one week, rather than a couple of days at a time.
- My church and bible study groups, for keeping me grounded.
- My parents (Pat and Dave Smith) for their faith in my ability and constant encouragement.
- The Oxford University Department for Continuing Education and Kellogg College staff, for their endless patience and willingness to help me through the University regulations and paper work.
- My examiners (Dr. Andrew Simpson and Dr. Andrew Butterfield) who provided many useful comments that have significantly improved the presentation of this thesis.

My DPhil fees were kindly paid for by the UK Ministry of Defence's "Defence Evaluation and Research Agency" and then by QinetiQ (following privatisation in July 2002). I would also like to thank Kellogg College for their bursary that enabled me to fund attendance and participation in three European conferences.

Contents

This page is intentionally blank

Chapter 1

Introduction

1.1 Context and Motivation

QinetiQ's Systems Assurance (SA) team uses a variety of formal method tools for reasoning about existing systems, such as the ProofPower theorem prover [Lem00] and the FDR (Failures-Divergence and Refinement) model checker [Fse05, Sca98]. The successful use of these tools on industrial scale projects has highlighted a variety of issues, such as the difficulty in translating results from one formal framework into another. One reason for this is that the notion of program correctness in one formal framework can be subtly different from the notion of correctness in another framework, as illustrated by Bolton's work on the refinement of state-based and event-based models [Bol02, BL05].

Within the SA team it was proposed that Hoare and He's Unifying Theories of Programming (UTP) [HH98] be adopted as a common underpinning framework. This solution is in principle fine for the majority of the team's current tools, as they are designed to model either imperative or functional programs, possibly within a concurrent context (e.g. multiple threads). However, the support for object-oriented (OO) programs within the UTP is currently limited to class-based object orientation [HLL02a, HLL06, CSW03, CSW05], with the exception of our own work [SG07] (and this dissertation). These class-based models of object orientation support languages with single inheritance, where subtyping is subclassing.

In the slightly wider field of predicative programming, Kassios has introduced the notion of an object-specification [Kas04], which is a generalisation of a class, rather than an object. Here both objects and classes are defined as object-specifications that meet certain requirements. This work builds upon Hehner's work on programming as "boolean expressions" [Heh93].

In contrast to the above work, our model of object orientation directly supports an object-based OO-paradigm, rather than a class-based OO-paradigm. Having said this, it is straightforward to use an object to define a class, as is discussed in Appendix B. Further, object references are handled in our model via a *heap*, which is initially represented as a map from shareable locations to values.

Aside 1.1: It would be possible to base the modelling of objects on that of processes, where an object is seen as a special sort of process. Such objects would inherit the underlying process' model of concurrency. Therefore, it is likely that a process-based model of objects would focus on one notion of concurrency, at the expense of other notions of concurrency. An alternative view is to separate the modelling of objects and concurrency. In principle, this would enable the appropriate combination of object and concurrency models to be chosen for a given problem. This is the view adopted; thus as concurrency is already well catered for within the UTP, the modelling of objects is considered.
□

The UTP framework is abstract, hence any model of objects for the UTP ought to be similarly abstract. This raises the issue of how to validate the UTP's model of objects. What is required

is a formal notion of objects that has a simple semantic model that is straightforward to understand and can be executed for the purposes of validation.

1.2 What is Object Orientation?

There are currently no generally accepted notions of some of the key aspects of object orientation, such as precisely what is meant by the terms 'object' and 'inheritance'. In general, the OO-paradigm can be thought of as describing a broad family of OO-frameworks, that provide facilities for creating, manipulating, and relating objects.

From our point of view, an *object* is an entity that has both *state* and *operations* for accessing and manipulating that state. Typically, the state is stored in *fields* (or *attributes*) and the operations are provided by *methods*. Conceptually, access to the fields is via the published methods (or interface). However, in practice, it is frequently possible to enable the state of an object to be directly visible (accessible) and updateable. The important point, from our perspective, is that an object conceptually owns both the fields and operations that can be applied to it. Further, the only way of updating its state is via invocation of one of its methods (operations); here unprotected fields are considered to have implicit 'get' and 'set' methods for reading and updating their data.

Inheritance is a more tricky issue. It is used for a variety of purposes, such as *code reuse* and *subtyping*. Meyer's taxonomy of object orientation describes ten different forms of inheritance [Mey96, Mey97]. From our point of view inheritance is not an issue that we will consider in any depth, as the inheritance mechanisms can be defined (modelled) in terms of basic objects.

The most appropriate model of objects depends on what are considered to be *true* OO-features, as illustrated by the following example.

Example 1.2: Languages such as CLOS (Common Lisp Object System) [DG87] are considered by some, but not all, to be examples of the OO-paradigm. Essentially, this is because it is an example of a multi-method language, thus has "no receiver of a message" [Bru02]. Here the methods are treated more like procedures, which may be syntactically associated with a class of objects, but are conceptually collected into a central pool. This pool is referred to by the dynamic method dispatch algorithm, which is used to determine the method that *best fits* the call, where a call essentially consists of the method's name and actual parameter types. The important point is that the method selection is not based on an individual (receiver) object, but rather on a collection of objects. Hence, standard OO-notions such as *self* (i.e. the current object) are meaningless. Further, the natural OO-concepts of encapsulation and design by contract (visible interface) are also difficult to model. But other things are easier – e.g. binary methods.
□

From our point of view, method invocations in an OO-language ought to specify the object that the method is going to be invoked on (i.e. the invocation message 'receiver'). Therefore, we do not support multi-method flavours of object orientation; Castagna [Cas97] provides a detailed presentation of the underlying theory of multi-methods.

Having discarded multi-method languages, there are essentially two remaining families of OO-language, namely class-based and object-based languages. A pure class-based language organises its objects into classes of similar objects, where a class essentially provides a template for the construction of an object. In object-based languages, new objects have to be constructed from existing objects, and modified as necessary.

Appendix A provides a more detailed examination of typical OO-concepts and briefly discusses different styles of semantic model.

1.3 Formal Models of Objects and Object Orientation

Reynold's article on user defined types and procedural data structures [Rey75] is considered by some to be the first 'important' contribution to the formalisation of OO-programming [GM94, Bru02]. Having said this, there are earlier works that contribute to this area including:

- the formalisation of type systems, such as the simply typed lambda calculus [Chu40], and polymorphic lambda calculus [Rey74];
- the defacto operational formalisation of compilers (and interpreters) for existing object oriented languages, such as Simula [BDMN79] and Smalltalk [Lew95].

Other important works in the formalisation of object-orientation include Cardelli and Wegner's on types and data abstraction [CW85], Cardelli's semantics of inheritance [Car88], and Cook's denotational semantics of inheritance [Coo89]. The last of these works is reprinted in [GM94], which collects what the editors consider to be the fifteen most important works on the formalisation of OO-programming prior to 1993; note that [Rey75] is also reprinted in this collection.

Currently there is an extensive collection of approaches to the formalisation of the OO-paradigm, from relatively intuitive operation descriptions based on abstract machines, through to more abstract mathematics such as higher-order logics and category theory. Here, the intuitive models can be used to validate the more abstract mathematical models, which are better suited for reasoning and proof.

A special sort of algebra, known as an initial algebra, is a traditional way of modelling an abstract data type [GTWW77, MT92]. Essentially each algebra defines an interface, which explicitly states what constants and operations are provided to generate values of that algebra's type (i.e. carrier set). Algebraic formalisms of objects and OO-features include: Rees' algebraic theory of software interfaces [Ree00]; and Wei's framework for studying the semantics of object-oriented programs [Wei01].

The problem with the algebraic approach is that it focuses on how to build (or construct) an entity, rather than how the entity behaves (responds to requests for information). Co-algebras address this issue, by modelling an entity's behaviour, rather than how it can be constructed. Co-algebraic formalisms include: Reichel's work on object semantics [Rei95]; Jacobs and colleagues' work on reasoning about classes and objects [Jac96, Jac98, HHJT98]; and Poll's work on the semantics of subtyping [Pol01].

Various calculi have been proposed for the modelling of the OO-paradigm. Some are adaptations of the untyped lambda calculus (λ-calculus), such as: Castagna's work on the unified foundation of OO-programming [Cas97]; Pierce's work on the typed programming languages [Pie02]; and Bruce, Pierce and Cardelli's work on comparing object encodings [BCP99]. Others are built on process calculi, notably the pi-calculus, such as: Sangiorgi and Walker's work on objects as mobile processes [SW01, Part 7]; Jones' work on the Pobl ($\pi o \beta \lambda$) language [Jon93]; and the PhD theses of Turner [Tur95], Hordhagen [Nor98], Lumpe [Lum99], and Merro [Mer00]. Then there are the various object calculi based on Abadi and Cardelli's theory of objects [AC96], such as: Gordon, Hankin and Lassen's work on imperative objects [GHL99]; Bugliesi and Crafa's work on dynamic messages [BC99]; and Clarke's work on object ownership and containment [Cla02].

Logics for modelling OO-programs and OO-specification languages include: Abadi and Leino's logic of object-oriented programs [AL98]; Lano, Bicarregui and Evans' structured axiomatic semantics for UML (Unified Modelling Language) models [LBE00]; and Smith's logic for object-Z [Smi95].

All of the above approaches have merit, although none are considered to be the definitive definition of an object or object orientation. The Abadi and Cardelli style of object calculus was chosen as the starting point of the *base-line* semantics for validating the proposed UTP model of object orientation, as discussed in Section 1.5.2.

1.4 A Brief Introduction to the UTP and the Object Calculus

Hoare and He's Unifying Theories of Programming (UTP) [HH98] 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: the predicate variables x and x' denote the input and output values of the program variable x; and the set $\{x, x'\}$ denotes its alphabet. One key observation is that predicates with the same alphabets can be partially ordered by logical implication (e.g. $x' = 3 \Rightarrow x' > 0$). Here, the predicates $x' = 3$ and $x' > 0$ can be considered to represent the implementation and specification respectively. In general, this forms an implication-based refinement ordering, where false is the strongest element in the order and true the weakest.

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 (as discussed in Section 1.1). Each of these specialisations introduce some *observational variables* to model various aspects of the program's context, such as: whether the program has started or terminated; the history of past communication events; and the inheritance relationship between class definitions. Alongside these observational variables, the specialisations also include some *healthiness conditions*, which are used to constrain a healthy program's behaviour. For example, a healthy program should not be able to alter the history of past communication events; a formalisation of this constraint is one of the healthiness conditions that applies to all the models of concurrency presented in [HH98]. The UTP is presented in more detail in Section 2.3.

Abadi and Cardelli's untyped object calculus (ς-calculus) [AC96] provides a mathematical foundation for objects that is defined in terms of a collection of reduction rules. These rules are typically used to form the basis of an operational semantics. One characteristic of the ς-calculus is the explicit representation of the *self* parameter; this differs from some class-based languages, such as Java and C^{++}, where the *self* is implicitly accessible via the this keyword. The self parameter mechanism allows several notions of self to be visible within the same scope (e.g. that of an inner method definition); it provides the calculus with some of its expressive power.

The ς-calculus has been extended, by various authors, to include several other notions, such as literal values and their operations, typing, object references, functions, and object ownership. We provide our own object calculus – the $\mathcal{O}$-calculus – which directly supports value-based objects, eagerly evaluated functions, references, literal values and their operations, conditional evaluation, local declaration blocks, and sequential composition – but not types, classes, or other class-based notions such as inheritance. The $\mathcal{O}$-calculus is presented in Section 2.2.

1.5 Thesis

The thesis that this dissertation argues is as follows: it is possible to provide an object-based object orientation to the UTP with value- and reference-based objects, and a fully abstract model of references.

1.5.1 Approach

The high level approach we take is to: identify an existing object-based framework that can be embedded into the UTP; construct a corresponding denotational model of this framework within the UTP; demonstrate that the models are consistent; and then demonstrate that the model of references is fully abstract. An Abadi–Cardelli-style of object calculus [AC96] was chosen as the existing object framework (Section 2.2). Three UTP models of this framework

are presented in Chapter 3, and shown to be consistent in Chapter 4. A model of locations is introduced in Chapter 5, which is used to define a fully abstract model of references (using a heap). This abstract heap is then used to construct a fourth model of the chosen object framework (Section 6.2).

1.5.2 Decisions taken

We now present seven key decisions taken during our studies. First, we limit our consideration of object-based object orientation to those languages in which one invokes a method on a specified 'receiver' object, rather than passing an object as the first parameter of a more general method invocation mechanism. Hence, we do not cover multi-method flavours of object orientation – such as presented by [Cas97].

Second, the untyped Abadi–Cardelli-style object calculus [AC96] was chosen as the object-based framework to embed. Essentially, this is because it provides a small framework that is computationally complete, reasonably clear, and straightforward to extend. The small framework was thought to be the best way of ensuring the feasibility of providing both a UTP encoding and a demonstration that it is consistent. The computational completeness is demonstrated by a straightforward encoding of the λ-calculus [AC96]. The clarity is important for illustrating the language-specific definitions of OO-features. This is assisted by being able to add some straightforward extensions, which can be used to remove some of the more obscure encoding such as the Abadi–Cardelli object encoding of both conditional evaluation and the natural numbers.

Third, our variant of the Abadi–Cardelli calculus directly represents both value and reference objects. Such distinctions are important for modelling the atomicity of assignment, and for being able to control which elements of a compound structure can be directly referenced. For example, it mirrors the distinction between boxed and unboxed values with the C# language.

Fourth, the operational semantics of the object calculus is presented in a deterministic small-step reduction-rule style [Pie02]. The consistency proofs between this small-step semantics and the UTP semantics is done as a structural induction over these steps.

Fifth, we present multiple UTP encodings of the object calculus. Here the idea is to start with simple encodings that mirror the nature of the object calculus, and gradually adapt them to remove unwanted features. This approach provides a mechanism for informally explaining why the UTP models are consistent with the object calculus.

Sixth, the last of the UTP models completely changes the way in which a program is represented, from a combination of local variables and a map from heap locations to values, to a trace-based graph of program state. In order to facilitate this change, a concrete model of the trace-based graph has been introduced to aid understanding and validate the modelling approach.

Seventh (and last) we consider the model of references to be fully abstract iff every modelled location is reachable from the root (program) node and every location's value is uniquely defined by its position within the graph, and not by some arbitrary label.

1.5.3 Contributions

The work in this dissertation makes three significant contributions to our area of study (i.e. object-based object orientation within the UTP). First, we provide the UTP with an object-based model of object-oriented programs, in contrast to the existing class-based models.

Second, we provide an alternative model of pointers (references) for UTP that handles both value- and reference-based compound values (e.g. objects), in contrast to existing models, which handle one or the other. This is intended to contribute to the current discussions in the community as to what a general UTP pointer model should include and how one could generalise it.

Third, we provide an encoding of a prominent formalism of object orientation in the UTP (namely the Abadi–Cardelli-style of object calculus [AC96]) in contrast to developing our own notion of objects. Note that the encoding is performed in a systematic manner, with the hope that this will enable our work to be straightforwardly adapted for use with other Abadi–Cardelli-style object calculi.

1.5.4 Publications

This dissertation builds on two of our previously published works, [SG07] and [SG08]. The [SG07] paper introduces a cut down version of our chosen object calculus, a corresponding UTP model, and an outline of the consistency proof. Both the model and the consistency proofs are extended in this dissertation, in Chapter 3 and Chapter 4 respectively.

The [SG08] paper presents a model of locations that can be used to provide an abstract layout of memory, where each location is identified by precisely one path-set (i.e. the one that defines it). This paper also presents a brief outline of how this model can be linked to the basic relational model of the UTP, which is extended to the UTP model of designs in Chapter 6 of this dissertation.

Between them these publications provide evidence for the originality of the thesis being presented. They form the basis of work being presented in this dissertation. Here [SG07] corresponds to Section 2.2, Chapter 3 and Chapter 4, and [SG08] corresponds to Chapter 5.

1.5.5 Scope

At this point it is worth stating some areas that are beyond the scope of our consideration. First, we do not consider typing. This is consistent with the untyped nature of the UTP, at least as presented by Hoare and He in their seminal book [HH98], which introduces the subject. A significant body of research has gone into typed versions of the object calculus, e.g. [AC96, GH98, Cla02]. It ought to be possible to take advantage of this work, such as by assuming that the corresponding UTP model is provided with type-correct programs. In this circumstance it would be reasonable to use properties that follow from this type correctness in our UTP model proofs – for example, that named object members always exist when selected.

Second, we do not consider concurrency. It ought to be possible to add the UTP notion of reactive processes to the UTP model of objects. Having done this, it would be a matter of choosing a reactive process model that could straightforwardly support the operational semantics of an Abadi–Cardelli-style calculi.

Third, we do not consider refinement. The models were not intended to be used for the basis of a refinement calculus. It is not clear that it would be straightforward to use these models for the purposes of a refinement calculus without significant adaptation. Current work on providing the UTP with an OO-refinement calculus includes [HLL06, Kas06].

Fourth, we do not consider object ownership or reference encapsulation techniques. For example, object *representation exposure* is not considered. There is a significant amount of work in this area [NVP98, NCP99, Cla02, Par05, PNCB06, PB08] (and indeed, an entire workshop series on 'Aliasing, Confinement and Ownership'). These works use a variety of formalisms including the object calculus (e.g.[Cla02]) and separation logic (e.g. [Par05]).

Chapter 2

Background

This chapter introduces the mathematical frameworks and notations that are used throughout this dissertation. In particular, it presents:

- an overview of Abadi and Cardelli's untyped object calculus (ς-calculus) [AC96], along with a definition of our $\mathcal{O}$-calculus [SG07]; and
- an overview of Hoare and He's Unifying Theories of Programming (UTP) [HH98].

Before we introduce the Abadi–Cardelli calculus and the UTP, it is worth presenting some of the common notations that are used throughout this dissertation.

2.1 Common Notation and Conventions

Within this dissertation various notations and conventions are used to clarify the presentation. They will typically be introduced as they are needed. The remainder of this section contains some basic notation and conventions that are used throughout the dissertation.

2.1.1 The use of colour

So far we have only used colour to highlight cross-references and citations, both of which are coloured in blue (and hyper-linked in the electronic version). Brown is used to highlight meta-data within our $\mathcal{O}$-calculus reduction rules; this meta-data status is also highlighted using a Quine's corner notation ($\ulcorner \ldots \urcorner$) [Qui40] when it is not clear from context. Lastly, purple is used to highlight literal label values (e.g. an object's member names); a literal label is also typeset in `teletype` font.

2.1.2 Repeated terms

Comma-separated lists: It is frequently useful to denote an arbitrary comma-separated list of repeated terms. For example, $t_{i=1}^{k}$ is used to denote the list of terms t_1, t_2, …, t_k whenever $k \geq 1$, and the empty list of terms whenever $k < 1$. Such an empty list of terms is explicitly denoted by $\_$.

It may be necessary to scope the range of a repeated term, and this is denoted in the usual manner, i.e. $(t)_{i=1}^{k}$; in such cases, the outer brackets are not included in its expansion, which is t_1, t_2, …, t_k. When outer brackets are to be kept, the following alternative notation may be used:

$$\begin{array}{lcl} (_{i=1}^{k}\ t_i) & \widehat{=} & (t_{i=1}^{k}) \\ \{_{i=1}^{k}\ t_i\} & \widehat{=} & \{t_{i=1}^{k}\} \\ [_{i=1}^{k}\ t_i] & \widehat{=} & [t_{i=1}^{k}] \end{array}$$

Occasionally, it is useful to start counting at a different number, index over sets, or count backwards. These cases are denoted by $t^{k}_{i=j}$, $t_{i \in I}$, and $t^{i=k}_{j}$ respectively, where j is an integer and I is a set of indexes (such as program variable names). Note that the indexed terms $t^{i=3}_{2}$ and $t^{2}_{i=3}$ produce the results 't_3, t_2' and $\llcorner$ respectively; the latter empty case follows from the observation that there is no index i that is simultaneously greater than or equal to 3 and less than 2; i.e. $\neg(\exists\, i \bullet i \geq 3 \land i < 2)$.

Indexed operators: It is frequently useful to be able to repeatedly apply an associative binary operator to a list of arguments. For example, the indexed sequential composition operator $\fatsemi^{k}_{i=1}\ t_i$ denotes $t_1 \fatsemi t_2 \fatsemi \ldots t_k$ whenever $k \geq 1$, and skip (the unit of composition) whenever $k < 1$. Note that such operators are only defined for empty lists of arguments if they have a *unit* (e.g. 0 for addition, 1 for multiplication, true for conjunction, and false for disjunction).

2.1.3 Identifying subterms

It is sometimes useful to write functions that operate over the abstract syntax tree representation of a program. For example, it is sometimes possible to identify the *free* variables of a program in this manner. Here the only interesting cases are the variable identifier terms, and the variable binding terms. All other cases simply merge the results of applying the *free* variable function to their subterms. Therefore, it would be convenient to be able to define these other cases by a single generic case. This is achieved via the use of a notation for formally denoting the subterms of an abstract syntax tree term.

Generic subterms: Let a generic abstract syntax tree term t containing k subterms $t^{k}_{i=1}$ be denoted by $t\{t^{k}_{i=1}\}$, where a leaf term ($k = 0$) has the empty set of subterms.

In principle the *free* variable function (FV) can now be defined by the following three case equations, where: the first case whose left hand side (LHS) pattern matches the provided argument is taken; and the meta-term $bind(x, t)$ represents any term that binds the variable identifier x to the term t.

$$\begin{array}{lcl} \mathsf{FV}(bind(x,t)) & \widehat{=} & \mathsf{FV}(t) \setminus \{x\} \\ \mathsf{FV}(x) & \widehat{=} & \{x\} \\ \mathsf{FV}(t\{t^{k}_{i=1}\}) & \widehat{=} & \bigcup\{^{k}_{i=1}\mathsf{FV}(t_i)\} \end{array}$$

This style of function definition places certain restrictions on the structure of the language's abstract syntax tree. Specifically, in the case of the free-variable function that has just been defined, it must not be applied to the syntactic form of the standard UTP terms for representing variable introduction (and elimination); here, a variable is introduced (and terminated) as a subterm of sequential composition, so the scope of the variable is not contained as a subterm of the variable introduction operation, thus cannot be easily modelled using this approach. Fortunately, this issue can be avoided by constraining a program to use higher level *variable block* commands, which have the appropriate form, and whose semantics is defined in terms of the standard lower-level variable introduction (and elimination) commands.

2.1.4 Meta-variable identifiers

Within this dissertation, various meta-variables will be used to represent different types of entity within a syntactic description or formula. For example, the meta-variables l and o are used to represent a label and an object entity respectively. When more than one label or object is required, new label and object meta-variables can be constructed by the addition of subscripts and prime marks (e.g. l'_3, a primed label meta-variable with subscript 3). The following table summaries the typical mapping of meta-variables to entity types.

Meta-Variable	
Ids	Type
b	Boolean
cv	Compound Value
d	Definition
D	Design
e	Expression
f	Function
i, j	Integer (Index)
k	Integer (Size)
l	Label
lv	Literal Value
m	Method
n	Node
N	Set of nodes
o	Object
ov	Operand Value

Meta-Variable	
Ids	Type
p, q	Conditional (predicate)
P, Q	(Relational) predicate
lp	(Labelled) path
Lp	(Labelled) path-set
pt	Program text
r	Reference Value
t	Term
tl	Trace label
tr	Trace
Tr	Trace-set
u	List of variables
v	Value
w	List of variables
x, y	(Program) variables
z	Fix-point identifier

Table 2.1: Meta-variable identifiers

2.2 Object Calculus

The object calculus used in this dissertation is an extension of the ς-calculus presented in Chapter 6 of Abadi and Cardelli's book on objects [AC96]. Our object calculus is known as the $\mathcal{O}$-calculus; it is essentially the same as the heap extended core object calculus we presented in [SG07]. The main differences are in the order of presentation and the inclusion of two further extensions, which directly support conditional evaluation and untyped lambda calculus (λ-calculus) functions.

The operational semantics of the object calculus operations is defined in terms of a collection of small-step evaluation rules. These rules are similar to those in [Pie02], 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 Section 2.2.3, can be selected and set as follows:

$\{\textsc{heap} \mapsto H\}$	Let H denote the HEAP context.
$\{\textsc{heap} \leftarrowtail H\}$	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:

$$\frac{\langle \mathit{condition}_1 \rangle \quad \ldots \quad \langle \mathit{condition}_n \rangle}{\langle \mathit{concluded\ term\ evaluation\ step} \rangle}\ \textsc{RuleName} \qquad \boxed{\begin{array}{l} \langle \mathit{side\ condition}_1 \rangle \\ \ldots \\ \langle \mathit{side\ condition}_m \rangle \end{array}}$$

Here, a $\mathit{condition}_i$ is either a logical constraint or an evaluation step, where as a $\mathit{side\ condition}_j$ is either a meta-level logical constraint or a meta-variable definition. Note that if there are no *conditions* (i.e. $n = 0$), then the concluded term evaluation step can be used whenever the term on its left hand side pattern matches with that of the actual term.

We now define our $\mathcal{O}$-calculus in five stages. The first stage presents an overview of the ς-calculus (Section 2.2.1). The second stage extends this with a few basic enhancements, for

handling literal values and field assignment (Section 2.2.2). The third stage adds an abstract heap (Section 2.2.3). Finally, the last two stages provide direct support for handling conditional evaluation and eagerly evaluated λ-calculus functions respectively (Section 2.2.4).

2.2.1 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.

$[\overset{k}{\underset{i=1}{}} l_i = m_i]$	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 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.

Here, the meta-variables

o, l	denote an object value and a label 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.

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

Here, the meanings of the components are as follows

$l \in o$	label l is in the domain of object o.
$o_1 \oplus o_2$	object map o_2 overrides object map o_1.
$\ulcorner e \urcorner$	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.

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

Here:

$o(l)$	is the method of object o with label l.
$m \mathrel{\widehat{=}} e$	defines variable m to be the evaluation of meta-expression e.
$\varsigma(x)\ e \doteq m$	binds x and e to the self-variable and body of method m.
$e\{\!\|x \rightarrowtail o\|\!\}$	the substitution of object o for free variable x in expression e (it is defined in Section 2.2.5).

The rule for evaluating an evaluable argument before the base rule can be applied is defined in a similar manner to that of the method update operation.

$$\frac{\Gamma \bullet e \longrightarrow \Gamma' \bullet e'}{\Gamma \bullet e.l \longrightarrow \Gamma' \bullet e'.l}\ \textsc{InvM-1}$$

2.2.2 Basic extensions

The basic extensions introduce four sorts of – non-object – literal value: the booleans, the integers, the strings, and the unset unit sort. Here the integers are denoted by their usual Arabic numbers, the booleans by the constants true and false, the strings by double-quoted text (e.g. "Some text"), and the unset sort by the constant ¿.

Aside 2.1: The unset value serves the same role as the Perl language's undef value; e.g. it provides a mechanism for testing whether a variable contains, or a function returns, a defined value.
□

The basic extension also provides some logical, arithmetic, and text operations, for working with the booleans, integers, and strings respectively. The precise collection of operations is not important, so they are abstracted by a generalised literal value operation ($\boxdot_k$), which takes k literal arguments and returns a literal value; here k is omitted when it is clear from context. Lastly, the basic extension provides a field assignment operation, which is used to assign the evaluation of an expression to a given object member.

$\boxdot(e_{i=1}^k)$	denotes a k-ary operation on literal values (e.g. binary '+' and '≤', with $k = 2$).
$o.l := e$	denotes the field assignment operation, which evaluates the expression e to a value v, then applies the method update operation $o.l \Leftarrow \varsigma(\_)\ v$.

The following rules specify the base cases for a generic literal operation (LitOp) and the field assignment operation (FldA).

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

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

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

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

$$\frac{\Gamma \bullet e_1 \longrightarrow \Gamma' \bullet e_1'}{\Gamma \bullet e_1.l := e_2 \longrightarrow \Gamma' \bullet e_1'.l := e_2}\ \textsc{FldA-1} \qquad \frac{\Gamma \bullet e \longrightarrow \Gamma' \bullet e'}{\Gamma \bullet o.l := e \longrightarrow \Gamma' \bullet o.l := e'}\ \textsc{FldA-3}$$

Note that the digit distinguishing the field assignment rules FldA-1 and FldA-3 identifies the argument to evaluate.

2.2.3 Heap extensions

We now introduce a copy-based heap storage model [Pie02], 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.
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.

Here, the meta-variables r, v, and i denote the reference, general, and integer values respectively.

The following rules specify the base cases for the fresh, dereference, and assignment (reference update) operators; these rules select and manipulate the heap component of the rule-context Γ. The other cases for these operators are defined to follow the usual left to right evaluation order.

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

$$\frac{r \in H}{\Gamma\{\mathtt{heap} \mapsto H\} \bullet *r \longrightarrow \Gamma \bullet \ulcorner H(r) \urcorner}\ \textsc{DeRef}$$

$$\frac{r \in H}{\Gamma\{\mathtt{heap} \mapsto H\} \bullet r *= v \longrightarrow \Gamma\{\mathtt{heap} \leftarrow \ulcorner H \oplus \{r \mapsto v\} \urcorner\} \bullet r}\ \textsc{UpdL}$$

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

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

$$\frac{\Gamma \bullet e \longrightarrow \Gamma' \bullet e'}{\Gamma \bullet *e \longrightarrow \Gamma' \bullet *e'}\ \textsc{DeRef-1}$$

$$\frac{\Gamma \bullet e_1 \longrightarrow \Gamma' \bullet e_1'}{\Gamma \bullet e_1 *= e_2 \longrightarrow \Gamma' \bullet e_1' *= e_2}\ \textsc{UpdL-1} \qquad \frac{\Gamma \bullet e_2 \longrightarrow \Gamma' \bullet e_2'}{\Gamma \bullet r *= e_2 \longrightarrow \Gamma' \bullet r *= e_2'}\ \textsc{UpdL-2}$$

2.2.4 Remaining extensions

The remaining extensions introduce some support for directly handling conditional evaluation, sequential composition, eagerly evaluated λ-calculus functions and local blocks.

if b then e_1 else e_2	denotes the operation that evaluates expression e_1 when the boolean b is true, and expression e_2 otherwise.
$o?l$	denotes the membership test operation, which evaluates to true whenever object o contains a method with label l, and false otherwise.
$e_1 \fatsemi e_2$	denotes the sequential composition of the expressions e_1 and e_2. Here, the result of evaluating e_1 is ignored; hence, e_1's contribution is via its side-effects (e.g. updating the heap).
$\lambda(x)\ e$	denotes a function with formal parameter x whose body is defined by the expression e.

$f\ v$	denotes the function application operation, which evaluates the body of the function f, once its parameter has been replaced by a copy of value v, wherever it occurs (free within the body).
$\mathsf{let}\ d_{i=1}^{k}\ \mathsf{in}\ e$	denotes the local block that introduces a comma separated list of local declarations $d_{i=1}^{k}$ that can be used in expression e.

Here, the meta-variables f and d denote a function and a local declaration respectively, and the local declarations have one of two forms:

$x \equiv t$	denotes the binding of an unevaluated term t to the identifier x.
$x \mathrel{\widehat{=}} e$	denotes the binding of the evaluation of an expression e to the identifier x.

The following conditional operation rules have a slightly different form to those previously defined. Specifically, they have two base cases; and these cases may contain some potentially unevaluated evaluable arguments. This is important to model the usual notion of conditional evaluation, where only the branch to be selected is evaluated. An alternative would be to evaluate both branches, before selecting the relevant branch. Note that this alternative scheme provides the usual semantics for conditional evaluation so long as a branch terminates and does not contain any globally visible side effects.

$$\frac{}{\Gamma \bullet \mathsf{if\ true\ then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow \Gamma \bullet e_1}\ \textsc{IfT} \qquad \frac{}{\Gamma \bullet \mathsf{if\ false\ then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow \Gamma \bullet e_2}\ \textsc{IfF}$$

$$\frac{\Gamma \bullet e \longrightarrow \Gamma' \bullet e'}{\Gamma \bullet \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \longrightarrow \Gamma' \bullet \mathsf{if}\ e'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2}\ \textsc{If}$$

The operations for membership test, function invocation and sequential composition can be specified in the usual manner. Their base cases are now defined by the following rules.

$$\frac{}{\Gamma \bullet o?l \longrightarrow \Gamma \bullet \ulcorner l \in o \urcorner}\ \textsc{HasM} \qquad \frac{}{\Gamma \bullet v \fatsemi e \longrightarrow \Gamma \bullet e}\ \textsc{SeqC}$$

$$\frac{}{\Gamma \bullet f\ v \longrightarrow \Gamma \bullet \ulcorner e\{\!|x \mapsto v|\!\}\urcorner}\ \textsc{InvF} \qquad \boxed{\lambda(x)\ e \doteq f}$$

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

$$\frac{\Gamma \bullet e \longrightarrow \Gamma' \bullet e'}{\Gamma \bullet e?l \longrightarrow \Gamma' \bullet e'?l}\ \textsc{HasM-1} \qquad \frac{\Gamma \bullet e_1 \longrightarrow \Gamma' \bullet e_1'}{\Gamma \bullet e_1 \fatsemi e_2 \longrightarrow \Gamma' \bullet e_1' \fatsemi e_2}\ \textsc{SeqC-1}$$

$$\frac{\Gamma \bullet e_1 \longrightarrow \Gamma' \bullet e_1'}{\Gamma \bullet e_1\ e_2 \longrightarrow \Gamma' \bullet e_1'\ e_2}\ \textsc{InvF-1} \qquad \frac{\Gamma \bullet e \longrightarrow \Gamma' \bullet e'}{\Gamma \bullet f\ e \longrightarrow \Gamma' \bullet f\ e'}\ \textsc{InvF-2}$$

Finally, the local block is introduced as a form of function definition and invocation, as follows:

$$\frac{}{\Gamma \bullet \mathsf{let}\ x \equiv t\ \mathsf{in}\ e \longrightarrow \Gamma \bullet \ulcorner e\{\!|x \mapsto t|\!\}\urcorner}\ \textsc{LetM} \qquad \frac{}{\Gamma \bullet \mathsf{let}\ x \mathrel{\widehat{=}} e_1\ \mathsf{in}\ e_2 \longrightarrow \Gamma \bullet (\lambda(x)\ e_2)\ e_1}\ \textsc{LetD}$$

$$\frac{}{\Gamma \bullet \mathsf{let}\ d_{i=1}^{k}\ \mathsf{in}\ e \longrightarrow \Gamma \bullet \mathsf{let}\ d\ \mathsf{in\ let}\ d_{i=2}^{k}\ \mathsf{in}\ e}\ \textsc{LetL} \qquad \boxed{k > 1}$$

Here, $d_{i=1}^{k}$ denotes a list of local declarations with at least two elements.

2.2.5 Substitution definition and object calculus examples

Substitution: Three of the reduction rules for our $\mathcal{O}$-calculus have used the substitution operation to replace free occurrences of an identifier within a term with another term. This

can be formally defined as follows:

$$
\begin{array}{lcl}
x\{\!|x \leftarrowtail t|\!\} & \widehat{=} & t \\
(\ \varsigma(x)\ e\ )\{\!|x \leftarrowtail t|\!\} & \widehat{=} & \varsigma(x)\ e \\
(\ \lambda(x)\ e\ )\{\!|x \leftarrowtail t|\!\} & \widehat{=} & \lambda(x)\ e \\
(\ \mathsf{let}\ x \equiv t_1\ \mathsf{in}\ e\ )\{\!|x \leftarrowtail t|\!\} & \widehat{=} & \mathsf{let}\ x \equiv t_1\ \mathsf{in}\ e \\
(\ \mathsf{let}\ x \mathrel{\widehat{=}} e_1\ \mathsf{in}\ e\ )\{\!|x \leftarrowtail t|\!\} & \widehat{=} & \mathsf{let}\ x \mathrel{\widehat{=}} e_1\ \mathsf{in}\ e \\
(\ t_0\{t_{i=1}^{k}\}\ )\{\!|x \leftarrowtail t|\!\} & \widehat{=} & t_0\{\!|_{i=1}^{k}\ t_i\{\!|x \leftarrowtail t|\!\}\ \}
\end{array}
$$

Here, the first of the pattern matching definition lines matches only when the term being substituted is an identifier which is syntactically identical to that of the identifier being substituted. The following four definition lines show the four ways of introducing a local identifier (variable); these prevent the substitution of bound variables with the same name. And the last of the definition lines shows the general case, where the substitution of any other term amounts to substituting its subterms (if there are any). For example:

$$
\begin{array}{rl}
(\ \lambda(y)\ y + (\lambda(x)\ \ldots)(x+1)\ )\{\!|x \leftarrowtail 3|\!\} & = \lambda(y\{\!|x \leftarrowtail 3|\!\})\ (\ y + (\lambda(x)\ \ldots)(x+1)\ )\{\!|x \leftarrowtail 3|\!\} \\
& = \lambda(y)\ (\ y + (\lambda(x)\ \ldots)(x+1)\ )\{\!|x \leftarrowtail 3|\!\} \\
& = \lambda(y)\ y\{\!|x \leftarrowtail 3|\!\} + (\ (\lambda(x)\ \ldots)(x+1)\ )\{\!|x \leftarrowtail 3|\!\} \\
& = \lambda(y)\ y + (\ (\lambda(x)\ \ldots)(x+1)\ )\{\!|x \leftarrowtail 3|\!\} \\
& = \lambda(y)\ y + (\lambda(x)\ \ldots)\{\!|x \leftarrowtail 3|\!\}(x+1)\{\!|x \leftarrowtail 3|\!\} \\
& = \lambda(y)\ y + (\lambda(x)\ \ldots)(x+1)\{\!|x \leftarrowtail 3|\!\} \\
& = \lambda(y)\ y + (\lambda(x)\ \ldots)(x\{\!|x \leftarrowtail 3|\!\} + 1\{\!|x \leftarrowtail 3|\!\}) \\
& = \lambda(y)\ y + (\lambda(x)\ \ldots)(3 + 1\{\!|x \leftarrowtail 3|\!\}) \\
& = \lambda(y)\ y + (\lambda(x)\ \ldots)(3+1)
\end{array}
$$

Value example: The simplest $\mathcal{O}$-calculus programs (expressions) are the values, as they are already evaluated, and thus represent their own results. The integer 42 is an example of such a program. This example is used to illustrate how to interpret the results of compiling a $\mathcal{O}$-calculus program into one of our UTP object models.

Restorable cell example: This example is essentially the third variant of the Abadi–Cardelli's untyped storage cell [AC96, Section 6.5.5]. Here we have an uninitialised (unset) `data` field, whose value can be accessed via `get` and `set` methods. The `set` method also stores the previous value of the `data` field in a dynamically updated `undo` method.

$$
\begin{array}{lcl}
\mathit{UnsetRestCell} & = & [\,\texttt{data} = \text{¿},\ \texttt{get} = \varsigma(x)\ x.\texttt{data},\ \texttt{set} = \varsigma(x)\ \lambda(y)\,(\ x.\texttt{undo} \Leftarrow \varsigma(z)\ z.\mathit{data} := x.\texttt{data}\ ).\texttt{data} := y,\ \texttt{undo} = \varsigma(x)\ x.\texttt{data} := \text{¿}\,]
\end{array}
$$

Note that the `set` method makes use of two distinct *self* variables x and z, which represent the current and future values respectively. Further, both of these variables are in scope in `set`'s inner method (i.e. $x.\texttt{undo} \Leftarrow \varsigma(z)\ z.\mathit{data} := x.\texttt{data}$). This is why the Abadi–Cardelli calculus uses a named self parameter, rather than the *self* (or *this*) keyword that is used in many object-oriented (OO) languages.

Factorial example: A simple recursive algorithm to calculate the factorial of a number can be written in the $\mathcal{O}$-calculus as follows:

$$
\mathit{Factorial} \quad \widehat{=} \quad [\texttt{fact} = \varsigma(x)\ \lambda(i)\ \mathsf{if}\ i < 2\ \mathsf{then}\ 1\ \mathsf{else}\ i \times x.\texttt{fact}(i-1)]
$$

Here, the *Factorial* object provides a single method that contains an anonymous function, which takes a natural number and outputs its factorial value. It is later used as a concrete example of how one of the UTP encodings of the $\mathcal{O}$-calculus is compiled and reasoned with.

Sharing methods example: Appendix B provides a collection of $\mathcal{O}$-calculus programs that enable the methods of an object to be shared. These techniques can be used when modelling a class or an inheritance scheme, in terms of $\mathcal{O}$-calculus objects.

2.3 Unifying Theories of Programming

Various notations and conventions are adopted in the UTP book [HH98]. Only some of these are required for the purposes of this dissertation; in general, they will be introduced when needed. However, the following notation and conventions, concerning the alphabets of a predicate, are worth highlighting at this point.

First, it is useful to be able to partition the predicate P's alphabet of *free*, i.e. not quantified, variables into inputs and outputs; convention dictates that only the output variables are dashed, thus the output value of a variable x is denoted by x', whereas the input value is denoted by x itself. The alphabet associated with a predicate P, its inputs, and its outputs are denoted by αP, $in\alpha P$, and $out\alpha P$ respectively.

Second, as variables in real programing languages have both input and output values, even if they are the same, it is frequently assumed that the set of output variables can be derived from the set of input variables by dashing the input variables. In such cases, the output alphabet of a program represented by predicate P is $out\alpha P = \{x' \mid x \in in\alpha P\}$, and is denoted by $in\alpha' P$.

Last, the letters w and w' represent the list of all input and output variables of a predicate respectively, where $w(i)$ represents the i^{th} variable in w that has $\#w$ elements. This is done so that it is possible to do mass renaming of these variables, for an arbitrary predicate P, using the notation $P[w_1/w_2]$, to stand for the predicate that is formed by replacing all free occurrences of variable $w_2(i)$ with variable $w_1(i)$ in P, where $i \in 1\,..\,\#w_1$ and $\#w_1 = \#w_2$.

2.3.1 Relational model

The semantics of the UTP is based on the notion of a relation between variables, specifically input and output variables, at observable points in a program. Arguably the simplest program is the one that does nothing. This is modelled in the UTP by the skip program, which is defined by the predicate $w' = w$, where: w represents both the list of program variables and their initial (input) values; and w' the final (output) values of the same list of program variables. The usual UTP convention of denoting the input and output values of a program variable x, by logical variables x and x' is followed.

Formally, a UTP program is a relational predicate, denoted by the pair $(\alpha P, P)$, where P is a predicate that relates inputs and output variables within its alphabet αP, which is at least the set of *free*, i.e. not quantified, variables in P. Normally, the alphabet of a predicate is taken to be those variables that are free within it (or become required in order to compose it with another UTP program). Hence, the predicate part of the relation is often used to stand for the relation as a whole. This practise is adopted in this dissertation.

Aside 2.2: Most predicate operators assume that the alphabets of the predicates they combine are the same. Hence, there is a special operation, P_{+A}, which expands a predicate P for a given a set of input variable names A, by creating both an associated set of output variables, i.e. $\{a : A \bullet a'\}$, and insisting that they are bound to their associated inputs. Formally this operation is defined by $P_{+A} \mathrel{\widehat{=}} P \wedge (\bigwedge_{a \in A} \bullet a' = a)$, provided the sets of input names and free variables are disjoint, i.e. $A \cap FV(P) = \emptyset$.
□

Having said that a program can be represented as a predicate relation, there must be a way of representing the semantics of a program's constructs as predicate operators (i.e. operators that take and return predicates). Chapters 1 and 2 of the UTP book [HH98] demonstrate how the semantics of basic imperative constructs, such as assignment, conditional execution, and loops, can be written as predicates. This is straightforward for all the constructs except loops, which require an understanding of the *fix-point* theory associated with lattices [DP02, Chapters 2 and 4] and [BCG00].

Example 2.3: The sequential composition predicate operator $(\_ \fatsemi \_)$ constructs a new program fragment predicate from two existing program fragment predicates, as follows:

$$\begin{array}{lcll} P \fatsemi Q & \widehat{=} & \exists\, w_0 \bullet P[w_0/w'] \wedge Q[w_0/w] & \text{provided } out\alpha P = in\alpha' Q = \{w'\} \\ in\alpha(P \fatsemi Q) & \widehat{=} & in\alpha P & \\ out\alpha(P \fatsemi Q) & \widehat{=} & out\alpha Q & \end{array}$$

□

Correctness: Having introduced the notion of representing relations between input and output variables as predicates, what is the method of stating that a program correctly implements its specification? It is natural to choose the logical implication ordering to determine program correctness, where a program P is considered to correctly implement its specification S if it implies its specification, i.e. $P \Rightarrow S$; one caveat to this statement is that both the program and specification must have the same alphabet, i.e. $\alpha P = \alpha S$. The formal notion of program correctness is the universal quantification of $P \Rightarrow S$, and this is denoted $[P \Rightarrow S]$ in accordance with Dijkstra and Scholten's convention [DS90].

$$[P \Rightarrow S] = \forall\, x^k_{i=1} \bullet P \Rightarrow S \qquad \text{provided } \alpha P = \alpha S = \{x^k_{i=1}\}$$

Note that the implication ordering also defines what it means for one program to refine another in the UTP relational model.

Refinement: A program P is said to refine a specification S whenever it implies the specification.

$$P \sqsupseteq S \quad \widehat{=} \quad P \Rightarrow S \qquad \text{provided } \alpha P = \alpha S$$

The transitive nature of the refinement (implication) ordering allows for a staged development of concepts. Critically, the various steps in the development form a complete lattice of programs with alphabet A, where $\mathbf{false}_A$ is the strongest element of the predicate lattice ordered by implication and $\mathbf{true}_A$ is the weakest element. Here the notion of truth and falsehood is in relation to a specific alphabet of free variables. Essentially, $\mathbf{true}_A$ is modelled by the relation that does not restrict the values of its outputs, and is sometimes referred to in programming languages as *abort* or *chaos*. Unfortunately, $\mathbf{false}_A$ does not have such an intuitive description, but it is technically defined to be the result of the disjunction of an empty set of relations over alphabet A, i.e. $\bigvee_{P \in \emptyset}$, where:

$$\begin{array}{lcl} \bigvee_{P \in \emptyset} & \widehat{=} & \mathsf{false} \\ \bigvee_{P \in \{P^k_{i=1}\}} & \widehat{=} & P_1 \vee \bigvee_{P \in \{P^k_{i=2}\}} \end{array}$$

The false relation is sometimes called the *miracle*, as it implements an arbitrary specification S; i.e. $\forall\, S \bullet [\mathsf{false} \Rightarrow S]$.

2.3.2 Design model

A UTP *design* is a special form of relation that introduces the notions of successful initialisation and termination, via a couple of special variables, namely Π_{OK} and Π_{OK}' respectively. These variables are *free* in every design, and represent a program's execution context; as such they cannot be referred to by any of the program's variables or constructs. Instead, they are used to change some of the constraints that the basic relational model imposes on the implementation context of a program, such as backwards execution; that is only evaluating those inputs that are actually needed for the output. Though this style of execution can be supported, it is not supported by typical imperative languages, such as C, Pascal and Ada. Hence, the UTP notion of a design provides for the optimisation of forward execution typical of imperative languages; that is the evaluation of inputs as they appear, rather than as they are needed.

Aside 2.4 – Context variable naming conventions: The UTP models of the $\mathcal{O}$-calculus use special, non-program, variables to model various aspects of the OO-context. Such contextual variables are denoted by identifiers that start with the capital Greek letter pi (Π), so that they can be easily distinguished from normal program variables. In particular, the usual name of the special imperative programming (design) variable, used to model program initialisation and termination of a program, is changed from *ok* to Π_{OK} so that it can conform to this convention. Another example of such a contextual variable is Π_{RES}, which is used to denote the *result* of a program (or subprogram).
□

All of the UTP models of the $\mathcal{O}$-calculus presented in this thesis extend the notion of a UTP design, which captures the semantics of first-order imperative programs without subroutines (e.g. procedures and functions). However, instead of using the notion of the design defined in the seminal book on UTP [HH98], we use He's updated variant of a design [HLL02a]. It updates the original definition in [HH98] by ensuring that "the assumption is a precondition, containing only undashed [input] variables [... which ...] corresponds exactly to the third healthiness condition" of [HH98, page 84].

Design definition: 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.

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

provided $\alpha p \subseteq in\alpha P$ and $\Pi_{OK}, \Pi_{OK}' \notin \alpha P$

Two consequences of this design model are: it is impossible to say what the initial values of a variable are prior to the starting of a program; and it is impossible to say what the final values of a variable are prior to the termination of a program. Here $\neg\Pi_{OK}'$ represents a situation where a program has yet to terminate. Having said this, if the program is a design where both Π_{OK} and p hold, then it is guaranteed to terminate. The issue is in demonstrating (proving) that the precondition p is met. This is non-trivial and, in general, at least as hard as the halting problem.

Correctness: The notion of correctness for a design is the same as that for a relation, namely, that of an implementation-design implying the specification-design. Equivalently, the notion of program correctness can be defined as the conjunction of: the specification's pre-condition implying the implementation's pre-condition; and the implementation's post-condition implying the specification's post-condition, within the context of the specification's pre-condition.

In other words, supposing that implementation and specification designs are represented by $D_I = p_I \vdash P_I$ and $D_S = p_S \vdash P_S$ respectively, then the notion of program correctness is either $[D_I \Rightarrow D_S]$ or equivalently $[p_S \Rightarrow p_I] \wedge [(p_S \wedge P_I) \Rightarrow P_S]$. The proof of the equivalence of these two notions of design correctness is given in theorem 3.1.2 of the UTP book [HH98].

Aside 2.5: The notion of implication is used for both the definition of a design and the correctness of a design. In the first case, the pre-condition implies the post-condition; and in the second case, an implementation design implies the specification design. It is all too easy to confuse or merge these distinct usages of the implication ordering. Having said this, it is precisely the interaction between the usage of implication in the definition and correctness of a design which leads to the alternative (second) definition of design correctness in the preceding paragraph.
□

2.3.3 UTP design commands

In this section we present a slightly extended version of the standard commands for a UTP design; i.e. those operations that can be used to construct a UTP program.

We start by introducing the notion of a design *frame*, which is used to clarify the definition of the *basic* UTP commands, by focusing attention on only those elements of the alphabet that the commands actually change. Here, a *basic* command is considered to be any command that does not take a UTP program as an argument.

Each of the *non-basic* commands is defined in terms of how it affects the UTP programs that it consumes, which may be designs. We also show that if all the program arguments are designs, then the result is a design. Hence, the commands that we present are closed, when applied to designs. This closure is important for the use of Tarski's fix-point theory, which is used to define the semantics of loops (and recursion – see Chapter 3).

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 $p \vdash P \wedge w' = w$ with alphabet $\{V, V', w, w'\}$, where the vector w contains the variables within the input alphabet of $p \vdash P$ except those in the set V (i.e. $\{x \mid x \in w\} = in\alpha(p \vdash P) \setminus V$).

Skip command: The $\mathsf{skip}^{\mathrm{D}}_A$ command does nothing and terminates successfully.

$$\begin{array}{lcl} \mathsf{skip}^{\mathrm{D}}_A & \widehat{=} & \emptyset : (\ \mathsf{true} \vdash \mathsf{true}\ ) \\ \alpha(\mathsf{skip}^{\mathrm{D}}_A) & \widehat{=} & A \end{array}$$

Miracle command: The miracle places an impossible collection of restrictions on the outputs of the variables within its alphabet (i.e. false).

$$\begin{array}{lcl} \mathsf{miracle}_A & \widehat{=} & \emptyset : (\ \mathsf{true} \vdash \mathsf{false}\ ) \\ \alpha\mathsf{miracle}_A & \widehat{=} & A \end{array}$$

Note that the miracle element simplifies to the predicate $\neg \Pi_{\mathrm{OK}}$, and is the top of the design refinement ordering ($\top_D$). For a program to be implementable its result must not be reducible to a miracle. However, miracles are useful for modelling contractual guarantees, as failure of the guarantee to hold is not the fault of the design (or program).

Chaos command: The chaos_A command, also known as *abort*, behaves unpredictably.

$$\begin{array}{lcl} \mathsf{chaos}_A & \widehat{=} & \emptyset : (\ \mathsf{false} \vdash \mathsf{true}\ ) \\ \alpha(\mathsf{chaos}_A) & \widehat{=} & A \end{array}$$

Note that the chaos element simplifies to the predicate true, and is the bottom of the design refinement ordering ($\bot_D$). It could also have been defined by (false $\vdash$ false), which is the last of the four programs involving only truth and falsehood.

Conditional command: The conditional command $(P \lhd b \rhd Q)$ behaves like the design P if the boolean expression b is true, otherwise it behaves like Q.

$$\begin{array}{lcl} P \lhd b \rhd Q & \widehat{=} & b \wedge P \vee \neg b \wedge Q, \quad \text{provided } \alpha P = \alpha Q \\ \alpha(P \lhd b \rhd Q) & \widehat{=} & \alpha P \end{array}$$

When both of a conditional command's program arguments are designs, and its boolean argument is defined, then the result of the conditional is also a design. This is proved in Appendix C.1.1 – specifically it is shown that:

$$(p \vdash P) \lhd b \rhd (q \vdash Q) \quad = \quad (p \lhd b \rhd q) \vdash (P \lhd b \rhd Q)$$

In English, the resultant design states that whenever the boolean variable is defined and the selected precondition is met, then the selected postcondition will also hold.

Non-deterministic choice command: The non-deterministic choice $(P \sqcap Q)$ behaves either like design P or design Q.

$$\begin{array}{lcl} P \sqcap Q & \widehat{=} & P \vee Q, \quad \text{provided } \alpha P = \alpha Q \\ \alpha(P \sqcap Q) & \widehat{=} & \alpha P \end{array}$$

When both of a non-deterministic command's arguments are designs, then the result of the non-deterministic command is also a design. It is straightforward to prove this using the propositional calculus and the definitions of the non-determinism command and the UTP design. Here the only interesting point is that the disjunction of designs $(p \vdash P)$ and $(q \vdash Q)$ forms a conjunction of their pre-conditions.

$$(p \vdash P) \sqcap (q \vdash Q) \quad = \quad p \wedge q \vdash P \sqcap Q$$

Sequential composition command: The sequential composition $(P \fatsemi Q)$ behaves like design P and then design Q, where the final state of P is also the initial state of Q.

$$\begin{array}{lcl} P \fatsemi Q & \widehat{=} & \exists\, w_0 \bullet P[w_0/w'] \wedge Q[w_0/w], \quad \text{provided } out\alpha P = in\alpha' Q = \{w'\} \\ in\alpha(P \fatsemi Q) & \widehat{=} & in\alpha P \\ out\alpha(P \fatsemi Q) & \widehat{=} & out\alpha Q \end{array}$$

The proof that the composition of two designs is also a design is in Appendix C.1.2. This differs from the proof given in the UTP book, as we are using He's more recent version of a design, which is slightly more restrictive. In particular, it is possible to simplify the representation of the precondition for the composed design, by side-stepping the need for sequential composition with the bottom element in the standard relational calculus (i.e. true).

$$(p \vdash P) \fatsemi (q \vdash Q) \quad = \quad p \wedge \neg(P \fatsemi \neg q) \vdash P \fatsemi Q$$

In English, this says that the composition of two designs is the design that performs $P \fatsemi Q$ whenever: the first design's precondition p is met; and the result of the first design produces a context which ensures that the second design's precondition q is met.

Assignment command: The assignment command $(x := e)$ assigns the evaluation of expression e to the variable x so long as the expression e is defined and the variable x is in the alphabet of the assignment $(\alpha(\_:=\_))$. We adopt the usual UTP practice of assuming that such expressions are well defined and thus do not put in a definedness pre-condition check in the definition of assignment. The assignment command can also be generalised to the simultaneous assignment command, which takes vectors of distinct variables $(x^k_{i=1})$ and general expressions $(e^k_{i=1})$.

$$\begin{array}{lcl} x :=_A e & \widehat{=} & \{x\} : (\ \mathsf{true} \vdash x' = e\ ), \quad \text{provided } x' \in A \\ \alpha(x :=_A e) & \widehat{=} & A \end{array}$$

$$\begin{array}{lcl} x^k_{i=1} :=_A e^k_{i=1} & \widehat{=} & \{x^k_{i=1}\} : (\ \mathsf{true} \vdash x'^k_{i=1} = e^k_{i=1}\ ), \quad \text{provided } x'^k_{i=1} \in A \\ \alpha(x^k_{i=1} :=_A e) & \widehat{=} & A \end{array}$$

This assignment operation differs from that of the standard higher-order programming found in Chapter 9 of [HH98], which defines assignment in terms of refinement. Here, the assignment is defined by the relational predicate $x := e \mathrel{\widehat{=}} (x' \sqsupseteq e \wedge w' = w)$, where a first order value (v) is only refined by itself (i.e. $x' \sqsupseteq v \ \Leftrightarrow \ x' = v$). In particular, it allows an implementation of a procedure P to refine its specification; i.e. $(P \sqsupseteq S) \Rightarrow (x := P \sqsupseteq x := S)$.

Iteration command: The iteration command $b * P$ repeats the program P as long as the condition b is true.

$$b * P \ \widehat{=}\ \mu\, X \bullet P \fatsemi X \lhd b \rhd \mathsf{skip}^{\mathrm{D}},$$

Here, $\mu\, X \bullet F(X)$ stands for the *Tarski weakest fix-point* [Tar55] of the equation $X = F(X)$, and X is a predicate from the lattice of UTP relational predicates [HH98].

The program $b * (p \vdash P)$ results in a design because either: it infinitely loops, in which case it is equivalent to the chaos design; or it sequentially composes a finite number of designs, which is a design.

Variable introduction command: The local variables $x^k_{i=1}$ are introduced by the command $(\mathsf{var}\, x^k_{i=1})$, so long as they are not already in the alphabet of the program.

$$\begin{array}{lcl} \mathsf{var}\, x^k_{i=1} & \widehat{=} & \exists\, x^k_{i=1} \bullet \mathsf{skip}^{\mathrm{D}}_A \\ & = & \mathsf{true} \vdash \exists\, x^k_{i=1} \bullet \mathsf{skip}_{A1} \end{array}$$

Here, the alphabet of the command as a whole, i.e. $\alpha(\mathsf{var}\, x^k_{i=1})$, is $A \setminus \{x^k_{i=1}\}$ and the alphabet $A1 = A \setminus \{\Pi_{\mathrm{OK}}, \Pi_{\mathrm{OK}}'\}$. If $k < 1$ then an empty variable list is introduced; this is considered to have the same effect as $\mathsf{skip}^{\mathrm{D}}_A$ (i.e. the no operation command). The proof that this command produces a design is presented in Appendix C.1.3.

When the introduction of variable x is composed with program Q, x must be in the alphabet of Q (i.e. $x \in \alpha Q$), and the scope of the existential quantification is extended to include program Q; that is $(\mathsf{var}\, x) \fatsemi Q = (\exists\, x \bullet Q)$. This result is proved in Appendix C.2.2; it relies on the (existential) renaming in the definition of sequential composition.

Initialised variable introduction: It is sometimes convenient to initialise the variables as they are introduced.

$$\mathsf{var}\, x^k_{i=1} := e^k_{i=1} \ \widehat{=}\ (\mathsf{var}\, x^k_{i=1}) \fatsemi (x^k_{i=1} := e^k_{i=1})$$

Variable completion command: The local variables $x_{i=1}^k$ are removed from the current scope by the command $(\mathsf{end}\, x_{i=1}^k)$.

$$\begin{array}{lcl} \mathsf{end}\, x_{i=1}^k & \widehat{=} & \exists\, x'^k_{i=1} \bullet \mathsf{skip}_A^{\mathrm{D}} \\ & = & \mathsf{true} \vdash \exists\, x'^k_{i=1} \bullet \mathsf{skip}_{A1} \end{array}$$

Here, the alphabet of the command as a whole, i.e. $\alpha(\mathsf{end}\, x_{i=1}^k)$, is $A \setminus \{x_{i=1}^k\}$ and the alphabet $A1 = A \setminus \{\Pi_{\mathrm{OK}}, \Pi_{\mathrm{OK}}'\}$. If $k < 1$ then an empty variable list is completed in a similar manner to that of variable introduction; i.e. it is considered to have the same effect as $\mathsf{skip}_A^{\mathrm{D}}$. The proof that this command produces a design is presented in Appendix C.1.3.

When a program Q is composed with the command representing the completion of variable x's scope, then the scope of the existential quantification is extended to include program Q; i.e. $Q \fatsemi (\mathsf{end}\, x) = (\exists\, x' \bullet Q)$. This result is proved in Appendix C.2.2, in a similar manner to that of variable introduction.

Variable declaration command: In this dissertation local variables are used in a *traditional* block-scoping role; i.e. variables are introduced for a known limited scope. The following two block-scoping variable declaration commands are used to introduce an uninitialised collection and an initialised map of local variables respectively.

$$\begin{array}{lcl} \mathsf{decl}\, x_{i=1}^k \,\mathsf{in}\, P & \widehat{=} & (\mathsf{var}\, x_{i=1}^k) \fatsemi P \fatsemi (\mathsf{end}\, x_{i=1}^k) \\ \mathsf{decl}\{{}_{i=1}^k\, x_i \mapsto v_i\} \,\mathsf{in}\, P & \widehat{=} & (\mathsf{var}\, x_{i=1}^k := v_{i=1}^k) \fatsemi P \fatsemi (\mathsf{end}\, x_{i=1}^k) \end{array}$$

Note that both of the syntactic forms of block scoping are suitable for a top-down structural free variable identification algorithm as previously discussed in Section 2.1.3.

Given that the block variable declaration commands are applied to designs, then the resulting program is also a design, as follows, where u and V stand for $x_{i=1}^k$ and $v_{i=1}^k$ respectively.

$$\begin{array}{lcl} \mathsf{decl}\, u \,\mathsf{in}\, p \vdash P & = & (\forall\, u \bullet p) \vdash (\exists\, u, u' \bullet P) \\ \mathsf{decl}\{{}_{i=1}^k\, x_i \mapsto v_i\} \,\mathsf{in}\, p \vdash P & = & p[V/u] \vdash (\exists\, u' \bullet P[V/u]) \end{array}$$

These results are proved in Appendix C.1.5.

2.3.4 Hiding and filtering variables from programs

It is sometimes useful to hide some variables from a subprogram. For example, if a subprogram happens to declare a local variable with the same name as a global variable that it does not use, then, according to the standard UTP rules on variable introduction, either the local or global variable must be renamed. Within this context such hiding would provide a convenience mechanism for not having to rename variables. However, there are contexts where it is important to reduce the alphabet of a subprogram so that it can conform to external constraints. For example, in the context of recursion it is important that the alphabets either side of an unwinding of the fix-point are the same. The ability to hide unwanted local variables from the fix-point iteration then becomes a rather more important issue.

Hide variables command: The hide variables command $\mathsf{hide}\ x_{i=1}^k\ \mathsf{from}\ P$ removes the variables $x_{i=1}^k$ from the alphabet of P, but following P's completion restores these variables with their previous values.

$$\mathsf{hide}\ \_\ \mathsf{from}\ P \ \widehat{=}\ P$$

$$\mathsf{hide}\ x_{i=1}^k\ \mathsf{from}\ P \ \widehat{=}\ \exists\, y_{i=1}^k \mid y_{i=1}^k = x_{i=1}^k \bullet (\mathsf{end}\, x_{i=1}^k) \fatsemi P \fatsemi (\mathsf{var}\, x_{i=1}^k := y_{i=1}^k)$$

$$\alpha(\mathsf{hide}\ \{x_{i=1}^k\}\ \mathsf{from}\ P) = \{{}_{i=1}^k\, x_i, x_i'\} \cup \alpha P$$

where

$$\forall_{i=1}^{k}\ x_i \notin \alpha P \wedge y_i \notin \alpha P$$
$$out\alpha P = (in\alpha P)'$$

Note that the hide variables command is similar to that of the standard UTP alphabet extension command (P_{+A}), in the sense that both can be used to enable the sequential composition of a program with a smaller alphabet. The difference is that the hide variables command is specialised to work with designs, rather than general relations; applying the hide operator to a design $p \vdash P$ returns a design of the form $p \vdash P \wedge u' = u$, rather than $(p \vdash P) \wedge u' = u$, where u represents the vector of variables being hidden.

Lemma 2.6: Hide design

$$\mathsf{hide}\ u\ \mathsf{from}\ p \vdash P \quad = \quad p \vdash P \wedge u' = u$$

□

This lemma is proved in Appendix C.1.4.

Sometimes it is useful to limit a subprogram to a specific set of variables. This can be achieved by hiding all variables in the alphabet except for those of interest. For example, it is possible to limit the scope of subprogram P to the alphabet $\{u, u'\}$, within a program that has an alphabet $\{w, w'\}$, using the command

$$\mathsf{hide}\ w \lfloor u\ \mathsf{from}\ P$$

where $w \lfloor u$ is the vector w without the variables contained in vector u. One issue with this approach is that it is not possible to infer what the alphabet of the surrounding program should be. Therefore, any *compilation scheme* that used this form of hiding would become context dependant, in the sense that it would have to maintain the program's current alphabet. Our original UTP constant–map-design ($\mathcal{U}_{\mathrm{M}}$-design) of the $\mathcal{O}$-calculus did precisely this, in order to enable the definition of the method invocation command to appropriately reduce the scope of the program representing the target method (the current $\mathcal{U}_{\mathrm{M}}$-design is presented in Section 3.4). This significantly complicated the consistency proofs for this model, and thus support for a context independent compilation scheme was sought.

The adopted solution to the context independent compilation issue was inspired by the observation that scoped (hidden) subprograms were being sequentially composed with subprograms that could be compiled in a context independent manner – specifically that the alphabets of these independent subprograms could be used to infer the alphabet of a scoped subprogram. Here, the new UTP command would take three arguments: the independent subprogram; the list of variables to be kept; and the scoped subprogram.

Filter-in composition command: The left- and right-hand filter-in composition commands perform a sequential composition of subprograms P and Q, where only the variables contained in the vector u are allowed in the subprogram P.

$$P \restriction u \circledS Q \quad \widehat{=} \quad (\mathsf{hide}\ w \lfloor u\ \mathsf{from}\ P) \fatsemi Q$$
$$Q \circledS u \restriction P \quad \widehat{=} \quad Q \fatsemi (\mathsf{hide}\ w \lfloor u\ \mathsf{from}\ P)$$

Here, vector w contains all the variables in the input alphabet of Q, and $w \lfloor u$ is the vector w without the variables contained in vector u.

It is also possible to define left- and right-hand filter-out composition commands in a similar manner, though in this case it is not required for simplifying the compilation process.

Filter-out composition command: The left- and right-hand filter-out composition commands perform a sequential composition of subprograms P and Q, where the variables contained in the vector u are removed from (hidden in) the subprogram P.

$$P \lfloor u \rfloor Q \mathrel{\widehat{=}} (\mathsf{hide}\ u\ \mathsf{from}\ P) \fatsemi Q$$
$$Q \lfloor u \rfloor P \mathrel{\widehat{=}} Q \fatsemi (\mathsf{hide}\ u\ \mathsf{from}\ P)$$

2.3.5 Higher order programming

Church's λ-calculus provides a simple, yet very powerful, model for the whole of computing, based on an abstract idea of what a function is [Chu36, Chu85, Pie02]. Briefly, the main idea is that of the lambda abstraction, i.e. "$f = \lambda(x)\ e$", where f is a single parameter function, whose formal parameter x may be used in the definition of its body (expression) e. Having provided a syntax for defining a function, the λ-calculus also provides a syntax for applying it, namely "$f\ x$", where the function f is applied to the argument x, which itself can be a function. It is this concept of applying a function to a function (i.e. treating a function as data or a value) that is given the term *higher order* in this context.

In general, the concept of higher order programming can be used to enable programs – such as macros, procedures and subroutines – to be treated as data. The important point is that suitable program variables can have inner programs assigned to them. Chapter 9 of the UTP book [HH98] provides a succinct introduction to these concepts, from both imperative and functional language points of view, which can be adapted to model OO-methods. Having said this, some would argue that [HH98] simulates, rather than provides, higher ordered programming to the UTP. Here, functions (and procedures) are modelled in terms of predicate relations, rather than being a first class concept. Specifically they are modelled as UTP predicate texts that are extracted when needed.

Before continuing with the discussion on higher order programming we illustrate how higher order programming can be simulated in a set-theoretic context, such as provided by the Z specification language [ISO02, Spi92, WD96].

Example 2.7: Suppose that a programming language has two basic data types, the natural numbers ($\mathbb{N}$) and the booleans ($\mathbb{B}$). Further, let there be two operations for constructing new data types from old, namely, the usual power-set ($\mathbb{P}\_$) and Cartesian product ($\_ \times \_ \ldots \times \_$) type constructors. This enables the type of an anonymous record (a tuple) that contains a natural number followed by a boolean to be represented by $\mathbb{N} \times \mathbb{B}$. Now a function *even* can be modelled by a set of such tuples, one for each natural number n, where: the first element of the tuple is set to n, and the second is set to $true_{\mathbb{B}}$ when n is even and the value $false_{\mathbb{B}}$ otherwise. Formally this can be defined by $even \mathrel{\widehat{=}} \{(n, n \bmod 2 = 0) \mid n \in \mathbb{N}\}$, where the *even* function can be represented as an element of the type $\mathbb{P}(\mathbb{N} \times \mathbb{B})$.
□

The previous example demonstrates how a function can be modelled by a set of pairs. In general this modelling technique can be extended to any relation, and illustrates the key insight behind the simulation of higher order functionality in the UTP.

Object-oriented modelling: Two teams have made significant progress with adding class-based OO-support to the UTP. They have both used higher order programming concepts, in the sense that object values essentially contain the methods (functions) that can be invoked on them. The first team's work [HLL02a, HLL02b, HLXS04, HLL06, CHH+07] has resulted in the development of the Refinement Calculus of Object Systems (rCOS) language; it supports object references. The second team's work [CSW05, SCS06, CHW06, HCW08] is being used to extend the Circus language; here the work on objects and pointers has been done independently, but with a view to merging the results.

Our work is similar to these in that we use higher order concepts to represent methods as designs, whose semantics is determined by an appropriate weakest fix-point construction. The key difference, is that in our approach methods are not associated with a class, but with an object.

2.4 Heap Storage Locations

Sharing (*aliasing*) objects is supported by most mainstream OO-languages, such as C^{++} and Java. Here objects are dynamically created via the new *ClassName*(*parameters*) constructor, which allocates space on the heap for the object of type *ClassName*, initialises its fields using the supplied *parameters*, and then returns a pointer (object reference) to the location (address) on the heap where the object is stored. Such object references can then be assigned (copied) to other variables, and thus the objects that they are pointing at can be aliased (shared). Appendix A.3 contains a slightly more detailed discussion on aliasing.

An object's location can be considered to represent its unique *identity*, as once created an object retains its heap location (identity) for the duration of its existence; in other words the object's identity is immutable. Some consider that "object identity is the foundation of object-oriented programming" [NVP98]. Our $\mathcal{O}$-calculus does not support this notion of unique object identity. Instead, it supports an explicit notion of *self* that is independent of where the object is located. From our point of view, heap storage provides a means for sharing objects, such as required by typical implementations of the doubly–linked-list.

Example 2.8: A record-like object for representing both an unset node and an example middle node in a doubly–linked-list could be defined as follows in our $\mathcal{O}$-calculus.

$$\begin{array}{lcl} \mathit{UnsetNode} & \equiv & [\ \mathtt{data} = \varsigma(\_)\ \text{¿} \\ & & \ \ \mathtt{next} = \varsigma(\_)\ \mathsf{nil} \\ & & \ \ \mathtt{prev} = \varsigma(\_)\ \mathsf{nil} \\ & &] \end{array} \qquad \begin{array}{lcl} \mathit{MidNode} & \equiv & [\ \mathtt{data} = \varsigma(\_)\ 42 \\ & & \ \ \mathtt{next} = \varsigma(\_)\ \ell_{57} \\ & & \ \ \mathtt{prev} = \varsigma(\_)\ \ell_{15} \\ & &] \end{array}$$

Here, the *MidNode* node in the middle of the list is shared by its two adjacent nodes (at locations ℓ_{57} and ℓ_{15}), where the preceding node's (ℓ_{15}'s) `next` field and the following node's (ℓ_{57}'s) `prev` field both contain an object reference to the *MidNode* node.
□

Semantics: Two broad categories of semantic modelling approaches are value semantics and location (or reference) semantics. Here a value semantics is typically written in terms of a map from variables to values, whereas a location semantics is typically written in terms of two maps, from variables to locations and locations to values. Having said this, some variants of a location semantics simply add a map from locations to values and treat a location as a value.

Alternatively, the layout of storage can be viewed as a directed graph. One possibility is to view variables and values as nodes within the graph. Here the node representing a compound value, such as a record, would have an outbound edge for each of its fields, which is labelled by the distinct field name, to the value for that field. This corresponds to the concrete graph of locations presented in Section 5.2. Another possibility is to view the nodes as locations and the edges as the values. This corresponds to the abstract graph of locations presented in Section 5.3. Further, these abstract graphs underpin the fully abstract model of memory, that is used by our last UTP model $\mathcal{O}$-calculus.

Chapter 3

Unifying Theories of Objects

This chapter presents a family of three Unifying Theories of Programming (UTP) models of our $\mathcal{O}$-calculus:

1. the UTP operand–stack-design ($\mathcal{U}_{\mathrm{S}}$-design), which stores the result of evaluating an operand, i.e. a subprogram, on an operand stack;
2. the UTP result–value-design ($\mathcal{U}_{\mathrm{R}}$-design), which replaces this stack with a combination of local variables and a result value; and
3. the UTP constant–map-design ($\mathcal{U}_{\mathrm{M}}$-design), which extends the use of variables so that they represent the direct and inherited parameters of function calls and method invocations.

These models extend the notion of a UTP design with values and commands for modelling objects, methods, and pointers (references). Here the approach is to convert each $\mathcal{O}$-calculus expression into a UTP command that denotes its semantics.

Aside 3.1: The reason that an $\mathcal{O}$-calculus expression is transformed into a UTP command, rather than a UTP expression, is because programs are represented by expressions in the $\mathcal{O}$-calculus and commands in the UTP.
□

The notion of a *command-expression* is introduced to systematically transform the majority of the $\mathcal{O}$-calculus's expressions into equivalent UTP programs, as follows:

1. transform each argument in the expression to a UTP program that calculates its result;
2. evaluate the arguments in the same order as that of the $\mathcal{O}$-calculus's operational semantics;
3. store the evaluated arguments (e.g. in an operand stack or some local variables); and
4. apply the underlying $\mathcal{O}$-calculus operation on the stored values, and store the result.

Item 4 is handled by a model specific *transformation-command*, which applies the $\mathcal{O}$-calculus operation to the evaluated arguments and appropriately stores the result as a UTP program. Specifically, a UTP program that produces the given result.

Some operations such as method invocation do not follow this model, as they require more sophisticated handling, in this case the use of fix-points.

3.1 Extending the UTP design

The UTP notion of a design is extended so that it can model $\mathcal{O}$-calculus entities such as objects, labels, methods, functions, and abstract locations (along with their associated heap).

3.1.1 Objects and labels

An object is modelled by a partial map from labels to method definitions, where a label is denoted by an alphanumeric identifier and a method as a tuple of program-texts (Section 3.1.2). This mirrors the approach of the $\mathcal{O}$-calculus, thus has similar benefits and limitations. The main benefit is that all members of an object have the same form: they are methods. The main limitation is that it is difficult to distinguish a method that represents a field from a method whose body consists of a single – possibly compound – value.

Aside 3.2: Originally we planned to model an object as a partial map from object member-labels to method-labels, which were used to index a global method table. The idea was that this table would be useful for the fix-point definition of a mutually recursive method's semantics. A benefit of this approach was that an object only contained scalar values, i.e. member and method labels, rather than the compound method definitions and object values. However, the indirection through a global method table proved to be both unnecessary and a hindrance. First, it introduced the problem of naming methods, and managing these names. Second, the extra indirection complicated the definition of method invocation, by introducing two extra steps in the following invocation process – specifically steps 2 and 3.

1. check whether the object contained the member-label;
2. get the corresponding global method-label;
3. check whether this method-label is in the global method table;
4. get the corresponding method;
5. call the method with the object (in step 1) as its self-parameter.

Only stages 1, 4, and 5 are required in our current model of method invocation.
□

3.1.2 Methods and functions

Within the higher-order chapter of the UTP book [HH98] subroutines (e.g. functions) are essentially modelled by *program texts* that are unpacked (extracted) and then sequentially composed with their calling environment. An object's methods can be viewed in a similar manner. One important observation is that the calling context sets the method's initial alphabet. Therefore care must be taken to ensure that we know what the calling context's alphabet is. A simple approach to address this problem is to constrain the program alphabet at the point of a method (or function) call so that it only contains the UTP model's observation variables (and not any program variables). This approach is sufficient for our purposes and is implemented by each of our systematic compilation schemes. For convenience when we wish to talk about both a method and a function, we shall use the term procedure.

One significant consequence of having a constant alphabet at the point of a procedure call is that the initial alphabet of every procedure is identical. This uniformity is essential for the way in which mutually recursive procedures have been handled. Here, each model of the $\mathcal{O}$-calculus provides a single command for invoking a procedure, which identifies both the argument value and the program-text from the UTP model's state variables; that is, the values of the variables in the program's alphabet. This enables the definition of mutually recursive procedures to be given in terms of a fix-point over program states.

Aside 3.3: An alternative way of modelling the $\mathcal{O}$-calculus method would be to treat it in a similar manner to that of a single argument untyped lambda calculus (λ-calculus) function – specifically, that the method's object is passed by value as the argument to the function, which is then treated as an immutable parameter (constant) within its definition. Inner methods could then be defined in a similar manner to the way in which inner functions are defined in

the λ-calculus. Here, the parameters containing the immutable object arguments of the outer scope methods are available for use within a inner method, as read-only global variables. These global variables must be included within the alphabets of the methods that use them, but not in the alphabets of the methods that declare (introduce) them. The important point is that this approach leads to methods which have different alphabets, and thus do not conform to the limitations that we have chosen to adopt.
□

Within each of our UTP object models, non-local variables (e.g. from an outer method self-variables or function parameters) must be either substituted for their values prior to their procedure's invocation or passed as part of their procedure's invocation. This ensures that any references to the parent's formal parameter are available for use within the inner procedure invocation.

Example 3.4: In Section 2.2.5 we introduced a *Factorial* object that contained a single method, which made use of an anonymous inner function. This is now reproduced for our convenience.

$$\mathit{Factorial} \quad \widehat{=} \quad [\texttt{fact} = \varsigma(x)\ \lambda(i)\ \mathsf{if}\ i < 2\ \mathsf{then}\ 1\ \mathsf{else}\ i \times x.\texttt{fact}(i-1)]$$

Note that the anonymous inner function body (i.e. $\mathsf{if}\ i < 2\ \mathsf{then}\ 1\ \mathsf{else}\ i \times x.\texttt{fact}(i-1)$) contains a reference to the outer method's self parameter (x). It is this outer variable (x) that needs to be substituted for its value or passed to the inner procedure as part of its invocation.
□

The simplest approach to the non-local variable issue is to ensure that the procedure invocation processes substitute (a syntactic variant of) their argument's value for their formal parameter in their *program-text*, prior to its extraction. Having said this, there are a couple of hurdles to overcome:

1. the ability to identify all *free* occurrences of the variable (parameter) within a program-text, so that only these get substituted;
2. the ability to transform any semantic value, which can be passed as an argument to a method or function, into a suitable program text.

The standard UTP technique for overcoming the first hurdle is to ensure that inner bound variables never have the same name as a free (alphabet) variable. This constraint causes problems with the $\mathcal{O}$-calculus compilation process to the UTP. It can be easily overcome by restricting our program texts to a UTP syntax that has a convenient block scoping structure. Having restricted our program texts to this convenient form, it is straightforward to define a free-variable substitution processes, for each of our UTP models, in a manner similar to that of the $\mathcal{O}$-calculus (Section 2.2.5).

The second hurdle is also straightforward to overcome due to the form of the values within our model. Here, most values are their own texts (e.g. the literal 3). Functions (and methods) are represented by a tuple of texts, which already conform to the variable scoping compilation restriction, thus their textual representation can be this tuple. Objects are represented as a map from labels to methods, which we textually denote by a set of pairs, where labels are their own texts and methods are tuples of texts.

3.1.3 Abstract locations and the heap

The $\mathcal{O}$-calculus directly models the heap as a map from abstract locations to values. This map is stored as an environmental context variable, which is updated as a *side effect* of reducing a term, using the small step reduction rules. Having said this, the main purpose of the rules for creating, updating, and dereferencing heap locations is to manage such side effects.

Before continuing with our discussion on the modelling of the $\mathcal{O}$-calculus, it is worth recalling that its fresh operation has the side effect of updating the HEAP context variable. This side effect has important implications on the ordering of evaluation as illustrated by the following example.

Example 3.5: Consider creating a zero-initialised fresh abstract location, whose value (not its contents) is used to initialise two fields of a *new* object.

$$\mathsf{let}\ loc \mathrel{\widehat{=}} (\mathsf{fresh} *= 0)\ \mathsf{in}\ [\mathtt{f1} = \varsigma(\_)\ loc, \mathtt{f2} = \varsigma(\_)\ loc]$$

Note that it is not possible to replace the definition of the location variable *loc* in the body of the local block, as this change in execution order would result in fields `f1` and `f2` pointing to different abstract locations, rather than sharing the same abstract location. Essentially, this is because the fresh command takes an environmental parameter, which is updated on each use. Therefore, the UTP predicate that formally models the operational semantics of the fresh command needs to be parametrised by this environmental information.

If two distinct fresh locations were required then the following, subtly different, $\mathcal{O}$-calculus program could be used.

$$\mathsf{let}\ loc \equiv (\mathsf{fresh} *= 0)\ \mathsf{in}\ [\mathtt{f1} = \varsigma(\_)\ loc, \mathtt{f2} = \varsigma(\_)\ loc]$$

Here, the only difference is in the let variable-binder which changes from definition ($\widehat{=}$) to macro-expansion ($\equiv$).
□

The simple heap approach presented in this chapter mirrors that of the $\mathcal{O}$-calculus. Therefore, this chapter's UTP models of a heap all inherit the *garbage* detection and collection issues of the $\mathcal{O}$-calculus; that is, the issues with being able to detect and delete locations on the (abstract) heap that cannot be referenced by the program's execution state. Note that the operational semantics of the $\mathcal{O}$-calculus ignores such garbage collection issues, as this complication was not necessary for illustrating the usefulness of the calculus. Within the context of the UTP models this issue is perhaps more important, as we wish to be able to say when one program is equal to another, and this will involve a comparison of abstract heap locations.

A garbage collection mechanism can be added to the $\mathcal{O}$-calculus, via a function that examines the current usage of the heap by the terms representing a program and its context, and removes the unreferenceable locations. A similar approach could be adopted for the first of our UTP models, but not the latter two, as they use existential quantification to temporarily hide references to the heap (whereas the first does not). Chapter 5 provides an alternative – fully abstract – model of a heap, which is used to address these issues. It is based on the idea of representing store as a directed graph, which has anonymous nodes with distinctly labelled outbound edges. Section 3.4.3 presents an alternative compilation scheme which ensures that all compound terms (e.g. objects and methods) have distinctly named components, and thus can be directly represented by such a directed graph.

3.1.4 Modelling notes

Badly behaved programs: A program is considered to be *badly behaved* if it does not successfully terminate. Infinitely recursing (looping) programs are badly behaved and are denoted by the chaos command in each of our UTP object models, which is the weakest fixpoint of a non-terminating design [HH98, WC02]. Therefore, chaos appears to be a good choice for modelling badly behaved programs. However, an $\mathcal{O}$-calculus program can also be badly behaved by becoming *stuck*.

Aside 3.6: Recall that an $\mathcal{O}$-calculus program's operational semantics are written in terms of a collection of – non overlapping – small-step reduction rules. Therefore, given a program term, at most one rule can be applied to the whole term (rather than one of its subterms). If no rules can be applied, one of two conditions can hold: either the term representing the program is now a value, in which case it has successfully terminated, or else the program has become stuck (and thus is badly behaved).
□

The standard UTP approach is to not distinguish between different types of badness, and thus model stuck $\mathcal{O}$-calculus programs by chaos. This is what we have done.

Note that the dynamic conditions under which a program becomes stuck are identifiable, as they correspond to when the arguments of an $\mathcal{O}$-calculus term have the wrong form (e.g. type). In principle, the form of the arguments could be explicitly checked for within our UTP models, which in turn would enable them to generate a different sort of failure. This might be useful in the future, if the distinction between stuck and non-terminating is required. Currently, some of the checks are implicitly captured in the pre-conditions of the designs we use to model the $\mathcal{O}$-calculus operations.

UTP program texts and compilation: Both UTP operations and arithmetic expressions are typically considered to directly denote semantic entities, rather than specific syntactic forms, which are then provided with a semantics.

Example 3.7: The arithmetic expressions $x \times 2$ and $x + x$ both denote the same mathematical entity, thus they can be substituted for each other in any (semantic) context. Here, it is assumed that the syntactic presentation of such a substitution automatically adds brackets as necessary (in order to prevent change of meaning due to the precedence of surrounding operators).

The UTP relational predicates $x := 4$ and $x := 2 \fatsemi x := x + x$ both denote the same mathematical entity, thus they can be substituted for each other in any (semantic) context, in a similar manner to that of the arithmetic example.
□

Within our models of the $\mathcal{O}$-calculus methods are represented by UTP program texts whose syntactic form has the block scoping property; i.e. the scope of a bound variable can be limited to the children of the term that introduces the variable.

One consequence of introducing the concept of a UTP program text, is that we need to be able to convert between such texts and their semantic entities. Converting a UTP program text to its corresponding semantic entity, is straight forward; it is denoted by the usual semantic meaning brackets as follows

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

Here, t is a *valid* output of the program compilation process, which ensures that the syntactic form of its terms have the block scoping property. The reverse process of taking a semantic entity and producing a syntactic representation of is defined, in Section 3.2.5, by the text_{J} operation (where J is the UTP model identifier).

Having provided some notation for converting between syntax and semantics, we introduce a notation for defining the systematic transformation (i.e. a compilation) of an $\mathcal{O}$-calculus program into the corresponding UTP program. This compilation process is denoted by $\langle\!\langle t \rangle\!\rangle^{\text{M}}_{\text{J}}$, where t is the $\mathcal{O}$-calculus term to be compiled, m is the optional – staged compilation – mode, and J is the UTP model identifier.

We also provide a shorthand for the presenting the UTP semantics of an $\mathcal{O}$-calculus term as follows:

$$[\![\langle\!\langle t \rangle\!\rangle^{\text{M}}_{\text{J}}]\!] \;\widehat{=}\; [\![\, [\![\langle\!\langle t \rangle\!\rangle^{\text{M}}_{\text{J}}]\!] \,]\!]$$

3.2 Operand Stack Model

The UTP operand–stack-design ($\mathcal{U}_{\text{S}}$-design) 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}, Π_{STK}', and Π_{HEAP}', which represent the input and output states (values) of the operand stack and heap storage respectively.

The contents of the stack are the values (e.g. the integers, objects, functions, and heap locations) and other semantic entities (i.e. labels and methods), which between them represent the operands of the $\mathcal{O}$-calculus operations. 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 $\mathcal{O}$-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.

3.2.1 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 $\mathcal{O}$-calculus program must result in the $\mathcal{U}_{\text{S}}$-design that pushes a single value onto the operand stack. This is achieved by the following $\mathcal{U}_{\text{S}}$-design evaluation ($\mathcal{E}_{\text{S}}$), which pushes an operand value (ov) onto the operand stack.

$$\mathcal{E}_{\text{S}}\, ov \;\widehat{=}\; \{\Pi_{\text{STK}}\} : (\ \mathsf{true} \vdash \Pi_{\text{STK}}' = \langle ov \rangle \frown \Pi_{\text{STK}}\ )$$

Note that the evaluation command has been subscripted by the operand stack model identifier (i.e. S). The evaluation command in the other models will be subscripted by their model identifiers, so that these commands can be distinguished. In general, this form of subscripting is applied to all the model specific commands (and utility operations).

The compilation of a literal value lv to the $\mathcal{U}_{\text{S}}$-design 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.

$$\langle\!\langle lv \rangle\!\rangle_{\text{S}} \;\widehat{=}\; \mathcal{E}_{\text{S}} \langle\!\langle lv \rangle\!\rangle_{\text{S}}^{\text{E}} \qquad\qquad \langle\!\langle lv \rangle\!\rangle_{\text{S}}^{\text{E}} \;\widehat{=}\; lv$$

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

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

Here, t is a *valid* output of the program compilation process, which ensures that the syntactic forms of its terms have the *block scoping* property; i.e. the scope of a bound variable can be limited to the children of the term that introduces the variable. Therefore, the texts for standard UTP variable introduction and elimination commands (i.e. var and end) cannot be output as part of the compilation process. This is important for defining the free-variable substitution function (Section 3.2.6) using a structural induction over UTP program texts.

Before continuing with the model description it is worth illustrating the UTP semantics of a simple literal value program.

Example 3.8: The $\mathcal{U}_{\text{S}}$-design for the $\mathcal{O}$-calculus program in the 'value example' on page 14 is:

$$\begin{aligned} [\![42]\!]_{\text{S}} &= \mathcal{E}_{\text{S}} [\![42]\!]_{\text{S}}^{\text{E}} = \mathcal{E}_{\text{S}}\, 42 \\ &= \Pi_{\text{STK}} := \langle 42 \rangle \frown \Pi_{\text{STK}} = \{\Pi_{\text{STK}}\} : (\, \mathsf{true} \vdash \Pi_{\text{STK}}' = \langle 42 \rangle \frown \Pi_{\text{STK}}\,) \end{aligned}$$

Recall that the resultant value is that contained on the top of the stack.
□

3.2.2 Function values

In the $\mathcal{U}_{\mathrm{S}}$-design a function value is defined as a pair of compiled program texts that represent the function's parameter and body. It is denoted by $(\!|\ x, P\ |\!)$, where the scope of the parameter x is the program-text P.

$$\langle\!\langle \lambda(x)\ e \rangle\!\rangle_{\mathrm{S}} \ \widehat{=}\ \mathcal{E}_{\mathrm{S}}\langle\!\langle \lambda(x)\ e \rangle\!\rangle^{\mathrm{E}}_{\mathrm{S}} \qquad \langle\!\langle \lambda(x)\ e \rangle\!\rangle^{\mathrm{E}}_{\mathrm{S}} \ \widehat{=}\ (\!|\ \langle\!\langle x \rangle\!\rangle^{\mathrm{E}}_{\mathrm{S}}, \langle\!\langle e \rangle\!\rangle_{\mathrm{S}}\ |\!)$$

Here, a variable is represented by itself in the declaration (evaluation) context and the evaluation of itself in the general program context.

$$\langle\!\langle x \rangle\!\rangle_{\mathrm{S}} \ \widehat{=}\ \mathcal{E}_{\mathrm{S}}\langle\!\langle x \rangle\!\rangle^{\mathrm{E}}_{\mathrm{S}} \qquad \langle\!\langle x \rangle\!\rangle^{\mathrm{E}}_{\mathrm{S}} \ \widehat{=}\ x$$

Note that the evaluation ($\mathcal{E}_{\mathrm{S}}$) of a variable is not defined by the $\mathcal{U}_{\mathrm{S}}$-design, but it may be contained in the function's body (i.e. a compiled program text). Such variables are substituted by their values, prior to the program text being extracted to a $\mathcal{U}_{\mathrm{S}}$-design subprogram. Details of the program text variable-substitution and extraction processes are presented in the discussion of function and method invocation (Section 3.2.6).

3.2.3 Object values and method definitions

In the $\mathcal{U}_{\mathrm{S}}$-design an object value is defined as a map from labels to methods. This is denoted by $\{^{k}_{i=1}\ l_i \mapsto m_i\}$, 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.

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

A method is defined as a pair of compiled program texts that represent the method's *self* variable and body. Like the function, it is denoted by $(\!|\ x, P\ |\!)$, where the scope of the *self* variable (parameter) x is the program text P. However, methods differ from functions in their compilation; specifically, a method cannot occur as a top level program as it is not considered to be a value, thus it only has an evaluation moded compilation scheme.

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

3.2.4 Command expressions

In the $\mathcal{O}$-calculus almost all the programming operations are expressions. Such expressions are converted into UTP commands that evaluate each of their arguments, store the results on the operand stack, and then apply an appropriate stack transformation command. This is modelled by the $\mathsf{cmdExp}_{\mathrm{S}}$ command, which applies a transformation program (command) P to the result of executing a list of programs $P^{k}_{i=1}$ – representing the eager evaluation of its arguments – in a left to right order.

$$\mathsf{cmdExp}_{\mathrm{S}}(P, (P^{k}_{i=1})) \ \widehat{=}\ (\fatsemi^{k}_{i=1}\ P_i) \fatsemi P$$

Note that some of the $\mathcal{O}$-calculus operations that we wish to transform have *unevaluable* arguments, such as the method update operation $e.l \Leftarrow m$, where the expression e should evaluate to an object containing an unevaluable label l, which indexes the method that is going to be replaced by the unevaluable method m. In general, the contents of unevaluable arguments are considered to be already evaluated, and are thus loaded on to the operand stack.

An example of a $\mathcal{U}_{\mathrm{S}}$-design stack transformation program is the $\mathsf{trans}_{\mathrm{S}}(f, k)$ command, which takes a k-parameter function f – that defines the operation being modelled – and constructs

a UTP program that applies this function to the top k elements 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 $x_{i=1}^{k}$ represent the arguments for function f, then the updated stack can be modelled by $\langle f(x_{i=1}^{k})\rangle \frown (\mathsf{tail}^k\ \Pi_{\mathrm{STK}})$, assuming that: it has started ($\Pi_{\mathrm{OK}} = \mathsf{true}$); there are sufficient arguments ($k \leq \#\Pi_{\mathrm{STK}}$); and these arguments are in the domain of the function being modelled ($x_{i=1}^{k} \Elt f$).

$$\begin{array}{lcl} \mathsf{trans}_{\mathrm{S}}(f, k) & \widehat{=} & k \leq \#\Pi_{\mathrm{STK}} \land (\Pi_{\mathrm{STK}}\ \mathsf{proj}\ k) \Elt f \\ & & \vdash \\ & & \Pi_{\mathrm{STK}}' = \langle f((\Pi_{\mathrm{STK}}\ \mathsf{proj}\ k))\rangle \frown (\mathsf{tail}^k\ \Pi_{\mathrm{STK}}) \\ (s\ \mathsf{proj}\ k) & \widehat{=} & (_{i=1}^{k}\ \mathsf{head}(\mathsf{tail}^{k-i}\ s)) \end{array}$$

Example 3.9: The $\mathcal{O}$-calculus addition operation can be modelled in terms of the $\mathsf{cmdExp}_{\mathrm{S}}$ operation as follows:

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

□

Note, the definition of the $\mathsf{cmdExp}_{\mathrm{S}}$ generalises the one presented in [SG07] by removing the limitation that only the $\mathsf{trans}_{\mathrm{S}}$ command could be used to specify the appropriate transformation. It was generalised in order to minimised the difference between the UTP object models in this chapter and the one in Section 6.2.

3.2.5 Method updates and field assignments

Method update in the $\mathcal{O}$-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.

$$\begin{array}{lcl} \langle\!\langle e.l \Leftarrow m \rangle\!\rangle_{\mathrm{S}} & \widehat{=} & \mathsf{cmdExp}_{\mathrm{S}}(\\ & & \quad \mathsf{trans}_{\mathrm{S}}(methUpd, 3), \\ & & \quad (\ \langle\!\langle e \rangle\!\rangle_{\mathrm{S}},\ \mathcal{E}_{\mathrm{E}}\langle\!\langle l \rangle\!\rangle_{\mathrm{S}}^{\mathrm{E}},\ \mathcal{E}_{\mathrm{E}}\langle\!\langle m \rangle\!\rangle_{\mathrm{S}}^{\mathrm{E}}\ ) \\ & &) \\ methUpd & \widehat{=} & \{\ (o, l, m) \mapsto o \oplus \{l \mapsto m\} \mid \\ & & \quad (o, l, m) \in Object \times Label \times Method \land l \Elt o \\ & & \} \end{array}$$

Note that the *methUpd* function can never be called outside its domain, as it is always used in the context of the $\mathsf{trans}_{\mathrm{S}}$ UTP program, which checks to see if the arguments are in the domain of the transformation function to be used – in this case *methUpd*; if the arguments are in the domain, then the transformation function is applied and the result stored on the top of the operand stack, otherwise the program is defined to become chaotic (i.e. become the chaos program).

A field assignment is compiled in a similar manner, except for the use of a utility function (text_S) for converting a variable, a value, a method, and some programs to their corresponding program texts.

$$\begin{aligned}
\langle\!\langle\, e_1.l := e_2 \,\rangle\!\rangle_\mathrm{S} \;\widehat{=}\; & \mathsf{cmdExp}_\mathrm{S}(\\
& \quad \mathsf{trans}_\mathrm{S}(\mathit{fldUpd}, 3), \\
& \quad (\, \langle\!\langle\, e_1 \,\rangle\!\rangle_\mathrm{S},\ \mathcal{E}_\mathrm{S}\langle\!\langle\, l \,\rangle\!\rangle^\mathrm{E}_\mathrm{S},\ \langle\!\langle\, e_2 \,\rangle\!\rangle_\mathrm{S} \,) \\
&) \\
\mathit{fldUpd} = \{\ & (o, l, v) \mapsto o \oplus \{l \mapsto (\!|\ _, \mathsf{text}_\mathrm{S}(v)\ |\!)\} \mid \\
& \quad (o, l, v) \in \mathit{Object} \times \mathit{Label} \times \mathit{Value} \wedge l \notin o \\
\}&
\end{aligned}$$

Recall that in Section 3.1.4 we introduced the text_S helper operation, and said that it takes an UTP semantic entity and produces a textual representation of it. What we did not say was what such a textual representation would be.

In Section 3.1.2 we provided an outline of how operand values could be converted back into their textual forms. Here, variables and atomic operand values (which include object labels) are their own texts. Procedures are represented by a tuple of texts, which already conform to the variable scoping compilation restriction, thus their textual representation can be this tuple. Objects are represented as a map from labels to methods, which we textually denote by a set of label-method pairs.

In addition to these operand-values we provide textual representations for those programs that are supplied as a fix-point argument (as discussed in Section 3.2.6). For the moment it is sufficient to say that these can be constructed solely from the call_S, $\mathsf{apply}_\mathrm{S}$ and chaos programs, each of which can be represented by its own name, within a program text. Further, as we do not need to convert other semantic entities into program texts, no definition for these semantic entities is given.

It is now possible to indirectly define the text_S operation by a set of equations it must obey, as follows:

$$\begin{array}{lcl}
[\![\, \mathsf{text}_\mathrm{S}(x) \,]\!] & = & x \\
[\![\, \mathsf{text}_\mathrm{S}(av) \,]\!] & = & av \\
[\![\, \mathsf{text}_\mathrm{S}((\!|\ x, t\ |\!)) \,]\!] & = & (\!|\ x, t\ |\!) \\
[\![\, \mathsf{text}_\mathrm{S}([\![\, \{^k_{i=1}\, l_i \mapsto m_i\}) \,]\!] & = & \{^k_{i=1} \mathsf{text}_\mathrm{S}([\![\, l_i \,]\!]) \mapsto \mathsf{text}_\mathrm{S}([\![\, m_i \,]\!])\} \\
[\![\, \mathsf{text}_\mathrm{S}(\mathsf{call}_\mathrm{S}) \,]\!] & = & \mathsf{call}_\mathrm{S} \\
[\![\, \mathsf{text}_\mathrm{S}(\mathsf{apply}_\mathrm{S}(z)) \,]\!] & = & \mathsf{apply}_\mathrm{S}(\mathsf{text}_\mathrm{S}([\![\, z \,]\!])) \\
[\![\, \mathsf{text}_\mathrm{S}(\mathsf{chaos}) \,]\!] & = & \mathsf{chaos}
\end{array}$$

Note that the reason that this operation is not defined constructively is because more than one program text can represent the same semantic entity. For example, the identity function $(\!|\ x, \mathcal{E}_\mathrm{S}\, x\ |\!)$, can also be written as $(\!|\ y, \mathcal{E}_\mathrm{S}\, y\ |\!)$. Here, the proof that these to functions are semantically equal would involve demonstrating that for every context Γ and for all values v the equation $(\!|\ x, \mathcal{E}_\mathrm{S}\, x\ |\!)(v) = (\!|\ y, \mathcal{E}_\mathrm{S}\, y\ |\!)(v)$ holds. In practise this is not a problem for our proofs, as our textual denotation of the semantic entities mirrors that of the above equations, and we choose the obvious identity like conversion to a program text; that is for $\mathsf{text}_\mathrm{S}([\![\, pt \,]\!])$, we choose the program text pt. We could have chosen any other program text pt', so long as $[\![\, pt' \,]\!] = [\![\, pt \,]\!]$, but then we would have to prove that this equation holds.

3.2.6 Method and function invocation

Method invocation in the $\mathcal{O}$-calculus is compiled in two parts. First an object-member pair is constructed from the invocation arguments: an expression (e) representing the target 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_S). It executes the body of the method where the method's self variable has been instantiated with the calling object's value.

$$\begin{array}{lcl} \langle\!\langle e.l \rangle\!\rangle_\mathrm{S} & \widehat{=} & \mathsf{cmdExp}_\mathrm{S}(\\ & & \quad \mathsf{trans}_\mathrm{S}(omPair, 2), \\ & & \quad (\langle\!\langle e \rangle\!\rangle_\mathrm{S}, \langle\!\langle l \rangle\!\rangle_\mathrm{S}) \\ & &) \fatsemi \mathsf{call}_\mathrm{S} \\ omPair & \widehat{=} & \{(o, l) \mapsto (o, o(l)) \mid (o, l) \in Object \times Label \wedge l \in o\} \end{array}$$

Function invocation is compiled in a similar manner. The key difference is that the function's argument is a general expression, rather than a label.

$$\begin{array}{lcl} \langle\!\langle e_1\, e_2 \rangle\!\rangle_\mathrm{S} & \widehat{=} & \mathsf{cmdExp}_\mathrm{S}(\\ & & \quad \mathsf{trans}_\mathrm{S}(vtPair, 2), \\ & & \quad (\langle\!\langle e_1 \rangle\!\rangle_\mathrm{S}, \langle\!\langle e_2 \rangle\!\rangle_\mathrm{S}) \\ & &) \fatsemi \mathsf{call}_\mathrm{S} \\ vtPair & \widehat{=} & \{(v, pt) \mapsto (pt, v) \mid (pt, v) \in ProgramText \times Value\} \end{array}$$

Before formally defining the generic call_S command, it is worth presenting three helper functions for operating with program texts – specifically, one for substituting value-texts for variable-texts, one for substituting a call-text with a fix-point value-text, and one for extracting a program text (i.e. converting a program text into a $\mathcal{U}_\mathrm{S}$-design). These functions are defined by cases, where the first case that matches is taken.

The program text substitution function ($t\{\!|x \leftarrowtail ov|\!\}$) performs the same role as that of its $\mathcal{O}$-calculus counterpart, in that it replaces all free occurrences of the program variable x with the operand value ov in the program text t.

$$\begin{array}{lcl} x\{\!|x \leftarrowtail ov|\!\}_\mathrm{S} & \widehat{=} & ov \\ (\!|\, x, t \,|\!)\{\!|x \leftarrowtail ov|\!\}_\mathrm{S} & \widehat{=} & (\!|\, x, t \,|\!) \\ t\{\!|^k_{i=1}\, t_i|\!\}\{\!|x \leftarrowtail ov|\!\}_\mathrm{S} & \widehat{=} & t\{\!|^k_{i=1}\; t_i\{\!|x \leftarrowtail ov|\!\}_\mathrm{S}\; |\!\} \end{array}$$

The fix_S function constructs a program text by substituting the terms representing a procedure invocation (call_S) with the program text representing the instantiation of the fix-point variable z. Note that this does not directly substitute the call_S texts within inner procedures; inner procedures texts are appropriately updated by the procedure application command $\mathsf{apply}_\mathrm{S}$, which is defined at the end of this section.

$$\begin{array}{lcl} \mathsf{fix}_\mathrm{S}(\mathsf{call}_\mathrm{S}, z) & \widehat{=} & z \\ \mathsf{fix}_\mathrm{S}(\!|\, x, pt \,|\!) & \widehat{=} & (\!|\, x, pt \,|\!) \\ \mathsf{fix}_\mathrm{S}(t\{\!|^k_{i=1}\, t_i|\!\}, z) & \widehat{=} & t\{\!|^k_{i=1}\, \mathsf{fix}_\mathrm{S}(t_i, z)|\!\} \end{array}$$

The following ext_S function extracts a method's program text (t) given that there is a program text to represent the value of fix-point variable (z).

$$\mathsf{ext}_\mathrm{S}(t, z) \;\widehat{=}\; [\![\, \mathsf{fix}_\mathrm{S}(t, \mathsf{text}_\mathrm{S}(z)) \,]\!]$$

It turns out that the value of the fix-point variable can always be represented in the form $\mathsf{apply}^i_\mathrm{S}(\mathsf{chaos})$, where i is a natural number. Here, non-terminating programs are represented by $i = 0$ (i.e. chaos) and terminating programs by $n \geq 1$, where n is the number of method and function invocations that are actually invoked. In either case, programs of this form are handled by the existing program text generation function (text_S).

We are now in a position to define the call_S command. It is defined as the least fix-point of the $\mathsf{apply}_\mathrm{S}$ function, which substitutes the argument v for the parameter x in program text pt.

$$\begin{array}{lcl}
\mathsf{call}_\mathrm{S} & \widehat{=} & \mu\, z \bullet \mathsf{apply}_\mathrm{S}(z) \\
\mathsf{apply}_\mathrm{S}(z) & \widehat{=} & (\ \exists\, v, x, pt \mid (v, ((\!|\ x, pt\ |\!))) = \mathsf{head}\,\Pi_{\mathrm{STK}} \bullet \\
 & & \qquad \mathsf{pop}_\mathrm{S} \fatsemi \mathsf{ext}_\mathrm{S}(pt\{\!|x \longmapsto \mathsf{text}_\mathrm{S}(v)|\!\}_\mathrm{S}, z) \\
 & &) \\
 & & \lhd \#\Pi_{\mathrm{STK}} > 0 \land (\mathsf{head}\,\Pi_{\mathrm{STK}}) \in \mathit{Value} \times \mathit{ProgramText} \rhd \\
 & & \mathsf{chaos} \\
\mathsf{pop}_\mathrm{S} & \widehat{=} & \{\Pi_{\mathrm{STK}}\} : (\#\Pi_{\mathrm{STK}} > 0 \vdash \Pi_{\mathrm{STK}}' = \mathsf{tail}\,\Pi_{\mathrm{STK}})
\end{array}$$

3.2.7 Other programming mechanisms

Conditional Execution: The UTP provides a notion of conditional execution, where a boolean expression is used to select the appropriate subprogram. It differs from the $\mathcal{O}$-calculus's conditional execution operator in two significant ways:

1. The $\mathcal{O}$-calculus allows the execution of an arbitrary program to determine the value of its boolean control argument, whereas the UTP conditional command only allows a predicate that is completely defined in terms of the conditional's input alphabet (i.e. a UTP conditional).

2. The evaluation of the $\mathcal{O}$-calculus conditional expression need not produce a boolean value (or for that matter any result), whereas the UTP conditional expression is required to be both a predicate and defined.

The first of these differences is catered for by extracting the program that determines the value of the conditional expression and composing it with the UTP $\mathsf{switch}_\mathrm{S}$ program.

$$\langle\!\langle \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \rangle\!\rangle_\mathrm{S} \quad \widehat{=} \quad \langle\!\langle e_1 \rangle\!\rangle_\mathrm{S} \fatsemi \mathsf{switch}_\mathrm{S}(\langle\!\langle e_2 \rangle\!\rangle_\mathrm{S}, \langle\!\langle e_3 \rangle\!\rangle_\mathrm{S})$$

The $\mathsf{switch}_\mathrm{S}$ program essentially selects one of a pair of subprograms depending on the value on the top of the operand-stack. However, it also has to cater for the possibility that the evaluation of the conditional program did not produce a boolean value. In such cases, the corresponding $\mathcal{O}$-calculus operational semantics would become *stuck*, and thus fail to produce a useful result; recall that we have chosen to model all non-useful programs by the UTP chaos command.

$$\begin{array}{lcl}
\mathsf{switch}_\mathrm{S}(P_1, P_2) & \widehat{=} & (\mathsf{pop}_\mathrm{S} \fatsemi P_1 \lhd \mathsf{head}\,\Pi_{\mathrm{STK}} \rhd \mathsf{pop}_\mathrm{S} \fatsemi P_2) \\
 & & \lhd \#\Pi_{\mathrm{STK}} > 0 \land (\mathsf{head}\,\Pi_{\mathrm{STK}}) \in \mathit{Boolean} \rhd \\
 & & \mathsf{chaos}
\end{array}$$

Membership test: An example of the test that can precede a conditional execution is the object membership test $o?l$. Such a test is compiled using a command expression as follows.

$$\begin{array}{lcl}
\langle\!\langle e?l \rangle\!\rangle_\mathrm{S} & \widehat{=} & \mathsf{cmdExp}_\mathrm{S}(\\
 & & \quad \mathsf{trans}_\mathrm{S}((_ \Subset _), 2), \\
 & & \quad (\langle\!\langle l \rangle\!\rangle_\mathrm{S}, \langle\!\langle e \rangle\!\rangle_\mathrm{S}) \\
 & &)
\end{array}$$

Note that the order of the parameter evaluation has been swapped. This is fine because the program representing the evaluation of a label ($\mathcal{E}_\mathrm{S}\, l$) does not depend on or alter the environment; i.e. it is independent of the program representing the expression e.

Sequential Composition: The sequential composition of two expressions can be straightforwardly represented by the sequential composition of their compiled programs, where the result of the first program is ignored (i.e. popped from the operand stack).

$$\langle\!\langle e_1;\ e_2 \rangle\!\rangle_{\mathrm{S}} \ \widehat{=}\ \langle\!\langle e_1 \rangle\!\rangle_{\mathrm{S}} \fatsemi \mathsf{pop}_{\mathrm{S}} \fatsemi \langle\!\langle e_2 \rangle\!\rangle_{\mathrm{S}}$$

Local declaration: The compilation of a local declaration block uses a similar rewriting strategy as that of our $\mathcal{O}$-calculus (Section 2.2.4). Here, the *macro* declaration block is expanded (at compilation time), the *definition* declaration block is rewritten in terms of a function and its application, and the *list* declaration block is expanded.

$$\begin{aligned}
&\langle\!\langle \mathsf{let}\ x \equiv t\ \mathsf{in}\ e \rangle\!\rangle_{\mathrm{S}} \ \widehat{=}\ \langle\!\langle e_0 \rangle\!\rangle_{\mathrm{S}} \\
&\langle\!\langle \mathsf{let}\ x \mathrel{\widehat{=}} e_1\ \mathsf{in}\ e \rangle\!\rangle_{\mathrm{S}} \ \widehat{=}\ \langle\!\langle (\lambda(x)\ e)\ e_1 \rangle\!\rangle_{\mathrm{S}} \\
&\langle\!\langle \mathsf{let}\ d_{i=1}^{k}\ \mathsf{in}\ e \rangle\!\rangle_{\mathrm{S}} \ \widehat{=}\ \langle\!\langle \mathsf{let}\ d_1\ \mathsf{in} \\
&\qquad\qquad\qquad\qquad \mathsf{let}\ d_{i=2}^{k}\ \mathsf{in}\ e \rangle\!\rangle_{\mathrm{S}}
\end{aligned}$$

where $e_0 \mathrel{\widehat{=}} e\{\!|x \leftarrowtail t|\!\}$ and $k \geq 2$

Note that we precompute the macro substitution before its body is compiled to ensure that this does not get confused with the compilation of the substitution scheme that is presented in Chapter 4.

3.2.8 Modelling the heap operations

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

Fresh Operator: The $\mathcal{O}$-calculus fresh operation is compiled to its corresponding $\mathcal{U}_{\mathrm{S}}$-design command.

$$\langle\!\langle \mathsf{fresh} \rangle\!\rangle_{\mathrm{S}} \ \widehat{=}\ \mathsf{fresh}_{\mathrm{S}}$$

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.

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

Note that this operation deliberately uses the same fresh location generation function as in the $\mathcal{O}$-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 $\mathcal{O}$-calculus operation is compiled by evaluating the expression representing the heap location, then applying the $\mathcal{U}_{\mathrm{S}}$-design's command for dereferencing the current result.

$$\langle\!\langle *e \rangle\!\rangle_{\mathrm{S}} \ \widehat{=}\ \langle\!\langle e \rangle\!\rangle_{\mathrm{S}} \fatsemi \mathsf{deref}_{\mathrm{S}}$$

The heap dereference command ($\mathsf{deref}_{\mathrm{S}}$) takes the heap location on the top of the stack and replaces it with a copy of the associated heap value.

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

Heap Update Operator: The heap update $\mathcal{O}$-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.

$$\begin{array}{lcl} \langle\!\langle\, e_1 \mathrel{*}= e_2 \,\rangle\!\rangle_{\mathrm{S}} & \widehat{=} & \mathsf{cmdExp}_{\mathrm{S}}(\\ & & \quad \mathsf{trans}_{\mathrm{S}}(\mathit{lvPair}, 2), \\ & & \quad (\langle\!\langle\, e_1 \,\rangle\!\rangle_{\mathrm{S}}, \langle\!\langle\, e_2 \,\rangle\!\rangle_{\mathrm{S}}) \\ & &) \fatsemi \mathsf{update}_{\mathrm{S}} \end{array}$$

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

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

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 $\mathcal{O}$-calculus. This helps to highlight the semantic, rather than syntactic, differences between the models.

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.

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

3.3 Result Value Model

The $\mathcal{U}_{\mathrm{R}}$-design removes the need for an operand stack by storing intermediate results in local variables, which are introduced in the standard UTP way. The cost of this modelling approach is that the alphabets of subprograms are no longer uniform, and thus have to be managed to ensure that they can be composed. In particular, care has to be taken to ensure that the alphabet at the point of a recursive method invocation is the same as the alphabet of all existing method invocations. This restriction ensures that the sequential composition of the current program state with the unwinding of a method invocation is well defined.

In the $\mathcal{U}_{\mathrm{R}}$-design an object value is represented as a map from object labels to methods, where a method definition is a compiled – but unevaluated – program text. Such program texts are processed as part of the method invocation command.

In this model we introduce the Π_{OK} and Π_{RES} special modelling variables, where:

- Π_{OK} represents the starting and termination of a program;
- Π_{RES} represents the result of executing a program (or method).

The remainder of this section presents only the differences between this model and the previous $\mathcal{U}_{\mathrm{S}}$-design. Specifically, when the only action required to use a previous definition is to replace each old model identifier subscript (S) with this model's identifier subscript (R), then the definition is not reproduced.

3.3.1 Operand values

The values, labels, and methods used within a $\mathcal{U}_{\text{R}}$-design are the same as those used within a $\mathcal{U}_{\text{S}}$-design. One consequence of this is that both the definitions and compilation scheme for these operand-values are the same, up to renaming of model identifiers, for both models, and thus are not repeated here. However, the definition of the evaluation command ($\mathcal{E}_{\text{R}}$), which is used to model the simplest object calculus program (i.e. a literal value), needs to be updated to use the result-value (Π_{RES}) instead of the operand-stack (Π_{STK}). This is achieved by the following $\mathcal{U}_{\text{R}}$-design evaluation command, which assigns an operand value (ov) to the result-value (Π_{RES}).

$$\mathcal{E}_{\text{S}}\, ov \;\widehat{=}\; \Pi_{\text{RES}} := ov$$

Example 3.10: The $\mathcal{U}_{\text{R}}$-design for the $\mathcal{O}$-calculus program in the 'value example' on page 14 is:

$$\begin{array}{lcl} [\![42]\!]_{\text{R}} & = & \mathcal{E}_{\text{R}}[\![42]\!]^{\text{E}}_{\text{R}} \;=\; \mathcal{E}_{\text{R}}\, 42 \\ & = & \Pi_{\text{RES}} := 42 \;=\; \{\Pi_{\text{RES}}\} : (\, \mathsf{true} \vdash \Pi_{\text{RES}}' = 42 \,) \end{array}$$

□

3.3.2 Command expressions

In the $\mathcal{O}$-calculus almost all the programming operations are expressions. As previously discussed, such operations are modelled by a command-expression $\mathsf{cmdExp}_{\text{R}}(P, P^k_{i=1})$ that applies the transformation command P, which represents the operation, to the results of executing the subprograms $P^k_{i=1}$. Here, each of the subprogram results is stored in an *intermediate* variable, before being consumed by the *transformation* command that applies the operation. Specifically, the $\mathcal{U}_{\text{R}}$-design's command-expression is defined in three stages, as follows:

1. introduce a local variable for each operand, which is scoped to the next two commands;
2. calculate each argument and store its value in the appropriate local variable;
3. takes the contents of these local variables and apply the transformation command to them.

$$\begin{array}{lll} \mathsf{cmdExp}_{\text{R}}(P, (P^k_{i=1})) \;\widehat{=}\; \mathsf{decl}\, x^k_{i=1}\, \mathsf{in} & & (1) \\ \quad (\fatsemi^k_{j=1}\; P_j \upharpoonright x^k_{i=1} \fatsemi x_j := \Pi_{\text{RES}}) \fatsemi & & (2) \\ \quad P & & (3) \end{array}$$

Note that Item 2 uses one of the scoped sequential composition commands of Section 2.3.4 – specifically, the form $P \upharpoonright x^k_{i=1} \fatsemi Q$ that has the effect of hiding the variables $x^k_{i=1}$ from the scope of a subprogram P before sequentially composing it with subprogram Q.

Item 3's result transformation command can be systematically constructed by supplying a function f to the $\mathsf{trans}_{\text{R}}(f, (x^k_{i=1}))$ command, where $x^k_{i=1}$ are the names of the local variables storing the argument values. The result of this command is stored in the special result variable (Π_{RES}), so long as the values of the parameters are in the domain of the function f; if this conditions fails to hold then the resulting program behaves chaotically.

$$\mathsf{trans}_{\text{R}}(f, 2) \;\widehat{=}\; \{\Pi_{\text{RES}}\} : (\, (x^k_{i=1}) \in f \vdash \Pi_{\text{RES}}' = f(x^k_{i=1}) \,)$$

Example 3.11: The $\mathcal{O}$-calculus addition operation can be modelled in terms of the $\mathsf{trans}_{\text{R}}$ function by sequentially processing the programs represented by expressions e_1 and e_2, storing

their results in some temporary local variables, and then applying the binary addition function to this result.

$$
\begin{array}{lcl}
\langle\!\langle e_1 + e_2 \rangle\!\rangle_{\mathrm{R}} & \widehat{=} & \mathsf{cmdExp}_{\mathrm{R}}(\\
& & \quad \mathsf{trans}_{\mathrm{R}}((\_ + \_), 2), \\
& & \quad (\langle\!\langle e_1 \rangle\!\rangle_{\mathrm{R}}, \langle\!\langle e_2 \rangle\!\rangle_{\mathrm{R}}) \\
& &) \\
& = & \mathsf{decl}\, x_1, x_2 \,\mathsf{in} \\
& & \quad (\langle\!\langle e_1 \rangle\!\rangle_{\mathrm{R}} \lfloor x_1, x_2 \rfloor x_1 := \Pi_{\mathrm{RES}}) \fatsemi \\
& & \quad (\langle\!\langle e_2 \rangle\!\rangle_{\mathrm{R}} \lfloor x_1, x_2 \rfloor x_2 := \Pi_{\mathrm{RES}}) \fatsemi \\
& & \quad \mathsf{trans}_{\mathrm{R}}((\_ + \_), 2) \\
& = & \mathsf{decl}\, x_1, x_2 \,\mathsf{in} \\
& & \quad ((\mathsf{hide}\ x_1, x_2 \ \mathsf{from}\ \langle\!\langle e_1 \rangle\!\rangle_{\mathrm{R}}) \fatsemi x_1 := \Pi_{\mathrm{RES}}) \fatsemi \\
& & \quad ((\mathsf{hide}\ x_1, x_2 \ \mathsf{from}\ \langle\!\langle e_2 \rangle\!\rangle_{\mathrm{R}}) \fatsemi x_2 := \Pi_{\mathrm{RES}}) \fatsemi \\
& & \quad \mathsf{trans}_{\mathrm{R}}((\_ + \_), 2)
\end{array}
$$

□

3.3.3 Function and method invocation

The procedure invocation $\mathcal{O}$-calculus operations are compiled in a similar manner to that of their $\mathcal{U}_{\mathrm{S}}$-design counterparts.

Aside 3.12: Recall that as the $\mathcal{U}_{\mathrm{R}}$-design's command expression ($\mathsf{cmdExp}_{\mathrm{R}}$) hides the variables it introduces to store intermediate results from the evaluation of its argument subprograms, these variables will not be in scope of any procedure invocation. Further, as this is the only way of introducing program variables, the alphabet of the method invocation will only contain the six model variables (i.e. Π_{OK}, Π_{RES}, Π_{HEAP}, and their dashed variants).
□

The only aspect of function and method invocation that has changed from the previous model is the definition of the $\mathsf{call}_{\mathrm{R}}$ command, or more specifically, its fix-point function. Here, the $\mathsf{call}_{\mathrm{R}}$ command is defined as the least fix-point of the $\mathsf{apply}_{\mathrm{R}}$ function, which substitutes the value v in the program-text t for its parameter x.

$$
\begin{array}{lcl}
\mathsf{call}_{\mathrm{R}} & \widehat{=} & \mu\, z \bullet \mathsf{apply}_{\mathrm{R}}(z) \\
\mathsf{apply}_{\mathrm{R}}(z) & \widehat{=} & (\ \exists\, v, x, t \mid (v, (\!|\ x, t\ |\!)) = \Pi_{\mathrm{RES}} \bullet \\
& & \qquad \mathsf{ext}_{\mathrm{R}}(t\{\!|x \leftarrowtail \mathsf{text}_{\mathrm{R}}(v)|\!\}_{\mathrm{R}}, z) \\
& &) \\
& & \lhd\, \Pi_{\mathrm{RES}} \in \mathit{Value} \times \mathit{ProgramText} \,\rhd \\
& & \mathsf{chaos}
\end{array}
$$

3.3.4 Modelling the heap operations

The $\mathcal{U}_{\mathrm{R}}$-design is a straightforward update of the $\mathcal{U}_{\mathrm{S}}$-design. The only change is to replace the definitions of the $\mathsf{fresh}_{\mathrm{S}}$, $\mathsf{deref}_{\mathrm{S}}$ and $\mathsf{update}_{\mathrm{R}}$, with versions of the commands that use the result value instead of the operand stack.

Fresh Operator: The fresh command creates a new location on the heap, which is initialised to the explicit unset value; it sets the result value to the new heap location.

$$
\begin{array}{lcl}
\mathsf{fresh}_{\mathrm{R}} & \widehat{=} & \exists\, r \mid r = \mathsf{fresh}_{\mathrm{LOC}}(\mathrm{dom}\, \Pi_{\mathrm{HEAP}}) \bullet \\
& & \qquad \Pi_{\mathrm{RES}}, \Pi_{\mathrm{HEAP}} := r, \Pi_{\mathrm{HEAP}} \oplus \{r \mapsto \text{¿}\}
\end{array}
$$

Dereference Operator: The heap dereference command ($\mathsf{deref}_{\mathrm{R}}$) takes the location stored in the result variable, and replaces it with a copy of the associated heap value.

$$\begin{array}{lcl} \mathsf{deref}_{\mathrm{R}} & \widehat{=} & \{\Pi_{\mathrm{RES}}\} : (\\ & & \quad \Pi_{\mathrm{RES}} \notin \Pi_{\mathrm{HEAP}} \\ & & \quad \vdash \\ & & \quad \Pi_{\mathrm{RES}}{}' = \Pi_{\mathrm{HEAP}}\ \Pi_{\mathrm{RES}} \\ & &) \end{array}$$

Heap Update Operator: The heap update command assumes that the input result value (Π_{RES}) contains a location-value pair, where the location already exists on the heap. If this is the case, then the value of that location is updated appropriately.

$$\begin{array}{lcl} \mathsf{update}_{\mathrm{R}} & \widehat{=} & (\ \exists\, r, v \mid (r, v) = \Pi_{\mathrm{RES}} \bullet \\ & & \quad \Pi_{\mathrm{RES}}, \Pi_{\mathrm{HEAP}} := r, \Pi_{\mathrm{HEAP}} \oplus \{r \mapsto v\} \\ & & \quad \lhd\, r \notin \Pi_{\mathrm{HEAP}} \,\rhd \\ & & \quad \mathsf{chaos} \\ & &) \\ & & \lhd\, \Pi_{\mathrm{RES}} \in \mathit{Location} \times \mathit{Value} \,\rhd \\ & & \mathsf{chaos} \end{array}$$

Garbage collection revisited: When discussing the modelling of the heap in Section 3.1.3, we stated that the hiding of local variables, via existential quantification, could lead to situations where it is not possible to determine if a heap location is *garbage* by just examining the state variables within a program. In such situations, knowing when a heap location is genuinely unreachable is very difficult. This is illustrated by the following example.

Example 3.13: Consider the following $\mathcal{O}$-calculus program, which creates a fresh location on the heap, assigns the value 42 to it, and then dereferences that location.

$$*(\mathsf{fresh} *= 42)$$

It can be compiled to the following UTP program, using the $\mathcal{U}_{\mathrm{R}}$-design compilation scheme.

$$\begin{array}{l} (\ \mathsf{cmdExp}_{\mathrm{R}}(\\ \quad \mathsf{trans}_{\mathrm{R}}(\mathit{lvPair}, 2), \\ \quad (\mathsf{fresh}_{\mathrm{R}}, \mathcal{E}_{\mathrm{R}}\, 42) \\ \ \ ) \\ ) \fatsemi \mathsf{update}_{\mathrm{R}} \fatsemi \mathsf{deref}_{\mathrm{R}} \end{array}$$

Having done this, we expand the definition of the $\mathsf{cmdExp}_{\mathrm{R}}$ command to generate the following program, which illustrates the hidden variable issue.

$$\begin{array}{l} (\ \mathsf{decl}\, x_1, x_2 \ \mathsf{in} \\ \quad (\mathsf{hide}\ x_1, x_2 \ \mathsf{from}\ \mathsf{fresh}_{\mathrm{R}}) \fatsemi x_1 :=_{\Pi_{\mathrm{RES}}} \fatsemi \\ \quad (\mathsf{hide}\ x_1, x_2 \ \mathsf{from}\ \mathcal{E}_{\mathrm{R}}\, 42) \fatsemi x_2 :=_{\Pi_{\mathrm{RES}}} \fatsemi \\ \quad \mathsf{trans}_{\mathrm{R}}(\mathit{lvPair}, 2) \\ ) \fatsemi \mathsf{update}_{\mathrm{R}} \fatsemi \mathsf{deref}_{\mathrm{R}} \end{array}$$

We focus on the evaluation of the subprogram $\mathcal{E}_{\mathrm{R}}\, 42$, in the context where the subprogram $\mathsf{fresh}_{\mathrm{R}}$ has evaluated to the abstract location ℓ_{17}. First, we observe that the alphabet of this subprogram does not include either of the hidden local variables x_1 or x_2. Second, we observe that just prior to the evaluation of our nominated subprogram, only the Π_{RES} result variable and the hidden local variable x_1 contain references to location ℓ_{17}. Third, we observe that

the evaluation overwrites the Π_{RES} result variable with the value 42, thus leaving the hidden variable x_1 with the only reference to location ℓ_{17}. Therefore, from our nominated subprograms point of view the location ℓ_{17} is not referenced by any variable within its alphabet, and thus is garbage and can be removed.
□

3.3.5 Other mechanisms

Control flow mechanisms: The conditional evaluation command is compiled in the same manner as before. It executes subprogram P_1, P_2, or chaos depending on whether the result variable (Π_{RES}) contains true, false, or a non-boolean value respectively.

$$\begin{aligned}\mathsf{switch}_{\text{R}}(P_1, P_2) \;\widehat{=}\; (\;& (P_1 \lhd \Pi_{\text{RES}} \rhd P_2) \\ & \lhd \Pi_{\text{RES}} \in \mathit{Boolean} \rhd \\ & \mathsf{chaos} \\ )\;&\end{aligned}$$

Sequential Composition: The sequential composition of two expressions can be straightforwardly represented by the sequential composition of their compiled programs. It differs from the $\mathcal{U}_{\text{S}}$-design in that it implicitly ignores the result of the first program, rather than explicitly popping the unwanted result from the operand stack.

$$\langle\!\langle e_1;\ e_2 \rangle\!\rangle_{\text{S}} \;\widehat{=}\; \langle\!\langle e_1 \rangle\!\rangle_{\text{S}} \fatsemi \langle\!\langle e_2 \rangle\!\rangle_{\text{S}}$$

3.4 Constant Map Model

The $\mathcal{U}_{\text{M}}$-design is a variant of the $\mathcal{U}_{\text{R}}$-design, which represents procedures as triples rather than pairs: in addition to the formal parameter and procedure body text there is a constant variable map. This map is used to represent the environmental bindings of the procedure body's free variables (which may include the formal parameter). The idea is that:

1. each of the variables within such a map is defined before the completion of the procedure invocation command, which itself establishes the binding of its formal parameter to both the procedure being invoked and any nested procedures;

2. the procedure invocation command introduces the variable-value pairs in the constant map as read-only variables for the scope of this procedure's body, but excluding both procedure invocations and inner-procedure definitions.

The two exclusions in the scope of a procedure's context variables follow from the constraints of modelling recursive procedure invocations, specifically, that the alphabet before (and after) each procedure invocation call must be the same. This can be achieved by ensuring that the current procedure's context variables are hidden before each procedure invocation. Following the completion of the procedure invocation the hidden context variables are reinstated. The inner procedure exclusion follows from the observation that the invocation of any procedure removes the context of the calling (enclosing) procedure context.

3.4.1 Procedure definition

A procedure is defined as a triple of compiled program texts that represent the procedure's *parameter*, *constant-map* and *body*, where the constant map contains an entry for each free-variable within the procedure's body. These variables are set – once – prior to or during the

loading of the procedure's body. A procedure definition is denoted by $(\!|\ x, M, pt\ |\!)$, where the scope of the parameter x and the constant-map M is the program text pt.

In contrast to the previous UTP models of the $\mathcal{O}$-calculus the tuple denoting the procedure is not considered to be its meaning. Instead, we define its meaning by a pair $(\!|\ x, body\ |\!)$, where the second and third arguments of the tuple are combined to form its body – specifically the variable bindings in the constant map are used to provide a local declaration block for the program text.

$$(\!|\ x, M, pt\ |\!) \mathrel{\widehat{=}} (\!|\ x, \mathsf{decl}\ M\ \mathsf{in}\ pt\ |\!)$$

There are two reasons for denoting a procedure by a triple and defining it as a pair. First, the triple represents the conceptual aspects of the procedure definition, which are later used to structure the layout of a graph based model of state (Section 6.2.2). Second, the pair provides a representation of a method that simplifies the consistency proofs, which follow in the next chapter.

A method only has an evaluation context compilation mode, as it is not considered to be a value. It is defined as follows:

$$\begin{array}{lll} \langle\!\langle \varsigma(x)\ e \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}} \mathrel{\widehat{=}} & (\!| & x, \\ & & \{y \mapsto \mathsf{free} \mid y \in Y\}, \\ & & \langle\!\langle e \rangle\!\rangle_{\mathrm{M}} \\ & |\!) & \end{array} \quad \left| \begin{array}{l} \text{Self variable} \\ \text{Constant map} \\ \text{Body} \end{array} \right.$$

Here, $Y = \mathsf{FV}(e) \cup \{x\}$ and free is a special marker value, for unbound variables. Note, the reason we use free to denote an unbound variable rather than the existing unset value (¿) is because ¿ is a legitimate value for a variable to contain.

A function is compiled in a similar fashion to that of the method; the main difference is that functions are considered to be values, thus have a normal compilation mode (as well).

$$\begin{array}{lll} \langle\!\langle \lambda(x)\ e \rangle\!\rangle_{\mathrm{M}} & \mathrel{\widehat{=}} & \mathcal{E}_{\mathrm{M}} \langle\!\langle \lambda(x)\ e \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\ ' \\ \langle\!\langle \lambda(x)\ e \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}} & \mathrel{\widehat{=}} & \langle\!\langle \varsigma(x)\ e \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}} \end{array}$$

The method update and field assignment are essentially the same as those defined for the $\mathcal{U}_{\mathrm{R}}$-design, which in turn are inherited from the $\mathcal{U}_{\mathrm{S}}$-design (see Section 3.2.5 for details).

3.4.2 Method and function invocation

The method invocation $\mathcal{O}$-calculus operation is compiled in a similar manner to that of the $\mathcal{U}_{\mathrm{R}}$-design, which is inherited from the $\mathcal{U}_{\mathrm{S}}$-design. The key difference is in the use of the ternary variable filtering command, which ensures that only the special program variables Π_{OK} and Π_{RES} are in the alphabet of the $\mathsf{call}_{\mathrm{M}}$ command.

$$\begin{array}{lll} \langle\!\langle e.l \rangle\!\rangle_{\mathrm{M}} & \mathrel{\widehat{=}} & \mathsf{cmdExp}_{\mathrm{M}}(\\ & & \quad \mathsf{trans}_{\mathrm{M}}(omPair, 2), \\ & & \quad (\ \langle\!\langle e \rangle\!\rangle_{\mathrm{M}},\ \mathcal{E}_{\mathrm{M}} \langle\!\langle l \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\ ) \\ & &)\ ;;\ \Pi_{\mathrm{OK}}, \Pi_{\mathrm{RES}} \upharpoonright \mathsf{call}_{\mathrm{M}} \end{array}$$

Here, the *omPair* function is as defined in the operand stack model (Section 3.2.6).

Function invocation $\mathcal{O}$-calculus command is compiled in a similar manner to that of the method invocation.

$$\begin{array}{lll} \langle\!\langle e_1\, e_2 \rangle\!\rangle_{\mathrm{M}} & \mathrel{\widehat{=}} & \mathsf{cmdExp}_{\mathrm{M}}(\\ & & \quad \mathsf{trans}_{\mathrm{M}}(vtPair, 2), \\ & & \quad (\langle\!\langle e_1 \rangle\!\rangle_{\mathrm{M}}, \langle\!\langle e_2 \rangle\!\rangle_{\mathrm{M}}) \\ & &)\ ;;\ \Pi_{\mathrm{OK}}, \Pi_{\mathrm{RES}} \upharpoonright \mathsf{call}_{\mathrm{M}} \end{array}$$

Here, the *vtPair* function is as defined in the operand stack model (Section 3.2.6).

Before formally defining the generic $\mathsf{call}_{\mathrm{M}}$ command it is worth presenting an updated helper function for parameter substitution, which handles the new definition of a procedure. The notion of variable substitution is similar to that of the previous models, in the sense that it replaces a *free* variable for its definition. The difference is that a variable is only considered to be *free* if it is contained in the domain of a method's constant map and has the unbound value (free). This leads to the following algorithm for substituting the invoking object for the self variable (with name x):

1. If the method's constant map contains an unset version of the named constant x, then set it to this constant.

2. For each *direct* inner method, check whether the self variable shares the name x. If it does then this inner method's processing is complete; otherwise go to step 1.

Unfortunately this algorithm is awkward to formally define using our structural induction technique, due to the special (contextual) handling of the first method invocation. Therefore, we take the inner-method's point of view (step 2 in the algorithm), and defer the special case processing of the invoking method to the $\mathsf{call}_{\mathrm{M}}$ command.

$$
\begin{array}{lcl}
(x \mapsto \mathsf{free})\{\!|x \leftarrowtail v|\!\}_{\mathrm{M}} & \mathrel{\widehat{=}} & (x \mapsto v) \\
(\!| x, M, t |\!)\{\!|x \leftarrowtail v|\!\}_{\mathrm{M}} & \mathrel{\widehat{=}} & (\!| x, M, t |\!) \\
t\{\!|_{i=1}^{k}\ t_i|\!\}\{\!|x \leftarrowtail v|\!\}_{\mathrm{M}} & \mathrel{\widehat{=}} & t\{\!|_{i=1}^{k}\ t_i\{\!|x \leftarrowtail v|\!\}_{\mathrm{M}}\ |\!\}
\end{array}
$$

We are now in a position to define the $\mathsf{call}_{\mathrm{M}}$ command. It is defined as the least fix-point of the $\mathsf{apply}_{\mathrm{M}}$ function, which substitutes the argument value v in the program-text t for its formal parameter x. However, if either the argument value or the program-text does not exist, then the resultant program behaves chaotically.

$$
\begin{array}{lcl}
\mathsf{call}_{\mathrm{M}} & \mathrel{\widehat{=}} & \mu\, z \bullet \mathsf{apply}_{\mathrm{M}}(z) \\
\mathsf{apply}_{\mathrm{M}}(z) & \mathrel{\widehat{=}} & (\ \exists\, v, x, t \mid (v, (x, t)) = \Pi_{\mathrm{RES}} \bullet \\
 & & \quad \mathsf{ext}_{\mathrm{M}}(t\{\!|x \leftarrowtail \mathsf{text}_{\mathrm{M}}(v)|\!\}, z) \\
 & &) \\
 & & \lhd\, \Pi_{\mathrm{RES}} \in \mathit{Value} \times \mathit{ProgramText} \,\rhd \\
 & & \mathsf{chaos}
\end{array}
$$

3.4.3 Alternative method and function definition compilation scheme

Section 6.2 presents the trace-based model of the $\mathcal{O}$-calculus, which has a requirement to uniquely name both the inner procedures and the objects that a procedure directly contains. It is straightforward to update the $\mathcal{U}_{\mathrm{M}}$-design being presented here to meet this requirement, once we observe that such inner procedures and object values could be bound to free variables outside the scope of this method's definition. However, we do not want to insist that methods are written in this fashion; instead we simulate the external declaration of these values as part of the compilation process.

The first stage of this simulation is to identify those elements (subterms) of a procedure that need to be extracted; namely, the directly defined inner procedure definitions and object values. The following $\mathsf{objsAndProcs}$ function extracts the set of terms that representing these inner procedures and objects, where o, m, f, and t, have their usual meanings; namely an object value term, a method definition term, a function definition term, and a general term respectively.

$$
\begin{array}{lcl}
\mathsf{objsAndProcs}(o) & \mathrel{\widehat{=}} & \{o\} \\
\mathsf{objsAndProcs}(m) & \mathrel{\widehat{=}} & \{m\} \\
\mathsf{objsAndProcs}(f) & \mathrel{\widehat{=}} & \{f\} \\
\mathsf{objsAndProcs}(t\{\!|t_{i=1}^{k}|\!\}) & \mathrel{\widehat{=}} & \bigcup\{\!|_{i=1}^{k}\ \mathsf{objsAndProcs}(t_i)|\!\}
\end{array}
$$

The second stage is to generate a *fresh* name to identify each of these extracted terms.

$$\begin{array}{lcl} \mathsf{freshMap}(Y, \emptyset) & \widehat{=} & \emptyset \\ \mathsf{freshMap}(Y, \{t^k_{i=1}\}) & \widehat{=} & \{y \mapsto t_1\} \cup \mathsf{freshMap}(\{y\} \cup Y, \{t^k_{i=2}\}) \end{array}$$

Here, $y = \mathsf{fresh}_{\mathrm{VAR}}(Y)$ generates a variable that is not in Y.

The third (and last) stage is to use these functions in the compilation of a procedure. Here, the identified inner procedures and objects are replaced by their *freshly* generated names within the *body* of the procedure, and the constant map is updated to include the entries for these inner procedures and objects. The compilation of a $\mathcal{O}$-calculus method now follows.

$$\begin{array}{l} \langle\!\langle \varsigma(x)\ e \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}} \ \widehat{=} \\ \quad (\!|\ x, \\ \qquad M_1 \cup M_2, \\ \qquad \langle\!\langle e \{\!| o \leftarrowtail z \mid z \mapsto o \in M_2 |\!\} \rangle\!\rangle_{\mathrm{M}} \\ \quad |\!) \end{array} \quad \left| \begin{array}{l} \text{Self variable} \\ \text{Constant map} \\ \text{Body} \end{array} \right.$$

where

$$\begin{array}{lll} Y & = \mathsf{FV}(e) \cup \{x\} & \text{The method body's } \textit{free} \text{ variables} \\ M & = \mathsf{freshMap}(Y, \mathsf{objsAndProcs}(e)) & \text{The named inner procedure and object map} \\ M_1 & = \{y \mapsto \mathsf{free} \mid y \in Y\} & \text{The map of, un-set (free), external variables} \\ M_2 & = \{y \mapsto \langle\!\langle e \rangle\!\rangle_{\mathrm{M}} \mid y \mapsto e \in M\} & \text{The map of simulated external variables} \end{array}$$

Note that as the compilation of an $\mathcal{O}$-calculus function is defined in terms of the compilation of an $\mathcal{O}$-calculus method, its definition does not need to be updated.

3.5 Summary

We have presented three UTP models of the $\mathcal{O}$-calculus, along with their corresponding systematic compilation schemes. These models differ in the way that they handle both the final and intermediate values within a calculation (i.e. the execution of a program), as follows:

1. the $\mathcal{U}_{\mathrm{S}}$-design uses an operand stack to store the results of both the final and intermediate calculations;
2. the $\mathcal{U}_{\mathrm{R}}$-design replaces the operand stack with a combination of a result variable and some temporary local variable declaration blocks, which are used to store the final and intermediate calculations respectively; and
3. the $\mathcal{U}_{\mathrm{M}}$-design extends the use of local variable declaration blocks, so that they also model procedure parameters.

All three of these models share the same straightforward model of a heap, which is a partial map from abstract locations to values. Such a model of a heap is not fully-abstract as discussed in Section 1.5.2. Here, we have focused on the core modelling of object-based object orientation, which in turn is focused on the modelling of potentially mutually recursive method (and function) invocation.

Potentially mutually recursive procedure calls are defined in terms of the UTP weakest fixpoint operator, where each procedure in the recursive loop is essentially sequentially composed with the subprogram that invoked it and the subprogram that follows it. One consequence of this is that the procedure invocation mechanism must guarantee that the alphabets of the procedure and the calling contexts will always agree. Our solution to this issue, was to ensure that the alphabet at the point of procedure invocation was guaranteed to contain just the model's observational variables (i.e. the non program variables).

It is now possible to argue that the $\mathcal{U}_{\text{S}}$-design is the simplest of the UTP models, as it does not use the UTP's variable management facilities (e.g. introduction and elimination), and thus the alphabet of all subprograms within this UTP model are guaranteed to be the same. The $\mathcal{U}_{\text{R}}$-design is only slightly more complicated as the local variables it uses to store intermediate results of calculations (i.e. subprograms) are guaranteed to be outside the scope of the subprograms themselves, and thus the alphabet at point of method invocation will always be the same. Lastly, the $\mathcal{U}_{\text{M}}$-design guarantees that the alphabet of a procedure invocation is always the same, by temporarily hiding all local program variables from scope just prior to invoking a procedure.

These models form the basis for two of our three contributions (as presented in Section 1.5.3 and discussed in Chapter 7). Specifically, we have encoded – via a compilation scheme – an Abadi–Cardelli-style object calculus in the UTP. What we have not shown (so far) is that this compilation scheme produces UTP programs that have the same semantics as that of their source (i.e. an $\mathcal{O}$-calculus program). This is the topic of Chapter 4.

Chapter 4

Object Model Consistency

The three Unifying Theories of Programming (UTP) object models in Chapter 3 each provide a semantics for our $\mathcal{O}$-calculus, which was presented in Section 2.2. We demonstrate that the denotational semantics of each UTP model is consistent with the operational semantics of the $\mathcal{O}$-calculus, via a structural induction over the $\mathcal{O}$-calculus's terms; i.e. the UTP denotational semantics of an $\mathcal{O}$-calculus operation is the same as that of its result.

One complication is that the $\mathcal{O}$-calculus's operational semantics introduces some dynamic terms, such as the *fresh location values*. These values index (point at) entries on the abstract heap, which is considered to be empty at the beginning of a program's execution. Another dynamic term is that of the substitution meta-operation, which replaces an $\mathcal{O}$-calculus parameter with its value. The fresh location terms have already been included in our UTP models of the $\mathcal{O}$-calculus, via the modelling of the three heap operations (fresh, $*$, and $*=$). However, the substitution operation has not been directly modelled, as it was not required for defining the UTP semantics of the $\mathcal{O}$-calculus. Having said this, the substitution operation does appear in the resultant part of several $\mathcal{O}$-calculus small-step reduction rules, and thus could be considered as part of the structure that the consistency proof is inducting over. This is the view that we take.

This chapter presents the dynamic operations for the three UTP models (Section 4.1), an outline of the structural induction argument (Section 4.2), the main proofs for each of the models (Section 4.3 to Section 4.5), and an example (Section 4.6). Here, the example provides an illustration of the correspondence between the operational $\mathcal{O}$-calculus and denotational UTP semantics via one iteration of a factorial program. Additional laws and proofs are contained in Appendix C and are referenced where appropriate.

4.1 Dynamic Operations

The previous chapter defined three UTP models of the $\mathcal{O}$-calculus (Section 2.2), where each model had at least one compilation scheme for taking an $\mathcal{O}$-calculus program to its corresponding UTP model. However, these compilation schemes did not include translations for the *run-time* information that the $\mathcal{O}$-calculus's operational rules introduce. In particular, we do not provide compilation schemes for

- the variable substitution meta-operation ($t\{\!|x \leftarrowtail ov|\!\}$); and
- the execution context (Γ) (i.e. the heap).

We now update our compilation schemes to handle this dynamic run-time information, so that it is straightforward to define the UTP semantics of an $\mathcal{O}$-calculus term at any point (step) during its operational evaluation (execution).

4.1.1 Variable substitution compilation scheme

The meta-function for substituting variables in an $\mathcal{O}$-calculus program is recursively defined over the syntactic structure of an $\mathcal{O}$-calculus term (program). In particular, it cannot be applied to the meanings of its arguments, just their syntactic form. For the UTP operand–stack-design ($\mathcal{U}_{\text{S}}$-design) this amounts to compiling the term representing the object calculus program, and then applying the $\mathcal{U}_{\text{S}}$-design's syntactic variable substitution process.

$$\llbracket t\{\!|x \leftarrowtail ov|\!\}\rrbracket_{\text{S}} \;\widehat{=}\; \llbracket t\rrbracket_{\text{S}}\{\!|x \leftarrowtail \llbracket ov\rrbracket^{\text{E}}_{\text{S}}|\!\}_{\text{S}}$$

The same compilation scheme is used for the UTP result–value-design ($\mathcal{U}_{\text{R}}$-design), but not for the UTP constant–map-design ($\mathcal{U}_{\text{M}}$-design). The $\mathcal{U}_{\text{M}}$-design explicitly models parameters via the introduction of UTP variables. Therefore, the variable substitution compilation scheme is adapted to use this approach. Specifically, it introduces a variable to hold the *constant* value of the variable substitution, for all terms within the current program-text, except for (inner) procedures. Such inner procedures contain a *constant map*, which records the values of its *inherited* parameters as they are set, via the use of the $\mathcal{U}_{\text{M}}$-design's variable substitution function.

$$\begin{aligned}\llbracket t\{\!|x \leftarrowtail ov|\!\}\rrbracket_{\text{M}} \;\widehat{=}\; & \mathsf{decl}\ \{x \mapsto \llbracket ov\rrbracket^{\text{E}}_{\text{M}}\}\ \mathsf{in} \\ & \llbracket t\rrbracket_{\text{M}}\{\!|x \leftarrowtail \llbracket ov\rrbracket^{\text{E}}_{\text{M}}|\!\}_{\text{M}}\end{aligned}$$

4.1.2 The heap context compilation scheme

The heap in the environmental context is a, potentially empty, finite map from $\mathcal{O}$-calculus dynamic locations to values. Each of the three UTP models in Chapter 3 mirrors the $\mathcal{O}$-calculus model, in the sense that it introduces a Π_{HEAP} observation variable to store the map from dynamic locations to values; in these cases, the locations and values are of the relevant UTP model. Therefore, the compilation scheme for the heap context straightforwardly transforms each of the entries in the $\mathcal{O}$-calculus map to the corresponding entries in the UTP model's map, as follows:

$$\llbracket \{^{k}_{i=1}\, r_i \mapsto v_i\}\rrbracket_{\text{J}} \;\widehat{=}\; \{^{k}_{i=1}\, \llbracket r_i\rrbracket_{\text{J}} \mapsto \llbracket v_i\rrbracket_{\text{J}}\}$$

Here, J stands for one of the three UTP design model identifiers S, R, and M.

The dynamic aspects of the $\mathcal{O}$-calculus are introduced by the application of the small-step reduction rules to an $\mathcal{O}$-calculus expression that represents a program. These rules are applied to (and return) dynamic terms of the form $C \bullet e$, where C denotes a context and e an expression. The UTP semantics for such dynamic terms is the sequential composition of the UTP designs that denote the context C and the expression e respectively, as follows:

$$\llbracket C \bullet e\rrbracket_{\text{J}} \;\widehat{=}\; \llbracket C\rrbracket_{\text{J}} \fatsemi \llbracket e\rrbracket_{\text{J}}$$

Here, the context C is in one of the following three styles:

$\llbracket \Gamma\{\mathtt{heap} \mapsto H\}\rrbracket_{\text{J}}$	$\widehat{=}$	$\Pi_{\text{HEAP}} :=_{\text{J}} \llbracket H\rrbracket_{\text{J}}$	Read context heap value
$\llbracket \Gamma\{\mathtt{heap} \leftarrowtail H\}\rrbracket_{\text{J}}$	$\widehat{=}$	$\Pi_{\text{HEAP}} :=_{\text{J}} \llbracket H\rrbracket_{\text{J}}$	Write context heap value
$\llbracket \Gamma\rrbracket_{\text{J}}$	$\widehat{=}$	skip^{D}	Context not accessed

Recall that the last form of the context occurs in $\mathcal{O}$-calculus small-step reduction rules that do not use (or alter) the contextual information, and thus can be modelled by the unit of composition.

4.2 Structural Induction

Each valid $\mathcal{O}$-calculus expression (program) is either a value or reducible to a value using the deterministic small-step reduction rules. From our structural induction perspective, the interesting cases are those terms (structures) that correspond to the left hand side (LHS) of a reduction rule, as the UTP semantics of the before and after states of these rules can be compared and shown to be consistent (i.e. equal). Those rules that enable a subterm (i.e. an argument) to be reduced via the application of another rule to the subterm are already catered for by the induction hypothesis; i.e. that a sub-term has a consistent UTP semantics. Therefore, we focus our attention on the remaining rules, which typically apply to fully evaluated arguments; one notable exception is that of the IfT and IfF conditional evaluation rules, both of which only require one of their arguments to be evaluated (Section 2.2.4).

$\mathcal{O}$-calculus expressions can be invalid for two main reasons. First, a non-value expression may not match any of the small-step reduction rules, in which case the program's execution becomes *stuck*. For example, there is no rule that enables the logical negation of an integer (e.g. $\neg 3$ cannot be reduced). Second, the expression may be *infinitely reducible*, as the program that it represents does not terminate (e.g. $[l = \varsigma(x)\, x.l].l$). Our UTP model does not distinguish between these two forms of invalidity; they are both modelled by chaos (Section 3.1.4). The stuck behaviours are modelled by precondition checks within each of the UTP models; if such preconditions fail the resulting UTP designs are equivalent to chaos, as required. The infinitely looping terms result from recursive (or mutually recursive) method and function calls that never reduce to a value. The UTP weakest fix-point of such terms is equivalent to chaos due to Tarski's theorem as discussed in [HH98, WC04], as required. We still have to show that the reduction of an individual method and function call is consistent, so as to preserve the non-terminating sequence of calls. This consistency check is already being performed as part of the valid term consistency check, so no further work is required.

Another constraint that we add to the structural induction is that it can only be used to show the consistency of closed $\mathcal{O}$-calculus programs; i.e. $\mathcal{O}$-calculus programs that have no free variables. This syntactically checkable constraint is required for the assumption that all variables within the program are bound to a value before they are used. The other requirement for this assumption to hold is that all the variables are initialised before they are used. This is guaranteed by the syntactic structure of the program and small-step reduction rules, which ensure: that variables are initialised as part of function or method invocation; and that let expressions are reduced to function invocations.

The remainder of this section introduces the use of commuting diagrams to present one step of the consistency structural induction argument, and then applies such diagrams to the reducible terms. Here, we merely state the lemmas; the proofs for the three UTP models follow in Sections 4.3 to 4.5.

4.2.1 Commuting Diagrams

The commuting diagram in Figure 4.1 illustrates the structure of the proof that the semantic models for an object calculus operation op with k subterms are consistent, where:

- st_i is the i^{th} subterm of the original operation.
- ov_i is the i^{th} subterm of the operation after its arguments (i.e. operand values) have been evaluated in the correct order.
- $[\![t]\!]^{\mathrm{M}}_{\mathrm{J}}$ is the combination of the semantic meaning and compilation functions (i.e. $[\![\langle\!| t |\!\rangle^{\mathrm{M}}_{\mathrm{J}}]\!]$), where M is the compilation mode and J is the model identifier.
- 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 subterm mappings (i.e. $\forall_{i=1}^{k} \langle\!\langle ov_i \rangle\!\rangle_{\mathrm{J}} = \mathcal{E}_{\mathrm{J}}\ ov_i$).

$A1$ is the assumption that the arguments are in the domain of the operation being modelled (i.e. $(ov_{i=1}^{k}) \Subset op$).

$A2$ is the assumption that the result of executing the operation with arguments $ov_{i=1}^{k}$ is the expression e.

$\rightarrow, \twoheadrightarrow$ are the one-step and multi-step $\mathcal{O}$-calculus operational semantics.

$\downarrow$ is a compilation and/or semantic evaluation function.

$=, \|$ are two different representations of the equality relation, for horizontal and vertical display contexts respectively.

$$\begin{array}{ccccc}
\Gamma_0 \bullet op\{st_{i=1}^{k}\} & \xrightarrow{A0} & \Gamma_1 \bullet op\{ov_{i=1}^{k}\} & \xrightarrow{A1,\,A2} & \Gamma_2 \bullet e \\
\Big\downarrow{\scriptstyle [\![-]\!]_{\mathrm{J}}} & & \Big\downarrow{\scriptstyle [\![-]\!]_{\mathrm{J}}} & & \Big\downarrow{\scriptstyle [\![-]\!]_{\mathrm{J}}} \\
[\![\Gamma_0]\!]_{\mathrm{J}} \fatsemi [\![op\{st_{i=1}^{k}\}]\!]_{\mathrm{J}} & \overset{A0}{=\!=} & [\![\Gamma_1]\!]_{\mathrm{J}} \fatsemi [\![op\{ov_{i=1}^{k}\}]\!]_{\mathrm{J}} & \overset{A1,\,A2}{=\!=} & [\![\Gamma_2]\!]_{\mathrm{J}} \fatsemi [\![e]\!]_{\mathrm{J}}
\end{array}$$

Figure 4.1: Commuting diagram principle

The left hand square of the commuting diagram in Figure 4.1 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 4.2.

$$\begin{array}{ccc}
\Gamma_1 \bullet op\{ov_{i=1}^{k}\} & \xrightarrow{A1,\,A2} & \Gamma_2 \bullet e \\
\Big\downarrow{\scriptstyle [\![-]\!]_{\mathrm{J}}} & & \Big\downarrow{\scriptstyle [\![-]\!]_{\mathrm{J}}} \\
[\![\Gamma_1]\!]_{\mathrm{J}} \fatsemi [\![op\{ov_{i=1}^{k}\}]\!]_{\mathrm{J}} & \overset{A1,\,A2}{=\!=} & [\![\Gamma_2]\!]_{\mathrm{J}} \fatsemi [\![e]\!]_{\mathrm{J}}
\end{array}$$

Figure 4.2: Commuting diagram practice

4.2.2 Command expressions

The $\mathcal{O}$-calculus provides a variety of operations that are non-recursively defined, such as the simple arithmetic and logical operations. As these operations are systematically compiled, it is possible to present a generic proof that these operations are consistently modelled. The commuting diagram in Figure 4.3 outlines the structure of the proof that a function f with k scalar parameters has a consistent denotational semantics. Here we assume that:

$A3$ The scalar values $ov_{i=1}^{k}$ are in the domain of the k-ary function; i.e. $(ov_{i=1}^{k}) \Subset f$.

$A4$ The scalar value ov is the result of evaluating the function f; i.e. $ov = f(ov_{i=1}^{k})$.

Lemma 4.1 is the key step in this consistency proof, which demonstrates that a command expression has the expected semantics. This is denoted by $L\,4.1$ in Figure 4.3's commuting diagram, which in turn forms a template for all the $\mathcal{O}$-calculus operations that are defined as command expressions within the three UTP models. In particular, field assignment and method update are also covered by this proof template; the commuting diagram that specifically corresponds to the method update operation is supplied at the end of this section (in Figure 4.4).

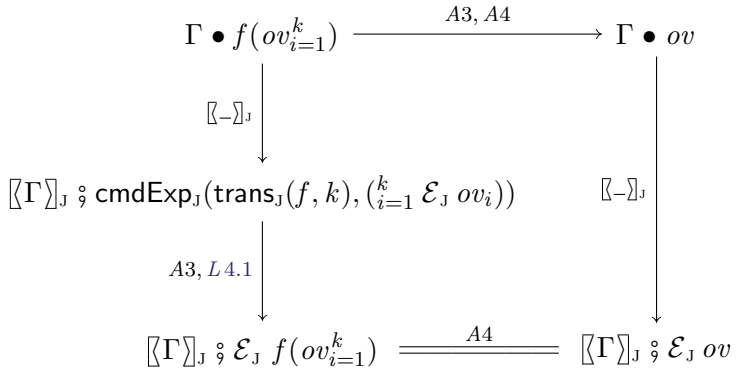


Figure 4.3: Generic command expression commuting diagram

Lemma 4.1 – Command Expression Lemma: The effect of applying a cmdExp_J command to a function f, with *pre-evaluated* arguments $ov_{i=1}^k$, is the same as the effect of applying the evaluation command to the result of the function f on its arguments, assuming that the arguments are in the domain of the function (i.e. $(ov_{i=1}^k) \Subset f$).

$$(ov_{i=1}^k) \Subset f \quad \Rightarrow \quad \mathsf{cmdExp}_J(\mathsf{trans}_J(f,k),(\overset{k}{i=1}\ \mathcal{E}_J\ ov_i)) = \mathcal{E}_J(f(ov_{i=1}^k))$$

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

Member update: Figure 4.4 specialises the command expression commuting diagram template of Figure 4.3 to show the consistency argument for the method update operation.

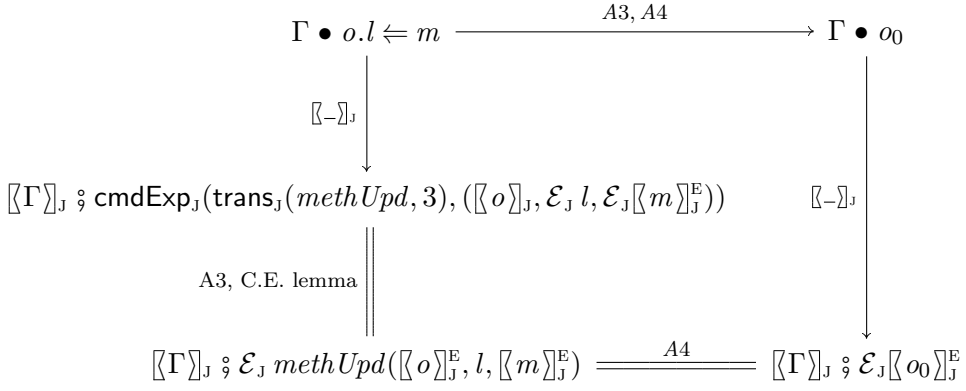


Figure 4.4: Method update commuting diagram

4.2.3 Procedure invocation

There are two similar forms of procedure invocation: method invocation and function invocation. The commuting diagram in Figure 4.5 demonstrates that the $\mathcal{O}$-calculus method invocation is consistent with one unwinding of the fix-point function (call_J) that defines it, where assumption $A5$ is that the label l of object o has the method $\varsigma(x)\ e$. Similarly, the commuting diagram in Figure 4.6 demonstrates that the $\mathcal{O}$-calculus function invocation is consistent with one unwinding of the fix-point function (call_J) that defines it, where assumption $A6$ is that the function f is denoted by $\lambda(x)\ e$.

The consistency diagrams in Figure 4.5 and Figure 4.6 rely on three lemmas: the command expression lemma (Lemma 4.1); the unwinding lemma (Lemma 4.2), which uses the fix-point

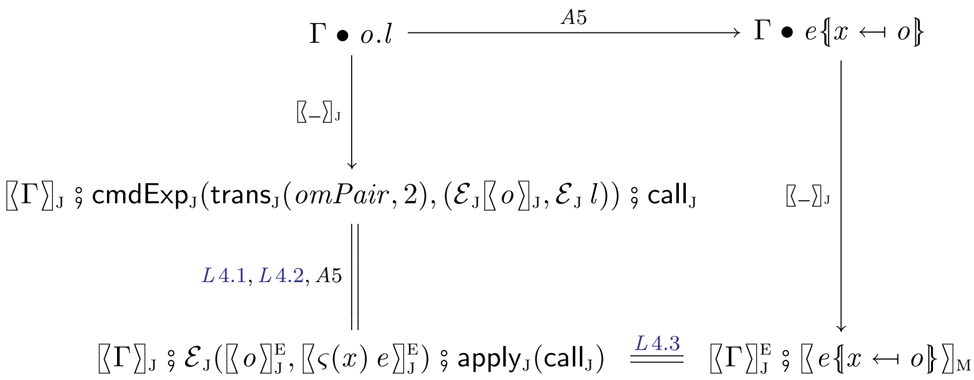


Figure 4.5: Method invocation commuting diagram

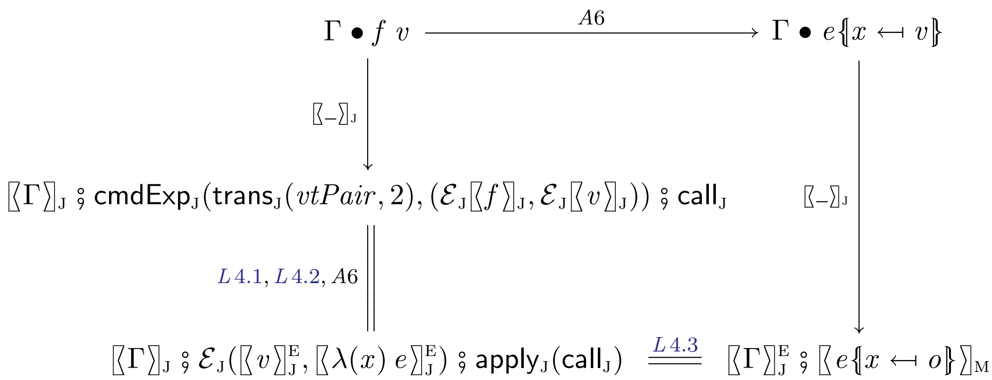


Figure 4.6: Function invocation commuting diagram

definition to provide a single unwinding; and the procedure call lemma (Lemma 4.3), which demonstrates that this unwinding is correct.

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

Lemma 4.2 – Unwinding Lemma:

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

□

Proof

$$\begin{aligned}
& \mathsf{call}_J \\
= & \quad \text{Defn. of } \mathsf{call}_J \\
& (\mu\, z \bullet \mathsf{apply}_J(z)) \\
= & \quad \text{Defn of fix-point } \mu \\
& \mathsf{apply}_J(\mu\, z \bullet \mathsf{apply}_J(z)) \\
= & \quad \text{Defn. of } \mathsf{call}_J \\
& \mathsf{apply}_J(\mathsf{call}_J)
\end{aligned}$$

□

The following procedure call lemma does not directly meet the requirements of the commuting diagram, in that it does not start with the pre-compiled form of a function or method. Instead, it starts with the compiled form of the function or method, which depends on the UTP model being used. A similar comment can be applied to the dynamic substitution operation. Both of these definitional transformations are straightforward, and thus the procedure call lemma indirectly meets the requirements of the commuting diagram.

Lemma 4.3 – Procedure Call Lemma: The effect of applying the $\mathsf{call}_{\mathrm{J}}$ command is equivalent to the effect of applying one iteration of this command to itself.

$$\begin{array}{lcl} \mathcal{E}_{\mathrm{J}}(\llbracket v \rrbracket^{\mathrm{E}}_{\mathrm{J}}, (\!| x, \llbracket e \rrbracket^{\mathrm{E}}_{\mathrm{J}} |\!)) \fatsemi \mathsf{apply}_{\mathrm{J}}(\mathsf{call}_{\mathrm{J}}) & = & [\![\langle\!\langle e \rangle\!\rangle_{\mathrm{J}} \{\!| x \mapsto \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{J}} |\!\}_{\mathrm{J}}]\!] \\ \mathcal{E}_{\mathrm{M}}(\llbracket v \rrbracket^{\mathrm{E}}_{\mathrm{M}}, (\!| x, M, pt |\!)) \fatsemi \mathsf{apply}_{\mathrm{M}}(\mathsf{call}_{\mathrm{M}}) & = & [\![\mathsf{decl}\{x \mapsto v\} \mathsf{\ in} \\ & & \quad (\!| pt |\!) \{\!| x \mapsto \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}} |\!\}_{\mathrm{M}}]\!] \end{array}$$

where $\mathrm{J} \in \{ \mathrm{S}, \mathrm{R} \}$ and

$$M = \{x \mapsto \mathsf{free}, (y_i \mapsto ov_i)^k_{i=1}\}$$
$$pt = \langle\!\langle e \rangle\!\rangle_{\mathrm{M}} \{\!|^k_{i=1} y_i \mapsto ov_i |\!\}_{\mathrm{M}}$$

Note that operand values ov_1^k represent the values of *all* the inherited parameters y_1^k respectively; this follows from limiting our consistency analysis to closed $\mathcal{O}$-calculus programs (as if one of the parent parameters was not set, then it would be free in the $\mathcal{O}$-calculus program, and thus contradict our limitation).
□

4.2.4 Heap operations

Fresh heap locations: The commuting diagram in Figure 4.7 outlines the structure of the proof that the fresh operator in the $\mathcal{O}$-calculus has a consistent denotational semantics. Here we assume that:

A7 The expression $\mathsf{fresh}_{\mathrm{LOC}}(\mathrm{dom}\, H_0)$ evaluates to ℓ_j.
A8 The context-heap value H_1 is $H_0 \oplus \{\ell_j \mapsto \text{¿}\}$.

$$\begin{array}{ccc} \Gamma\{\mathtt{heap} \mapsto H_0\} \bullet \mathsf{fresh} & \xrightarrow{A7,\, A8} & \Gamma\{\mathtt{heap} \leftarrowtail H_1\} \bullet \ell_j \\ \downarrow \llbracket - \rrbracket_{\mathrm{J}} & & \downarrow \llbracket - \rrbracket_{\mathrm{J}} \\ (\Pi_{\mathrm{HEAP}} := \llbracket H_0 \rrbracket^{\mathrm{E}}_{\mathrm{J}}) \fatsemi \mathsf{fresh}_{\mathrm{J}} & \overset{L\,4.4,\, A7,\, A8}{=\!=\!=} & (\Pi_{\mathrm{HEAP}} := \llbracket H_1 \rrbracket^{\mathrm{E}}_{\mathrm{J}}) \fatsemi (\mathcal{E}_{\mathrm{J}}\, \ell_j) \end{array}$$

Figure 4.7: Fresh location commuting diagram

Lemma 4.4 – Fresh location lemma:

$$(\Pi_{\mathrm{HEAP}} := H) \fatsemi \mathsf{fresh}_{\mathrm{J}} \quad = \quad (\Pi_{\mathrm{HEAP}} := H \oplus \{r \mapsto \text{¿}\}) \fatsemi (\mathcal{E}_{\mathrm{J}}\, r)$$

where $r = \mathsf{fresh}_{\mathrm{LOC}}(\mathrm{dom}\, \Pi_{\mathrm{HEAP}})$
□

Dereferencing a heap location: The commuting diagram in Figure 4.8 outlines the structure of the proof that the dereference operator $(*\_)$ in the $\mathcal{O}$-calculus has a consistent denotational semantics. Here, assumption *A9* is that $r \notin \Pi_{\mathrm{HEAP}}$ and $v = \Pi_{\mathrm{HEAP}}(r)$.

$$\begin{array}{ccc} \Gamma\{\mathtt{heap} \mapsto H\} \bullet *r & \xrightarrow{A9} & \Gamma\{\mathtt{heap} \leftarrowtail H\} \bullet v \\ \downarrow \llbracket - \rrbracket_{\mathrm{J}} & & \downarrow \llbracket - \rrbracket_{\mathrm{J}} \\ (\Pi_{\mathrm{HEAP}} := \llbracket H \rrbracket^{\mathrm{E}}_{\mathrm{J}}) \fatsemi \llbracket r \rrbracket_{\mathrm{J}} \fatsemi \mathsf{deref}_{\mathrm{J}} & \overset{L\,4.5,\, A9}{=\!=\!=} & (\Pi_{\mathrm{HEAP}} := \llbracket H \rrbracket^{\mathrm{E}}_{\mathrm{J}}) \fatsemi \llbracket v \rrbracket_{\mathrm{J}} \end{array}$$

Figure 4.8: Dereferencing location commuting diagram

Lemma 4.5 – Dereferencing location lemma:

$$(\Pi_{\text{HEAP}} := H) \fatsemi (\mathcal{E}_{\text{J}}\ r) \fatsemi \mathsf{deref}_{\text{J}} \quad = \quad (\Pi_{\text{HEAP}} := H) \fatsemi (\mathcal{E}_{\text{J}}\ v)$$

where $r \notin \Pi_{\text{HEAP}}$ and $v = \Pi_{\text{HEAP}}(r)$

□

Updating a heap location: The commuting diagram in Figure 4.9 outlines the structure of the proof that the reference update operator ($\_ *= \_$) in the $\mathcal{O}$-calculus has a consistent denotational semantics. Here, assumption $A10$ is that the context-heap value $H_1 = H_0 \oplus \{r \mapsto v\}$.

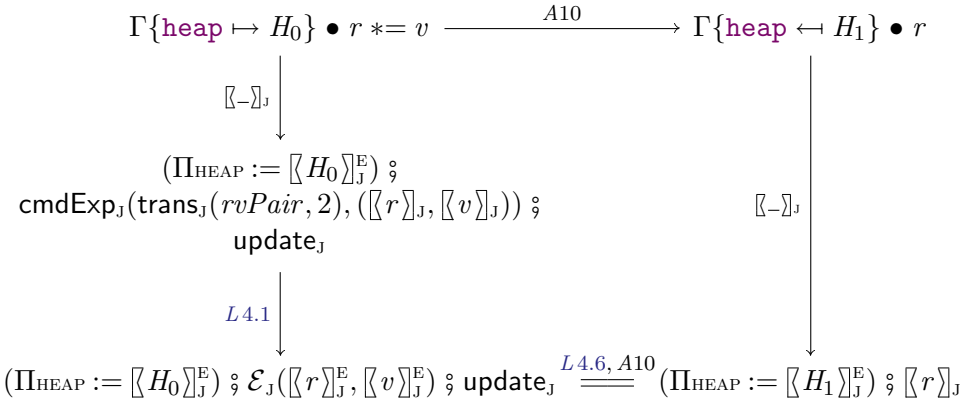


Figure 4.9: Updating location commuting diagram

Lemma 4.6 – Updating location lemma:

$$(\Pi_{\text{HEAP}} := H) \fatsemi \mathcal{E}_{\text{J}}(r, v) \fatsemi \mathsf{update}_{\text{J}} \quad = \quad (\Pi_{\text{HEAP}} := H \oplus \{r \mapsto v\}) \fatsemi (\mathcal{E}_{\text{J}}\ r)$$

where $r \notin \Pi_{\text{HEAP}}$

□

4.2.5 Other programming mechanisms

Sequential composition: The composition of an $\mathcal{O}$-calculus value and an expression results in the evaluation of the expression. The two commuting diagrams in Figure 4.10 illustrate the consistency argument for the $\mathcal{U}_{\text{S}}$-design and the other UTP models respectively. In the latter

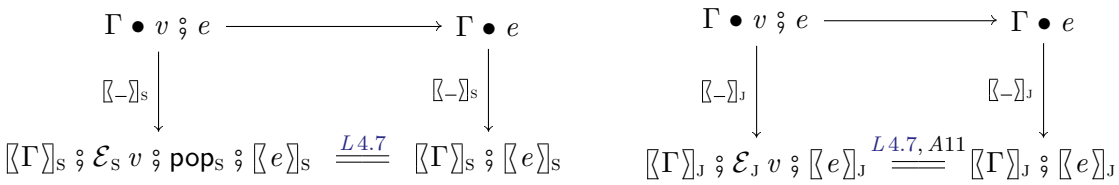


Figure 4.10: Sequential composition commuting diagram

– right hand side (RHS) – case, the assumption $A11$ is that the semantics of expression e is independent of the value v. This assumption holds whenever the LHS of the sequential composition does not update the evaluation context (Γ), thus as the evaluation of a value can never update the context, assumption $A11$ holds in this case.

Lemma 4.7 – Sequential Composition:

$$\begin{array}{lcl} \mathcal{E}_{\text{S}}\ ov \fatsemi \mathsf{pop}_{\text{S}} \fatsemi P & = & P \\ \mathcal{E}_{\text{J}}\ ov \fatsemi P & = & P \end{array}$$

Here, for J in { R, M } we also require that the side condition $\forall\, ov \bullet P = P[ov/\Pi_{\text{RES}}]$, which is assumption $A11$.
□

Conditional choice: The conditional term is split into two cases, depending on the evaluation of its first argument (the conditional), which is either true or false. The commuting diagrams in Figure 4.11 illustrate the consistency argument for both of these cases, where:

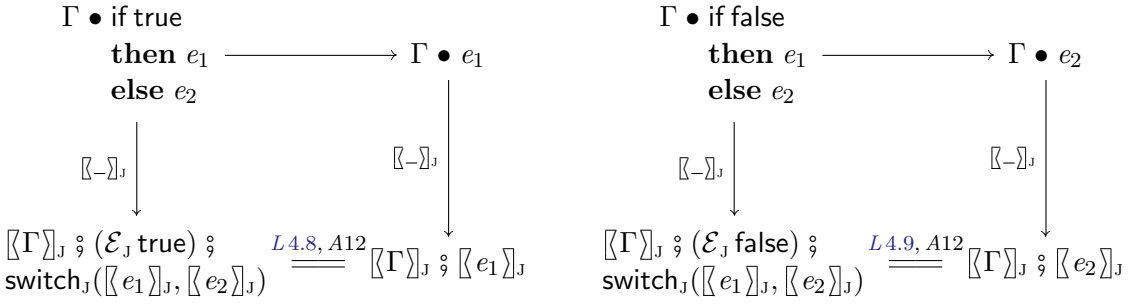


Figure 4.11: Conditional commuting diagrams

assumption $A12$ is that neither expression e_1 or e_2 are dependent on the evaluation of the conditional argument; and $L\,4.8$ and $L\,4.9$ denote the following true and false conditional lemmas respectively.

Lemma 4.8 – Conditional True:

$$\mathcal{E}_{\text{J}}\ \mathsf{true} \fatsemi \mathsf{switch}_{\text{J}}(P, Q) \quad = \quad P$$

Here, for J in { R, M }, we also require that the side conditions $\forall\, b \bullet P = P[b/\Pi_{\text{RES}}]$ and $\forall\, b \bullet Q = Q[b/\Pi_{\text{RES}}]$ hold, which is assumption $A12$.
□

Lemma 4.9 – Conditional False:

$$\mathcal{E}_{\text{J}}\ \mathsf{false} \fatsemi \mathsf{switch}_{\text{J}}(P, Q) \quad = \quad Q$$

Here, for J in { R, M }, we also require that the side conditions $\forall\, b \bullet P = P[b/\Pi_{\text{RES}}]$ and $\forall\, b \bullet Q = Q[b/\Pi_{\text{RES}}]$ hold, which is assumption $A12$.
□

The proofs for these lemmas follow almost immediately from the definition of the $\mathsf{switch}_{\text{J}}$ command, which mirrors that of the $\mathcal{O}$-calculus conditional expression.

4.2.6 Proof notation

We introduce an alternative notation for the design framing mechanism presented in Section 2.3.3, to simplify the presentation of the proofs that follow in the remainder of this chapter.

$$p \vdash^{V} P \quad \widehat{=} \quad V : (p \vdash P)$$

4.3 Operand Stack Model

We now prove the lemmas that require model-specific steps for the $\mathcal{U}_{\mathrm{S}}$-design of the $\mathcal{O}$-calculus.

4.3.1 Command expressions

The command expression lemma (Lemma 4.1) is now specialised for the $\mathcal{U}_{\mathrm{S}}$-design model as follows

$$(ov_{i=1}^{k}) \in f \quad \Rightarrow \quad \mathsf{cmdExp_S}(\mathsf{trans_S}(f,k),(\overset{k}{}_{i=1}\ \mathcal{E}_{\mathrm{S}}\ ov_i)) = \mathcal{E}_{\mathrm{S}}(f(ov_{i=1}^{k}))$$

Within the following proof, the LHS of the initial implication is added as an assumption to the proof context.

Proof

$$\begin{array}{ll}
& \mathsf{cmdExp_S}(\mathsf{trans_S}(f,k),(\overset{k}{}_{i=1}\ \mathcal{E}_{\mathrm{S}}\ ov_i)) \\
= & \cdots\cdots \quad \text{Defn. of } \mathsf{cmdExp_S} \\
& (\fatsemi_{i=1}^{k}\ \mathcal{E}_{\mathrm{S}}\ ov_i) \fatsemi \mathsf{trans_S}(f,k) \\
= & \cdots\cdots \quad \text{Defn. of } \mathcal{E}_{\mathrm{S}} \\
& (\fatsemi_{i=1}^{k}\ (\mathsf{true} \vdash^{\{\Pi_{\mathrm{STK}}\}} \Pi_{\mathrm{STK}}' = \langle ov_i \rangle \frown \Pi_{\mathrm{STK}})) \fatsemi \\
& \mathsf{trans_S}(f,k) \\
= & \cdots\cdots \quad \text{Defn. of } _ \fatsemi _ \\
& (\mathsf{true} \vdash^{\{\Pi_{\mathrm{STK}}\}} \Pi_{\mathrm{STK}}' = \langle \overset{i=k}{}_{1}\ ov_i \rangle \frown \Pi_{\mathrm{STK}}) \fatsemi \quad \text{and predicate logic} \\
& \mathsf{trans_S}(f,k) \\
= & \cdots\cdots \quad \text{Defn. of } \mathsf{trans_S} \\
& (\mathsf{true} \vdash^{\{\Pi_{\mathrm{STK}}\}} \Pi_{\mathrm{STK}}' = \langle \overset{i=k}{}_{1}\ ov_i \rangle \frown \Pi_{\mathrm{STK}}) \fatsemi \\
& (\ \#\Pi_{\mathrm{STK}} \geq k \wedge (\Pi_{\mathrm{STK}}\ \mathsf{proj}\ k) \in f \\
& \quad \vdash^{\{\Pi_{\mathrm{STK}}\}} \\
& \quad \Pi_{\mathrm{STK}}' = \langle f(\Pi_{\mathrm{STK}}\ \mathsf{proj}\ k) \rangle \frown (\mathsf{tail}^k\ \Pi_{\mathrm{STK}}) \\
&) \\
= & \cdots\cdots \quad \text{Law C.13 assign comp.} \\
& (\#\Pi_{\mathrm{STK}} \geq k \wedge (\Pi_{\mathrm{STK}}\ \mathsf{proj}\ k) \in f)[\langle ov_1^{i=k} \rangle \frown \Pi_{\mathrm{STK}}/\Pi_{\mathrm{STK}}] \\
& \vdash^{\{\Pi_{\mathrm{STK}}\}} \\
& (\Pi_{\mathrm{STK}}' = \langle f(\Pi_{\mathrm{STK}}\,\mathsf{proj}\,k) \rangle \frown (\mathsf{tail}^k\ \Pi_{\mathrm{STK}}))[\langle ov_1^{i=k} \rangle \frown \Pi_{\mathrm{STK}}/\Pi_{\mathrm{STK}}] \\
= & \cdots\cdots \quad \text{Semantic substitution} \\
& \#(\langle ov_1^{i=k} \rangle \frown \Pi_{\mathrm{STK}}) \geq k \wedge ((\langle ov_1^{i=k} \rangle \frown \Pi_{\mathrm{STK}})\ \mathsf{proj}\ k) \in f \\
& \vdash^{\{\Pi_{\mathrm{STK}}\}} \\
& \Pi_{\mathrm{STK}}' = \langle f((\langle ov_1^{i=k} \rangle \frown \Pi_{\mathrm{STK}})\ \mathsf{proj}\ k) \rangle \frown (\mathsf{tail}^k \langle ov_1^{i=k} \rangle \frown \Pi_{\mathrm{STK}}) \\
= & \cdots\cdots \quad \text{Simplification} \\
& ((\langle ov_1^{i=k} \rangle \frown \Pi_{\mathrm{STK}})\ \mathsf{proj}\ k) \in f \qquad \#\langle ov_1^{i=k} \rangle = k \\
& \vdash^{\{\Pi_{\mathrm{STK}}\}} \qquad \mathsf{tail}^k(\langle ov_1^{i=k} \rangle = k) = \langle\rangle \\
& \Pi_{\mathrm{STK}}' = \langle f((\langle ov_1^{i=k} \rangle \frown \Pi_{\mathrm{STK}})\ \mathsf{proj}\ k) \rangle \frown \Pi_{\mathrm{STK}} \\
= & \cdots\cdots \quad \langle t_1^{i=k} \rangle \frown t\ \mathsf{proj}\ k \ = \ t_{i=1}^{k}. \\
& (ov_{i=1}^{k}) \in f \\
& \vdash^{\{\Pi_{\mathrm{STK}}\}} \\
& \Pi_{\mathrm{STK}}' = \langle f(ov_{i=1}^{k}) \rangle \frown \Pi_{\mathrm{STK}} \\
= & \cdots\cdots \quad \text{A3, i.e. } (ov_{i=1}^{k}) \in f \\
& \mathsf{true} \vdash^{\{\Pi_{\mathrm{STK}}\}} \Pi_{\mathrm{STK}}' = \langle f(ov_{i=1}^{k}) \rangle \frown \Pi_{\mathrm{STK}} \\
= & \cdots\cdots \quad \text{Defn. of } \mathcal{E}_{\mathrm{S}} \\
& \mathcal{E}_{\mathrm{S}}(f(ov_{i=1}^{k}))
\end{array}$$

□

4.3.2 Procedure invocation

The only lemma that has not already been proven for showing consistency of either the method or function invocation operations is the procedure call lemma (Lemma 4.3), which is now specialised for the $\mathcal{U}_S$-design as follows

$$\mathcal{E}_S(\llbracket v \rrbracket^E_S, (\!|\ x, \llbracket e \rrbracket_S\ |\!)) \fatsemi \mathsf{apply}_S(\mathsf{call}_S) \quad = \quad \mathsf{ext}_S(\langle\!\langle e \rangle\!\rangle_S \{\!| x \leftarrow \langle\!\langle v \rangle\!\rangle^E_S |\!\}_S, \mathsf{call}_S)$$

Proof

$$\mathcal{E}_S(\llbracket v \rrbracket^E_S, (\!|\ x, \langle\!\langle e \rangle\!\rangle_S\ |\!)) \fatsemi \mathsf{apply}_S(\mathsf{call}_S)$$

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

$$(\mathsf{true} \vdash^{\{\Pi_{STK}\}} \Pi_{STK}' = \langle(\llbracket v \rrbracket^E_S, (\!|\ x, \langle\!\langle e \rangle\!\rangle_S\ |\!))\rangle \frown \Pi_{STK}) \fatsemi \\ \mathsf{apply}_S(\mathsf{call}_S)$$

= .... Defn. of apply_S

$$\begin{array}{l} (\mathsf{true} \vdash^{\{\Pi_{STK}\}} \Pi_{STK}' = \langle(\llbracket v \rrbracket^E_S, (\!|\ x, \langle\!\langle e \rangle\!\rangle_S\ |\!))\rangle \frown \Pi_{STK}) \fatsemi \\ (\ \ (\ \ \exists v_1, x_1, t_1 \mid (v_1, (\!|\ x_1, t_1\ |\!)) = \mathsf{head}\,\Pi_{STK} \bullet \\ \qquad \mathsf{pop}_S \fatsemi \mathsf{ext}_S((\!|\ t_1\{\!| x_1 \leftarrow \mathsf{text}_S(v_1) |\!\}_S\ |\!), \mathsf{call}_S) \\ \quad) \\ \quad \lhd \#\Pi_{STK} > 1 \ \ \wedge\ \ (\mathsf{head}\,\Pi_{STK}) \in \mathit{Value} \times \mathit{ProgramText} \rhd \\ \quad \mathsf{chaos} \\ ) \end{array}$$

= .... Defn. of pop_S

$$\begin{array}{l} (\mathsf{true} \vdash^{\{\Pi_{STK}\}} \Pi_{STK}' = \langle(\llbracket v \rrbracket^E_S, (\!|\ x, \langle\!\langle e \rangle\!\rangle_S\ |\!))\rangle \frown \Pi_{STK}) \fatsemi \\ (\ \ (\ \ \exists v_1, x_1, t_1 \mid (v_1, (\!|\ x_1, t_1\ |\!)) = \mathsf{head}\,\Pi_{STK} \bullet \\ \qquad (\#\Pi_{STK} > 1 \vdash^{\{\Pi_{STK}\}} \Pi_{STK}' = \mathsf{tail}\,\Pi_{STK}) \fatsemi \\ \qquad \mathsf{ext}_S((\!|\ t_1\{\!| x_1 \leftarrow \mathsf{text}_S(v_1) |\!\}_S\ |\!), \mathsf{call}_S) \\ \quad) \\ \quad \lhd \#\Pi_{STK} > 1 \ \ \wedge\ \ (\mathsf{head}\,\Pi_{STK}) \in \mathit{Value} \times \mathit{ProgramText} \rhd \\ \quad \mathsf{chaos} \\ ) \end{array}$$

= .... Defn. of $\_ \fatsemi \_$ and predicate logic

$$\begin{array}{l} (\ \ \exists v_1, x_1, t_1 \mid (v_1, (\!|\ x_1, t_1\ |\!)) = (\llbracket v \rrbracket^E_S, (\!|\ x, \langle\!\langle e \rangle\!\rangle_S\ |\!)) \bullet \\ \qquad (\mathsf{true} \vdash^{\{\Pi_{STK}\}} \Pi_{STK}' = \Pi_{STK}) \fatsemi \\ \qquad \mathsf{ext}_S((\!|\ t_1\{\!| x_1 \leftarrow \mathsf{text}_S(v_1) |\!\}_S\ |\!), \mathsf{call}_S) \\ ) \\ \lhd \mathsf{true} \ \ \wedge\ \ (\llbracket v \rrbracket^E_S, (\!|\ x, \langle\!\langle e \rangle\!\rangle_S\ |\!)) \in \mathit{Value} \times \mathit{ProgramText} \rhd \\ \mathsf{chaos} \end{array}$$

= .... Defn. of $\_ \lhd \_ \rhd \_$ and predicate logic

$$\begin{array}{l} \exists v_1, x_1, t_1 \mid (v_1, (\!|\ x_1, t_1\ |\!)) = (\llbracket v \rrbracket^E_S, (\!|\ x, \langle\!\langle e \rangle\!\rangle_S\ |\!)) \bullet \\ \quad (\mathsf{true} \vdash^{\{\Pi_{STK}\}} \Pi_{STK}' = \Pi_{STK}) \fatsemi \\ \quad \mathsf{ext}_S((\!|\ t_1\{\!| x_1 \leftarrow \mathsf{text}_S(v_1) |\!\}_S\ |\!), \mathsf{call}_S) \end{array}$$

= .... One point rule and defn of skip^D

$$\mathsf{skip}^D \fatsemi \\ \mathsf{ext}_S((\!|\ \langle\!\langle e \rangle\!\rangle_S\{\!| x \leftarrow \mathsf{text}_S(\llbracket v \rrbracket^E_S) |\!\}_S\ |\!), \mathsf{call}_S)$$

= .... skip^D unit of $\_ \fatsemi \_$ and defn. of text_S

$$\mathsf{ext}_S((\!|\ \langle\!\langle e \rangle\!\rangle_S\{\!| x \leftarrow \langle\!\langle v \rangle\!\rangle^E_S |\!\}_S\ |\!), \mathsf{call}_S)$$

□

4.3.3 Heap operations

Fresh heap locations: The commuting diagram in Figure 4.7 outlines the structure of the proof that the fresh operator in the $\mathcal{O}$-calculus has a consistent denotational semantics. Here we assume that:

$A7$ The expression $\mathsf{fresh}_{\mathrm{LOC}}(\mathrm{dom}\, H_0)$ evaluates to ℓ_j.
$A8$ The context-heap value H_1 is $H_0 \oplus \{\ell_j \mapsto ¿\}$.

The fresh location lemma (Lemma 4.4) is specialised for the $\mathcal{U}_{\mathrm{S}}$-design as follows

$$(\Pi_{\mathrm{HEAP}} := H) \,⨾\, \mathsf{fresh}_{\mathrm{S}} \quad = \quad (\Pi_{\mathrm{HEAP}} := H \oplus \{r \mapsto ¿\}) \,⨾\, (\mathcal{E}_{\mathrm{S}}\, r)$$

where $r = \mathsf{fresh}_{\mathrm{LOC}}(\mathrm{dom}\, \Pi_{\mathrm{HEAP}})$.

Proof

$(\Pi_{\mathrm{HEAP}} := H) \,⨾\, \mathsf{fresh}_{\mathrm{S}}$

$=$ .... Defn. of $\mathsf{fresh}_{\mathrm{S}}$

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

$=$ .... Law C.13 assign comp.

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

$=$ A7

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

$=$ .... 1-point rule

$\Pi_{\mathrm{STK}}, \Pi_{\mathrm{HEAP}} := \langle \ell_j \rangle \frown \Pi_{\mathrm{STK}}, H \oplus \{\ell_j \mapsto ¿\}$

$=$ .... Defns. of $\_ \,⨾\, \_$ and $\mathcal{E}_{\mathrm{S}}$

$(\Pi_{\mathrm{HEAP}} := H \oplus \{\ell_j \mapsto ¿\}) \,⨾\, (\mathcal{E}_{\mathrm{S}}\, \ell_j)$

□

Dereferencing heap locations: The commuting diagram in Figure 4.8 outlines the structure of the proof that the dereference operator in the $\mathcal{O}$-calculus has a consistent denotational semantics. Here we assume that $r \Subset H$ and $v = H(r)$.

The dereference location lemma (Lemma 4.5) is specialised for the $\mathcal{U}_{\mathrm{S}}$-design as follows

$$(\Pi_{\mathrm{HEAP}} := H) \,⨾\, \mathsf{fresh}_{\mathrm{S}} \quad = \quad (\Pi_{\mathrm{HEAP}} := H \oplus \{r \mapsto ¿\}) \,⨾\, (\mathcal{E}_{\mathrm{S}}\, r)$$

where $r = \mathsf{fresh}_{\mathrm{LOC}}(\mathrm{dom}\, \Pi_{\mathrm{HEAP}})$.

Proof

$\Pi_{\mathrm{HEAP}} := H \,⨾\, \mathcal{E}_{\mathrm{S}}\, r \,⨾\, \mathsf{deref}_{\mathrm{S}}$

$=$ .... Defn. of $\mathcal{E}_{\mathrm{S}}$

$\Pi_{\mathrm{HEAP}} := H \,⨾\, \Pi_{\mathrm{STK}} := \langle r \rangle \frown \Pi_{\mathrm{STK}} \,⨾\, \mathsf{deref}_{\mathrm{S}}$

$=$ .... Independent assignments

$\Pi_{\mathrm{HEAP}}, \Pi_{\mathrm{STK}} := H, \langle r \rangle \frown \Pi_{\mathrm{STK}} \,⨾\, \mathsf{deref}_{\mathrm{S}}$

$$
\begin{array}{ll}
= & \cdots\cdots \text{Defn. of } \mathsf{deref}_{\mathrm{S}} \\
& \Pi_{\mathrm{HEAP}}, \Pi_{\mathrm{STK}} := H, \langle r \rangle \frown \Pi_{\mathrm{STK}} \fatsemi \\
& (\ \#\Pi_{\mathrm{STK}} > 0 \wedge (\mathsf{head}\, \Pi_{\mathrm{STK}}) \in \Pi_{\mathrm{HEAP}} \\
& \quad \vdash_{\{\Pi_{\mathrm{STK}}\}} \\
& \quad \Pi_{\mathrm{STK}}' = \langle \Pi_{\mathrm{HEAP}}(\mathsf{head}\, \Pi_{\mathrm{STK}}) \rangle \frown (\mathsf{tail}\, \Pi_{\mathrm{STK}}) \\
&) \\
= & \cdots\cdots \text{Law C.13 assign comp.} \\
& (\ \#(\langle r \rangle \frown \Pi_{\mathrm{STK}}) > 0 \wedge \mathsf{head}(\langle r \rangle \frown \Pi_{\mathrm{STK}}) \in H \\
& \quad \vdash \\
& \quad \Pi_{\mathrm{HEAP}}' = H \wedge \\
& \quad \Pi_{\mathrm{STK}}' = \langle H(\mathsf{head}(\langle r \rangle \frown \Pi_{\mathrm{STK}})) \rangle \frown \mathsf{tail}(\langle r \rangle \frown \Pi_{\mathrm{STK}}) \\
&) \\
= & \cdots\cdots \text{Simplification} \\
& (\ 1 + \#\Pi_{\mathrm{STK}} > 0 \wedge r \in H \\
& \quad \vdash \\
& \quad \Pi_{\mathrm{HEAP}}' = H \wedge \Pi_{\mathrm{STK}}' = \langle H(r) \rangle \frown \Pi_{\mathrm{STK}} \\
&) \\
= & \cdots\cdots \text{Assumption A9} \\
& (\ \mathsf{true} \\
& \quad \vdash \\
& \quad \Pi_{\mathrm{HEAP}}' = H \wedge \Pi_{\mathrm{STK}}' = \langle v \rangle \frown \Pi_{\mathrm{STK}} \\
&) \\
= & \cdots\cdots \text{Defn. of } \_ := \_ \\
& \Pi_{\mathrm{HEAP}}, \Pi_{\mathrm{STK}} := H, \langle v \rangle \frown \Pi_{\mathrm{STK}} \\
= & \cdots\cdots \text{Independent assignments} \\
& \Pi_{\mathrm{HEAP}} := H \fatsemi \Pi_{\mathrm{STK}} := \langle v \rangle \frown \Pi_{\mathrm{STK}} \\
= & \cdots\cdots \text{Defn. of } \mathcal{E}_{\mathrm{S}} \\
& \Pi_{\mathrm{HEAP}} := H \fatsemi \mathcal{E}_{\mathrm{S}}\, v
\end{array}
$$

□

4.3.4 Other programming operations

Sequential composition operation: The sequential composition operation structural induction (of Section 4.2.5) had one outstanding lemma (Lemma 4.7), which is specialised for the $\mathcal{U}_{\mathrm{S}}$-design as follows

$$\mathcal{E}_{\mathrm{S}}\, ov \fatsemi \mathsf{pop}_{\mathrm{S}} \fatsemi P \quad = \quad P$$

Proof

$$
\begin{array}{ll}
& \mathcal{E}_{\mathrm{S}}\, ov \fatsemi \mathsf{pop}_{\mathrm{S}} \fatsemi P \\
= & \cdots\cdots \text{Defn. of } \mathcal{E}_{\mathrm{S}} \\
& (\Pi_{\mathrm{STK}} := \langle ov \rangle \frown \Pi_{\mathrm{STK}}) \fatsemi \mathsf{pop}_{\mathrm{S}} \fatsemi P \\
= & \cdots\cdots \text{Defn. of } \mathsf{pop}_{\mathrm{S}} \\
& (\Pi_{\mathrm{STK}} := \langle ov \rangle \frown \Pi_{\mathrm{STK}}) \fatsemi \\
& (\#\Pi_{\mathrm{STK}} > 0 \vdash^{\{\Pi_{\mathrm{STK}}\}} \Pi_{\mathrm{STK}}' = \mathsf{tail}\, \Pi_{\mathrm{STK}}) \fatsemi \\
& P
\end{array}
$$

$= \cdots$ Law C.13 assign comp.

$$\left(\begin{array}{l} \#(\langle ov \rangle \frown \Pi_{\text{STK}}) > 0 \\ \vdash^{\{\Pi_{\text{STK}}\}} \\ \Pi_{\text{STK}}' = \mathsf{tail}(\langle ov \rangle \frown \Pi_{\text{STK}}) \end{array} \right) ⨟ P$$

$= \cdots$ Simplification

$$(1 + \#\Pi_{\text{STK}} > 0 \vdash^{\{\Pi_{\text{STK}}\}} \Pi_{\text{STK}}' = \Pi_{\text{STK}}) ⨟ P$$

$= \cdots$ Simplification

$$(\mathsf{true} \vdash^{\emptyset} \mathsf{true}) ⨟ P$$

$= \cdots$ Defn. skip^{D}

$$\mathsf{skip}^{\text{D}} ⨟ P$$

$= \cdots$ skip^{D} unit of $\_ ⨟ \_$

$$P$$

□

Conditional operation: The conditional operation structural induction (of Section 4.2.5) left two undischarged lemmas, which corresponded to the true and false branches of the conditional statement. We now prove the false conditional lemma (Lemma 4.9), which is specialised for the $\mathcal{U}_{\text{S}}$-design as follows

$$\mathcal{E}_{\text{S}}\, \mathsf{false} ⨟ \mathsf{switch}_{\text{S}}(P, Q) \quad = \quad Q$$

Proof

$$\mathcal{E}_{\text{S}}\, \mathsf{false} ⨟ \mathsf{switch}_{\text{S}}(P, Q)$$

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

$$\Pi_{\text{STK}} := \langle \mathsf{false} \rangle \frown \Pi_{\text{STK}} ⨟ \mathsf{switch}_{\text{S}}(P, Q)$$

$= \cdots$ Defn. of $\mathsf{switch}_{\text{S}}$

$$\begin{array}{l} \Pi_{\text{STK}} := \langle \mathsf{false} \rangle \frown \Pi_{\text{STK}} ⨟ \\ (\ (\ \mathsf{pop}_{\text{S}} ⨟ P \ \lhd \mathsf{head}\, \Pi_{\text{STK}} \rhd \ \mathsf{pop}_{\text{S}} ⨟ Q\ ) \\ \quad \lhd \#\Pi_{\text{STK}} > 0 \wedge (\mathsf{head}\, \Pi_{\text{STK}}) \in \mathit{Boolean} \rhd \\ \quad \mathsf{chaos} \\ ) \end{array}$$

$= \cdots$ Defn. of pop_{S}

$$\begin{array}{l} \Pi_{\text{STK}} := \langle \mathsf{false} \rangle \frown \Pi_{\text{STK}} ⨟ \\ (\ (\ (\#\Pi_{\text{STK}} > 0 \vdash^{\{\Pi_{\text{STK}}\}} \Pi_{\text{STK}}' = \mathsf{tail}\, \Pi_{\text{STK}}) ⨟ P \\ \qquad \lhd \mathsf{head}\, \Pi_{\text{STK}} \rhd \\ \qquad (\#\Pi_{\text{STK}} > 0 \vdash^{\{\Pi_{\text{STK}}\}} \Pi_{\text{STK}}' = \mathsf{tail}\, \Pi_{\text{STK}}) ⨟ Q \\ \quad) \\ \quad \lhd \#\Pi_{\text{STK}} > 0 \wedge (\mathsf{head}\, \Pi_{\text{STK}}) \in \mathit{Boolean} \rhd \\ \quad \mathsf{chaos} \\ ) \end{array}$$

$= \cdots$ Law C.13 assign comp.

$$\begin{array}{l} (\ (\#(\langle \mathsf{false} \rangle \frown \Pi_{\text{STK}}) > 0 \vdash^{\{\Pi_{\text{STK}}\}} \Pi_{\text{STK}}' = \mathsf{tail}(\langle \mathsf{false} \rangle \frown \Pi_{\text{STK}}) ⨟ \\ P \\ \quad \lhd \mathsf{head}(\langle \mathsf{false} \rangle \frown \Pi_{\text{STK}}) \rhd \\ \quad (\#(\langle \mathsf{false} \rangle \frown \Pi_{\text{STK}}) > 0 \vdash^{\{\Pi_{\text{STK}}\}} \Pi_{\text{STK}}' = \mathsf{tail}(\langle \mathsf{false} \rangle \frown \Pi_{\text{STK}}) ⨟ \\ \quad Q \\ ) \\ \lhd \#(\langle \mathsf{false} \rangle \frown \Pi_{\text{STK}}) > 0 \wedge \mathsf{head}(\langle \mathsf{false} \rangle \frown \Pi_{\text{STK}}) \in \mathit{Boolean} \rhd \\ \mathsf{chaos} \end{array}$$

$$
\begin{array}{ll}
= & \cdots\cdots \text{Simplification} \\
& (\ (1 + \#\Pi_{\text{STK}} > 0 \vdash^{\{\Pi_{\text{STK}}\}} \Pi_{\text{STK}}' = \Pi_{\text{STK}}) \fatsemi P \\
& \quad \lhd \mathsf{false} \rhd \\
& \quad (1 + \#\Pi_{\text{STK}} > 0 \vdash^{\{\Pi_{\text{STK}}\}} \Pi_{\text{STK}}' = \Pi_{\text{STK}}) \fatsemi Q \\
&) \\
& \lhd 1 + \#\Pi_{\text{STK}} > 0 \land \mathsf{false} \in \mathit{Boolean} \rhd \\
& \mathsf{chaos} \\
= & \cdots\cdots \text{Simplification} \\
& (\ (\mathsf{true} \vdash^{\emptyset} \mathsf{true}) \fatsemi P \\
& \quad \lhd \mathsf{false} \rhd \\
& \quad (\mathsf{true} \vdash^{\emptyset} \mathsf{true}) \fatsemi Q \\
&) \\
& \lhd \mathsf{true} \rhd \\
& \mathsf{chaos} \\
= & \cdots\cdots \text{Defn. of } \_ \lhd \_ \rhd \_ \\
& (\mathsf{true} \vdash^{\emptyset} \mathsf{true}) \fatsemi Q \\
= & \cdots\cdots \text{Defn. of } \mathsf{skip}^{\text{D}} \\
& \mathsf{skip}^{\text{D}} \fatsemi Q \\
= & \cdots\cdots \mathsf{skip}^{\text{D}} \text{ unit of } \_ \fatsemi \_ \\
& Q
\end{array}
$$

□

The true conditional lemma (Lemma 4.8) is specialised and proved in a similar manner; the $\mathcal{U}_{\text{S}}$-design specialised variant is

$$\mathcal{E}_{\text{S}}\,\mathsf{true} \fatsemi \mathsf{switch}_{\text{S}}(P, Q) \quad = \quad P$$

4.4 Result Variable Model

We now prove the lemmas that require model specific steps for the $\mathcal{U}_{\text{R}}$-design of the $\mathcal{O}$-calculus.

4.4.1 Command expressions

The command expression lemma (Lemma 4.1) is now specialised for the $\mathcal{U}_{\text{R}}$-design model, as follows

$$(ov_{i=1}^{k}) \Elzinv f \quad \Rightarrow \quad \mathsf{cmdExp}_{\text{R}}(\mathsf{trans}_{\text{R}}(f, k), (\fatsemi_{i=1}^{k} \mathcal{E}_{\text{R}}\ ov_i)) = \mathcal{E}_{\text{R}}(f(ov_{i=1}^{k}))$$

Within the following proof, the LHS of the initial implication is added as an assumption to the proof context.

Proof

$$
\begin{array}{ll}
& \mathsf{cmdExp}_{\text{R}}(\mathsf{trans}_{\text{R}}(f, k), (\fatsemi_{i=1}^{k} \mathcal{E}_{\text{R}}\ ov_i)) \\
= & \cdots\cdots \text{Defn. of } \mathsf{cmdExp}_{\text{R}} \\
& \mathsf{decl}\ x_{i=1}^{k}\ \mathsf{in} \\
& \quad (\fatsemi_{j=1}^{k} \mathcal{E}_{\text{R}}\ ov_j \upharpoonright x_{i=1}^{k} \fatsemi x_j := \Pi_{\text{RES}}) \fatsemi \\
& \quad \mathsf{trans}_{\text{R}}(f, k) \\
= & \cdots\cdots \text{Defn. of } \_ \upharpoonright \_ \fatsemi \_ \\
& \mathsf{decl}\ x_{i=1}^{k}\ \mathsf{in} \\
& \quad (\fatsemi_{j=1}^{k} (\mathsf{hide}\ x_{i=1}^{k}\ \mathsf{from}\ \mathcal{E}_{\text{R}}\ ov_j) \fatsemi x_j := \Pi_{\text{RES}}) \fatsemi \\
& \quad \mathsf{trans}_{\text{R}}(f, k)
\end{array}
$$

$=$ ..... Defn. of $\mathcal{E}_{\mathrm{R}}$ and $\_ := \_$

$$
\begin{array}{l}
\mathsf{decl}\ x_{i=1}^{k}\ \mathsf{in} \\
\quad (\fatsemi_{j=1}^{k} \\
\qquad (\mathsf{hide}\ x_{i=1}^{k}\ \mathsf{from} \\
\qquad\quad (\mathsf{true} \vdash \Pi_{\mathrm{RES}}{}' = ov_j \wedge \Pi_{\mathrm{HEAP}}{}' = \Pi_{\mathrm{HEAP}}) \\
\qquad) \fatsemi \\
\qquad x_j := \Pi_{\mathrm{RES}} \\
\quad) \fatsemi \\
\quad \mathsf{trans}_{\mathrm{R}}(f,k)
\end{array}
$$

$=$ ..... Law C.7 hide design

$$
\begin{array}{l}
\mathsf{decl}\ x_{i=1}^{k}\ \mathsf{in} \\
\quad (\fatsemi_{j=1}^{k} \\
\qquad (\ \mathsf{true} \\
\qquad\quad \vdash \\
\qquad\quad \Pi_{\mathrm{RES}}{}' = ov_j \wedge \Pi_{\mathrm{HEAP}}{}' = \Pi_{\mathrm{HEAP}} \wedge x'^{k}_{i=1} = x^{k}_{i=1} \\
\qquad) \fatsemi \\
\qquad x_j := \Pi_{\mathrm{RES}} \\
\quad) \fatsemi \\
\quad \mathsf{trans}_{\mathrm{R}}(f,k)
\end{array}
$$

$=$ ..... defn. $\_ := \_$

$$
\begin{array}{l}
\mathsf{decl}\ x_{i=1}^{k}\ \mathsf{in} \\
\quad (\fatsemi_{j=1}^{k}\ \Pi_{\mathrm{RES}} := ov_j \fatsemi x_j := \Pi_{\mathrm{RES}}) \fatsemi \\
\quad \mathsf{trans}_{\mathrm{R}}(f,k)
\end{array}
$$

$=$ ..... Transitivity of assignment

$$
\begin{array}{l}
\mathsf{decl}\ x_{i=1}^{k}\ \mathsf{in} \\
\quad (\fatsemi_{j=1}^{k}\ \Pi_{\mathrm{RES}} := ov_j \fatsemi x_j := ov_j) \fatsemi \\
\quad \mathsf{trans}_{\mathrm{R}}(f,k)
\end{array}
$$

$=$ ..... Independent assignments

$$
\begin{array}{l}
\mathsf{decl}\ x_{i=1}^{k}\ \mathsf{in} \\
\quad (\fatsemi_{j=1}^{k}\ x_j := ov_j \fatsemi \Pi_{\mathrm{RES}} := ov_j) \fatsemi \\
\quad \mathsf{trans}_{\mathrm{R}}(f,k)
\end{array}
$$

$=$ ..... Simplification

$$
\begin{array}{l}
\mathsf{decl}\ x_{i=1}^{k}\ \mathsf{in} \\
\quad x_{j=1}^{k}, \Pi_{\mathrm{RES}} := ov_{j=1}^{k}, ov_k \fatsemi \\
\quad \mathsf{trans}_{\mathrm{R}}(f,k)
\end{array}
$$

$=$ ..... Defn. of $\mathsf{trans}_{\mathrm{R}}$

$$
\begin{array}{l}
\mathsf{decl}\ x_{i=1}^{k}\ \mathsf{in} \\
\quad x_{j=1}^{k}, \Pi_{\mathrm{RES}} := ov_{j=1}^{k}, ov_k \fatsemi \\
\quad (x_{i=1}^{k} \Subset f \vdash^{\{\Pi_{\mathrm{RES}}\}} \Pi_{\mathrm{RES}}{}' = f(x_{i=1}^{k})
\end{array}
$$

$=$ ..... Law C.13 assign comp.

$$
\begin{array}{l}
\mathsf{decl}\ x_{i=1}^{k}\ \mathsf{in} \\
\quad (ov_{i=1}^{k}) \Subset f \vdash^{\{\Pi_{\mathrm{RES}}\}} \Pi_{\mathrm{RES}} = f(ov_{i=1}^{k})
\end{array}
$$

$=$ ..... A3, i.e. $(ov_{i=1}^{k}) \Subset f$

$$
\begin{array}{l}
\mathsf{decl}\ x_{i=1}^{k}\ \mathsf{in} \\
\quad \mathsf{true} \vdash^{\{\Pi_{\mathrm{RES}}\}} \Pi_{\mathrm{RES}}{}' = f(ov_{i=1}^{k})
\end{array}
$$

$=$ ..... Defn. of $\mathcal{E}_{\mathrm{R}}$

$$
\begin{array}{l}
\mathsf{decl}\ x_{i=1}^{k}\ \mathsf{in} \\
\quad \Pi_{\mathrm{RES}} := f(ov_{i=1}^{k})
\end{array}
$$

$=$ ..... Defn. of decl

$$
\mathsf{var}\ x_{i=1}^{k} \fatsemi \Pi_{\mathrm{RES}} := f(ov_{i=1}^{k}) \fatsemi \mathsf{end}\ x_{i=1}^{k}
$$

$$
\begin{array}{lll}
= & \cdots & x_{i=1}^{k} \text{ not in assignment} \\
 & \mathsf{var}\ x_{i=1}^{k} \mathbin{⨾} \mathsf{end}\ x_{i=1}^{k} \mathbin{⨾} \Pi_{\text{RES}} := f(ov_{i=1}^{k}) & \\
= & \cdots & \text{Unused local vars.} \\
 & \mathsf{skip}^{\text{D}} \mathbin{⨾} \Pi_{\text{RES}} := f(ov_{i=1}^{k}) & \\
= & \cdots & \text{Defn. of } \mathcal{E}_{\text{R}}\text{, and } \mathsf{skip}^{\text{D}} \text{ unit of } \_ \mathbin{⨾} \_ \\
 & \mathcal{E}_{\text{R}}\, f(ov_{i=1}^{k}) &
\end{array}
$$

□

4.4.2 Heap operations

Dereferencing a heap location: The dereference location lemma (Lemma 4.5) is specialised for the $\mathcal{U}_{\text{S}}$-design as follows

$$(\Pi_{\text{HEAP}} := H) \mathbin{⨾} \mathsf{fresh}_{\text{R}} \quad = \quad (\Pi_{\text{HEAP}} := H \oplus \{r \mapsto ¿\}) \mathbin{⨾} (\mathcal{E}_{\text{R}}\, r)$$

where $r = \mathsf{fresh}_{\text{LOC}}(\operatorname{dom} \Pi_{\text{HEAP}})$.

Proof

$$
\begin{array}{lll}
 & \Pi_{\text{HEAP}} := H \mathbin{⨾} \mathcal{E}_{\text{R}}\, r \mathbin{⨾} \mathsf{deref}_{\text{R}} & \\
= & \cdots & \text{Defn. of } \mathcal{E}_{\text{R}} \\
 & \Pi_{\text{HEAP}} := H \mathbin{⨾} \Pi_{\text{RES}} := r \mathbin{⨾} \mathsf{deref}_{\text{R}} & \\
= & \cdots & \text{Independent assignments} \\
 & \Pi_{\text{HEAP}}, \Pi_{\text{RES}} := H, r \mathbin{⨾} \mathsf{deref}_{\text{R}} & \\
= & \cdots & \text{Defn. of } \mathsf{deref}_{\text{R}} \\
 & \Pi_{\text{HEAP}}, \Pi_{\text{RES}} := H, r \mathbin{⨾} & \\
 & (\Pi_{\text{RES}} \Subset \Pi_{\text{HEAP}} \vdash^{\{\Pi_{\text{RES}}\}} \Pi_{\text{RES}}' = \Pi_{\text{HEAP}}(\Pi_{\text{RES}})) & \\
= & \cdots & \text{Law C.13 assign comp.} \\
 & r \Subset H \vdash \Pi_{\text{HEAP}}' = H \wedge \Pi_{\text{RES}}' = H(r) & \\
= & \cdots & \text{Assumption A9} \\
 & \mathsf{true} \vdash \Pi_{\text{HEAP}}' = H \wedge \Pi_{\text{RES}}' = v & \\
= & \cdots & \text{Defn. of } \_ := \_ \\
 & \Pi_{\text{HEAP}}, \Pi_{\text{RES}} := H, v & \\
= & \cdots & \text{Independent assignments} \\
 & \Pi_{\text{HEAP}} := H \mathbin{⨾} \Pi_{\text{RES}} := v & \\
= & \cdots & \text{Defn. of } \mathcal{E}_{\text{R}} \\
 & \Pi_{\text{HEAP}} := H \mathbin{⨾} \mathcal{E}_{\text{R}}\, v &
\end{array}
$$

□

Updating a heap location: The updating location lemma (Lemma 4.6) is specialised for the $\mathcal{U}_{\text{R}}$-design as follows

$$(\Pi_{\text{HEAP}} := H) \mathbin{⨾} \mathcal{E}_{\text{R}}(r, v) \mathbin{⨾} \mathsf{update}_{\text{R}} \quad = \quad (\Pi_{\text{HEAP}} := H \oplus \{r \mapsto v\}) \mathbin{⨾} (\mathcal{E}_{\text{R}}\, r)$$

where $r \Subset \Pi_{\text{HEAP}}$.

Proof (for the $\mathcal{U}_{\text{R}}$-design).

$$
\begin{array}{lll}
 & \Pi_{\text{HEAP}} := H \mathbin{⨾} \mathcal{E}_{\text{R}}(r, v) \mathbin{⨾} \mathsf{update}_{\text{R}} & \\
= & \cdots & \text{Defn. of } \mathcal{E}_{\text{R}} \\
 & \Pi_{\text{HEAP}} := H \mathbin{⨾} \Pi_{\text{RES}} := (r, v) \mathbin{⨾} \mathsf{update}_{\text{R}} & \\
= & \cdots & \text{Independent assignments} \\
 & \Pi_{\text{HEAP}}, \Pi_{\text{RES}} := H, (r, v) \mathbin{⨾} \mathsf{update}_{\text{R}} &
\end{array}
$$

$=$ ····· Defn. of $\mathsf{update}_{\mathrm{R}}$

$\Pi_{\mathrm{HEAP}}, \Pi_{\mathrm{RES}} := H, (r, v)$ ⨟
($\exists\, r_1, v_1 \mid (r_1, v_1) = \Pi_{\mathrm{RES}} \bullet$
  $\Pi_{\mathrm{RES}}, \Pi_{\mathrm{HEAP}} := r_1, \Pi_{\mathrm{HEAP}} \oplus \{r_1 \mapsto v_1\}$
  $\lhd\, r_1 \notin \Pi_{\mathrm{HEAP}} \,\rhd$
  chaos
)
$\lhd\, \Pi_{\mathrm{RES}} \in \mathit{Location} \times \mathit{Value} \,\rhd$
chaos

$=$ ····· Law C.13 assign comp.

($\exists\, r_1, v_1 \mid (r_1, v_1) = (r, v) \bullet$
  $\Pi_{\mathrm{RES}}, \Pi_{\mathrm{HEAP}} := r_1, H \oplus \{r_1 \mapsto v_1\}$
  $\lhd\, r_1 \notin H \,\rhd$
  chaos
)
$\lhd\, (r, v) \in \mathit{Location} \times \mathit{Value} \,\rhd$
chaos

$=$ ····· $(r, v) \in \mathit{Location} \times \mathit{Value}$

$\exists\, r_1, v_1 \mid (r_1, v_1) = (r, v) \bullet$
  $\Pi_{\mathrm{RES}}, \Pi_{\mathrm{HEAP}} := r_1, H \oplus \{r_1 \mapsto v_1\}$
  $\lhd\, r_1 \notin H \,\rhd$
  chaos

$=$ ····· One point rule

$\Pi_{\mathrm{RES}}, \Pi_{\mathrm{HEAP}} := r, H \oplus \{r \mapsto v\}$
$\lhd\, r \notin H \,\rhd$
chaos

$=$ ····· Assumption A10, $r \notin H$

$\Pi_{\mathrm{RES}}, \Pi_{\mathrm{HEAP}} := r, H \oplus \{r \mapsto v\}$

$=$ ····· Independent assignments

$\Pi_{\mathrm{HEAP}} := H \oplus \{r \mapsto v\}$ ⨟ $\Pi_{\mathrm{RES}} := r$

$=$ ····· Defn. of $\mathcal{E}_{\mathrm{R}}$

$\Pi_{\mathrm{HEAP}} := H \oplus \{r \mapsto v\}$ ⨟ $\mathcal{E}_{\mathrm{R}}\, r$

□

4.4.3 Other programming operations

Sequential composition operation: The sequential composition operation structural induction (of Section 4.2.5) had one outstanding lemma (Lemma 4.7), which is specialised for the $\mathcal{U}_{\mathrm{R}}$-design as follows

$$\mathcal{E}_{\mathrm{R}}\, ov \fatsemi P \quad = \quad P$$

where $\forall\, ov \bullet P = P[ov/\Pi_{\mathrm{RES}}]$, which is assumption A9.

Proof

$\mathcal{E}_{\mathrm{R}}\, ov \fatsemi P$

$=$ ····· Defn. of $\mathcal{E}_{\mathrm{R}}$

$\Pi_{\mathrm{RES}} := ov \fatsemi P$

$=$ ····· Law C.13 assign comp.

$P[ov/\Pi_{\mathrm{RES}}]$

$=$ ····· A11, $\forall\, ov \bullet P = P[ov/\Pi_{\mathrm{RES}}]$

P

□

Conditional operation: The conditional operation structural induction (of Section 4.2.5) left two undischarged lemmas, which corresponded to the true and false branches of the conditional statement. We now prove the false conditional lemma (Lemma 4.9), which is specialised for the $\mathcal{U}_{\mathrm{R}}$-design as follows

$$\mathcal{E}_{\mathrm{R}}\,\mathsf{false} \fatsemi \mathsf{switch}_{\mathrm{R}}(P, Q) \quad = \quad Q$$

where $\forall\, b \bullet P = P[b/\Pi_{\mathrm{RES}}]$ and $\forall\, b \bullet Q = Q[b/\Pi_{\mathrm{RES}}]$, which is assumption $A10$.

Proof

$$\begin{array}{ll}
& \mathcal{E}_{\mathrm{R}}\,\mathsf{false} \fatsemi \mathsf{switch}_{\mathrm{R}}(P, Q) \\
= & \cdots\cdots \text{Defn. of } \mathcal{E}_{\mathrm{R}} \\
& \Pi_{\mathrm{RES}} := \mathsf{false} \fatsemi \mathsf{switch}_{\mathrm{R}}(P, Q) \\
= & \cdots\cdots \text{Defn. of } \mathsf{switch}_{\mathrm{R}} \\
& \Pi_{\mathrm{RES}} := \mathsf{false} \fatsemi \\
& (\ (P \lhd \mathsf{head}\,\Pi_{\mathrm{RES}} \rhd Q) \\
& \quad \lhd \Pi_{\mathrm{RES}} \in \mathit{Boolean} \rhd \\
& \quad \mathsf{chaos} \\
&) \\
= & \cdots\cdots \text{Law C.13 assign comp.} \\
& (\ (P[\mathsf{false}/\Pi_{\mathrm{RES}}] \lhd \mathsf{false} \rhd Q[\mathsf{false}/\Pi_{\mathrm{RES}}]) \\
& \quad \lhd \mathsf{false} \in \mathit{Boolean} \rhd \\
& \quad \mathsf{chaos}[\mathsf{false}/\Pi_{\mathrm{RES}}] \\
&) \\
= & \cdots\cdots \text{Defn. of } _ \lhd _ \rhd _ \\
& Q[\mathsf{false}/\Pi_{\mathrm{RES}}] \\
= & \cdots\cdots \text{A12, } Q = Q[\mathsf{false}/\Pi_{\mathrm{RES}}] \\
& Q
\end{array}$$

□

The true conditional lemma (Lemma 4.8) is specialised and proved in a similar manner; the $\mathcal{U}_{\mathrm{R}}$-design specialised variant is

$$\mathcal{E}_{\mathrm{R}}\,\mathsf{true} \fatsemi \mathsf{switch}_{\mathrm{R}}(P, Q) \quad = \quad P$$

where $\forall\, b \bullet P = P[b/\Pi_{\mathrm{RES}}]$ and $\forall\, b \bullet Q = Q[b/\Pi_{\mathrm{RES}}]$, which is assumption $A10$.

4.5 Constant Map Model

We now prove the lemmas that require model specific steps for the $\mathcal{U}_{\mathrm{M}}$-design of the $\mathcal{O}$-calculus.

4.5.1 Procedure invocation

The proof for the $\mathcal{U}_{\mathrm{M}}$-design's procedure call lemma (Lemma 4.3) makes use of the following call iteration lemma, which reduces the proof to that of a single iteration (step) of a potentially recursive (or mutually recursive) procedure call.

Lemma 4.10 – Call iteration:

$$\begin{array}{lll}
\mathcal{E}_{\mathrm{M}}(v, (\!|\ x, M, pt\ |\!)) \fatsemi \mathsf{apply}_{\mathrm{M}}(\mathsf{call}_{\mathrm{M}}) & = & \mathsf{decl}\ [\![\ M\{\!|x \longmapsto \mathsf{text}_{\mathrm{M}}(v)|\!\}_{\mathrm{M}}\ ]\!]\ \mathsf{in} \\
& & [\![\ pt\{\!|x \longmapsto \mathsf{text}_{\mathrm{M}}(v)|\!\}_{\mathrm{M}}\ ]\!]
\end{array}$$

□

The proof of this lemma follows at the end of this section.

We now specialise the procedure call lemma for the $\mathcal{U}_{\mathrm{M}}$-design as follows

$$\mathcal{E}_{\mathrm{M}}(\llbracket v \rrbracket^{\mathrm{E}}_{\mathrm{M}}, (\!| x, M, pt |\!)) \fatsemi \mathsf{apply}_{\mathrm{M}}(\mathsf{call}_{\mathrm{M}}) \quad = \quad \mathsf{decl}\ \{x \mapsto \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}\ \mathsf{in} \\ \llbracket\ \langle\!\langle e \{\!|^{k}_{i=1} y_i \leftarrowtail v_i |\!\} \rangle\!\rangle_{\mathrm{M}} \{x \leftarrowtail \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}_{\mathrm{M}}\ \rrbracket$$

where

$$pt \mathrel{\widehat{=}} e\{\!|^{k}_{i=1} y_i \leftarrowtail v_i |\!\}$$

$$M \mathrel{\widehat{=}} \{x \mapsto \mathsf{free}, (y_i \mapsto \llbracket v_i \rrbracket^{\mathrm{E}}_{\mathrm{M}})^{k}_{i=1}\}$$

Proof

$\mathcal{E}_{\mathrm{M}}(\llbracket v \rrbracket^{\mathrm{E}}_{\mathrm{M}}, (\!| x, M, pt |\!)) \fatsemi \mathsf{apply}_{\mathrm{M}}(\mathsf{call}_{\mathrm{M}})$

$=$ ..... Lemma 4.10 call iter.

$\mathsf{decl} \llbracket\ M\{x \leftarrowtail \mathsf{text}_{\mathrm{M}}(\llbracket v \rrbracket^{\mathrm{E}}_{\mathrm{M}})\}_{\mathrm{M}}\ \rrbracket\ \mathsf{in}$
$\quad \llbracket\ pt\{x \leftarrowtail \mathsf{text}_{\mathrm{M}}(\llbracket v \rrbracket^{\mathrm{E}}_{\mathrm{M}})\}_{\mathrm{M}}\ \rrbracket$

$=$ ..... Defn. $\mathsf{text}_{\mathrm{M}}$

$\mathsf{decl} \llbracket\ M\{x \leftarrowtail \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}_{\mathrm{M}}\ \rrbracket\ \mathsf{in}$
$\quad \llbracket\ pt\{x \leftarrowtail \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}_{\mathrm{M}}\ \rrbracket$

$=$ ..... Defn. M

$\mathsf{decl} \llbracket\ \{x \mapsto \mathsf{free}, (y_i \mapsto \langle\!\langle v_i \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}})^{k}_{i=1}\}\{x \leftarrowtail \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}_{\mathrm{M}}\ \rrbracket\ \mathsf{in}$
$\quad \llbracket\ pt\{x \leftarrowtail \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}_{\mathrm{M}}\ \rrbracket$

$=$ ..... Defn. of $\_\{\!|\_ \leftarrowtail \_|\!\}_{\mathrm{M}}$

$\mathsf{decl}\ \{x \mapsto \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}, (y_i \mapsto \llbracket\ \langle\!\langle v_i \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\{x \leftarrowtail \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}_{\mathrm{M}}\ \rrbracket)^{k}_{i=1}\}\ \mathsf{in}$
$\quad \llbracket\ pt\{x \leftarrowtail \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}_{\mathrm{M}}\ \rrbracket$

$=$ ..... Decl. by cases

$\mathsf{decl}\ \{x \mapsto \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}\ \mathsf{in}$
$\quad \mathsf{decl}\ \{\!|^{k}_{i=1} y_i \mapsto \llbracket\ \langle\!\langle v_i \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\{x \leftarrowtail \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}_{\mathrm{M}}\ \rrbracket\}\ \mathsf{in}$
$\quad\quad \llbracket\ pt\{x \leftarrowtail \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}_{\mathrm{M}}\ \rrbracket$

$=$ ..... Defn. of pt

$\mathsf{decl}\ \{x \mapsto \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}\ \mathsf{in}$
$\quad \mathsf{decl}\ \{\!|^{k}_{i=1} y_i \mapsto \llbracket\ \langle\!\langle v_i \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\{x \leftarrowtail \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}_{\mathrm{M}}\ \rrbracket\}\ \mathsf{in}$
$\quad\quad \llbracket\ \langle\!\langle e \rangle\!\rangle_{\mathrm{M}}\{\!|^{k}_{i=1}\ y_i \leftarrowtail v_i |\!\}_{\mathrm{M}}\{x \leftarrowtail \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}_{\mathrm{M}}\ \rrbracket$

$=$ ..... Defn. of $\_\{\!|\_ \leftarrowtail \_|\!\}_{\mathrm{M}}$

$\mathsf{decl}\ \{x \mapsto \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}\ \mathsf{in}$
$\quad \llbracket\ (\ \mathsf{decl}\ \{\!|^{k}_{i=1} y_i \mapsto \langle\!\langle v_i \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}\ \mathsf{in}$
$\quad\quad\quad \langle\!\langle e \rangle\!\rangle_{\mathrm{M}}\{\!|^{k}_{i=1}\ y_i \leftarrowtail v_i |\!\}_{\mathrm{M}}$
$\quad\quad)\{x \leftarrowtail \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}_{\mathrm{M}}\ \rrbracket$

$=$ ..... Defn. of $\langle\!\langle \_\{\!|\_ \leftarrowtail \_|\!\} \rangle\!\rangle_{\mathrm{M}}$

$\mathsf{decl}\ \{x \mapsto \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}\ \mathsf{in}$
$\quad \llbracket\ \langle\!\langle e \{\!|^{k}_{i=1}\ y_i \leftarrowtail v_i |\!\} \rangle\!\rangle_{\mathrm{M}}\{x \leftarrowtail \langle\!\langle v \rangle\!\rangle^{\mathrm{E}}_{\mathrm{M}}\}_{\mathrm{M}}\ \rrbracket$

□

Proof of the call iteration lemma (Lemma 4.10).

$\mathcal{E}_{\mathrm{M}}(v, (\!| x, M, pt |\!)) \fatsemi \mathsf{apply}_{\mathrm{M}}(\mathsf{call}_{\mathrm{M}})$

$=$ ..... Defn. of $\mathcal{E}_{\mathrm{M}}$

$\Pi_{\mathrm{RES}} := (v, (\!| x, M, pt |\!)) \fatsemi$
$\mathsf{apply}_{\mathrm{M}}(\mathsf{call}_{\mathrm{M}})$

$=$ ..... Defn. of $(\!| \_, \_, \_ |\!)$

$\Pi_{\mathrm{RES}} := (v, (\!| x, \mathsf{decl}\ M\ \mathsf{in}\ pt |\!)) \fatsemi$
$\mathsf{apply}_{\mathrm{M}}(\mathsf{call}_{\mathrm{M}})$

$$
\begin{array}{ll}
= & \cdots\cdots \text{Law C.13 assign comp.} \\
& \mathsf{apply}_{\mathrm{M}}(\mathsf{call}_{\mathrm{M}})[(v, (\!|\ x, \mathsf{decl}\ M\ \mathsf{in}\ pt\ |\!)/\Pi_{\mathrm{RES}}] \\
= & \cdots\cdots \text{Defn. of } \mathsf{apply}_{\mathrm{J}} \\
& (\ (\ \exists\, v_1, x_1, t_1 \mid (v_1, (x_1, t_1)) = \Pi_{\mathrm{RES}} \bullet \\
& \qquad \mathsf{ext}_{\mathrm{M}}(t_1 \{\!| x_1 \leftarrowtail \mathsf{text}_{\mathrm{M}}(v_1) |\!\}_{\mathrm{M}}, \mathsf{call}_{\mathrm{M}}) \\
& \ ) \\
& \ \triangleleft\, \Pi_{\mathrm{RES}} \in \mathit{Value} \times \mathit{ProgramText} \,\triangleright \\
& \ \mathsf{chaos} \\
&)[(v, (\!|\ x, \mathsf{decl}\ M\ \mathsf{in}\ pt\ |\!)/\Pi_{\mathrm{RES}}] \\
= & \cdots\cdots \text{Law C.18 extract id.} \\
& (\ (\ \exists\, v_1, x_1, t_1 \mid (v_1, (x_1, t_1)) = \Pi_{\mathrm{RES}} \bullet \\
& \qquad [\![\ t_1 \{\!| x_1 \leftarrowtail \mathsf{text}_{\mathrm{M}}(v_1) |\!\}_{\mathrm{M}}\ ]\!] \\
& \ ) \\
& \ \triangleleft\, \Pi_{\mathrm{RES}} \in \mathit{Value} \times \mathit{ProgramText} \,\triangleright \\
& \ \mathsf{chaos} \\
&)[(v, (\!|\ x, \mathsf{decl}\ M\ \mathsf{in}\ pt\ |\!)/\Pi_{\mathrm{RES}}] \\
= & \cdots\cdots \text{\_[\_/\_] distributes through } (\_ \triangleleft \_ \triangleright \_) \\
& (\ \exists\, v_1, x_1, t_1 \mid (v_1, (x_1, t_1)) = (v, (\!|\ x, \mathsf{decl}\ M\ \mathsf{in}\ pt\ |\!)) \bullet \\
& \qquad [\![\ t_1 \{\!| x_1 \leftarrowtail \mathsf{text}_{\mathrm{M}}(v_1) |\!\}_{\mathrm{M}}\ ]\!] \\
& \ ) \\
& \ \triangleleft\, (v, (\!|\ x, \mathsf{decl}\ M\ \mathsf{in}\ pt\ |\!) \in \mathit{Value} \times \mathit{ProgramText} \,\triangleright \\
& \ \mathsf{chaos}[(v, (\!|\ x, \mathsf{decl}\ M\ \mathsf{in}\ pt\ |\!)/\Pi_{\mathrm{RES}}] \\
= & \cdots\cdots \text{Select true branch} \\
& \exists\, v_1, x_1, t_1 \mid (v_1, (x_1, t_1)) = (v, (\!|\ x, \mathsf{decl}\ M\ \mathsf{in}\ pt\ |\!)) \bullet \\
& \ [\![\ t_1 \{\!| x_1 \leftarrowtail \mathsf{text}_{\mathrm{M}}(v_1) |\!\}_{\mathrm{M}}\ ]\!] \\
= & \cdots\cdots \text{One point rule} \\
& [\![\ (\mathsf{decl}\ M\ \mathsf{in}\ pt) \{\!| x \leftarrowtail \mathsf{text}_{\mathrm{M}}(v) |\!\}_{\mathrm{M}}\ ]\!] \\
= & \cdots\cdots \text{Defn. of } \_\{\!|\_|\!\}_{\mathrm{M}} \\
& \mathsf{decl}[\![\ M \{\!| x \leftarrowtail \mathsf{text}_{\mathrm{M}}(v) |\!\}_{\mathrm{M}}\ ]\!] \ \mathsf{in}\ [\![\ pt \{\!| x \leftarrowtail \mathsf{text}_{\mathrm{M}}(v) |\!\}_{\mathrm{M}}\ ]\!]
\end{array}
$$

□

4.5.2 Other operations

The $\mathcal{U}_{\mathrm{M}}$-design inherits most of its functionality from the $\mathcal{U}_{\mathrm{R}}$-design; therefore, its proofs of the other operations are almost identical to those of the $\mathcal{U}_{\mathrm{R}}$-design. The main issue in updating these proofs would be to correctly pass through (but not change) the alphabet variables representing a procedure's parameters. The only changes to such variables occur during the procedure invocation process, which we have already presented (in Section 4.5.1).

4.6 Factorial Example

We now present a worked example of one iteration of a factorial calculation in both the $\mathcal{O}$-calculus and its associated $\mathcal{U}_{\mathrm{S}}$-design. First, we provide our $\mathcal{O}$-calculus definition of the mathematical factorial operation, as the only member of the following *sFact* object.

$$sFact \ \equiv\ [\mathtt{fact} = \varsigma(x)\ \lambda(i)\ \mathsf{if}\ i < 2\ \mathsf{then}\ 1\ \mathsf{else}\ i \times x.\mathtt{fact}(i-1)]$$

Note that the `fact` method returns a function that takes any integer parameter i and calculates the factorial of i, if i is a natural number; if i is not a natural number this function returns the value 1.

In the remainder of this section we introduce some extra notation for clarifying the presentation of the examples, we illustrate one iteration of the factorial function using the $\mathcal{O}$-calculus's

deterministic reduction rules (in four steps), and we demonstrate that the $\mathcal{U}_S$-design semantics is consistent for each step of the reduction.

4.6.1 Notation

In order to simplify the presentation of this example we introduce some extra notation. First, we introduce a named variant of the operational calculus's single-step evaluation arrow, which is denoted by $\_ \xrightarrow{\text{R-NAM}} \_$, where R-NAM is the name of the rule being used to perform the evaluation step. Note that in general, this might be a sequence of nested rules (which correspond to the rules used to evaluate an argument). When this situation arises we use a comma-separated list of rule names, starting at the outer-most rule.

Second, we introduce a new syntax for the cmdExp_S command for those cases when we want to apply either an infix relation (e.g. $\_ < \_$) or an infix operation (e.g. $\_ \times \_$) to literal values.

$$P\ rel_S^{CE}\ Q \equiv \mathsf{cmdExp}_S(\mathsf{trans}_S(\_\ rel\ \_), (P, Q))$$
$$P\ op_S^{CE}\ Q \equiv \mathsf{cmdExp}_S(\mathsf{trans}_S(\_\ op\ \_), (P, Q))$$

This notation is then generalised for any binary function fn whose arguments are either a literal value or program-text.

$$fn_S^{CE}(P, Q) \equiv \mathsf{cmdExp}_S(\mathsf{trans}_S(fn), (P, Q))$$

Third (and last), we introduce a notation for providing a staged compilation of an $\mathcal{O}$-calculus program into its $\mathcal{U}_S$-design counterpart, via providing parametrised macro definitions of the form $MACRO\_NAME(x_{i=1}^k) \equiv P[x_{i=1}^k]$, where the arguments associated with the parameters $x_{i=1}^k$ can replace the occurrences of these parameters within the compiled predicate P.

4.6.2 Object calculus iteration

The factorial of a number n can be calculated by evaluating the $sFact.\texttt{fact}(n)$ $\mathcal{O}$-calculus program. Further, in order to perform at least one iteration (which in this case is a recursive call) of the `fact` method, n must be greater than 1. Therefore, let us assume that n is a literal integer value that is greater than 1, in the following $\mathcal{O}$-calculus evaluation of one iteration of the factorial calculation.

$$\begin{aligned}
\Gamma \bullet sFact.\texttt{fact}(n) &\xrightarrow{\text{INV-M}} \Gamma \bullet (\lambda(i)\ \mathsf{if}\ i < 2\ \mathsf{then}\ 1\ \mathsf{else}\ i \times sFact.\texttt{fact}(i-1))(n) \\
&\xrightarrow{\text{INV-F}} \Gamma \bullet \mathsf{if}\ n < 2\ \mathsf{then}\ 1\ \mathsf{else}\ n \times sFact.\texttt{fact}(n-1) \\
&\xrightarrow{\text{IF,LITOP}} \Gamma \bullet \mathsf{if\ false\ then}\ 1\ \mathsf{else}\ n \times sFact.\texttt{fact}(n-1) \\
&\xrightarrow{\text{IFF}} \Gamma \bullet n \times sFact.\texttt{fact}(n-1)
\end{aligned}$$

It is useful to name each of these steps, as this enables a clearer presentation of the correspondence between this operation evaluation, and the UTP denotational simplification.

$$\begin{aligned}
sFactFn &\equiv \lambda(i)\ \mathsf{if}\ i < 2\ \mathsf{then}\ 1\ \mathsf{else}\ i \times sFact.\texttt{fact}(i-1) \\
sFactCond1 &\equiv \mathsf{if}\ n < 2\ \mathsf{then}\ 1\ \mathsf{else}\ n \times sFact.\texttt{fact}(n-1) \\
sFactCond2 &\equiv \mathsf{if\ false\ then}\ 1\ \mathsf{else}\ n \times sFact.\texttt{fact}(n-1) \\
sFactIter &\equiv n \times sFact.\texttt{fact}(n-1)
\end{aligned}$$

Note that only the function value (rather than) its call is named in the first step; therefore the $\mathcal{O}$-calculus program at the end of the first step is denoted by $sFactFn(n)$.

4.6.3 UTP operand-stack design iteration

We now demonstrate that each operational reduction step is consistent with the $\mathcal{U}_{\mathrm{S}}$-design's semantics. First, we introduce our staged compilation scheme, then we present a correspondence lemma for each of the four steps; these lemmas are proved in Appendix C.3.2.

Compilation: We now present a staged compilation of the five named $\mathcal{O}$-calculus programs and three additional programs which invoke either an object's method, a function, or both.

$$
\begin{array}{lcllcl}
\langle\!\langle sFact \rangle\!\rangle_{\mathrm{S}} & = & O & \langle\!\langle sFact.fact \rangle\!\rangle_{\mathrm{S}} & = & OF(x) \\
\langle\!\langle sFactFn \rangle\!\rangle_{\mathrm{S}} & = & F(O) & \langle\!\langle sFactFn(n) \rangle\!\rangle_{\mathrm{S}} & = & SF(F(O), n) \\
\langle\!\langle sFactCond1 \rangle\!\rangle_{\mathrm{S}} & = & E1(O, n) & \langle\!\langle sFact.fact(n) \rangle\!\rangle_{\mathrm{S}} & = & SF(OF(O), n) \\
\langle\!\langle sFactCond2 \rangle\!\rangle_{\mathrm{S}} & = & E2(O, n, \mathsf{false}) & & & \\
\langle\!\langle sFactIter \rangle\!\rangle_{\mathrm{S}} & = & ITER(O, n) & & &
\end{array}
$$

where

$$
\begin{array}{lcllcl}
O & \equiv & \{\texttt{fact} \mapsto (\!|\, x, F(x) \,|\!)\} & SW(x,i) & \equiv & \mathsf{switch}_{\mathrm{S}}(\mathcal{E}_{\mathrm{S}}\, 1, ITER(x,i)) \\
F(x) & \equiv & \mathcal{E}_{\mathrm{S}}(\!|\, i, E1(x,i) \,|\!) & ITER(x,i) & \equiv & \mathcal{E}_{\mathrm{S}}\, i \times^{\mathrm{CE}}_{\mathrm{S}} SF(OF(x), i) \\
E1(x,i) & \equiv & (\mathcal{E}_{\mathrm{S}}\, i <^{\mathrm{CE}}_{\mathrm{S}} \mathcal{E}_{\mathrm{S}}\, 2) \fatsemi SW(x,i) & SF(f,i) & \equiv & vfPair^{\mathrm{CE}}_{\mathrm{S}}(f, \mathcal{E}_{\mathrm{S}}\, i) \fatsemi \mathsf{call}_{\mathrm{S}} \\
E2(x,i,b) & \equiv & \mathcal{E}_{\mathrm{S}}\, b \fatsemi SW(x,i) & OF(x) & \equiv & omPair^{\mathrm{CE}}_{\mathrm{S}}(\mathcal{E}_{\mathrm{S}}\, x, \mathcal{E}_{\mathrm{S}}\, \texttt{fact}) \fatsemi \mathsf{call}_{\mathrm{S}}
\end{array}
$$

Step 1 – Method invocation to function invocation:

Lemma 4.11 – Factorial Example Step 1:

$$[\![sFact.\texttt{fact}(n)]\!]_{\mathrm{S}} = [\![sFactFn(n)]\!]_{\mathrm{S}}$$

□

Step 2 – Function invocation to conditional expression:

Lemma 4.12 – Factorial Example Step 2:

$$[\![sFactFn(n)]\!]_{\mathrm{S}} = [\![sFactCond1]\!]_{\mathrm{S}}$$

□

Step 3 – Conditional expression decision evaluation:

Lemma 4.13 – Factorial Example Step 3:

$$[\![sFactCond1]\!]_{\mathrm{S}} = [\![sFactCond2]\!]_{\mathrm{S}}$$

□

Step 4 – Selection of else conditional expression branch:

Lemma 4.14 – Factorial Example Step 4:

$$[\![sFactCond2]\!]_{\mathrm{S}} = [\![sFactIter]\!]_{\mathrm{S}}$$

□

4.7 Summary

We have proved that the UTP semantics of the $\mathcal{O}$-calculus is consistent with that of its operational semantics. Here, the UTP semantics of an $\mathcal{O}$-calculus program is equal to the UTP semantics of the same $\mathcal{O}$-calculus program after any number of operational execution steps. The only additional requirement of a faithful encoding is that the semantics of the resultant values are the same, as it might be possible to construct a consistent mapping that changes the semantics. Here, our compilation schemes straightforwardly map both literal and object values on to their corresponding UTP values; and the function values and methods are compiled into program texts that we have already proved to be consistent. Therefore, the UTP models of our $\mathcal{O}$-calculus, in Chapter 3, are faithful.

The main contribution of this chapter to our overall thesis is to provide further evidence that the contributions claimed in the previous chapter are well founded.

Chapter 5

Unifying Theories of Locations

This chapter presents an abstract model of locations that can be used to provide a fully abstract heap for a Unifying Theories of Programming (UTP) representation of our $\mathcal{O}$-calculus. It starts by discussing the notion of a location and some of the constraints that we place upon them Section 5.1. Having done this, we present a concrete model of locations (Section 5.2), which is abstracted (Section 5.3) and then integrated into the basic relational model of the UTP (Section 5.4).

5.1 Locations

The UTP model in [CHW06] supports the notion of a compound data structure via the introduction of a record datatype, which essentially maps distinct labels to values. These labels are also used when unambiguously specifying the location of a value and determining whether it is shared.

An object can be modelled in a similar manner to that of a record. For example, in C^{++} and $C^{\#}$ the object and record types are defined by the `class` and `struct` datatype constructors respectively. Here, a variable of an object type contains a pointer to an object, whereas a variable of a record type contains the record itself. It is this distinction between variables of object and record types that we believe is important to explicitly model in a general theory of UTP pointers. Specifically, the contents of a record are duplicated, whereas the contents of an object are aliased (shared).

The UML class diagram in Figure 5.1 provides a high level overview of our model; it illustrates the relationships between abstract locations and the values contained within them. Here an abstract location is either shareable or containable, but not both. All such locations store precisely one value, which can be a reference, a literal or a compound value. Reference values can contain either the null value or the address of a shareable location. Compound values can contain any number of distinctly named fields, where each field is stored in a separate containable location.

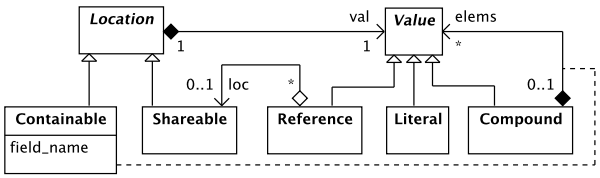


Figure 5.1: Location Model – Class Diagram

5.1.1 Terminology

The notions of a location, pointer, reference and object-identity are used to provide a means for storing or locating a value. Precisely what is meant by each of these terms varies; some typical definitions are:

Location a location in memory. It may be tagged with attributes, such as the type of data it can store, the owner of the data, and whether the data is read-only. For the purposes of the model being presented a location is essentially tagged with whether its contents can be directly addressed by a pointer.

Pointer a value that points to a directly addressable memory location. It may be tagged with attributes in a similar manner to that of a location. Note that a pointer's attributes need not match those of the location, they just ought to be consistent with them.

Reference a pointer whose value cannot be used in, or the result of, pointer arithmetic.

Object identity the identity of an object, which may be its address in memory, a name in a Java Remote Method Invocation (RMI) registry, a name in a CORBA (Common Object Request Broker Architecture) naming service, etc.

5.1.2 Scope

The model of locations we present is not intended to support the concepts of object ownership or reference containment, such as discussed in ownership models [Cla02, NCP99] and separation logics [Rey02]. Nor is this model of locations intended to support low-level pointer operations, such as those operations that create a new pointer by adding an arbitrary offset to an existing pointer (e.g. $ptr = ptr + 2$) or get the address of a record's element (e.g. $ptr = \&(rec.fld)$). Having said this, it is straightforward to write C^{++} and C$^{\#}$ programs that do not directly use such low-level pointer operations and this ought to be syntactically checkable. For example, in C$^{\#}$ this could be achieved by banning the use of the `unsafe` keyword.

5.1.3 Family tree

Within this chapter we use instantiations of data-types associated with a family tree, as illustrated by the class diagram in Figure 5.2, to provide examples of abstract data-structures. Essentially a family tree contains a relationship between a set of people, where each person has a name, a gender and a set of relations; a person may also have a record of dates associated with them to record either or both of their birth and death dates. Further, each entry in a person's relations set contains the type of relationship (as indicated by the 'rel' entry), the set of people that this is with, and a link back to itself (i.e. the containing person entry). We give a

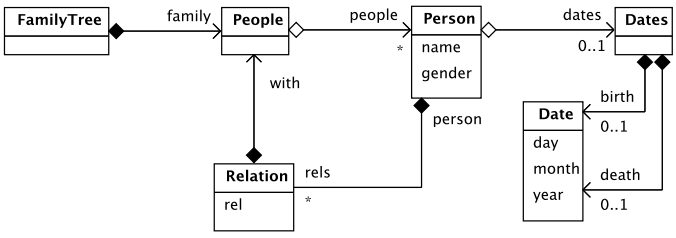


Figure 5.2: Family Tree Example – Class Diagram

specific interpretation to the aggregations within Figure 5.2 to link them to the concepts within our abstract location model (as previously illustrated in Figure 5.1). Here, shareable (hollow

diamond) and composite (solid diamond) aggregations are used to distinguish between shareable and containable locations, respectively; aggregations that have any number of instances are represented by lists.

5.2 Concrete Representation

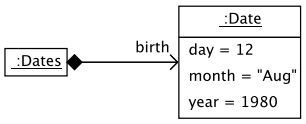
5.2.1 Concrete value notation

The two types of *literal value* used within this chapter are the integers (e.g. -32) and the strings (e.g. "Some text"). There is also a special *unset* literal constant, denoted by ¿; this is used to represent the contents of a freshly created location, and the value of a missing element.

The two remaining types of concrete value are the *compound* and *reference values*. A compound value is represented by a partial map from field names (which we identify with containable locations) to concrete values. It is denoted by $\{\overset{k}{\underset{i=1}{}} nm_i = v_i\}$, where the name nm_i indexes the concrete value v_i. A name is denoted by an alpha-numeric word starting with a letter or the dollar symbol. The name represented purely by a single dollar symbol, which we refer to as the 'dollar name', is reserved for denoting a shareable location, and thus cannot be used as a compound value's field name.

A reference value is either null or an index to a shareable location. Such values are denoted by $\ominus$ and ℓ_i respectively, where two non-null reference values ℓ_i and ℓ_j index distinct shareable locations whenever $i \neq j$.

Figure 5.3 provides both an instance of the family tree's Dates class and a concrete value representation of this instance (object). Here, the Dates object explicitly sets only one of its two optional Date fields, birth, to 12 Aug 1980. The other optional field, death, is left unset. This data structure can be drawn as a graph, as illustrated in Figure 5.4, where: a literal



$$\{\,\texttt{birth} = \{\,\texttt{day} \;\; = 12,\\ \qquad\qquad\;\; \texttt{month} = \text{"Aug"},\\ \qquad\qquad\;\; \texttt{year} \;\; = 1980\,\},\\ \;\texttt{death} = ¿\\ \}$$

Figure 5.3: Dates – object diagram and concrete representation

value is denoted by a boxed node containing the literal; and a compound value is denoted by a circular node, whose outgoing edges are labelled with its distinct field names. Reference values

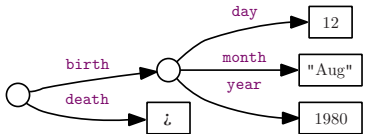


Figure 5.4: Dates – concrete graph representation

are denoted by a diamond node that contains the reference (Figure 5.6 and Figure 5.8).

In addition to defining the concrete representations of the values we use meta-variables for representing each of the different types of value, in accordance with Section 2.1.4. Here i, j and k represent the integers, and s, lv, cv, rv, v, and t represent strings, literal values, compound values, reference values, values and terms respectively. Note that the meta-variable t includes the concrete values, field names, and the yet-to-be terms such as location graphs. These meta-variables are typically used to define functions, as illustrated by the following example, which extracts the shareable locations contained within a term. This example also uses the generalised term notation that was introduced in Section 2.1.3, where $t\{\overset{k}{\underset{i=1}{}} t_i\}$ denotes

a term t with k subterms, $t^k_{i=1}$, where a leaf term (k = 0) has the empty set of subterms.

$$\begin{array}{lcl} sLocs\ \ell_i & \widehat{=} & \{\ell_i\} \\ sLocs\ t\{^k_{i=1} t_i\} & \widehat{=} & \bigcup\{^k_{i=1} sLocs\ t_i\} \end{array}$$

We read such definitions by pattern-matching from top to bottom, accepting the first equation that matches an actual argument. Thus, the order in which the lines of a function are presented may affect its meaning. In this case, swapping the order would produce a function that returns the empty set.

The *sLocs* function is applied to terms that are yet to be defined, such as the heap value term in Section 5.2.2. Note that this does not require an update to the *sLocs* function as these terms are already handled by the second definitional line, which can be applied to any term (i.e. a general term).

5.2.2 Concrete location heap

A *location heap* is a partial map from shareable locations to values; it is denoted by $\{_{i:N}\ell_i \mapsto v_i\}$, where N is some finite subset of the natural numbers. For example, the object diagram in Figure 5.5 illustrates that Jane Doe is married to John Doe, where only the instances of the Person class are considered to be shareable. It can be represented by the following concrete

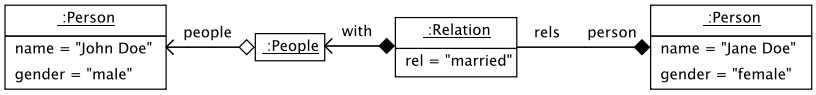


Figure 5.5: Marriage example – object diagram

location heap, where the contents of shared locations ℓ_3 and ℓ_5 contain the John Doe and Jane Doe Person objects respectively.

$$\begin{array}{l} \{\ \ell_3 \mapsto \{\ \texttt{name} = \text{"John Doe"},\ \texttt{gender} = \text{"male"},\ \texttt{dates} = \ominus,\ \texttt{rels} = \text{¿}\ \},\\ \quad \ell_5 \mapsto \{\ \texttt{name} = \text{"Jane Doe"},\ \texttt{gender} = \text{"female"},\ \texttt{dates} = \ominus,\\ \qquad\quad \texttt{rels} = \{\ \$1 = \{\ \texttt{rel} = \text{"married"},\ \texttt{person} = \ell_5,\ \texttt{with} = \{\ \texttt{people} = \{\ \$1 = \ell_3\ \}\ \}\ \}\ \}\\ \qquad \}\\ \} \end{array}$$

Note that the `rels` fields within the Jane Doe Person object contains a singleton list of relations, whose first (and only) entry is labelled with `$1`. Further, this entry's `with` field contains a list of People objects with a single entry, again labelled with `$1`, that contains a pointer to the John Doe Person object. In general, the n^{th} element of a list would be labelled with `$n`.

Figure 5.6 provides the alternative graph representation of the example, where the dashed edges are used to link a reference value to its contents. Note that these edges are labelled with the dollar name (`$`) as discussed in Section 5.2.1.

Before moving on to present the concrete location model, we observe that a concrete heap can reference a shareable location that it does not define; i.e. a concrete heap can contain reference values that are not in the domain of the heap's partial map. In order to classify location heaps that do not have this undesirable property, we introduce a healthiness condition, which considers a heap to be healthy whenever all references to shared locations within the graph's values are defined by the graph itself.

$$\mathrm{HC_H}\ H \ \widehat{=}\ sLocs\ H \subseteq \mathrm{dom}\ H$$

where

H	is the meta variable representing a concrete location heap
$\mathrm{dom}\_$	returns the domain of a relation or function

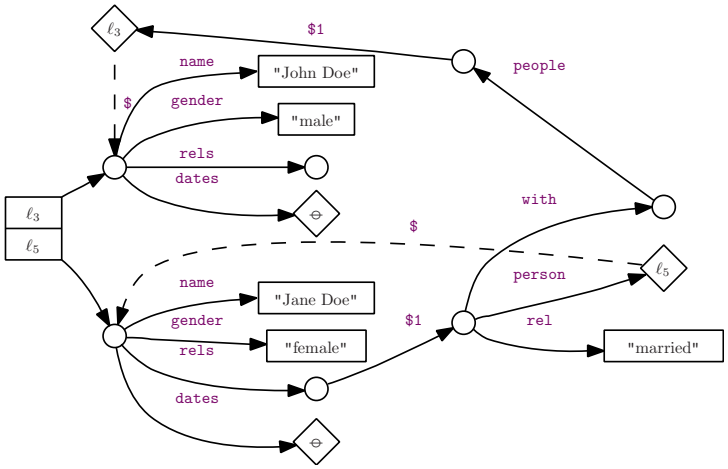


Figure 5.6: Marriage example – concrete heap graph

Further, any heap can be made healthy by adding an entry for each missing shared location and setting that location's value to the unset value (¿), as follows:

$$\mathrm{MH}_{\mathrm{H}}\ H \ \widehat{=}\ H \cup \{r \mapsto \text{¿} \mid r \in (sLocs\ H) \setminus (\mathrm{dom}\ H)\}$$

Note that a healthy heap is unaffected by the application of MH_{H} (and vice-versa), because all the shareable locations in the heap are contained within its domain; i.e. $(\mathrm{MH}_{\mathrm{H}}\ H = H) \Leftrightarrow \mathrm{HC}_{\mathrm{H}}\ H$.

The notion of equivalence ($\_ \equiv \_$) between location heaps is more complex than that for concrete values, which is mathematical equality (where the ordering of elements within a set or map is not significant). Here, the equivalence relationship between heaps allows for the renaming of shareable locations. Specifically, two heaps are considered to be equivalent if there exists a bijective map (f) that can be applied to one heap to produce the other.

$$H_1 \equiv H_2 \quad \Leftrightarrow \quad \exists f \bullet H_1 = rename(H_2, f)$$

where

$$\begin{array}{lcl} rename(\ell_i, f) & \widehat{=} & \ell_{f(i)} \\ rename(t\{\overset{k}{\underset{i=1}{}} t_i\}, f) & \widehat{=} & t\{\overset{k}{\underset{i=1}{}} rename(t_i, f)\} \end{array}$$

5.2.3 Concrete location model

Location models extend this notion of the heap by adding a starting point, which is represented by a concrete value. Therefore, a location model is denoted by a value-heap pair (v, H). Here, the idea is that the value v represents the root of a computational unit, such as a program, whose elements can share data via the shareable locations in the heap H.

Like the location heap that preceded it, location models have a healthiness condition which ensures that the heap is valid; that is, all the shareable locations referenced within a model are defined by the heap.

$$\mathrm{HC1}_{\mathrm{L}}(v, H) \ \widehat{=}\ sLocs(v, H) = \mathrm{dom}\ H$$

A location model can be made $\mathrm{HC1}_{\mathrm{L}}$-valid in a similar manner to a heap.

$$\mathrm{MH1}_{\mathrm{L}}(v, H) \ \widehat{=}\ (v, H \cup \{r \mapsto \text{¿} \mid r \in sLocs(v, H) \setminus (\mathrm{dom}\ H)\})$$

Locations in the model are considered to be *reachable* if they are either contained within the starting value v or indirectly contained within the contents of v's reference values. For $\mathrm{HC1_L}$-healthy models, this can be formalised by the following functions, where R and R' represent the shareable locations that have already been taken into account and are contained within a value respectively.

$$\begin{array}{lcl} \mathit{reachable}(v, H) & \widehat{=} & \mathit{reachValue}(v, H, \emptyset) \\ \mathit{reachValue}(v, H, R) & \widehat{=} & \mathit{sLocs}\ v \cup \mathit{reachDeref}(\mathit{sLocs}\ v, H, R) \\ \mathit{reachDeref}(R', H, R) & \widehat{=} & \bigcup\{\mathit{reachValue}(H\ r, H, R' \cup R) \mid r \in R' \setminus R\} \end{array}$$

The following normal-form healthiness condition ensures that there is no unreachable information within the model; i.e. every shareable location that is defined by a model's heap is reachable.

$$\mathrm{HC2_L}(v, H) \ \widehat{=}\ \mathrm{dom}\, H = \mathit{reachable}(\mathrm{MH1_L}(v, H))$$

Note that we apply the $\mathrm{MH1_L}$ healthiness constructor prior to performing the reachability calculation, in order to ensure that $\mathrm{HC2_L}$ calculation is defined. If a location model is not in normal form (i.e. $\mathrm{HC2_L}$-healthy), it can be made so by ensuring that it is $\mathrm{HC1_L}$-healthy and then removing all the unreachable locations.

$$\begin{array}{l} \mathrm{MH2_L}(v, H) \ \widehat{=}\ \mathsf{let}\ (v_1, H_1) \mathrel{\widehat{=}} \mathrm{MH1_L}(v, H)\ \mathsf{in} \\ \qquad\qquad (\ v_1, \{rv \mid rv \in H_1 \wedge (\mathit{first}\ rv) \in \mathit{reachable}(v_1, H_1)\}\ ) \end{array}$$

where $\mathit{first}(x, y) \mathrel{\widehat{=}} x$

We are now in a position to define an equivalence relation over location models. It is similar to that of heaps, except that we first ensure that models are made healthy before performing the check, as we only want to consider reachable elements in a model's heap. In other words, two location models are equivalent iff there exists some bijective shareable-location-renaming function f that enables two normalised heaps to be made equal.

$$\begin{array}{l} (v_1, H_1) \equiv (v_2, H_2) \\ \Leftrightarrow \\ \exists f \bullet \mathrm{MH2_L}(v_1, H_1) = \mathit{rename}(\mathrm{MH2_L}(v_2, H_2), f) \end{array}$$

It is this notion of equivalence up to which our UTP model of locations is fully abstract, as described in Section 5.4.3.

The family tree example can now be extended to illustrate a concrete location model, by adding an object to represent the family tree, as illustrated in Figure 5.7. The concrete graph

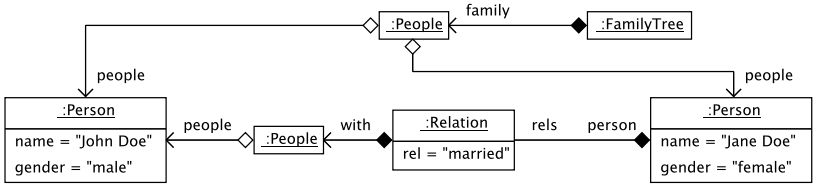


Figure 5.7: Family tree example – object diagram

representation of this example is provided by Figure 5.8, where the explicit visualisation of the heap has been removed, as it is no longer required for representing the shareable locations. Such locations are now represented by the dashed edges within the graph, which are now guaranteed to exist due to the reachability healthiness condition.

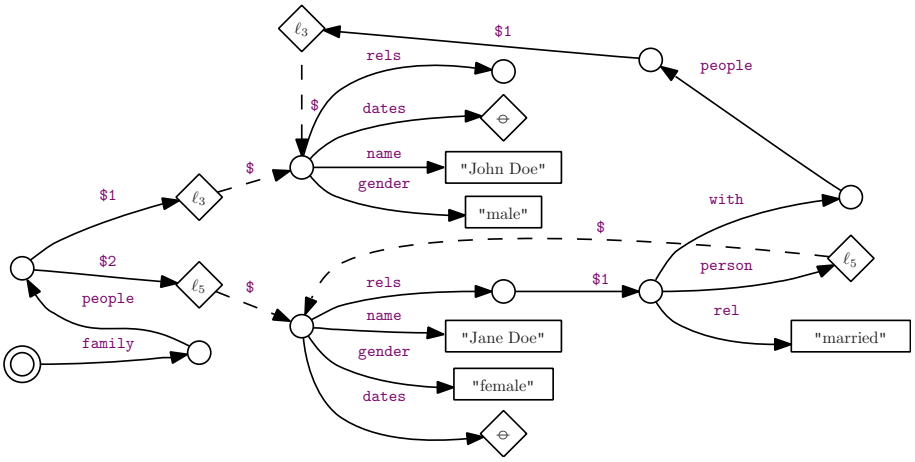


Figure 5.8: Family tree example – concrete model graph

5.2.4 Paths and their operations

A *compound value path* describes a route from a compound value to one of its elements, via a non-empty dot-separated sequence of field names (labels). Such labelled paths are essentially used to describe routes to contained locations, which we can access and update by using the following functions

$$
\begin{array}{lcl}
*cv.nm & \widehat{=} & cv\ \ nm \\
cv.nm := v & \widehat{=} & cv \oplus \{nm \mapsto v\} \\
cv.nm.lp := v & \widehat{=} & cv \oplus \{nm \mapsto (*cv.nm).lp := v\}
\end{array}
$$

where lp is the meta-variable for labelled paths, $(\_\ \_)$ is the function or map application operation, and $(\_ \oplus \_)$ is the function override operation. This notion of a path is extended to define location model update and access functions as follows:

$$
\begin{array}{lcl}
*(v, L) & \widehat{=} & v \\
*(v, L).lp & \widehat{=} & *v.lp \\
*(v, L).\ell_i & \widehat{=} & L\ \ell_i \\
*(v, L).\ell_i.lp & \widehat{=} & (L\ \ell_i).lp \\
(v, L) := v' & \widehat{=} & (v', L) \\
(v, L).lp := v' & \widehat{=} & (v.lp := v', L) \\
(v, L).\ell_i := v' & \widehat{=} & (v, L \oplus \{\ell_i \mapsto v'\}) \\
(v, L).\ell_i.lp := v' & \widehat{=} & (v, L \oplus \{\ell_i \mapsto (L\ \ell_i).lp := v'\})
\end{array}
$$

Further, it is possible to extend this notion to copy a value from one location to another, as follows

$$(v, L).lop := (v, L).llp' \quad \widehat{=} \quad (v, L).lop := *(v, L).llp'$$

where: lop denotes either a labelled path (lp), a shareable location index (ℓ_i), or a shareable location index followed by a path ($\ell_i.lp$); llp denotes a lop or the empty path (which is itself denoted by $\varnothing$; and $(v, L).\varnothing$ denotes (v, L).

It is also straightforward to define other operations, such as for deleting elements from compound values and the heap.

5.3 Abstract Model

In Section 5.2.3, graphs represented healthy concrete location models, where:

- the solid and dashed edges denote distinct compound and potentially shared shareable locations, respectively;
- the rectangular, circular, and diamond nodes denote literal, compound, and reference values, respectively.

This section presents: a brief overview of the trace-based graph abstraction; some utility operations for manipulating traces; a model of nodes as a set of traces; and an overview of the trace-based location graph model.

5.3.1 Graph abstraction

We can determine the value of dereferencing a reference node of a concrete location graph by following that node's outbound edge (as shown in Figure 5.8). That is, the shareable location index contained within a reference node is not required. Thus, this unused data can, and will, be ignored in our abstraction.

We observe that the outbound edges of each node within a healthy concrete location graph have distinct labels. Therefore, we can use a finite non-empty sequence of names to unambiguously define a path from a graph's root node to any other node. Such a path is from now on referred to as an *absolute path*.

The location of a node within a concrete location graph can be modelled by the set of all absolute paths to that node, which we from now on refer to as an *absolute path-set*. Hence, one way of providing a UTP model of locations would be as a partial map from such an absolute path-set to an appropriate abstraction of the data directly associated with its corresponding node. For example, the data associated with:

- a literal or null-reference node could be modelled by its concrete value;
- a compound node could be modelled by its set of outbound edge labels;
- a non-null reference node could be modelled by its outbound edge label.

Such a model of locations is similar to that presented in [CHW06], which uses the idea of an entity group to model shared locations. Here, each group contains the set (equivalence class) of fully qualified variables that share the same location.

Another approach is to change the notion of an absolute path-set, from representing the location of a node to representing both the location and contents of a node. To avoid confusion, we refer to such paths as *traces*. Here, the idea is that the last value in a trace represents its content, and the front of the trace its location. In other words, a trace is a path lp followed by a trace label l, where l represents either a name, a literal value, or the null-reference value; it is denoted by $lp.l$. This is the basis of our UTP model of locations, which we present in Section 5.4. Such a model of locations is similar to that presented in [HH99], which uses trace-sets to model both locations and values. Here, the main difference is in our introduction of a containable location and its effect on assignment (which [HH99] refers to as 'pointer swing'). Specifically, within our model the contents of contained locations need to be duplicated, whereas the contents of shared locations are referenced.

5.3.2 Traces

As previously stated, a trace is denoted by $lp.l$, where lp is a path and l is a trace label (i.e. a name, a literal, a null-reference). One consequence of this is that it is only possible to concatenate two traces (denoted by $tr_1.tr_2$) when the last label in the first trace tr_1 is a name, as only names are allowed within a path.

The remainder of this section defines some utility operations on traces and trace-sets, that are used in the construction of our abstract model of locations. First we introduce two

operations front and last for extracting the location and content components of a non-empty trace.

$$\mathsf{front}\ lp.l \mathrel{\widehat{=}} lp \qquad \mathsf{last}\ lp.l \mathrel{\widehat{=}} l$$

The front operation can be used to generate the set of locations visited by a trace, as characterised by their paths, where each path within this set is considered to be a *prefix* of the original trace. Such a set of paths is referred to as the *proper prefixes* of the given trace. The function $\mathsf{prefixes}_{\mathrm{T}}$ defines the non-proper version of the prefix set.

$$\begin{array}{ll} \mathsf{prefixes}_{\mathrm{T}}\ \varnothing & \mathrel{\widehat{=}} \{\varnothing\} \\ \mathsf{prefixes}_{\mathrm{T}}\ tr & \mathrel{\widehat{=}} \{tr\} \cup \mathsf{prefixes}_{\mathrm{T}}(\mathsf{front}\ tr) \end{array}$$

The prefixes also provide a natural ordering over traces.

$$tr_1 <_{\mathrm{T}} tr_2 \mathrel{\widehat{=}} \mathsf{prefixes}_{\mathrm{T}}(tr_1) \subset \mathsf{prefixes}_{\mathrm{T}}(tr_2)$$

Having defined an ordering over traces, it is now possible to use that ordering to define a subtraction operation. This is eventually used to define the relative paths between nodes in a set.

$$(\_ -_{\mathrm{T}} \_) \mathrel{\widehat{=}} \lambda\, tr_1, tr_2 \mid tr_2 \leq_{\mathrm{T}} tr_1 \bullet \mathsf{pick}\{tr \mid tr_1 = tr_2.tr\}$$

Here, the pick function picks the singleton element from a set (i.e. $\mathsf{pick}\{x\} \mathrel{\widehat{=}} x$).

Before leaving the trace utilities, we lift the definitions of the front, last, and $\mathsf{prefixes}_{\mathrm{T}}$ operations to trace-sets. The first two are lifted by applying their definitions to each non-empty trace with the set. The latter one is lifted to a trace-set (denoted by TR) by applying the prefixes operation to each trace within the set and merging the results.

$$\begin{array}{ll} \mathsf{frontTraces}\ TR & \mathrel{\widehat{=}} \{\mathsf{front}\ tr \mid tr \in (TR \setminus \{\varnothing\})\} \\ \mathsf{lastLabels}\ TR & \mathrel{\widehat{=}} \{\mathsf{last}\ tr \mid tr \in (TR \setminus \{\varnothing\})\} \\ \mathsf{prefixes}_{\mathrm{N}}\ TR & \mathrel{\widehat{=}} \bigcup\{\mathsf{prefixes}_{\mathrm{T}}\ tr \mid tr \in TR\} \end{array}$$

5.3.3 Trace-based node

A trace-based graph node is modelled by a set of traces that satisfies two healthiness conditions. Both of these conditions follow from the observation that the only way a concrete location graph node may have more than one incoming edge, is if all these edges are labelled with the dollar name. Consequently, every trace to a node is guaranteed to end with the same label, except for the root node which has no label. This is modelled by the first healthiness condition, which states that all incoming edges to a node (trace-set n) have the same label.

$$\mathrm{HC1}_{\mathrm{N}}(n) \mathrel{\widehat{=}} \#\,\mathsf{lastLabels}(n) \leq 1$$

Another consequence of the observation is that a node may only have multiple parents if it is stored in a shareable location. This is modelled by the second healthiness condition, which states that the trace to any node that has more than one parent must end with the special shareable location label.

$$\mathrm{HC2}_{\mathrm{N}}(n) \mathrel{\widehat{=}} \#\,\mathsf{lastLabels}(\mathsf{frontTraces}(n)) > 1 \Rightarrow \mathsf{lastLabels}(n) = \{\$\}$$

Any healthy node can be denoted by $Lp.l$, where each path in the path-set Lp is extended by the trace-label l to form the trace-set $\{lp.l \mid lp \in Lp\}$. The remainder of this section now presents some useful utility relations and operations on nodes.

Node relations: The *child-of* and *descendant-of* relations test whether one node is an immediate child of or a descendant of another node. These tests assume that the nodes come from a healthy graph, where all the routes to the parent are contained within the child.

$$\begin{array}{lcl} n_1 \ \mathsf{childOf}\ n_2 & \widehat{=} & n_2 \subseteq \mathsf{frontTraces}(n_1) \\ n_1 \ \mathsf{descendantOf}\ n_2 & \widehat{=} & n_2 \subseteq \mathsf{prefixes}_{\mathrm{N}}(\mathsf{frontTraces}(n_1)) \end{array}$$

In addition to knowing whether two nodes are related, it is sometimes useful to identify the relative traces from a parent to child node.

$$\mathsf{traces}_{\mathrm{N}}(n_1, n_2) \ \widehat{=}\ \{tr_2 -_{\mathrm{T}} tr_1 \mid tr_1 \in n_1 \wedge tr_2 \in n_2 \wedge tr_1 \leq_{\mathrm{T}} tr_2\}$$

Such trace-sets are used to determine whether two nodes are related via shareable or via containable locations. Here, two nodes are related by a shareable location if one of the traces within the trace-set includes the dollar label. Similarly they are related by a containable location if one of the traces within the trace-set does not contain the dollar label.

$$\begin{array}{lcl} n_1 \ \mathsf{shareDescOf}\ n_2 & \widehat{=} & n_1 \ \mathsf{descendantOf}\ n_2 \wedge \$ \in_{\mathrm{T}} \mathsf{traces}_{\mathrm{N}}(n_2, n_1) \\ n_1 \ \mathsf{containDescOf}\ n_2 & \widehat{=} & n_1 \ \mathsf{descendantOf}\ n_2 \wedge \$ \notin_{\mathrm{T}} \mathsf{traces}_{\mathrm{N}}(n_2, n_1) \end{array}$$

where $l \in_{\mathrm{T}} tr \mathrel{\widehat{=}} \exists\, lp \bullet lp.l \in \mathsf{prefixes}_{\mathrm{T}}\ tr$

Note that the only way a node can be both a shareable and a containable descendant of another node, is if the nodes are both contained in the same cycle. In this case, all the containable descendants are also shareable descendants.

Node unlinking (deletion): Part of the assignment process involves the removal of previously held data. This is the purpose of the following unlinking operations, which remove all traces of either a node (n_1) or its children from the specified target node (n_2).

$$\begin{array}{lcl} \mathsf{unlink}_{\mathrm{N}}\ n_1\ \mathsf{from}\ n_2 & \widehat{=} & n_2 \setminus \{tr_2 \mid tr_1 \in n_1 \wedge tr_2 \in n_2 \wedge tr_1 \leq_{\mathrm{T}} tr_2\} \\ \mathsf{unlinkChildren}_{\mathrm{N}}\ n_1\ \mathsf{from}\ n_2 & \widehat{=} & n_2 \setminus \{tr_2 \mid tr_1 \in n_1 \wedge tr_2 \in n_2 \wedge tr_1 <_{\mathrm{T}} tr_2\} \end{array}$$

These operations can be lifted to the graph context by unlinking a given node from a node-set.

$$\begin{array}{lcl} \mathsf{unlink}_{\mathrm{G}}(N, n) & \widehat{=} & \{(\mathsf{unlink}_{\mathrm{N}}\ n\ \mathsf{from}\ n') \mid n' \in N\} \\ \mathsf{unlinkChildren}_{\mathrm{G}}(N, n) & \widehat{=} & \{(\mathsf{unlinkChildren}_{\mathrm{N}}\ n\ \mathsf{from}\ n') \mid n' \in N\} \end{array}$$

Node duplication (replacement): It is sometimes useful to construct a new node from a pair of existing nodes, a source node (n_2) and one of its descendants (n_3). Here the idea is to extract the traces between the source and descendant nodes, and then append them to a new source node (n_1), which is the target of the duplication.

$$\mathsf{replace}_{\mathrm{N}}\ n_1\ \mathsf{for}\ n_2\ \mathsf{in}\ n_3 \ \widehat{=}\ \{tr_1.tr \mid tr_1 \in n_1 \wedge tr \in \mathsf{traces}_{\mathrm{N}}(n_2, n_3)\}$$

For example, consider the two graphs presented in Figure 5.9. Here, the traces between the source and destination node in the initial graph is the singleton set {`month`."Aug"}. The traces within this set are appended to each trace in the new source node (n_3) to produce the trace-set {`death.month`."Aug"}. This trace-set could be considered to define a new node within the graph, as illustrated in the updated part of Figure 5.9. Having said this, care needs to be taken to ensure that any formal graph duplication operation also appropriately copies the intermediate nodes (see later discussion on assignment in Section 5.3.4).

Instead of replacing one parent for another, we may want to add a parent; for example, when copying a reference to a shareable location. This is essentially achieved by performing the replacement operation and merging in the original data.

$$\mathsf{add}_{\mathrm{N}}\ n_1\ \mathsf{to}\ n_2\ \mathsf{in}\ n_3 \ \widehat{=}\ (\mathsf{replace}_{\mathrm{N}}\ n_1\ \mathsf{for}\ n_2\ \mathsf{in}\ n_3) \cup n_3$$

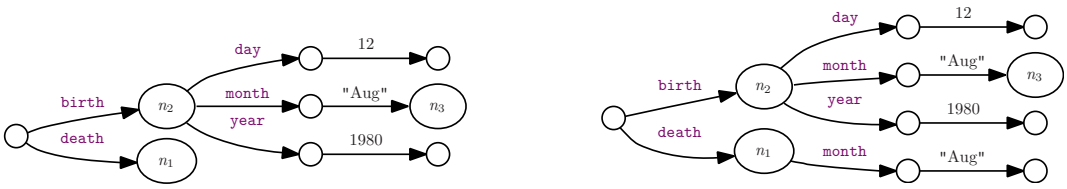


Figure 5.9: Simple node duplication

These operations can then lifted so that they operate on node-sets, by reparenting each node in the set.

$$\begin{array}{lcl} \mathsf{replace}_{\mathrm{G}}\ n_1\ \mathsf{for}\ n_2\ \mathsf{in}\ N & \mathrel{\widehat{=}} & \{(\mathsf{replace}_{\mathrm{N}}\ n_1\ \mathsf{for}\ n_2\ \mathsf{in}\ n) \mid n \in N\} \\ \mathsf{add}_{\mathrm{G}}\ n_1\ \mathsf{to}\ n_2\ \mathsf{in}\ N & \mathrel{\widehat{=}} & \{(\mathsf{add}_{\mathrm{N}}\ n_1\ \mathsf{to}\ n_2\ \mathsf{in}\ n) \mid n \in N\} \end{array}$$

We use these operations to prepare a subgraph for being moved or copied to a new location.

5.3.4 Trace-based graph

A *trace-based graph* is a set of trace-based nodes that satisfies four healthiness conditions. The first healthiness condition states that each of the graph's nodes is healthy.

$$\mathrm{HC1}_{\mathrm{G}}(G) \quad \mathrel{\widehat{=}} \quad \forall\, n \mid n \in G \bullet \mathrm{HC1}_{\mathrm{N}}(n) \land \mathrm{HC2}_{\mathrm{N}}(n)$$

The second healthiness condition states that the nodes of a graph are disjoint. This ensures that an absolute trace can be used to identify a single node.

$$\mathrm{HC2}_{\mathrm{G}}(G) \quad \mathrel{\widehat{=}} \quad \forall\, n_1, n_2 \mid \{n_1, n_2\} \subseteq G \land n_1 \neq n_2 \bullet n_1 \cap n_2 = \emptyset$$

For a graph (G) that satisfies condition $\mathrm{HC2}_{\mathrm{G}}$, it is possible to define an operation for extracting the node (n) that has an absolute trace (lp), so long as the trace is within the graph.

$$\mathsf{node}_{\mathrm{G}}(G, tr) \quad \mathrel{\widehat{=}} \quad \lambda\, G, tr \mid \mathrm{HC2}_{\mathrm{G}}(G) \land tr \Subset_{\mathrm{G}} G \bullet \mathsf{pick}(\{n \mid n \in G \land tr \in n\})$$

Here, the $(\_ \Subset_{\mathrm{G}} \_)$ relation determines whether a trace is in the graph:

$$lp \Subset_{\mathrm{G}} G \mathrel{\widehat{=}} lp \in (\bigcup G)$$

The third healthiness condition states that each of a node's traces is consistently extended; i.e. if it is possible to take an edge with label l from node n_1 to node n_2, then the trace-set formed by appending the label l to each of n_1's traces is a subset of n_2's trace-set.

$$\begin{array}{lcl} \mathrm{HC3}_{\mathrm{G}}(G) & \mathrel{\widehat{=}} & \forall\, n_1, n_2, tr_1, tr_2, l \mid \\ & & \quad \{n_1, n_2\} \subseteq G \land \{tr_1, tr_2\} \in n_1 \bullet \\ & & \quad\quad tr_1.l \in n_2 \Rightarrow tr_2.l \in n_2 \end{array}$$

The fourth healthiness condition states that the parents of a node are contained within the graph; in other words, the traces within a graph are prefix closed.

$$\mathrm{HC4}_{\mathrm{G}}(G) \quad \mathrel{\widehat{=}} \quad \forall\, tr, l \mid tr.l \Subset_{\mathrm{G}} G \bullet tr \Subset_{\mathrm{G}} G$$

The combination of the first three graph healthiness conditions defines what it means for the trace model to have a consistent, but not necessarily complete, set of nodes. Thus, these conditions should be satisfied by any healthy subgraph.

The remainder of this section provides operations for manipulating the contents of a location graph model, such as operations for: extracting a subgraph; extracting the value at a location; and assigning a value to a location.

Children and descendants subgraphs: Subgraphs can be formed by selecting only some of a graph's nodes. The childOf and descendantOf relations can be used to filter a graph to form children and descendants subgraphs respectively.

$$\begin{array}{lcl} \mathsf{children} & \widehat{=} & \lambda\, G, n \mid n \in G \bullet \{n' \mid n' \in G \land n'\ \mathsf{childOf}\ n\} \\ \mathsf{descendants} & \widehat{=} & \lambda\, G, n \mid n \in G \bullet \{n' \mid n' \in G \land n'\ \mathsf{descendantOf}\ n\} \end{array}$$

Note that a node can be a descendant of itself if, and only if, there is a non-empty sequence of edges back to itself.

Dereferencing a location's value: A location is represented by a healthy node whose last label is either a field name or the dollar name. Such a node can be represented by a path-set, as each trace within this node may only contain names. The value of a location node is determined by recursively examining its children, or more specifically its child labels. There are three cases to consider.

1. There is a single null-reference or literal value (nlv) child label. In this case, the label value is returned as the location's value.
2. There is a single child label that contains the dollar name. In this case the path-set (reference value) that models the child node is returned.
3. There is a set of child-labels that contain field names. In this case a compound value is recursively constructed from its children.

$$\begin{array}{lcl} *_{\mathrm{G}}(G, lp) & \widehat{=} & *_{\mathrm{G}}(G, \mathsf{node}_{\mathrm{G}}(G, lp)) \\ *_{\mathrm{G}}(G, Lp) & \widehat{=} & *_{\mathrm{G}}(G, \mathsf{children}(G, Lp)) \\ *_{\mathrm{G}}(G, \{Lp.nlv\}) & \widehat{=} & nlv \\ *_{\mathrm{G}}(G, \{Lp.\$\}) & \widehat{=} & Lp.\$ \\ *_{\mathrm{G}}(G, \{\overset{k}{_{i=1}}\ Lp.nm_i\}) & \widehat{=} & \{\overset{k}{_{i=1}}\ nm_i = *_{\mathrm{G}}(G, Lp.nm_i)\} \end{array}$$

Recall that we introduced $Lp.l$ as an alternative notation for denoting a healthy node, in Section 5.3.3, where $Lp.l \mathrel{\widehat{=}} \{lp.l \mid lp \in Lp\}$.

Preparing a location for assignment: The preparation required for assigning a value to a location depends on a number of factors, such as whether the location already exists. We could limit assignments to existing locations, but then this would not mirror our concrete model, which defined assignment in terms of the map overriding operation ($\_ \oplus \_$). Instead we categorise a potential location as either *existing* (E_{pm}), *freshly containable* (C_{pm}), *freshly shareable* (S_{pm}), or *invalid* (U_{pm}), as follows

$$prepMode(G, Lp.nm) \mathrel{\widehat{=}} \begin{cases} E_{pm}, & \text{if } Lp.nm \in G \\ S_{pm}, & \text{if } Lp.nm \notin G \land nm = \$ \\ C_{pm}, & \text{if } Lp.nm \notin G \land nm \neq \$ \land *_{\mathrm{G}}(G, Lp) \in CV \\ U_{pm}, & \text{otherwise} \end{cases}$$

where CV denotes the set of compound values (i.e. the compound value type). Note that the above definition of freshly created locations ensures that a compound value may only contain containable locations (and vice versa). In general, a path-set Lp is considered to represent an assignable location within a graph G whenever it has a valid assignable location mode.

$$Lp\ \mathsf{assignableIn}_{\mathrm{G}}\ G \mathrel{\widehat{=}} prepMode(G, Lp) \neq U_{pm}$$

It is now possible to define the preparation for an assignable location by ensuring that it exists and contains no contents. This can involve the clearing (unlinking) of an existing node's contents and the creation of a new location node.

$$\begin{array}{lcl} \mathsf{prep}_{\mathrm{G}}(G, Lp) & \widehat{=} & prep(G, Lp, prepMode(G, Lp)) \\ prep(G, Lp, E_{pm}) & \widehat{=} & \mathsf{unlinkChildren}_{\mathrm{G}}(G, Lp) \\ prep(G, Lp.l, S_{pm}) & \widehat{=} & \mathsf{unlinkChildren}_{\mathrm{G}}(G, Lp) \cup \{Lp.\$\} \\ prep(G, Lp.l, C_{pm}) & \widehat{=} & G \cup \{Lp.l\} \end{array}$$

Assigning a null-reference or literal value: A null-reference or literal value (nlv) can be assigned to a graph location by preparing the location and setting its contents to the given value.

$$\begin{array}{lcl} (G, lp) :=_{\mathrm{G}} nlv & \widehat{=} & (G, \mathsf{node}_{\mathrm{G}}(G, lp)) :=_{\mathrm{G}} nlv \\ (G, Lp) :=_{\mathrm{G}} nlv & \widehat{=} & \mathsf{prep}_{\mathrm{G}}(G, Lp) \cup \{Lp.nlv\} \end{array}$$

Assigning an encapsulated compound value: An encapsulated compound value is a concrete compound value that contains no shareable locations (i.e. a compound value in the set $\{cv \mid sLocs\, cv = \emptyset\}$). Such values are represented by the meta-variable ecv. It can be assigned to a location by preparing the location and setting its contents to the subtree that represents the compound value.

$$\begin{array}{lcl} (G, lp) :=_{\mathrm{G}} ecv & \widehat{=} & (G, \mathsf{node}_{\mathrm{G}}(G, lp)) :=_{\mathrm{G}} ecv \\ (G, Lp) :=_{\mathrm{G}} ecv & \widehat{=} & \mathsf{prep}_{\mathrm{G}}(G, Lp) \cup \bigcup \{Lp.tr \mid tr \in (cvTrs\, ecv)\} \end{array}$$

Here, the $cvTrs$ function converts an encapsulated compound value into a prefix closed set of traces, representing each trace through the compound value's structure.

Assigning the contents of an existing location: In the concrete model, we referred to this as the copying of a location's value. This is more tricky than the previous cases for a number of reasons. One significant reason is that the location we are copying may be contained within the target location that we are assigning to. In such a case, the location preparation process could remove (clear) the location we want to copy. This limitation can be overcome by a three-step process. First, copy the value to a fresh temporary location, which is not contained within the contents of the target location. Second, prepare the target location and copy the value of the temporary location to it. Last, remove the temporary location.

What would make a good temporary location is dependant on what the location graph is being used to model, so in general we cannot specify this. Having said that, what we can do is specify how to assign the contents of a location to a prepared location node.

Assigning to a cleared location node: When assigning the contents of a cleared location, care has to be taken to ensure that the contents of reference values are pointed to rather than duplicated. In order to facilitate this, two utility operations are defined: one for identifying the referenced nodes (copyRefSG); and the other to add the copied pointer (path) to these identified nodes (copyRefNodes).

$$\begin{array}{lcl} \mathsf{copyRefSG}(G, n) & \widehat{=} & \{n' \mid n' \in G \wedge n'\ \mathsf{shareDescOf}\ n\} \\ \mathsf{copyRefNodes}\ n_1\ \mathsf{to}\ n_2\ \mathsf{in}\ G & \widehat{=} & \mathsf{add}_{\mathrm{G}}\ n_2\ \mathsf{for}\ n_1\ \mathsf{in}\ \mathsf{copyRefSG}(G, n_1) \end{array}$$

Care also has to be taken to ensure that a duplicate of the value nodes are added to the copy node. This is facilitated by two utility operations: one for identifying the nodes to be

duplicated (copyValSG); and the other to perform the duplication (copyValNodes) using the node replacement operation.

$$\begin{array}{lcl} \mathsf{copyValSG}(G, n) & \widehat{=} & \{n' \mid n' \in G \land n' \ \mathsf{containDescOf}\ n\} \\ \mathsf{copyValNodes}\ n_1\ \mathsf{to}\ n_2\ \mathsf{in}\ G & \widehat{=} & \mathsf{replace}_{\mathrm{G}}\ n_2\ \mathsf{for}\ n_1\ \mathsf{in}\ \mathsf{copyValSG}(G, n_1) \end{array}$$

It is now possible to define the graph transformation operation of copying the contents of a source node to the empty location as the union of: the appropriately updated reference nodes; the descendant nodes that were not updated; the non-descendant nodes; and the duplicated value nodes.

$$\begin{array}{lcl} \mathsf{copy}_{\mathrm{G}}\ n_1\ \mathsf{to}\ n_2\ \mathsf{in}\ G & \widehat{=} & (\mathsf{copyRefNodes}\ n_1\ \mathsf{to}\ n_2\ \mathsf{in}\ G) \\ & \cup & \mathsf{descendants}(G, n_1) \setminus \mathsf{copyRefSG}(G, n_1) \\ & \cup & G \setminus \mathsf{descendants}(G, n_1) \\ & \cup & (\mathsf{copyValNodes}\ n_1\ \mathsf{to}\ n_2\ \mathsf{in}\ G) \end{array}$$

Now given that the location with path `$copy` is an assignable location that does not exist within the graph, the copy assignment can be defined as follows:

$$\begin{array}{lcl} (G, lp) :=_{\mathrm{G}} lp' & \widehat{=} & (G, lp) :=_{\mathrm{G}} \mathsf{node}_{\mathrm{G}}(G, lp') \\ (G, lp) :=_{\mathrm{G}} Lp' & \widehat{=} & (G, \mathsf{node}_{\mathrm{G}}(G, lp)) :=_{\mathrm{G}} Lp' \\ (G, Lp) :=_{\mathrm{G}} Lp' & \widehat{=} & \mathsf{let}\ \ G_1 \mathrel{\widehat{=}} (\mathsf{copy}_{\mathrm{G}}\ Lp'\ \mathsf{to}\ \{\texttt{\$copy}\}\ \mathsf{in}\ (G \cup \{\texttt{\$copy}\})) \\ & & \quad\ G_2 \mathrel{\widehat{=}} (\mathsf{copy}_{\mathrm{G}}\{\texttt{\$copy}\}\ \mathsf{to}\ Lp\ \mathsf{in}\ \mathsf{prep}_{\mathrm{G}}(G_1, Lp)) \\ & & \mathsf{in}\ \ \mathsf{unlink}_{\mathrm{G}}(G_2, \{\texttt{\$copy}\}) \end{array}$$

5.4 UTP Model

Our UTP model of locations uses the Abstract Location Trace Graph (ALTG) of Section 5.3.4 to provide a semantics of locations, where the special logical variables Π_{ALTG} and Π_{ALTG}' represent the before and after states of the graph. The contents of this ALTG are then linked to the normal UTP program variables, using a technique inspired by [CHW06]. In our case, the values of normal program variables are mirrored by correspondingly named first-level nodes in the graph. For example, the logical input and output variables for a UTP program variable x are represented by the node $\{x\}$ in the Π_{ALTG} and Π_{ALTG}' graphs respectively. Note that whenever there could be confusion between whether a variable is being used to denote its name rather than its value, we prefix the variable with a dash to get its name. For example, the predicate $x = *_{\mathrm{G}}(\Pi_{\text{ALTG}}, \text{'}x)$ holds whenever the value of variable x equals the value of extracting its corresponding element from the graph Π_{ALTG} (i.e. the one with the path name $\text{'}x$).

The remainder of this section introduces the healthiness conditions on the UTP model of locations, provides the definitions for a few operations, such as assignment, and relates the abstract and concrete models. Here the meta variable Q denotes a relational predicate that defines a UTP location model program.

5.4.1 Healthiness conditions

Before we formalise the relationship between a program's variables and the ALTG, it is worth introducing a healthiness condition to ensure that both the Π_{ALTG} and Π_{ALTG}' graphs are healthy (as defined in Section 5.3.4).

$$\begin{array}{lcl} \mathrm{HC1}_{\mathrm{U}}(Q) & \widehat{=} & Q = (\ Q \land \mathrm{HC}_{\mathrm{U}}(\Pi_{\text{ALTG}}) \land \mathrm{HC}_{\mathrm{U}}(\Pi_{\text{ALTG}}')\ ) \\ \mathrm{HC}_{\mathrm{U}}(G) & \widehat{=} & \mathrm{HC1}_{\mathrm{G}}(G) \land \mathrm{HC2}_{\mathrm{G}}(G) \land \mathrm{HC3}_{\mathrm{G}}(G) \land \mathrm{HC4}_{\mathrm{G}}(G) \end{array}$$

The first step in formalising the link between the graph and program variables is by insisting that the first-level nodes within the graph correspond precisely to the UTP program variables other than the observational variables (i.e. Π_{ALTG} and $\Pi_{\text{ALTG}}{}'$).

$$\begin{aligned}
\text{HC2a}_{\text{U}}(Q) &\mathrel{\widehat{=}} Q = (\ Q \wedge \{\,'x \mid x \in \mathit{inv}\alpha\, Q\} = \mathsf{labels}_{\text{G}}(\Pi_{\text{ALTG}}, \varnothing)\ ) \\
\text{HC2b}_{\text{U}}(Q) &\mathrel{\widehat{=}} Q = (\ Q \wedge \{\,'x \mid x' \in \mathit{outv}\alpha\, Q\} = \mathsf{labels}_{\text{G}}(\Pi_{\text{ALTG}}{}', \varnothing)\ ) \\
\text{HC2}_{\text{U}}(Q) &\mathrel{\widehat{=}} \text{HC2a}_{\text{U}}(Q) \wedge \text{HC2b}_{\text{U}}(Q)
\end{aligned}$$

Here, $\mathit{inv}\alpha\, Q$ and $\mathit{outv}\alpha\, Q$ represent the input and output alphabets of program Q except for the observational variables Π_{ALTG} and $\Pi_{\text{ALTG}}{}'$ respectively; and the child labels of graph path are defined by $\mathsf{labels}_{\text{G}}(G, p) \mathrel{\widehat{=}} \{l \mid P.l \in \mathsf{children}(G, \mathsf{node}_{\text{G}}(G, p))\}$.

The second, and last, step in formalising the link between the graph and program variables is to ensure that the value of a variable is the same as the value stored within the ALTG.

$$\begin{aligned}
\text{HC3a}_{\text{U}}(Q) &\mathrel{\widehat{=}} Q = (\ Q \wedge (\textstyle\bigwedge_{x \in \mathit{inv}\alpha\, Q} \bullet x = *_{\text{G}}(\Pi_{\text{ALTG}}, '\!x))\ ) \\
\text{HC3b}_{\text{U}}(Q) &\mathrel{\widehat{=}} Q = (\ Q \wedge (\textstyle\bigwedge_{x' \in \mathit{outv}\alpha\, Q} \bullet x' = *_{\text{G}}(\Pi_{\text{ALTG}}{}', '\!x))\ ) \\
\text{HC3}_{\text{U}}(Q) &\mathrel{\widehat{=}} \text{HC3a}_{\text{U}}(Q) \wedge \text{HC3b}_{\text{U}}(Q)
\end{aligned}$$

5.4.2 Operations

We only present those operations that significantly differ from those of the standard UTP model, as presented in Chapter 2 of [HH98]: specifically, the assignment and program variable management operations. A more detailed account is provided in Section 6.1, where we adapt these ideas to work with UTP designs.

The assignment operation is broken down into three cases, depending on the type of the r-value (i.e. the value to be assigned). These mirror the three cases presented in the trace-based graph model, except that the location is always defined in terms of a possibly empty path from a UTP program variable. It is defined as follows:

$$\begin{aligned}
x.lp := nlv &\mathrel{\widehat{=}} \text{HCs}_{\text{U}}(\Pi_{\text{ALTG}}{}' = ((\Pi_{\text{ALTG}}, '\!x.lp) :=_{\text{G}} nlv)) \\
x.lp := ecv &\mathrel{\widehat{=}} \text{HCs}_{\text{U}}(\Pi_{\text{ALTG}}{}' = ((\Pi_{\text{ALTG}}, '\!x.lp) :=_{\text{G}} ecv)) \\
x.lp := y.lp_1 &\mathrel{\widehat{=}} \text{HCs}_{\text{U}}(\Pi_{\text{ALTG}}{}' = ((\Pi_{\text{ALTG}}, '\!x.lp) :=_{\text{G}} '\!y.lp_1))
\end{aligned}$$

where $x.\varnothing = x$ and $\text{HCs}_{\text{U}}(Q) \mathrel{\widehat{=}} \text{HC3}_{\text{U}}(\text{HC2}_{\text{U}}(\text{HC1}_{\text{U}}(Q)))$

Note that the combined healthiness condition ensures that the consequences of updating shared values can be seen by all participating UTP program variables.

The variable introduction and elimination operations are also defined in terms of their effects on the ALTG. Here the variable introduction operation provides a default unset value to the introduced variable, and the variable elimination operation removes all references to the value from the graph.

$$\begin{aligned}
\mathsf{var}\, x &\mathrel{\widehat{=}} \exists\, x \bullet \text{HCs}_{\text{U}}(\Pi_{\text{ALTG}}{}' = ((\Pi_{\text{ALTG}}, '\!x) :=_{\text{G}} \text{¿})) \\
\mathsf{end}\, x &\mathrel{\widehat{=}} \exists\, x' \bullet \text{HCs}_{\text{U}}(\Pi_{\text{ALTG}}{}' = \mathsf{unlink}_{\text{G}}(\Pi_{\text{ALTG}}, '\!x))
\end{aligned}$$

5.4.3 Full abstraction

The ALTG-based UTP model of locations, outlined here, is fully abstract in the sense described earlier: two concrete location graphs are equivalent, as defined in Section 5.2.3, iff their corresponding ALTGs are equal. This is essentially because the underpinning ALTG model is fully abstract by design; it removes the need for explicitly indexed shareable locations. Here, each location has precisely one path-set that represents it.

5.5 Summary

We have introduced a UTP model of locations that has two types of location: shareable locations, which can be dereferenced by a pointer and are aliased when copied; and containable locations, which cannot be dereferenced by a pointer and are duplicated when copied. This differs from the other UTP models of pointers, where every location is considered to be shareable, as discussed in Section 7.3.

An advantage of our location approach is that it directly supports languages that have compound entities that are duplicated – rather than aliased – on assignment, without losing the ability to share data via pointer aliasing. For example, both a $C^{\#}$ struct and an $\mathcal{O}$-calculus object are duplicated on assignment, but can be shared by storing the whole value in a shareable location. Further, the model of locations presented here is fully-abstract, as required by our thesis (Section 1.5); it is the basis of the fully-abstract model of objects presented in Section 6.2.

Chapter 6

Combining Objects and Locations

This chapter illustrates how the Unifying Theories of Programming (UTP) model of locations presented in Section 5.4 can in principle be extended to support both the UTP theory of designs and our UTP theory of objects (Chapter 3). This model of locations uses an Abstract Location Trace Graph (ALTG) to model the state of a program (Section 5.3), where:

1. each location is modelled by a *path-set* that contains the potentially infinite set of paths from the root node to the node representing that location; and
2. each UTP program variable x corresponds to a first-level node in the ALTG with the same name (i.e. the node denoted by the path-set $\{\text{'}x\}$).

The second of these design choices is slightly updated within this chapter in order to simplify the presentation of the combined object and location model. Here, we associate the UTP program variables with the children of the first-level *program variables* node $\{\texttt{var}\}$. Therefore, each program variable x – within the alphabet of the UTP program – now corresponds to the node $\{\texttt{var}.\text{'}x\}$, where $\text{'}x$ denotes the name of program variable x rather than its value (as in the previous chapter).

Aside 6.1: The first-level nodes within the ALTG are also used to represent some meta-variables that are outside the alphabet of a UTP program; for example, those variables that are hidden from the scope of a subprogram that calculates the value of a command-expression's argument. Here, the $\{\texttt{hide}\}$ node is used to store a stack of such hidden local variable blocks (frames). Note that this stack is also used to store the call-frames of procedure invocation. Variable hiding is discussed in more detail towards the end of Section 6.1.2.
□

6.1 UTP Designs with Locations

We now illustrate a way of adapting the UTP model of locations presented in Section 5.4 so that it supports the notion of a design. Our outline of a UTP design-based model of locations uses an Abstract Location Trace Graph (ALTG) to provide a semantics of locations, where the logical observation variables Π_{ALTG} and $\Pi_{\text{ALTG}}{}'$ represent the before and after states of the graph. The contents of this ALTG are linked to the UTP program variables (in Section 6.1.1) as discussed in the introduction to this chapter. Having done this, we then (in Section 6.1.2) illustrate how those UTP design operations – that directly update the state of a program – can be adapted to work with this hybrid model. Operations, such as sequential composition and conditional choice, that do not directly update the state of a program, require no adaptation, as they do not directly change the values held in any program or observational variable. Hence, they do not change the contents of the ALTG.

6.1.1 Healthiness conditions

The healthiness conditions presented in this section provide a conservative extension to those of the UTP design, in the sense that when they are applied to a design they return a design. Our strategy is to define the healthiness conditions via the composition of two designs: one that specifies what it is to be healthy; and the other the program to be made healthy. Therefore, we have to transform (adapt) our existing – conjunctive – healthiness conditions for the UTP model of locations (Section 5.4.1) into this form.

Aside 6.2: All of our location healthiness conditions have the property that healthy designs can be written as the sequential composition of two parts: one that is written solely in terms of the input alphabet; and the other that is written solely in terms of the output alphabet. This is not possible for arbitrary conjunctive healthiness conditions in general.
□

Healthiness prefixing: We now consider those conjunctive healthiness conditions, of the form $P = P \wedge q$, where: the UTP program P is made healthy by being conjuctively combined with the – healthiness – predicate q; and the predicate q is written solely in terms of the input alphabet of P. Such a healthiness predicate can be systematically transformed into the first of the two sequentially composed designs by the following preHc utility function (UTP command).

$$\mathsf{preHc}(q) \;\widehat{=}\; \emptyset : (\, q \vdash \mathsf{true}\,)$$

Having systematically transformed the healthiness predicate into a healthiness design, the next step is to show that when it is sequentially combined with another design it returns a design of the expected form; i.e. a design where the healthiness predicate q is a precondition for the correctness of the design as a whole.

Lemma 6.3 – Left healthiness design lemma:

$$\mathsf{preHc}(q) \,;\, (p \vdash P) \;=\; q \wedge p \vdash P$$

provided that q only contains input variables (i.e. undashed variables).
□

Proof

$$\begin{array}{ll}
 & \mathsf{preHc}(q) \,;\, (p \vdash P) \\
= & \cdots\cdots\cdots \text{ Defn. of preHc} \\
 & (q \vdash w' = w) \,;\, (p \vdash P) \\
= & \cdots\cdots\cdots \text{ Law C.2 comp. design} \\
 & q \wedge \neg(w' = w \,;\, \neg p) \vdash w' = w \,;\, P \\
= & \cdots\cdots\cdots \text{ skip rel. unit of } (_ \,;\, _) \\
 & q \wedge \neg(\neg p) \vdash P \\
= & \cdots\cdots\cdots \text{ Law of the excluded middle} \\
 & q \wedge p \vdash P
\end{array}$$
□

Note that it is also straightforward to demonstrate that any design $p \vdash P$ is equal to $p \vdash p \wedge P$. Therefore, the healthiness predicate implicitly prefixes the resultant part (i.e. the post condition) of the design as well.

Healthiness postfixing: Healthiness postfixing is similar to prefixing, except that the conjunctive healthiness predicate is of the form $P = P \wedge Q$, where: the UTP program P is made healthy by being conjuctively combined with the healthiness predicate Q; and the predicate Q is written solely in terms of the output alphabet of P. Such a healthiness predicate can be systematically transformed into the second of the two sequentially composed designs by the following postHc utility function (UTP command).

$$\mathsf{postHc}(Q) \;\widehat{=}\; \emptyset : (\,\mathsf{true} \vdash Q\,)$$

Having systematically transformed the healthiness predicate into a healthiness design, the next step is to show that when it is sequentially combined with another design it returns a design of the expected form; i.e. a design where the healthiness predicate Q is a postcondition for the correctness of the design as a whole.

Lemma 6.4 – Right healthiness design lemma:

$$(p \vdash P) \,;\, \mathsf{postHc}(Q) \;=\; p \vdash P \wedge Q$$

provided that Q only contains output variables (i.e. dashed variables).
□

Proof

$$\begin{aligned}
& (p \vdash P) \,;\, \mathsf{postHc}(Q) \\
= & \quad \text{Defn. of postHc} \\
& (p \vdash P) \,;\, (\mathsf{true} \vdash Q \wedge w' = w) \\
= & \quad \text{Corol. C.3 comp. pre-true} \\
& p \vdash P \,;\, (Q \wedge w' = w) \\
= & \quad \text{Defn. of } (_ \,;\, _) \\
& p \vdash \exists\, w_0 \bullet P[w_0/w'] \wedge Q[w_0/w] \wedge w' = w_0 \\
= & \quad \text{No input vars in } Q \\
& p \vdash \exists\, w_0 \bullet P[w_0/w'] \wedge (Q \wedge w' = w_0) \\
= & \quad \text{One point rule } (w_0 = w') \\
& p \vdash P \wedge Q
\end{aligned}$$
□

Location design healthiness conditions: We are now in a position to systematically transform the three pairs of location healthiness conditions of the UTP location model presented in Section 5.4.1. The first pair of healthiness conditions are that the graphs representing both the input and output states of a program are healthy. That is the graph satisfies the four ALTG healthiness conditions presented in Section 5.3.4.

$$\begin{aligned}
\mathrm{HC1a_D}(P) \;&\widehat{=}\; \mathsf{preHc}(\mathrm{HC_G}(\Pi_{\mathrm{ALTG}})) \,;\, P \\
\mathrm{HC1b_D}(P) \;&\widehat{=}\; P \,;\, \mathsf{postHc}(\mathrm{HC_G}(\Pi_{\mathrm{ALTG}}')) \\
\mathrm{HC1_D}(P) \;&\widehat{=}\; \mathrm{HC1a_D}(\mathsf{skip^D}) \,;\, \mathrm{HC1b_D}(P)
\end{aligned}$$

where $\mathrm{HC_G}(G) = \mathrm{HC1_G}(G) \wedge \mathrm{HC2_G}(G) \wedge \mathrm{HC3_G}(G) \wedge \mathrm{HC4_G}(G)$

Note that the healthiness conditions are deliberately split into pre and post stages. This enables the post-stage healthiness conditions to be applied to the output of a UTP program – that specifies a change to its observational Π_{ALTG} variable – and produce a new UTP operation that healthily maps the UTP program (and other observational) variables to their Π_{ALTG} counterparts.

Having specified the healthiness conditions for ALTG itself, the next step is to formalise the link between the graph and the program variables. This is achieved in two steps, in a similar

manner to that of the UTP location model; they differ in the details of which graph nodes are used to represent the program variables (as discussed in the introduction to this chapter).

The first step in formalising the link between the graph and program variables is by insisting that the children of the {var} node within the graph correspond precisely to the UTP program variables other than the observational variables (i.e. Π_{OK}, Π_{OK}', Π_{ALTG} and Π_{ALTG}').

$$\begin{array}{lcl} \text{HC2a}_{\text{D}}(P) & \widehat{=} & \mathsf{preHc}(\ \{\ '\!x \mid x \in \mathit{inv}\alpha P\} = \mathsf{labels}_{\text{G}}(\Pi_{\text{ALTG}}, \texttt{var})\ ) \fatsemi P \\ \text{HC2b}_{\text{D}}(P) & \widehat{=} & P \fatsemi \mathsf{postHc}(\ \{\ '\!x \mid x' \in \mathit{outv}\alpha P\} = \mathsf{labels}_{\text{G}}(\Pi_{\text{ALTG}}', \texttt{var})\ ) \\ \text{HC2}_{\text{D}}(P) & \widehat{=} & \text{HC2a}_{\text{D}}(\mathsf{skip}^{\text{D}}) \fatsemi \text{HC2b}_{\text{D}}(P) \end{array}$$

Here, $\mathit{inv}\alpha P$ and $\mathit{outv}\alpha P$ represent the input and output alphabets of program P except for the observational variables; and $\mathsf{labels}_{\text{G}}(G, p)$ returns the child labels of graph path (as in Section 5.4.1).

The second step in formalising the link between the graph and program variables is to ensure that the value of a variable is the same as the value stored within the ALTG.

$$\begin{array}{lcl} \text{HC3a}_{\text{D}}(P) & \widehat{=} & \mathsf{preHc}(\ \bigwedge_{x \in \mathit{inv}\alpha P}\ x = *_{\text{G}}(\Pi_{\text{ALTG}}, \texttt{var}.\,'\!x)\ ) \fatsemi P \\ \text{HC3b}_{\text{D}}(P) & \widehat{=} & P \fatsemi \mathsf{postHc}(\ \bigwedge_{x' \in \mathit{outv}\alpha P}\ x' = *_{\text{G}}(\Pi_{\text{ALTG}}', \texttt{var}.\,'\!x)\ ) \\ \text{HC3}_{\text{D}}(P) & \widehat{=} & \text{HC3a}_{\text{D}}(\mathsf{skip}^{\text{D}}) \fatsemi \text{HC3b}_{\text{D}}(P) \end{array}$$

6.1.2 Operations

We now present a sample of four operations (UTP commands) that directly change the state of a program. The first two commands ($\mathsf{var}\ \_$) and ($\mathsf{end}\ \_$) introduce and remove a local variable from the scope of the program respectively. The third command ($\_ :=_{\text{D}} \_$) defines the assignment of a value to a variable (multiple assignment is left for future work). The fourth command ($\mathsf{hide}\ \_\ \mathsf{from}\ \_$) temporarily hides a collection of variables from the scope of a subprogram.

Each of the sample commands is written in terms of operations on the ALTG, which transform the input ALTG (Π_{ALTG}) into the resultant output ALTG (Π_{ALTG}'). They do not however specify the required updates to the linking of UTP variables (and their values) to the nodes in the ALTG. This is achieved by applying the latter two of the post healthiness condition constructors (i.e. HC2b_{D} and HC3b_{D}). For convenience these healthiness condition constructs are combined into the following output healthiness condition constructor.

$$\text{HC23b}_{\text{D}}(P) \ \widehat{=}\ \text{HC2b}_{\text{D}}(P) \fatsemi \text{HC3b}_{\text{D}}(\mathsf{skip})$$

Note that the first of the post healthiness conditions is not applied, as this is assumed to be guaranteed by the application of the ALTG graph operations to a healthy graph. Further, the pre healthiness conditions are not explicitly mentioned within the definition, in order to keep the definition as straightforward as possible. Given that these preconditions are met (and the operations are correctly specified), then it ought to be possible to show that the output design is guaranteed to be healthy. Here, we are only giving an illustration of how our previous work could be combined; it is for future work to iron out the details, and perform the necessary consistency proofs.

Local variable introduction and completion: The local variable x can be introduced by the command ($\mathsf{var}\, x$), so long as it is not already in this command's initial alphabet (i.e. $x \notin A$). When introduced its value is set to the explicit *unset* value ¿.

$$\begin{array}{lcl} \mathsf{var}_A\ x & \widehat{=} & \exists\, x \bullet \text{HC23b}_{\text{D}}(\\ & & \quad \mathsf{true} \\ & & \quad \vdash \\ & & \quad \Pi_{\text{ALTG}}' = ((\Pi_{\text{ALTG}}, \texttt{var}.\,'\!x) :=_{\text{G}} \text{¿}) \\ & &) \end{array}$$

Here, the alphabet of the command as a whole $\alpha(\mathsf{var}\, x)$ is $A \setminus \{x\}$.

The local variable x can be removed from the current scope by the command $(\mathsf{end}\, x)$.

$$\begin{array}{lcl} \mathsf{end}_A\, x & \widehat{=} & \exists\, x' \bullet \mathrm{HC23b_D}(\\ & & \quad \mathsf{true} \\ & & \quad \vdash \\ & & \quad \Pi_{\mathrm{ALTG}}{}' = \mathsf{unlink}_{\mathrm{G}}(\Pi_{\mathrm{ALTG}}, \mathtt{var}.\,'x) \\ & &) \end{array}$$

Here, the alphabet of the command as a whole $\alpha(\mathsf{end}\, x)$ is $A \setminus \{x'\}$.

Assignments: The assignment command is split into three cases, depending on the type of the rvalue (i.e. the value to be assigned). These mirror the three cases presented in the ALTG, except that the location is always defined in terms of a possibly empty path from a UTP program variable. Each of the assignments has the form $x.lp :=_{\mathrm{D}\,A} vp$, where: x represents a program variable in the alphabet A; lp is the path from x to assignable location in Π_{ALTG}; and vp is either a value that contains no references to shareable locations or a path to a location in Π_{ALTG}. For example, $\Pi_{\mathrm{RES}} :=_{\mathrm{D}} 42$ assigns the literal 42 to the special result variable. Note that in this case the empty path ($\varnothing$) has been omitted as it is the unit of path concatenation ($\_.\_$).

Assignment of a scalar: The scalar assignment command $(x.lp :=_{\mathrm{D}\,A} nlv)$ assigns a null or literal value (nlv) to the location $x.lp$, where A is the alphabet of the command, x is a program variable (i.e. $x \in A$) and lp is a possibly empty path from x to an assignable location in the ALTG.

$$\begin{array}{lcl} x.lp :=_{\mathrm{D}\,A} nlv & \widehat{=} & \mathrm{HC23b_D}(\\ & & \quad \mathsf{true} \\ & & \quad \vdash \\ & & \quad \Pi_{\mathrm{ALTG}}{}' = (\Pi_{\mathrm{ALTG}}, \mathtt{var}.\,'x.lp) :=_{\mathrm{G}} nlv \\ & &) \\ \alpha(x.lp :=_{\mathrm{D}\,A} e) & \widehat{=} & A \end{array}$$

Assignment of an encapsulated compound value: The compound assignment command $(x.lp :=_{\mathrm{D}\,A} ecv)$ assigns an encapsulated compound value to the variable x so long as the variable x is in the alphabet of the assignment. (Recall that on page 82 an 'encapsulated compound value' was defined as a compound value that does not contain shareable locations.)

$$\begin{array}{lcl} x.lp :=_{\mathrm{D}\,A} ecv & \widehat{=} & \mathrm{HC23b_D}(\\ & & \quad \mathsf{true} \\ & & \quad \vdash \\ & & \quad \Pi_{\mathrm{ALTG}}{}' = (\Pi_{\mathrm{ALTG}}, \mathtt{var}.\,'x.lp) :=_{\mathrm{G}} ecv \\ & &) \\ \alpha(x.lp :=_{\mathrm{D}\,A} e) & \widehat{=} & A \end{array}$$

Assignment of a program variable: The variable assignment command $(x.lp_1 :=_{\mathrm{D}\,A} y.lp_2)$ assigns the contents of location $y.lp_2$ to the location $x.lp_1$ so long as both x and y are program variables in the alphabet of the assignment and the locations $y.lp_2$ and $x.lp_1$ exist and are

assignable respectively.

$$
\begin{array}{lcl}
x.lp :=_{\mathrm{D}\,A}\ y & \widehat{=} & \mathrm{HC23b_D}(\\
 & & \quad \mathsf{true} \\
 & & \quad \vdash \\
 & & \quad \Pi_{\mathrm{ALTG}}{}' = (\Pi_{\mathrm{ALTG}}, \mathtt{var}.\text{'}x.lp) :=_{\mathrm{G}} \mathtt{var}.\text{'}y \\
 & &) \\
\alpha(x.lp :=_{\mathrm{D}\,A}\ e) & \widehat{=} & A
\end{array}
$$

Hiding and filtering variables from programs: In Section 2.3.4 we discussed why hiding was used in the modelling of the $\mathcal{O}$-calculus. Here, variables were temporarily hidden from the scope of a subprogram (or procedure call) and then restored following its completion.

Aside 6.5: This was achieved via the use of existential quantification, which essentially stored a *copy* of the value of each variable to be hidden, prior to that variable being deleted (i.e. ending the variable's scope).
□

Providing a facility to hide variables can, in principle, lead to a situation where some heap locations are only referenced by these hidden variables. Therefore, as our abstract heap model automatically removes unreferenced entries, we need to ensure that the hidden variables are stored in the abstract heap. Further, as the hiding operations are supplied with an unordered list of variables to hide, we support the creation and deletion of hidden local variable frames, where a frame is a mapping between variables and their values.

We now introduce another two first-level nodes – {`hide`} and {`hTop`} – to store the *stack* of hidden variable frames, where the node:

- {`hide`.i} contains the i^{th} frame on the stack; and
- {`hTop`} contains the index of the top stack entry (or zero if no such entry exists).

Having specified the structure of the hidden variable frame stack, it is useful to provide some operations to manage it. First, we provide the hTop macro that extracts the index on the top of the Π_{ALTG} hidden local variable frame stack.

$$\mathsf{hTop} \equiv *_{\mathrm{G}}(\Pi_{\mathrm{ALTG}}, \mathtt{hTop})$$

Second, we provide two UTP commands for incrementing and decrementing this counter.

$$
\begin{array}{lcl}
\mathsf{incrFr} & \widehat{=} & \{\Pi_{\mathrm{ALTG}}\} : (\ \mathsf{true} \vdash \Pi_{\mathrm{ALTG}}{}' = (\Pi_{\mathrm{ALTG}}, \mathtt{hTop}) :=_{\mathrm{G}} \mathsf{hTop} + 1\ ) \\
\mathsf{decrFr} & \widehat{=} & \{\Pi_{\mathrm{ALTG}}\} : (\ \mathsf{hTop} > 0 \vdash \Pi_{\mathrm{ALTG}}{}' = (\Pi_{\mathrm{ALTG}}, \mathtt{hTop}) :=_{\mathrm{G}} \mathsf{hTop} - 1\ )
\end{array}
$$

These commands are to be used along side the commands for adding and deleting a frame, where the adding of a frame creates an empty variable-value map, which then needs to be populated.

$$
\begin{array}{lcl}
\mathsf{addFrame} & \widehat{=} & \mathrm{HC23b_D}(\\
 & & \quad \mathsf{true} \\
 & & \quad \vdash \\
 & & \quad \Pi_{\mathrm{ALTG}}{}' = (\Pi_{\mathrm{ALTG}}, \mathtt{hide}.\mathsf{hTop}) :=_{\mathrm{G}} \{\} \\
 & &) \\
\mathsf{delFrame} & \widehat{=} & \mathrm{HC23b_D}(\\
 & & \quad \mathsf{true} \\
 & & \quad \vdash \\
 & & \quad \Pi_{\mathrm{ALTG}}{}' = \mathsf{unlink}_{\mathrm{G}}(\Pi_{\mathrm{ALTG}}, \mathtt{hide}.\mathsf{hTop}) \\
 & &)
\end{array}
$$

Once a hidden variable frame has been created (added) it is possible to populate it, by copying a current local variable (and its associated value) into it.

$$\mathsf{initVar}_A\, x \;\widehat{=}\; \mathrm{HC23b_D}(\; \mathsf{true} \vdash \Pi_{\mathrm{ALTG}}' = \mathsf{copy}_{\mathrm{G}}(\Pi_{\mathrm{ALTG}}, \mathtt{var}.\,'x, \mathtt{hide}.\mathsf{hTop}.\,'x)\;)$$

where x is in the alphabet A

The last operation on the hidden variable frames is to recover the data stored in the frame. This is achieved by copying the hidden variable into a variable of the same name in the current local variable block (frame).

$$\mathsf{restoreVar}_A\, x \;\widehat{=}\; \mathrm{HC23b_D}(\; \mathsf{true} \vdash \Pi_{\mathrm{ALTG}}' = \mathsf{copy}_{\mathrm{G}}(\Pi_{\mathrm{ALTG}}, \mathtt{hide}.\mathsf{hTop}.\,'x, \mathtt{var}.\,'x)\;)$$

where x is in the alphabet A

Hide variables command: The hide variables command $\mathsf{hide}\ x_{i=1}^{k}\ \mathsf{from}\ P$ removes the variables $x_{i=1}^{k}$ from the alphabet of P, but following P's completion restores these variables with their previous values.

$$\mathsf{hide}\ \_\ \mathsf{from}\ P \;\widehat{=}\; P$$

$$\mathsf{hide}\ x_{i=1}^{k}\ \mathsf{from}\ P \;\widehat{=}\; \mathsf{incrFr} \fatsemi \mathsf{addFrame} \fatsemi (\fatsemi_{i=1}^{k}\, \mathsf{initVar}\, x_i) \fatsemi (\fatsemi_{i=1}^{k}\, \mathsf{end}\, x_i) \fatsemi P \fatsemi (\fatsemi_{i=1}^{k}\, \mathsf{var}\, x_i) \fatsemi (\fatsemi_{i=1}^{k}\, \mathsf{restoreVar}\, x_i) \fatsemi \mathsf{delFrame} \fatsemi \mathsf{decrFr}$$

Aside 6.6: Limiting a subprogram to a specific set of variables can be achieved in an almost identical manner to that described in Section 2.3.4; the only difference is that this model's variant of the hide command is used in place of the original hide command.
□

6.2 Location Graph Model of the Object Calculus

We now present an outline of a way in which the combined theory of locations and designs can be used to model the $\mathcal{O}$-calculus. First, we discuss how the $\mathcal{O}$-calculus's concrete operand values can be represented in an ALTG. Second, we discuss how the UTP constant–map-design ($\mathcal{U}_{\mathrm{M}}$-design), presented in Section 3.4, can be adapted to use such ALTG-based values. Third, and last, we discuss the feasibility of completing this model.

6.2.1 Value representation

For the purposes of the following discussion we consider the $\mathcal{O}$-calculus labels and methods to be values; they were often treated as values in our previous UTP models of the $\mathcal{O}$-calculus (Chapter 3). Literal values, such as the integers, the booleans, the null pointer, and the explicit

unset value, can be represented in the ALTG by their texts. Here, the node that represents the location of the literal value has a single outgoing edge (to a leaf node) that is labelled with the literal value's text. An object method's label can be similarly represented by its own text.

An object essentially contains a set of distinctly labelled methods. Therefore, it can be represented by a compound value node where each outbound edge: connects to a node representing the location of a method value; and is labelled with the method's name. To avoid potential confusion with labelled values within the ALTG, the object method labels can be prefixed with the $ symbol.

An $\mathcal{O}$-calculus method is represented by a pair, a *self parameter* and an *expression*, where the expression (which represents the method's body) provides the method's definition. An obvious way of representing such a method is by a compound value node that has two outbound edges labelled `_self` and `_body`, where: the `_self` edge leads to a node with a single outbound edge (to a leaf node) that is labelled with the self parameter's name; and the `_body` edge leads to a node with a single outbound edge that is labelled with the program text that represents the method's definition. However, this representation turns out to be problematic, due to a combination of the following four issues:

1. the body of a method can contain inner methods that refer to outer methods' parameters;
2. the method parameters are replaced by their argument value on invocation;
3. the argument provided to a method invocation can be a pointer value; and
4. the ALTG pointer values are unstable – they change during program execution.

The first two issues require the ability to replace a parameter within the program text (and inner program texts) with a value. This is feasible so long as there is a textual representation of any value that is to be substituted; such a substitution approach was used for the first two object models in Chapter 3. The third issue explicitly mentions that pointers can be the values for substitution, thus these values would need a textual representation. The ALTG represents a pointer value by a potentially infinite path-set; such path-sets can be textually denoted by a finite regular expression. The fourth issue says that the path-sets denoting a pointer's value can change during the execution of a program (e.g. assigning the pointer to a new variable adds some additional paths to the path-set that denotes its value). Therefore, any changes in the abstract pointer's values would invalidate the textual representation of the pointer within a method's body. A method of preventing such an invalidation, would be to introduce a unique tag for each pointer, which could be stored textually in the method. However, this would prevent our model of pointers from being fully abstract, as we would have re-introduced pointer labelling.

An alternative approach to the modelling of a method is to split it into a triple, a *self parameter*, a *constant map*, and an *expression*, where the *constant map* is used to store the values of parameters as they are bound. Note that the reason it is called a constant map, is that by the time it is used for method invocation, all its parameters have been bound to their constant values. Having said this, the map is also used to provide unique names to, the otherwise anonymous, inner methods, functions, and objects (during $\mathcal{O}$-calculus compilation as discussed in Section 3.4.3). It is important that the anonymous inner methods and functions are extracted in this manner, because they also can contain outer (pointer) parameters.

Aside 6.7: Extracting the inner-objects is not strictly required, but it does means that their values are expanded into the ALTG during compilation, rather than being contained within the program-text label representing the method's body.
□

Hence, we model a method by a compound value node that has three outbound edges labelled `_self`, `_map`, and `_body`, where: the `_self` and `_body` edges are modelled as before; and the

`_map` edge leads to a compound value node that contains an entry (labelled outbound edge) for each parameter, inner method, inner function and inner object. These outbound edges lead to the location of general, method, function, and object value nodes respectively, and are labelled with their distinct (locally unique) names.

An $\mathcal{O}$-calculus function is represented in the ALTG in an identical manner to that of a method. Function and method definitions differ in their usage, rather than their data representation (e.g. they differ in how they are invoked).

6.2.2 Model definition

The UTP location–graph-design ($\mathcal{U}_{\text{L}}$-design) is similar to that of the $\mathcal{U}_{\text{M}}$-design (in Section 3.4), in the sense that it adapts this UTP model of objects to work with the combined theory of locations and designs. The significant difference is that of the heap representation, which has gone from a straightforward partial map between named abstract locations and operand values, to an abstract graph (i.e. an ALTG) that represents the value of any in scope (or hidden) variable capable of containing a reference value (pointer). Hence, the heap is only one aspect of the more general model of storage provided by the combined theory of locations and designs.

The $\mathcal{U}_{\text{L}}$-design uses the following three pairs of observational variables to define a UTP model of the $\mathcal{O}$-calculus.

- Π_{OK} and Π_{OK}' to represent the start and termination of a program;
- Π_{ALTG} and Π_{ALTG}' to represent the input and output states of a program;
- Π_{RES} and Π_{RES}' to represent the current (working) and final result of a program.

A significant amount of the $\mathcal{U}_{\text{M}}$-design can be directly inherited by the $\mathcal{U}_{\text{L}}$-design, such as its compilation of literal values and its definition of evaluation function, because:

1. the combined theory of locations and designs provides support for most of the operations that the original design supported (including variable declaration and hiding);
2. the UTP commands used to support the object modelling are frequently written in terms of these design operators; and
3. the $\mathcal{O}$-calculus compilation schemes mainly target these supported commands.

Note that these conditions are not sufficient (in all cases) to enable an operation to be inherited. For example, consider the function transformation command $\mathsf{trans}_{\text{M}}(f, k)$, which is inherited from the UTP result–value-design ($\mathcal{U}_{\text{R}}$-design), and defined in Section 3.3.2; it applies function f to the values of the local variables that represent its arguments (i.e. $x_{i=1}^{k}$). If any argument x_i contains a reference value (pointer), which the function f uses, then it is possible that the result will store the current representation of the pointer value, which can become invalid (as previously discussed in Section 6.2.1). Therefore, all functions that previously stored the state of a reference value, or accessed the heap (based on the pointer value), need to be updated or replaced so that they perform actions that are consistent with the ALTG's model of pointers.

Aside 6.8: The arithmetic and logical transformation functions worked only on numeric and boolean values respectively, thus they can be inherited from the $\mathcal{U}_{\text{M}}$-design. Further, their enclosing, command expression based, compilation schemes can also be inherited, as these are written in terms of operators that are supported by both the $\mathcal{U}_{\text{M}}$-design and $\mathcal{U}_{\text{L}}$-design.
□

We now consider how a sample of $\mathcal{U}_{\text{M}}$-design operations can be modelled – specifically those operations that manipulate pointers (references) and methods.

Heap operations: The three $\mathcal{O}$-calculus operations that manipulate the heap can be modelled in a straightforward manner in the $\mathcal{U}_{\text{L}}$-design. First, the fresh location operation fresh is compiled to the $\mathsf{fresh}_{\text{L}}$ UTP command. This command deliberately clears the result variable before assigning the explicitly unset value ($¿$) to a fresh shareable location. Note that if the clear operation had not been performed, then the assignment could set the value of an existing location to $¿$.

$$\mathsf{fresh}_{\text{L}} \mathrel{\widehat{=}} \mathsf{clear}(\texttt{res}) \fatsemi (\Pi_{\text{RES}}.\$:=_{\text{L}} ¿)$$

where

$$\begin{aligned}\mathsf{clear}(p) \mathrel{\widehat{=}}\ & \text{HC23b}_{\text{D}}(\\ & \quad \mathsf{true} \\ & \quad \vdash \\ & \quad \Pi_{\text{ALTG}}' = \mathsf{unlinkChildren}_{\text{G}}(\Pi_{\text{ALTG}}, p) \\ &)\end{aligned}$$

The dereferencing operation $*e$ is compiled to the UTP program $[\![e]\!]_{\text{L}} \fatsemi \mathsf{deref}_{\text{L}}$. Here, the program representing the expression e results in a value, which the $\mathsf{deref}_{\text{L}}$ command uses to dereference a pointer; if the value is not a pointer then the command is specified to behave chaotically, as this is the UTP semantics for a *stuck* $\mathcal{O}$-calculus program.

$$\begin{aligned}\mathsf{deref}_{\text{L}} \mathrel{\widehat{=}}\ & \Pi_{\text{RES}} :=_{\text{L}} \Pi_{\text{RES}}.\$ \\ & \lhd \mathsf{labels}_{\text{G}}(\Pi_{\text{ALTG}}, \texttt{res}) = \{\$\} \rhd \\ & \mathsf{chaos}\end{aligned}$$

The pointer assignment (update) operation is slightly more complex to define, as it requires two expressions to be evaluated: one to represent the pointer to the location to be updated; and the other the value of the update. In order to simplify its definition we introduce the pair_{L} utility command, which stores the results of two programs in a pair.

$$\begin{aligned}\mathsf{pair}_{\text{L}}(P_1, P_2) \mathrel{\widehat{=}}\ & \mathsf{decl}\, x_1, x_2 \,\mathsf{in} \\ & \quad (\mathsf{hide}\, x_1, x_2 \,\mathsf{from}\, P_1) \fatsemi x_1 :=_{\text{L}} \Pi_{\text{RES}} \fatsemi \\ & \quad (\mathsf{hide}\, x_1, x_2 \,\mathsf{from}\, P_2) \fatsemi x_2 :=_{\text{L}} \Pi_{\text{RES}} \fatsemi \\ & \quad \Pi_{\text{RES}} :=_{\text{L}} \{1 \mapsto ¿, 2 \mapsto ¿\} \fatsemi \\ & \quad \Pi_{\text{RES}}.1 :=_{\text{L}} x_1 \fatsemi \\ & \quad \Pi_{\text{RES}}.2 :=_{\text{L}} x_2\end{aligned}$$

We are now in a situation to define the pointer assignment compilation scheme, which pre-evaluates the arguments of the pointer assignment operation before applying the $\mathsf{update}_{\text{L}}$ command.

$$e_1 *= e2 \mathrel{\widehat{=}} \mathsf{pair}_{\text{L}}([\![e_1]\!]_{\text{L}}, [\![e_2]\!]_{\text{L}}) \fatsemi \mathsf{update}_{\text{L}}$$

The $\mathsf{update}_{\text{L}}$ command itself is now straightforward to define. First, it checks whether the first element of the pair is actually a pointer value. If the check fails, then the program is invalid and thus behaves chaotically. If the check passes, then the value stored in the second part of the pair to is assigned to this pointer. Having done this, the result variable is then updated to equal that of the pointer value (to mirror the $\mathcal{O}$-calculus definition).

$$\begin{aligned}\mathsf{update}_{\text{L}} \mathrel{\widehat{=}}\ & \Pi_{\text{RES}}.1.\$:=_{\text{L}} \Pi_{\text{RES}}.2 \fatsemi \\ & \Pi_{\text{RES}} :=_{\text{L}} \Pi_{\text{RES}}.1 \\ & \lhd \mathsf{labels}_{\text{G}}(\Pi_{\text{ALTG}}, \texttt{res}.1) = \{\$\} \rhd \\ & \mathsf{chaos}\end{aligned}$$

Note that the order of processing is slightly different to that of the $\mathcal{O}$-calculus being modelled. Here, the second expression gets evaluated before the check on the first argument's value is performed. This is not a significant problem, because the overall program semantics are preserved. First, we observe that the check does not alter the state of the program, therefore delaying its application does not affect the evaluation of the second expression. Second, we observe that the evaluation of the expression does not affect the outcome of the check, so long as it terminates. If it does not terminate or the check fails, then the $\mathcal{O}$-calculus program never returns or becomes stuck respectively; in either case the result is modelled by the chaotic UTP program, as our UTP semantics does not distinguish between types of failure (i.e. ways in which a program can be invalid).

Method definitions and updates: The method definition follows $\mathcal{U}_{\text{M}}$-design's alternative method definition scheme, presented in Section 3.4.3. This scheme enables the function values and method definitions to be represented by an ALTG, as it removes anonymous object values and program texts.

A method is defined as a triple of compiled program texts that represent the method's *self-variable*, *constant-map* and *body*, where the constant map provides definitions for the self-variable and each of the free-variables within the method's body.

$$
\begin{array}{l|l}
\langle\!\langle \varsigma(x)\ e \rangle\!\rangle_{\text{M}} \mathrel{\widehat{=}} & \\
\quad \{\ \texttt{\_self} \mapsto x, & \text{Self variable} \\
\quad\ \ \texttt{\_map} \mapsto M_1 \cup M_2, & \text{Constant map} \\
\quad\ \ \texttt{\_body} \mapsto \langle\!\langle e \{\!| t \leftarrowtail y \mid y \mapsto t \in M_2 |\!\} \rangle\!\rangle_{\text{M}} & \text{Body} \\
\quad \}
\end{array}
$$

where

$$
\begin{array}{ll|l}
Y & = \mathsf{FV}(e) \cup \{x\} & \text{The method body's } \textit{free} \text{ variables} \\
M & = \mathsf{freshMap}(Y, \mathsf{objsAndProcs}(e)) & \text{The named inner procedure and object map} \\
M_1 & = \{y \mapsto \text{¿} \mid y \in Y\} & \text{The map of, un-set (free), external variables} \\
M_2 & = \{y \mapsto \langle\!\langle e \rangle\!\rangle_{\text{M}} \mid y \mapsto e \in M\} & \text{The map of simulated external variables.}
\end{array}
$$

This compilation scheme ensures that no terms representing either a procedure definition or an object value are contained within a program text (which represents the definition of a method or function). Such procedures and objects have been replaced by local variables that refer to the definitions within a method's constant map. Therefore the $\mathcal{O}$-calculus method update operation is provided with a method identifier (variable), rather than the method itself. The compilation of the transformed method update operation now follows, where: the expression e representing the object to be updated is evaluated, then checked to see if it contains the method label l; if so it replaces the existing method indexed by label l with that stored in the method identifier variable x.

$$
\begin{array}{lcl}
\langle\!\langle e.l \Leftarrow x \rangle\!\rangle_{\text{L}} & \mathrel{\widehat{=}} & \langle\!\langle e \rangle\!\rangle_{\text{L}} \fatsemi (\ \Pi_{\text{RES}}.l :=_{\text{L}} \Pi_{\text{RES}}.l.\texttt{\_map.x} \\
& & \quad \lhd\ l \in \mathsf{labels}_{\text{G}}(\Pi_{\text{ALTG}}, \texttt{res}) \ \rhd \\
& & \quad \mathsf{chaos} \\
& &) \\
\end{array}
$$

Field assignment is actually slightly more complex to define, because it has an extra expression to evaluate, namely, the one that results in the value to be assigned. This is similar to the issue that we encountered for pointer assignment. Therefore, we follow a similar approach of pre-evaluating the expressions and storing them in a pair. Having done this, the next step is to check whether the value in the first argument is an object that contains the member to be updated (the one with label l). If the check fails, then the program is invalid and thus behaves

chaotically. If the check passes, then the program: constructs a method that returns the value to be assigned; and updates the relevant object member with this constructed method value.

$$\begin{array}{lcl} \langle\!\langle e_1.l := e_2 \rangle\!\rangle_{\mathrm{L}} & \widehat{=} & \mathsf{pair}_{\mathrm{L}}(\langle\!\langle e_1 \rangle\!\rangle_{\mathrm{L}}, \langle\!\langle e_2 \rangle\!\rangle_{\mathrm{L}}) \fatsemi \\ & & (\ \Pi_{\mathrm{RES}}.1.l :=_{\mathrm{L}} \{\, \_\mathtt{self} \mapsto \_,\ \_\mathtt{map} \mapsto \{\mathtt{x} \mapsto \text{¿}\},\ \_\mathtt{body} \mapsto \mathtt{x}\} \fatsemi \\ & & \quad \Pi_{\mathrm{RES}}.1.l.\_\mathtt{map}.\mathtt{x} :=_{\mathrm{L}} \Pi_{\mathrm{RES}}.2 \fatsemi \\ & & \quad \Pi_{\mathrm{RES}} :=_{\mathrm{L}} \Pi_{\mathrm{RES}}.2 \\ & & \quad \triangleleft\ l \in \mathsf{labels}_{\mathrm{G}}(\Pi_{\mathrm{ALTG}}, \mathtt{res}.1)\ \triangleright \\ & & \quad \mathsf{chaos} \\ & &)\end{array}$$

Note that the order of processing is slightly different to that of the $\mathcal{O}$-calculus being modelled, in precisely the same manner as that of the pointer assignment.

Method invocation: Method invocation in the $\mathcal{O}$-calculus is compiled in three parts. First, an object-member pair is constructed from the invocation arguments: an expression (e) representing the target object (o) and a label (l). This is achieved by retrieving the method with label l from object o. Second, all the program (non-observational) variables are hidden from view, in order to create the right conditions for method invocation. And third, the actual method invocation is performed using a generic procedure call command ($\mathsf{call}_{\mathrm{L}}$). This executes the body of the method, where the method's self variable has been instantiated with the calling object's value.

$$\begin{array}{lcl} \langle\!\langle e.l \rangle\!\rangle_{\mathrm{L}} & \widehat{=} & \mathsf{pair}_{\mathrm{L}}(\langle\!\langle e \rangle\!\rangle_{\mathrm{L}}, \mathsf{getMethod}_{\mathrm{L}} \langle\!\langle l \rangle\!\rangle^{\mathrm{E}}_{\mathrm{L}}) \\ & & \fatsemi\, \Pi_{\mathrm{OK}}, \Pi_{\mathrm{ALTG}}, \Pi_{\mathrm{RES}} \upharpoonright \\ & & \mathsf{call}_{\mathrm{L}}\end{array}$$

where

$$\begin{array}{lcl} \mathsf{getMethod}_{\mathrm{L}}\ l & \widehat{=} & \Pi_{\mathrm{RES}} :=_{\mathrm{L}} \Pi_{\mathrm{RES}}.l \\ & & \triangleleft\ l \in \mathsf{labels}_{\mathrm{G}}(\Pi_{\mathrm{ALTG}}, \mathtt{res})\ \triangleright \\ & & \mathsf{chaos}\end{array}$$

Note that $\mathsf{getMethod}_{\mathrm{L}}$ makes use of the result of the evaluation of the previous program within the pair, which produces the object that the method is going to be extracted from.

Function invocation is compiled in a similar manner. The key difference is that the function's argument is a general expression, rather than a label.

$$\begin{array}{lcl} \langle\!\langle\ e_1\ e_2\ \rangle\!\rangle_{\mathrm{R}} & \widehat{=} & (\ \mathsf{pair}_{\mathrm{L}}(\langle\!\langle e_1 \rangle\!\rangle_{\mathrm{L}}, \langle\!\langle e_2 \rangle\!\rangle_{\mathrm{L}}) \fatsemi \mathsf{swap}_{\mathrm{L}}(\Pi_{\mathrm{RES}}.1, \Pi_{\mathrm{RES}}.2)\ ) \\ & & \fatsemi\, \Pi_{\mathrm{OK}}, \Pi_{\mathrm{ALTG}}, \Pi_{\mathrm{RES}} \upharpoonright \\ & & \mathsf{call}_{\mathrm{L}} \\ \mathsf{swap}_{\mathrm{L}}(x, y) & \widehat{=} & \mathsf{decl}\ z\ \mathsf{in} \\ & & \quad z :=_{\mathrm{L}} x \fatsemi x :=_{\mathrm{L}} y \fatsemi y :=_{\mathrm{L}} z\end{array}$$

Note that the arguments within the pair are swapped so that they are in the same order as that for method invocation.

The generic procedure call is defined as in the earlier models as the least fix-point of the $\mathsf{apply}_{\mathrm{L}}$ command, which: checks to ensure that the result variable contains a value–program-text pair; assigns the value to the procedure's – and any inner-procedure's – parameter; initialises the procedure's local call block (the parameters in its constant map); and then executes the

program contained within the procedure's program text.

$$
\begin{aligned}
\mathsf{call}_{\mathrm{L}} \quad & \widehat{=} \quad \mu\, z \bullet \mathsf{apply}_{\mathrm{L}}(z) \\
\mathsf{apply}_{\mathrm{L}}(z) \quad & \widehat{=} \quad (\ \mathit{setParam} \fatsemi \\
& \qquad \mathit{getProc} \fatsemi \\
& \qquad \mathit{substCall}(z) \fatsemi \\
& \qquad \mathit{addCallFrame} \fatsemi \\
& \qquad \mathit{extProgText} \fatsemi \\
& \qquad \mathit{delCallFrame} \\
& \quad) \\
& \quad \lhd \Pi_{\mathrm{RES}}.1 \in \mathit{Value} \land \Pi_{\mathrm{RES}}.2 \in \mathit{ProgramText} \rhd \\
& \quad \mathsf{chaos}
\end{aligned}
$$

Here, the subprograms

setParam	Replaces all free occurrences of the procedure's parameter within the constant-map (texts) defined within this procedure. Note that for the purposes of this variable replacement the procedure being invoked is not considered to bind the variable being replaced.
getProc	Select the updated procedure (i.e. $\Pi_{\mathrm{RES}} :=_{\mathrm{L}} \Pi_{\mathrm{RES}}.2$).
substCall(z)	Replaces all occurrences of the program-text $\mathsf{call}_{\mathrm{L}}$ which are defined within the procedure (but not inner procedures), with the contents of the fix-point meta-variable z. Note this can always be represented by a program-text of the form $\mathsf{apply}_{\mathrm{L}}{}^{i}(\mathsf{chaos})$, where i is a natural number, $\mathsf{apply}_{\mathrm{L}}{}^{0}(P) = \mathsf{chaos}$ and $\mathsf{apply}_{\mathrm{L}}{}^{i+1}(P) = \mathsf{apply}_{\mathrm{L}}(\mathsf{apply}_{\mathrm{L}}{}^{i}(P))$.
addCallFrame	Moves the constant map representing the invoked procedure's call frame from the `res._var` node to the `var` node and adds these variables to the programs alphabet.
extProgText	Extracts the definition of the procedure's subprogram to execute from the `res._body` node, unsets the `res` node, and then starts executing the subprogram.
delCallFrame	Deletes the call frame variables introduced by the *addCallFrame* subprogram.

Each of these informally specified subprograms performs a significant role in the procedure invocation. Arguably the most awkward of these subprograms to define are the *setParams* and *substCall*(z), as both modify several locations within the ALTG, which correspond to the locations of the procedure's (and inner procedures') constant-map definitions and the procedure's recursive call program-texts respectively. Here, pattern matching can be used to determine the locations of all inner procedure nodes, which will always have three outbound edges with the reserved labels `_self`, `_map`, and `_body`. Note that care needs to be taken to ensure that the pattern match searching is stopped when a pointer edge (i.e. one with the label `$`) is encountered; otherwise program-texts outside the procedure's definition could be incorrectly updated.

6.2.3 Cell – Method Invocation Example

We now illustrate the $\mathcal{U}_{\mathrm{L}}$-design method invocation process via an example – specifically a cut down version of the restorable cell example, in Section 2.2.5, which only contains the `data` and `get` methods. This cut down version of the restorable cell object is denoted by the following *ExampleCell* $\mathcal{O}$-calculus object.

$$\mathit{ExampleCell} \quad \equiv \quad [\ \mathtt{data} = \varsigma(x)\, 42,\ \ \mathtt{get} = \varsigma(x)\, x.\mathtt{data}\ ]$$

The $\mathcal{U}_{\text{L}}$-design version of *ExampleCell* object can be systematically constructed by applying the compilation schemes previously presented in this chapter (and those inherited from Chapter 3). The result of this compilation is now denoted by the following *ExCel* UTP object.

$$\begin{array}{lcl} \mathit{ExCel} & \equiv & \{\ \texttt{data} \mapsto \{\ \texttt{\_self} \mapsto x,\ \texttt{\_map} \mapsto \{x \mapsto \text{¿}\},\ \texttt{\_body} \mapsto \mathcal{E}_{\text{L}}\, 42\}, \\ & & \quad \texttt{get} \mapsto \{\ \texttt{\_self} \mapsto x,\ \texttt{\_map} \mapsto \{x \mapsto \text{¿}\},\ \texttt{\_body} \mapsto G\_PT\} \\ & & \} \end{array}$$

where

$$\begin{array}{lcl} G\_PT & \equiv & \mathsf{pair}_{\text{L}}(\mathcal{E}_{\text{L}}\, x, \mathsf{getMethod}_{\text{L}}\, \texttt{data}) \\ & & \fatsemi\ \Pi_{\text{OK}}, \Pi_{\text{ALTG}}, \Pi_{\text{RES}}\ \restriction \\ & & \mathsf{call}_{\text{L}} \end{array}$$

The *ExCel* object can be represented by a tree, within an ALTG, as illustrated in Figure 6.1. Having said this, future ALTG illustrations that contain this tree, typically represent the whole

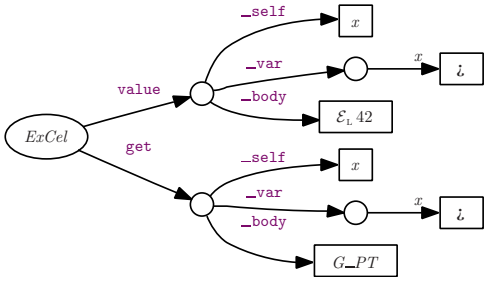


Figure 6.1: Example cell object – tree representation

tree by an *ExCel* labelled node, where multiple instances of the *ExCel* node represent distinct copies of the tree.

Representing the method invocation: We now consider the *ExampleCell*.get $\mathcal{O}$-calculus program, which invokes the *ExampleCell*'s get method. This is systematically compiled to the following $\mathcal{U}_{\text{L}}$-design program as follows.

$$\begin{array}{l} \mathsf{pair}_{\text{L}}(\mathcal{E}_{\text{L}}\, \mathit{ExCel}, \mathsf{getMethod}_{\text{L}}\, \texttt{get}) \\ \fatsemi\ \Pi_{\text{OK}}, \Pi_{\text{ALTG}}, \Pi_{\text{RES}}\ \restriction \\ \mathsf{call}_{\text{L}} \end{array}$$

Note that this variant of the ternary filter operation ($P \fatsemi\ \Pi_{\text{OK}}, \Pi_{\text{ALTG}}, \Pi_{\text{RES}}\ \restriction\ Q$) sequentially composes the result of the first sub-program (P) with that of the second sub-program (Q), where the non-observational variables within the first subprogram are hidden from the scope of the second sub-program (i.e. all variables but Π_{OK}, Π_{RES} and Π_{ALTG} are hidden). In this case, the first sub-program ($\mathsf{pair}_{\text{L}}(\mathcal{E}_{\text{L}}\, \mathit{ExCel}, \mathsf{getMethod}_{\text{L}}\, \texttt{get})$) is only defined in terms of the observational variables, so the empty set of variables is removed from the second sub-program.

$$\begin{array}{l} \mathsf{pair}_{\text{L}}(\mathcal{E}_{\text{L}}\, \mathit{ExCel}, \mathsf{getMethod}_{\text{L}}\, \texttt{get})\ \fatsemi \\ (\ \mathsf{hide}\ \_\ \mathsf{from} \\ \quad \mathsf{call}_{\text{L}} \\ ) \end{array}$$

This is equivalent to not performing any hiding at all. So, the resultant program for our method invocation can be simplified to:

$$\begin{array}{l} \mathsf{pair}_{\text{L}}(\mathcal{E}_{\text{L}}\, \mathit{ExCel}, \mathsf{getMethod}_{\text{L}}\, \texttt{get})\ \fatsemi \\ \mathsf{call}_{\text{L}} \end{array}$$

Here, the outer level sequential composition operator provides a convenient break point in our example presentation, as it forms an imperative step between two parts of the example calculation. In general, we aim to reduce our UTP program into a sequence of imperative steps, as this illustrates the sequence of steps that an execution of the program would essentially take. Having said this, we do not reduce the variable declaration and hiding to this form, as it is useful to keep their block scoped form. Further, we unwind the method invocation as each method is invoked, in order to manage the amount of program that is being viewed at any point in time.

Evaluating the object-method pair: For the moment we focus on the first part of the method invocation process, which evaluates the arguments, and stores them – as a pair – in the observational result variable (Π_{RES}).

$$\mathsf{pair}_{\text{L}}(\mathcal{E}_{\text{L}}\ \mathit{ExCel}, \mathsf{getMethod}_{\text{L}}\ \texttt{get})$$

This subprogram can be rewritten to a block structured imperative style by recursively applying the definitions of the $\mathcal{U}_{\text{L}}$-design commands (such as pair_{L}). This results in the following program definition, where each step of the execution is labelled by its step number.

$$\begin{array}{lr}
(\ \mathsf{decl}\ x_1, x_2\ \mathsf{in} & (1)\\
\quad (\ \mathsf{hide}\ x_1, x_2\ \mathsf{from} & (2)\\
\qquad \mathcal{E}_{\text{L}}\ \mathit{ExCel} & (3)\\
\quad)\ ⨟ & (4)\\
\quad x_1 :=_{\text{L}} \Pi_{\text{RES}}\ ⨟ & (5)\\
\quad (\ \mathsf{hide}\ x_1, x_2\ \mathsf{from} & (6)\\
\qquad \mathsf{getMethod}_{\text{L}}\ \texttt{get} & (7)\\
\quad)\ ⨟ & (8)\\
\quad x_2 :=_{\text{L}} \Pi_{\text{RES}} & (9)\\
\quad \Pi_{\text{RES}} :=_{\text{L}} (¿, ¿) & (10)\\
\quad \Pi_{\text{RES}}.1 :=_{\text{L}} x_1 & (11)\\
\quad \Pi_{\text{RES}}.2 :=_{\text{L}} x_2 & (12)\\
) & (13)
\end{array}$$

Here, step 1 introduces local variables x_1 and x_2 to store the result of evaluating the first and second arguments respectively. Steps 2–5 perform the evaluation of the first argument; i.e. the subprogram that evaluates the object that is the target of the method invocation. Note that the local variables are hidden from the evaluation of this sub-program ($\mathcal{E}_{\text{L}}\ \mathit{ExCel}$), as their only purpose is to temporarily store the result of executing the sub-program.

Steps 6–9 perform the evaluation of the second argument in a similar manner to that of the first argument. The difference is that the ($\mathsf{getMethod}_{\text{L}}\ \texttt{get}$) sub-program atypically makes use of the value stored within the Π_{RES} variable, which is that of the target object (i.e. the same value as stored in x_1).

Steps 10–12 perform the creation of the object-method pair, and the storing of the object and method values within the first and second elements of that pair respectively. Step 13 removes the local variables from scope.

The overall affect of these thirteen steps is summarised in Figure 6.2, which illustrates both the initial state of the UTP program's Π_{ALTG} and the state of the program's Π_{ALTG} after the thirteenth step of execution. Recall that *ExCel* and *G_PT* denote the tree representing the example cell object and the program text of the `get` method respectively. Appendix D contains a step-by-step illustration of the thirteen changes to the UTP program's Π_{ALTG}.

Expanding the first method call: The second part of the method invocation is the generic procedure call_{L} program, which is defined to be the weakest fix-point of the $\mathsf{apply}_{\text{L}}$ program (i.e.

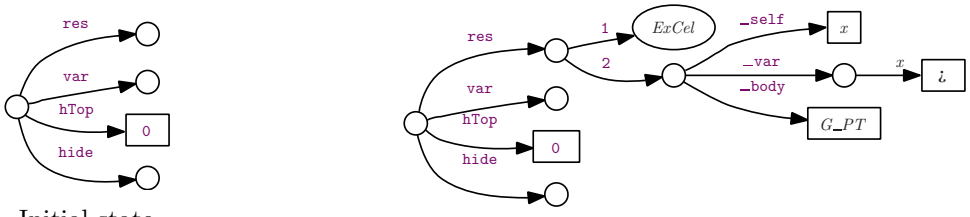


Figure 6.2: Evaluation pair summary

$\mu\, z \bullet \mathsf{apply}_{\mathrm{L}}(z)$). In the case of our example the weakest fix-point is the program $\mathsf{apply}^2_{\mathrm{L}}(\mathsf{chaos})$, which can be rewritten with the definitions of the $\mathsf{apply}^2_{\mathrm{L}}$ and $\mathsf{apply}_{\mathrm{L}}$ to generate the following UTP program.

$$
\begin{array}{l}
(\ \mathit{setParam} \fatsemi \\
\quad \mathit{getProc} \fatsemi \\
\quad \mathit{substCall}(\mathsf{apply}_{\mathrm{L}}(\mathsf{chaos})) \fatsemi \\
\quad \mathit{addCallFrame} \fatsemi \\
\quad \mathit{extProgText} \fatsemi \\
\quad \mathit{delCallFrame} \\
) \\
\lhd\, \Pi_{\mathrm{RES}}.1 \in \mathit{Value} \land \Pi_{\mathrm{RES}}.2 \in \mathit{ProgramText}\, \rhd \\
\mathsf{chaos}
\end{array}
$$

The above program can be further simplified by observing that following the evaluation of the object-method pair, the Π_{RES} variable contains a *Value* and a *ProgramText*, thus the true branch of the conditional can be taken. This produces the following program, where the twenty three steps (steps 18–40) involved with executing the body of the method itself are denoted by the *extProgText* program.

$\mathit{setParam} \fatsemi$	(14)
$\mathit{getProc} \fatsemi$	(15)
$\mathit{substCall}(\mathsf{apply}_{\mathrm{L}}(\mathsf{chaos})) \fatsemi$	(16)
$\mathit{addCallFrame} \fatsemi$	(17)
$\mathit{extProgText} \fatsemi$	$(18 - 40)$
$\mathit{delCallFrame}$	(41)

For the moment we focus on the first four steps, which are step 14–17 of our example's execution. These are illustrated in Figure 6.3, where G_PT' denotes the program-text that is created by substituting all the $\mathsf{call}_{\mathrm{L}}$ texts within the method's program-text with the text $\mathsf{apply}_{\mathrm{L}}(\mathsf{chaos})$, and:

$$
\begin{array}{lll}
G\_PT' & \widehat{=} & \mathsf{pair}_{\mathrm{L}}(\mathcal{E}_{\mathrm{L}}\, x, \mathsf{getMethod}_{\mathrm{L}}\, \texttt{data}) \\
& & \fatsemi\, \Pi_{\mathrm{OK}}, \Pi_{\mathrm{ALTG}}, \Pi_{\mathrm{RES}} \upharpoonright \\
& & \mathsf{apply}_{\mathrm{L}}(\mathsf{chaos})
\end{array}
$$

Step 18 is more involved as it extracts the above program text, which represents a method invocation that is similar to that of the initial program. The three main differences between this second invocation and the original are that:

1. the target object (*ExCel*) is stored in the local variable x, but is otherwise a duplicate of the original object;

2. the method to be selected is labelled with `data`, rather than `get`; and

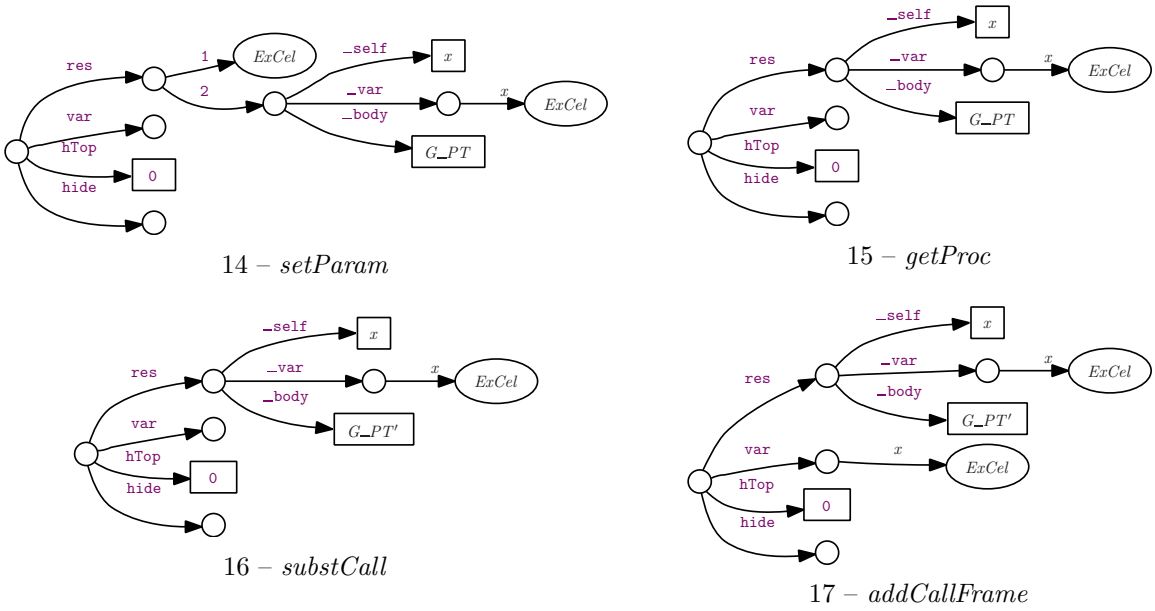


Figure 6.3: Steps 14 to 17 – Preparing the call frame

3. the $\mathsf{call}_{\mathrm{L}}$ program has been replaced by the $\mathsf{apply}_{\mathrm{L}}(\mathsf{chaos})$ (as part of the original method's weakest fix-point calculation).

The effect of this extraction is illustrated in Figure 6.4. Following the extraction, the program

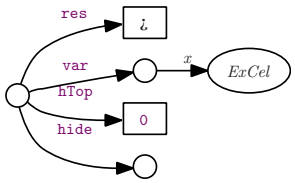


Figure 6.4: Inner program text extraction

contained within the inner method is essentially inserted into the execution path at this point in the program's execution; it takes twenty two steps (steps 19-40) to execute before returning control back to the original outer method.

Expanding the second method invocation: The second, inner, method invocation follows the same pattern as the original method invocation. First, the ternary filter is rewritten to produce the sequential composition of two program texts: the first performing the construction of an object-method pair; and the second performing the actual method invocation, once the original method's local variable x is hidden from scope.

$$
\begin{array}{ll}
\mathsf{pair}_{\mathrm{L}}(\mathcal{E}_{\mathrm{L}}\, x, \mathsf{getMethod}_{\mathrm{L}}\, \mathtt{data})\ ; & (19-31)\\
(\ \ \mathsf{hide}\ x\ \mathsf{from} & (32)\\
\qquad \mathsf{apply}_{\mathrm{L}}(\mathsf{chaos}) & (33-39)\\
) & (40)
\end{array}
$$

The thirteen steps (steps 19–31) to perform pair evaluation are almost identical to those of the original object-method pair evaluation. The overall effect of these steps on the Π_{ALTG} is summarised in Figure 6.5.

Having got the inner object-method pair, the next step (step 32) is to hide all the non-observational variables from the scope of the inner-method's execution, as illustrated in Figure 6.6, so that the program's alphabet at the point of a procedure call contains only the

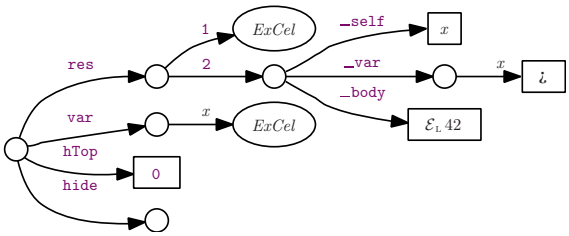


Figure 6.5: Steps 19–31 – Result of the inner method pair evaluation

$\mathcal{U}_L$-design observational variables. Recall that this is a requirement of the way our model handles potentially mutually recursive procedure invocation as discussed in Section 3.1.2.

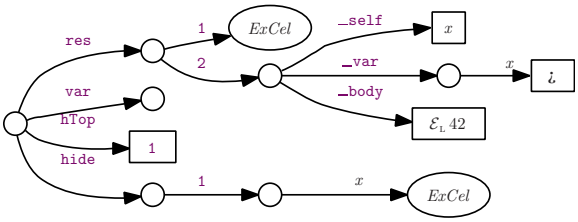
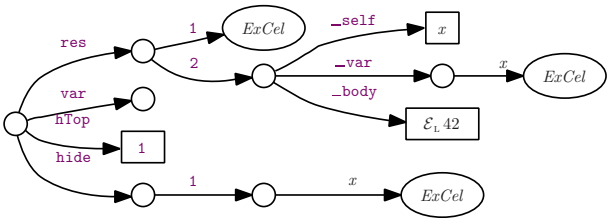


Figure 6.6: Step 32 – Hide the local variables

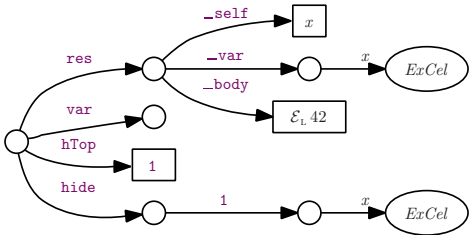
The apply_L program can now be rewritten in a similar manner to before, to produce the following program.

setParam ⨟	(33)
getProc ⨟	(34)
substCall(chaos) ⨟	(35)
addCallFrame ⨟	(36)
extProgText ⨟	(37 – 38)
delCallFrame	(39)

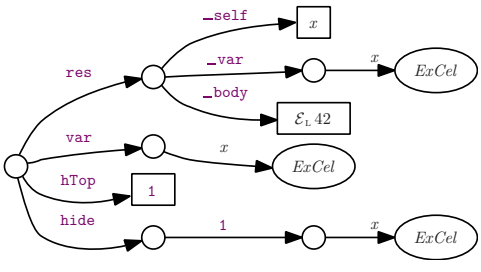
The first four steps (steps 33–36) prepare the inner method call frame as before, except that this time there is no call_L text to substitute, and thus no change in state following the execution of the *substCall* command. The affects of these steps on the Π_{ALTG} are illustrated in Figure 6.7.



Step 33 – *SetParam*



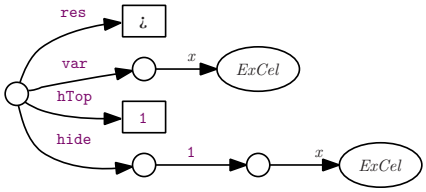
Step 34 – *GetProc* (and Step 35 – *SubstCall*)



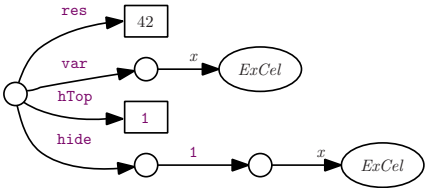
Step 36 – *AddCallFrame*

Figure 6.7: Steps 33–36 – Prepare the inner call frame

Following the preparation of the call frame the inner method is extracted and executed, as illustrated in Figure 6.8. Here the inner program consists of a single evaluation command ($\mathcal{E}_{\text{L}}\,42$), which is step 38.



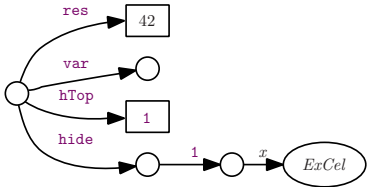
Step 37 – *ExtProg*



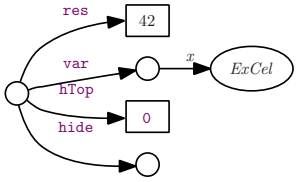
Step 38 – Evaluate the inner program

Figure 6.8: Steps 37–38 Extract and execute inner method

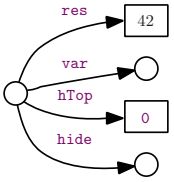
Complete the method invocations: Now all that remains is to complete the two method invocations. This amounts to deleting the inner method's call frame, restoring the outer method's local variable, and then deleting the outer method's call frame. Figure 6.9 illustrates the affect these last three steps (steps 39–41) have on the Π_{ALTG}. Note that the final



Step 39 – *DelCallFrame*



Step 40 – Restore the hidden state



Step 41 – *DelCallFrame*

Figure 6.9: Steps 39–40 – Complete both invocations

Π_{ALTG} contains the value 42 in its result variable, which is the expected result of executing the example program.

6.2.4 Feasibility

So far we have illustrated an approach to adapting the $\mathcal{U}_{\text{M}}$-design so that it can use the combined model of locations and designs. In particular, we have discussed how the ALTG can be used to store the various sorts of data used within the $\mathcal{O}$-calculus, and how these graphs can be used to represent the evolving state of a $\mathcal{U}_{\text{L}}$-design program.

There is still a significant amount of work required to complete the $\mathcal{U}_{\text{L}}$-design. Having said this, we have illustrated how some of the more awkward UTP commands could be modelled in terms of an ALTG. We argue that this is sufficient to show that it is possible to use an ALTG to represent the evolving state of a program. Hence, it is feasible to construct a $\mathcal{U}_{\text{L}}$-design from the combined model of locations and designs to represent the $\mathcal{O}$-calculus.

A more difficult question is whether the $\mathcal{U}_{\text{L}}$-design is better than the preceding models. It inherits a fully abstract model of heap storage from the UTP model of locations (Chapter 5), which is an improvement on the previous models. However, a cost of this improvement is that the abstract *path-set* representation of a pointer's address (i.e. the pointer's value rather than its contents) can change during the execution of a program, such as when a variable (x) containing a pointer is assigned to null. In this case the path-set representing the pointer is updated so that it no longer contains paths that originate from the given variable (i.e. those that are prefixed by `var.x`). Such changes in value reflect the updates to the ALTG that is being used to model the logical structure of a program's storage. Therefore, we cannot use existential quantification to represent the value of a pointer, whenever the underpinning ALTG that it is associated with might change.

Example 6.9: We used existential quantification to hide variables from the execution of a subprogram; when the subprogram had completed the variables were then reinstated with the existentially quantified values. Our solution to this issue was to add some extra nodes to the graph to store such previously existentially quantified values; the pointers within these values would then be maintained by the ALTG operations.
□

Whether the complications added by the introduction of the extra nodes to store variables that are temporarily outside the scope of a subprogram is worth the benefit of a fully abstract heap is a subjective matter of judgement. It is likely to depend on the purpose for which the model is being used, as this will determine how frequently two heaps will be compared to determine whether they are equal (or one contains the other). If this is happening frequently, then a fully abstract model of memory is likely to be beneficial, whereas if this is a rare occurrence then it may be easier to compare concrete heap representations. Note that it is straightforward to construct an abstract heap representation from a concrete heap (say for the purposes of such a comparison).

Overall, we argue that it is feasible to construct a $\mathcal{U}_{\mathrm{L}}$-design from the combined model of locations and designs to represent the $\mathcal{O}$-calculus. Thus, it is possible to provide an object-based object orientation to the UTP, which has value- and reference-based objects, and a fully abstract model of references; this was our thesis (Section 1.5).

6.3 Summary

We have outlined how the UTP model of locations presented in Chapter 5 can be extended to the standard UTP theory of designs and our UTP theory of objects (Chapter 3). In particular, we have shown that it is feasible to use the UTP model locations to provide a fully-abstract model of pointers for our UTP theory of objects. This completes the demonstration of our thesis, which stated that 'it is possible to provide an object-based object orientation to the UTP with value- and reference-based objects, and a fully abstract model of references' Section 1.5.

Chapter 7

Discussion

We have extended the Unifying Theories of Programming (UTP) [HH98, WC04] so that it can model object-based object-oriented (OO) programs, which are written in an Abadi–Cardelli-style object calculus [AC96] – specifically our $\mathcal{O}$-calculus [SG07] (as described in Section 2.2). This provides the $\mathcal{O}$-calculus with a relational predicate denotational semantics. 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 untyped object calculus (ς-calculus).

This chapter provides a summary of the work done (Section 7.1), a discussion of *closely* related work (Section 7.2 and Section 7.3), some future possibilities (Section 7.4), some personal reflections (Section 7.5), and the conclusions (Section 7.6). By 'closely related' we mean any work that considers using the UTP to model objects or pointers, or any work that we have been inspired by. Work that is not closely related has already been discussed in the introduction (Chapter 1) and Appendix A; this includes a general discussion of OO-concepts and alternative approaches to the formal modelling of object orientation (Section 1.3).

7.1 Summary

We presented four UTP models of an Abadi–Cardelli-style object calculus (the $\mathcal{O}$-calculus). This $\mathcal{O}$-calculus directly supports value-based objects, eagerly evaluated functions, references, literal values and their operations, conditional evaluation, local declaration blocks, and sequential composition – but not types, classes, or other class-based notions such as inheritance.

All four models extend the notion of a UTP design. The first three introduce a concrete model of a heap, which mirrors that of the $\mathcal{O}$-calculus; i.e. a partial map from abstract locations to values. The last model provides an alternative – fully abstract – model of the heap, which is based on a directed graph that has anonymous nodes and distinctly labelled edges. Here, the abstract locations are represented by the, potentially infinite, set of finite paths to a given location node within the Abstract Location Trace Graph (ALTG) from the root node.

Operand stack design: The UTP operand–stack-design ($\mathcal{U}_\mathrm{s}$-design) model of the $\mathcal{O}$-calculus does not make use of the UTP's local variable introduction (and completion) feature; instead all temporary values are stored on the operand stack. This simplifies the handling of method and function invocation, as the alphabet of all subprograms is the same; i.e. there are no alphabet issues to consider when determining the weakest fix-point of a procedure (method or function) call. Most of the other operations within the $\mathcal{O}$-calculus are straightforward expressions, in the sense that they apply a simple function (e.g. addition) to an evaluation of the expressions representing its arguments. Such operations (expressions) are transformed into UTP programs via the command expression ($\mathsf{cmdExp}_\mathrm{S}$) constructor, which stores the results of executing the programs representing the operation's arguments on the operand stack, before applying the appropriate underlying function.

Result variable design: The UTP result–value-design ($\mathcal{U}_{\mathrm{R}}$-design) replaces the operand stack with a combination of a special result variable (which is similar to an accumulator) and some local variables for storing intermediate results in a calculation (of a command expression). Note, the alphabet at the point of a procedure invocation call is guaranteed to contain only the observational variables; this follows from the observation that the local variables for storing the results of evaluating a command expression's arguments are hidden from the subprograms that evaluate its arguments.

Constant map design: The UTP constant–map-design ($\mathcal{U}_{\mathrm{M}}$-design) extends the $\mathcal{U}_{\mathrm{R}}$-design by using a local variable to represent a procedure's formal parameter. A nested procedure also *inherits* the parameters of its enclosing parent (grandparent, etc.) procedures, which are similarly represented by UTP local variables. Now as the UTP local variables that represent a procedure's parameters are visible throughout the definition of its body, they will be visible at the time of a procedure's invocation. A consequence of this is that the alphabet just prior to the procedure invocation can vary, as distinct procedures may have different parameter sets. This is at odds with our use of the weakest fix-point to model procedure invocation, which requires the alphabets at the point of the procedure invocation call to be the same. The solution we adopted was:

1. To hide the current procedure's local variables (parameters) as part of the procedure invocation, just prior to the actual procedure call; these variables are then reinstated once the called procedure has terminated.

2. To change the procedure definition from a pair (representing the explicit parameter and the body) to a triple (representing the explicit parameter, the body, and a constant parameter map); here the map is from variable names to values, which once set cannot be changed.

Note that the constant map is only changed as part of a procedure invocation, where the procedure's explicit parameter is bound in both its own map and the maps of any inner procedure (that does not redeclare that parameter).

Location graph design: The UTP location–graph-design ($\mathcal{U}_{\mathrm{L}}$-design) uses an ALTG to represent complex relationships between abstract locations and their data. Here, both shareable and containable locations are modelled by a path-set. They differ in that only shareable locations can be dereferenced by a pointer, whose value is the shareable location's path-set itself. The key point is that containable locations actually contain rather than reference their contents; thus, when they are copied their contents are duplicated rather than shared. This mirrors situations where the whole of a compound value, such as a Pascal `record` or a C^{++} `struct`, is duplicated on assignment. In general, being able to control the amount of data that gets duplicated on assignment provides a means for directly supporting different levels of containment within a data structure.

One consequence of modelling a pointer's value as the path-set that defines its location is that it is possible to directly represent the concept of a handle (i.e. a pointer to a pointer). The combination of having direct support for contained locations and pointer values mirrors the features of the $\mathcal{O}$-calculus.

Garbage collection: The $\mathcal{U}_{\mathrm{L}}$-design's model of a heap can only represent reachable locations. Therefore, any completely faithful implementation of this heap model must free memory as soon as it was no longer reachable. This places a significant obligation on the implementation, which may not be desirable, say for efficiency or applicability reasons; e.g. we may wish to use this model of a heap to describe a system that uses a less eager garbage collection scheme. In such cases, we can relax the acceptable implementation criterion, to say that all reachable locations

in the specification have a corresponding location in the implementation (which stores the same value).

Consistency: The $\mathcal{O}$-calculus's semantics is defined by a collection of small-step reduction rules, which are deterministic in the sense that at most one rule can be applied to any given $\mathcal{O}$-calculus term (in its reduction context). The semantics of the corresponding UTP designs are each defined as a relational predicate, which contains a four-member alphabet: two members for representing the input and output values of a program's heap storage (i.e. Π_{HEAP} and Π_{HEAP}'); and two members representing the input and output values of the program's result variable, which is either Π_{STK} and Π_{STK}' or Π_{RES} and Π_{RES}' depending on the UTP model being used. Note that there are no input parameters, as these are required to be bound to the program, as we only demonstrate the consistency of closed programs; i.e. programs without free variables.

We consider the $\mathcal{O}$-calculus and UTP semantics to be consistent so long as the UTP semantics of the $\mathcal{O}$-calculus term prior to an operational step *equals* that of the UTP semantics of the $\mathcal{O}$-calculus term after the operational step. This consistency was proven for the UTP operand-stack, result-variable, and constant-map designs; it has not been proven for the location-graph design, which was created to illustrate how our UTP object and location models could be combined.

7.2 Predicative Objects

Within this section we discuss the work on representing objects as (relational) predicates. Having done this, we identify those works that already support a refinement calculus, as this is an important step in enabling the theories to support practical programming techniques. We finish this section with a comparison between the work we have just discussed and that being presented in this dissertation.

This is not the first time object-oriented ideas have been added to (or modelled in) the UTP. Two teams have made significant progress with adding class-based OO-support to the UTP; for convenience we refer to them as the rCOS and Circus teams. They have both used higher-order programming concepts, in the sense that object values essentially contain the methods that can be invoked on them.

rCOS: The rCOS team's work [HLL02a, HLL02b, HLXS04, HLL06, CHH+07] has resulted in the development of the Refinement Calculus of Object Systems (rCOS). Here, an object value is denoted by a unique object-reference. Such references are bound to:

- the name of the class they are implementing (which is unique);
- the name of the class that they have been cast to (which is either the class's own name or the name of one of its ancestor classes); and
- the values for the class's – direct or inherited – attributes (i.e. fields).

Each class directly contains those fields and methods that it declares and defines respectively; it also indirectly contains any inherited methods and fields, from the class that it optionally extends. An rCOS OO-program is denoted by a pair, where the first element is a list of class declarations, and the second is the 'main' algorithm of a program that may make use of these classes. The semantics of an rCOS program is also split into two parts, which correspond to the two elements within the program pair. First, the class declaration list is used to construct a 'static' class-relationship environment. Second, this environment is used to determine the 'dynamic' evaluation of the program's main algorithm.

Circus: The Circus team's work [CSW03, CSW05, SCS06, CHW06, HCW08] is being used to extend the Circus language [WC02, OCW07], which is a concurrent language for refinement that unifies Z [Spi92, WD96], CSP (Communicating Sequential Processes) [Hoa85, Ros98], and the refinement calculus [Mor94, BvW98] in the UTP. Here, the work on objects and pointers has been done independently, but with a view to merging the results. We discuss the pointer work in Section 7.3.

The object extension to the Circus language introduces a mechanism for declaring classes and their associated methods, prior to the creation of the objects that use them. A significant difference between this work and the rCOS work is that the notion of object references is not bound in with the modelling of the objects themselves. Another significant difference is that methods are not directly contained by their classes. Instead, each method is an entity in its own right, which specifies the class it is directly bound to. This has the affect of allowing the definition of recursive (or mutually recursive) method invocations to be defined directly in terms of a fix-point over methods, rather than a fix-point over classes and their methods.

TCOZ: In addition to these teams, Qin, Dong, and Chin have adapted [HLL02a] to support the Timed Communicating Object-Z (TCOZ) class-based OO-specification language [QDC03]. This work integrates trace-based notions of concurrency, in a similar manner to that in [HH98]. Method invocation of passive methods – i.e. methods that update the local object state – is performed in a similar manner to that of the rCOS team. Active methods are used to interact with other threads of communication (which are called processes); here, the communication event contains the arguments that are being used in the active method invocation.

Predicative objects: Within the more general field of predicative programming [Heh93], another notion of class-based object orientation has been modelled [Kas04, Kas06]. Here, both objects and classes are special forms of a more general predicative construct. In [Kas04] these predicative constructs are called 'object-specifications', which are essentially a subset (though technically a subbunch) of the object-values (a named bunch) that conform to a given predicate. Such predicates, are used to define the relationships (e.g. axioms) between the object-values's methods. Objects and classes place further constraints on the object-specifications. Here, an object belongs to any class whose defining predicate is implied by the object's defining predicate.

In [Kas06] the object specifications are replaced by 'theories' each of which is a named collection of identifiers (called the vocabulary) and a boolean expression (called the axiom of the theory). This is similar to the UTP concept of a theory, where the UTP's observational variables and healthiness conditions correspond to the vocabulary and the axiom of [Kas06] respectively.

Aside 7.1: The axiom of the theory may use identifiers that are not in its vocabulary; such identifiers are called the external names of the theory, and are used to parametrise a theory. Theories can also be combined (by conjunction), so long as their defining predicates are consistent.
□

Objects and classes are defined in a [Kas06] programming theory – i.e. 'a theory that describes computations'. Here, an object consists of a pair: a type, which represents the class(es) of object it belongs to; and a record that binds distinctly named attributes to their mutually disjoint addresses. A class is essentially a set of objects that share the same type and attribute names. One consequence of this is that methods are not strictly part of a class, but rather are procedures whose first argument is the 'self' object (of a given type of class).

Refinement Calculus: The objective of a refinement calculus [Mor94, BvW98] is to provide a collection of laws that support an engineer when they perform the stepwise transformation of

a specification into an implementation, and the verification that an implementation meets its specification. Here, the usefulness of the refinement calculus can be subjectively measured, by the amount of support it provides to the engineer (e.g. does it provide simple transformations at the source code, rather than semantic model, level?). Technically, what we have done is a refinement calculus – it has a refinement ordering (i.e. implication) and some laws – but it is not useful as such because the laws do not support refinement at the source code level (i.e. $\mathcal{O}$-calculus level). Both [HLL06] and [Kas06] provide refinement calculi, for their variants of a predicative programming language.

Comparison: Both the class-based UTP and predicative programming models differ 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; i.e. the related approaches are class-based, whereas our approach is object-based. One consequence of this is that we can model classless objects, whereas the other approaches cannot. This opens the possibility of considering prototype-based languages, such as Self [US87, SU95] and JavaScript [Fla06].

Aside 7.2: Within our $\mathcal{O}$-calculus classes can be modelled as objects that provide certain facilities [AC96], such as a method which creates a new object instance and a repository of methods (or functions) that can be shared with each of its created objects. Different techniques for sharing methods between objects are presented in Appendix B. Note that it is not as straightforward to model an object as a special sort of class (rather than an instance of a class).
□

Having said that our work is not class-based, it is similar to the class-based UTP work in the sense that we also use higher-order concepts to represent methods as designs, whose semantics is determined by an appropriate weakest fix-point construction. Further, like rCOS each of our object models presents a conservative extension to the unifying theory of designs [HLL06]. Here, our extensions are designs that can access the additional observational variables for modelling heap storage and the accumulator (e.g. an operand stack or result variable). Therefore, all the existing laws concerning designs still hold.

Aside 7.3: Class-based languages frequently use the class (and interface) hierarchy to form a type hierarchy. Here, the idea is that an object of a child class can be used wherever a value of a parent class is required. This provides a form of structural typing, where code reuse through inheritance implies sub typing. This is not always what is wanted.

Object-based languages can also be typed (and even statically typed [AC96]). Here, the typing mechanisms tend to be based on the signature of the object (i.e. the collection of members that it contains and their signatures). Our $\mathcal{O}$-calculus is not typed, as the standard UTP theory of designs that we are embedding it into is not typed. The benefits of either object or class typing could be added in the future, along with their associated constraints on what can be legitimately expressed.
□

Our model of objects is not designed to support a refinement calculus, though it has the standard UTP notion of refinement, namely logical implication. Instead, we were focusing on the task of accurately capturing the operational semantics of $\mathcal{O}$-calculus in the UTP. Having done this, a future step is to consider how the UTP models can be adapted to support a refinement calculus (as discussed in Section 7.4).

7.3 Abstract Location Model

Within this section we relate our model of locations to those of: the trace-based models inspired by Hoare and He [HH99]; the UTP pointer models based on the notions of an *entity group*

[CHW06] and *automata* [HCW08]; and the rCOS object references [HLL06].

Trace-based modelling: Our model of locations was inspired by Hoare and He's trace-based model of pointers [HH99]. It introduces the notion of a containable location. This significantly complicates the — already non-trivial — notion of assignment, which in [HH99] is defined in terms of *swinging* the pointer of the assigned location to its new contents. In our model, several contained pointers can be swung at once by an assignment operation, as the contents of:

- a containable location are duplicated on assignment;
- a shareable location are referenced (shared) on assignment;
- a location can include many shareable and containable locations.

The benefit of this extra complexity is that our location model enables the atomicity of assignment to be directly specified (or supported). For example, the copying of a `struct` in C^{++} or a `record` in Pascal can be captured.

Schieder has also adapted Hoare and He's work on trace-based pointers to provide a weakest precondition semantics for pointers [Sch04]. Here, the object maps have been totalised in order to avoid undefinedness; this leads to the null pointer being modelled by a node that has outbound edges (all of which point to itself). However, like [HH99] it does not support the notion of a containable location.

Entity group modelling: Cavalcanti, Harwood, and Woodcock have an *entity group* [PO04] inspired model of pointers and records [CHW06]. Here, an *entity group* contains the set (equivalence class) of path names that can be used to access the same value, where a path name is either:

1. a *simple name* of a UTP user variable; or
2. a *rooted field name*, which is a simple name extended by a dot-separated sequence of record field labels.

This notion of a path is similar to the one we use, in the sense that both use a sequence of labels to define a route from a given starting point to a location. The main difference is that every location in [CHW06] is potentially shareable, whereas only some of the locations within our model are shareable. Specifically, [CHW06] does not support the notion of a containable location.

Automata modelling: Harwood, Cavalcanti, and Woodcock have an alternative UTP model of pointers, which models the heap in terms of an *automaton* [HCW08]. Their UTP model of pointers uses three observational variables to represent the automaton: one to represent the set of addresses (i.e. paths in [CHW06]) the automaton will accept; one to represent a partial mapping from *terminal* addresses to primitive values; and one to represent an equivalence relation on addresses. A feature of this model is that the value of a non-terminal address – i.e. one that does not contain a primitive value – is a partial function that maps addresses to values (that can be reached from this point). Having said this, there are no operations for assigning such values to variables.

This work also presents a theory of 'conjunctive healthiness conditions' that is used to simplify the formal reasoning. They demonstrate that this theory can be combined with the UTP theory of designs, via the existence of a Galois connection, provided that the conjunctive healthiness conditions do not mention the design's observational variables. We took a different approach, which was to adapt our location model's healthiness conditions so that they were written in terms of UTP designs (Section 6.1). Another difference is that every location in

[HCW08] is potentially shareable, whereas only some of the locations within our model are shareable. Therefore, like [CHW06], [HCW08] does not support the notion of a containable location.

rCOS object references: In rCOS [HLL06] a non-null object value is modelled as a member of an abstract set of *object identities*, which is the set of references, without the special null-reference (for the null object). The underpinning object itself is denoted by a data-structure that contains its identity (object reference), its constructed and current class-type index (class name), and a map from attributes of its constructed class-type to their values. Further, the 'dynamic' observational variable Π is used to store the set of currently allocated objects. The set of objects acts much like our simple heap model – a partial map from abstract locations to values. Here, each object within Π has a unique object identity, which corresponds to our abstract location, and the object itself can be considered the value. The main difference is that our heap can contain any type of value, whereas the rCOS equivalent (i.e. the Π set) can only contain object values.

First class pointer values: A further difference between our model and those of [HH99], [Sch04], [CHW06], [HCW08], and [HLL06] is that we have an explicit notion of a pointer value (i.e. shareable location), as represented by a path-set of the form '$Lp.\$$'. One consequence of this is that our model directly supports the notion of a *handle*, which is a pointer to a pointer, whereas the other approaches do not. Here, the second pointer value (path-set) includes a path of the form '$lp.\$.\$$'.

7.4 Future Directions

Within this section we consider three ways of progressing our work. First, we discuss how the existing models might be improved or extended to handle additional OO-features (e.g. exception handling). Second, we discuss some possible extensions to the abstract location model. Third, we discuss the possibility of adapting our models so that they can be used to support another programming language's refinement calculus.

Object modelling: An Abadi–Cardelli-style of object calculus [AC96] can be modelled in the UTP in a variety of different ways. One issue with the models we have presented is that the encoding (compilation) from the $\mathcal{O}$-calculus to the UTP is essentially one way. It would be interesting to consider whether one can provide a UTP encoding that was more bidirectional, in the sense that it is possible write a UTP program using a straightforward subset of the UTP predicate language, and have this represented as an $\mathcal{O}$-calculus program via a reversal (inverse) of the compilation function.

Another issue is that the $\mathcal{O}$-calculus model presented in this dissertation does not include several modern programming language features, such as exception handling and multi-threading (concurrency). Here, both the UTP and the object/lambda calculi have existing models of both exception handling, such as [He08, Pie02], and concurrency, such as [HH98, GH98]. A way forward would be to survey the approaches to modelling these features in both frameworks, and identify which of these models correspond (or almost correspond) to each other.

Location modelling: We have presented both a concrete and an abstract model of locations. What we have not done is to prove that the two models are consistent, so a potential future direction is to prove the consistency of these models; having done this the next step would be to prove the consistency of the $\mathcal{U}_{\text{L}}$-design against the $\mathcal{O}$-calculus, as the $\mathcal{U}_{\text{L}}$-design is underpinned by this abstract location model.

We have also not considered the issues of:

- location ownership and encapsulation (e.g. as presented in [Cla02, NCP99, Rey02]);
- location typing (e.g. augment a location with the type of its contents);
- location visibility (e.g. augment a location with read-only or scope modifiers).

Augmenting the UTP model of locations to handle any of these issues is a potential way forward.

Refinement calculus: Each of our UTP models of the $\mathcal{O}$-calculus have an implication refinement ordering. This implication ordering could be used to prove laws that define refinement relationships between $\mathcal{O}$-calculus expressions, which in turn could be used to prove refinement laws for any programming language that defines it semantics in terms of the $\mathcal{O}$-calculus. Hence, one line of future work could be to provide the $\mathcal{O}$-calculus with such a refinement calculus. Alternatively, it may be desirable to use one (or more) of our UTP object models to directly prove refinement laws for a programming language whose semantics is defined in terms of the $\mathcal{O}$-calculus. There is already a significant body of work on OO-refinement calculi; Shield's thesis [Shi04] provides a good starting point, as the first three chapters provide a survey of this area. Further, his own work provides a refinement calculus for an Abadi–Cardelli-style object calculus that does not contain references.

7.5 Reflection

We now reflect on some of the decisions we took regarding the embedding of the $\mathcal{O}$-calculus in the UTP. Within our work context the UTP had been proposed as a framework for combining the results of various industrial analysis tools. At that time object orientation had only just started to be addressed within the UTP community, and only for class-based OO-languages. This left object-based object orientation within the UTP unaddressed. Further, as class-based OO-languages were (and still are) dominating the popular languages it appeared that object-based object orientation might be overlooked by other UTP researchers. Therefore, it appeared that this was a challenging area, which was relevant to our job and had the potential to be successfully explored on a part-time basis (where it would not be feasible to compete with a full-time research group). This eventually led to the decision to identify a research issue within this area of study.

Abadi and Cardelli's work on the ς-calculus as defined in their book [AC96] had generated a significant amount of interest within the object-based research community. It was being used and adapted by several researchers to meet their own needs. Further, at its core was a small calculus whose core concept was that of an abstract representation of an object. Therefore, this calculus was adopted as the starting point for our modelling of object-orientation within the UTP.

We spent some time considering how the ς-calculus could be used, and the way in which it had been extended to directly support a range of programming issues. Part of this work is documented in Appendix B, which focuses on the modelling of classes via the sharing of methods. In hindsight, too much time was spent examining what the ς-calculus could do and support, rather than focusing on how it could be related to the UTP.

One significant concern was how to demonstrate the validity of any UTP model of objects. We already had a deterministic operational semantics for our variant of the ς-calculus (which we called the $\mathcal{O}$-calculus); it had been validated by running examples through a term-rewriting machine (which implemented the operational semantics). Therefore, we decided that the UTP semantics of objects should be consistent with this already validated operational semantics. This was one of the contributing factors that led to our decision to embed a variant of the $\mathcal{O}$-calculus within the UTP, rather than use it to inspire a UTP specific model of objects. Another

significant factor was that a model as fundamental as the ς-calculus ought to be representable in the UTP, if the UTP enabled different styles of programming to be unified (by relational predicates). In hindsight this led to the UTP being used to define an alternative denotational semantics for the $\mathcal{O}$-calculus, rather than having a model of objects that was inspired by the $\mathcal{O}$-calculus. Nevertheless, such an encoding is useful in its own right, as a denotational semantics is often useful for the proof of laws concerning its source language(s).

The precise choice of features that were included in the $\mathcal{O}$-calculus came from two conflicting objectives. First, to keep the language small, so that the proofs concerning its consistency were feasible. And second, to provide a selection of programming language operators to simplify examples and illustrate that it could be of practical use. In hindsight, it is arguable whether functions should have been included as first class values. We were able to generalise our encoding of methods in a straightforward, and possibly obvious, way to handle both function values and methods, but the cost in time and effort was significant.

7.6 Conclusions

Hoare and He's UTP [HH98, WC04] provides a rich model of programs as relational predicates. Our first contribution is to extend this work with a notion of object-based object orientation, in contrast with the existing class-based models. Some might argue that class-based object orientation dominates the mainstream OO-languages to such an extent that the alternative object-based OO-languages do not need to be addressed. However, the newer features of some of these languages, such as Java's dynamic proxies [FF04, D'A04], enable classes to be generated (compiled) and then used (loaded and linked) at run-time. These sorts of features blur the boundaries between classes and objects. Further, some of the prominent scripting languages, such as JavaScript [Fla06] are incorporating object-based object orientation. Therefore, we would argue that the UTP should support object-based OO-languages, and that our work provides a step in this direction.

Cavalcanti, Harwood, and Woodcock have provided the UTP model with a notion of pointers [CHW06, HCW08]. He, Liu, and Li have also provided a UTP model of pointers, which is limited to object references, where an object's value is its reference [HLL06]. Our second contribution is to provide an alternative model of pointers for the UTP that supports both value-based compound values (e.g. objects) and references, in contrast to existing UTP models with pointers that support only reference-based compound values. We argue that a general UTP model of pointers ought to consider both shareable and containable locations. Such models will provide support for languages like C$^{\#}$ and the $\mathcal{O}$-calculus, which have language constructs for building containable locations and handles (i.e. pointers to pointers).

Abadi and Cardelli's ς-calculi [AC96] provide a prominent formalism of object-based OO-programs, which models programs as objects. In general, we believe that the UTP ought to be able to model all significant computer programming language formalisms, in order for it to be considered a unifying theory. Hence, our third contribution is to model this Abadi–Cardelli notion of an object in the UTP, and thus demonstrate that it can unify this style of object formalism.

Finally, we observe that our work could be used to provide a denotational semantics for any programming language whose semantics was defined in terms of the $\mathcal{O}$-calculus. Here, the designer of the programming language could have the benefit of using an operational OO-semantics to define his language, and get our four corresponding denotational UTP models for 'free'. This would provide the designer with a choice of formal models for proving properties (and laws) concerning his programming language; which may well be easier in a denotational rather than operational semantic setting.

References

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, 1996.

[AL98] Martin Abadi and K. Rustan M. Leino. A logic of object-oriented programs. Research Report 161, Systems Research Center, Digital, September 1998.

[AZ96] Davide Ancona and Elena Zucca. An algebraic approach to mixins and modularity. In 5^{th} *International Conference on Algebraic and Logic Programming*, pages 179–193. Springer Verlag, 1996. 1139 in Lecture Notes in Computer Science.

[AZ98] Davide Ancona and Elena Zucca. A theory of mixin modules: Basic and derived operators. *Mathematical Structures in Computer Science*, 8(4):401–446, 1998. Cambridge University Press.

[AZ02] D. Ancona and E. Zucca. A calculus of module systems. *Journal of Functional Programming*, 11(2):91–132, March 2002.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *ECOOP 1990 – European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[BC99] Michele Bugliesi and Silvia Crafa. Object calculi for dynamic messages. In *FOOL 6 – Sixth International Workshop on the Foundations of Object-Oriented Languages*, 1999. Available via the FOOL web-site http://www.cs.hmc.edu/∼stone/FOOL/-index.html.

[BCG00] Roland Backhouse, Roy Crole, and Jeremy Gibbons, editors. *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*. Springer-Verlag, April 2000. International Summer School and Workshop, Oxford, Revised Lectures.

[BCP99] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, 1999.

[BDMN79] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula Begin*. Studentlitteratur (Lund, Sweden), Bratt Institute Fuer Neues Lerned (Goch, FRG), Chartwell-Bratt Ltd. (Kent, England), 2^{nd} edition, 1979.

[BL05] Christie Bolton and Gavin Lowe. A hierarchy of failures-based models: theory and application. *Theoretical Computer Science*, 330(3):407–438, 2005.

[BMvW00] Ralph-Johan Back, Anna Mikhajlova, and Joakim von Wright. Class refinement as semantics of correct object substitutability. Technical Report 333, Turku Centre for Computer Science, 2000.

[Bol02] Christie Bolton. *On the refinement of state-based and event-based models*. PhD thesis, Oxford University, 2002.

[BPG00] Michele Bugliesi and Santiago Pericas-Geersten. Depth subtyping and type inference for object calculi. In *FOOL 7 – Seventh International Workshop on the Foundations of Object-Oriented Languages*, 2000. Available via the FOOL web-site http://www.cs.hmc.edu/∼stone/FOOL/index.html.

[BPS99] Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *ECOOP 1999 – 13^{th} European Conference on Object-Oriented Programming*, volume 1628 of *Lecture Notes in Computer Science*, pages 43–66. Springer, 1999.

[Bra92] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.

[Bru96] Kim Bruce. Typing in object-oriented languages: Achieving expressiveness and safety. author's web site, September 1996. Report.

[Bru02] Kim B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

[BvW98] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998. Graduate Texts in Computer Science.

[Car88] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, 1988.

[Cas97] Giuseppe Castagna. *Object-Oriented Programming: A Unified Foundation*. Birkhauser Verlag AG, 1997.

[CD02] Daev Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA 2002 – Object-Oriented Programming Systems, Languages and Applications*, pages 292–310. ACM, 2002.

[CG98] Luca Cardelli and Andrew Gordon. Mobile ambients. In Maurice Nivat, editor, *FoSSaCS 1998 – First International Conference Foundations of Software Science and Computation Structure*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155, 1998.

[CHH+07] Zhenbang Chen, Abdel Hakim Hannousse, Dang Van Hung, Istvan Knoll, Xiaoshan Li, Zhiming Liu, Yang Liu, Qu Nan, Joseph C. Okika, Anders P. Ravn, Volker Stolz, Lu Yang, and Naijun Zhan. Modelling with relational calculus of object and component systems - rCOS. In Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors, *CoCoME – The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science*, pages 116–145, 2007.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[Chu85] Alonzo Church. *The Calculi of Lambda Conversion. (AM-6) (Annals of Mathematics Studies)*. Princeton University Press, Princeton, NJ, USA, 1985.

[CHW06] 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*, number 4010 in Lecture Notes in Computer Science. Springer-Verlag, 2006.

[Cla01] David Clarke. An object calculus with ownership and containment (extended abstract). In *FOOL 8 – Eighth International Workshop on the Foundations of Object-Oriented Languages*, 2001. Available via the FOOL web-site http://-www.cs.hmc.edu/∼stone/FOOL/index.html.

[Cla02] Daev Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, October 2002.

[CM92] R.C.H. Connor and R. Morrison. "Subtyping without tears". In *15th Australian Computer Science Conference*, pages 209–225, Hobart, Tasmania, 1992.

[CN00] Ana Cavalcanti and David A. Naumann. Simulation and class refinement for Java. In S. Drossopoulou, S. Eisenbach, B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *ECOOP Workshop on Formal Techniques for Java Programs*. Technical Report 269, Fernuniversität Hagen, 2000. Available from http://www.informatik.fernuni-hagen.de/pi5/publications.html.

[CN02] Ana Cavalcanti and David A. Naumann. Forward simulation for data refinement of classes. In L. Eriksson and P. A. Lindsay, editors, *FME 2002 – Formal Methods Europe*, volume 2391 of *Lecture Notes in Computer Science*, pages 471–490. Springer-Verlag, 2002.

[Coo89] William R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, May 1989. Tech Report CS-89-33.

[CSW03] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A unified language of classes and processes. In *St Eve: State-Oriented vs. Event-Oriented Thinking in Requirements Analysis, Formal Specification and Software Engineering*, Satellite Workshop at FM'03, 2003.

[CSW05] 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.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[D'A04] Lara D'Abreo. Java dynamic proxies: One step from aspect-oriented programming. Technical Report 21463, DevX, July 2004. http://www.devx.com/Java/-Article/21463.

[DG87] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An overview. In *ECOOP 1987 - European Conference for Object-Oriented Programming*, pages 151–170. Springer-Verlag, June 1987. LNCS, vol. 276.

[DP02] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2nd edition, 2002.

[DS90] E.W. Dijkstra and C.S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[FF98] Robert Bruce Findler and Mathew Flatt. Modular object-oriented programming with units and mixins. In 3rd *ACM International Conference on Functional Programming*, pages 94–104. ACM, September 1998.

[FF04] Ira R. Forman and Nate Forman. *Java Reflection in Action*. Manning Publications Co., 2004.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *POPL 1998 – 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183. ACM, 1998.

[Fla06] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, 5th edition, August 2006.

[FR00] Kathleen Fisher and John Reppy. Inheritance-based subtyping. In *FOOL 7 – Seventh International Workshop on the Foundations of Object-Oriented Languages*, 2000. Available via the FOOL web-site http://www.cs.hmc.edu/∼stone/FOOL/-index.html.

[Fse05] Formal Systems (Europe) Ltd. *Failures-Divergence and Refinement: FDR2 User Manual*, 6 edition, June 2005. Available from www.fsel.com/fdr_2_manual.html.

[GH98] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction types. *Electronic Notes in Theortical Computer Science*, 16(3):248–264, 1998.

[GHJV94] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[GHL99] Andrew D. Gordon, Paul D. Hankin, and Søren B. Lassen. Compilation and equivalence of imperative objects. *Journal of Functional Programming*, 9(4):373–426, 1999.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *Java Language Specification*. The Java Series. Addison Wesley, August 1996.

[GM94] Carl A. Gunter and John C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. MIT Press, 1994.

[Gol03] Michael Goldsmith. Personal correspondance concerning the semantics of CSP and its application to the development of FDR, April 2003.

[GR89] Adele Goldberg and David Robson. *Smalltalk 80: The Language*. Computer Science. Addison-Wesley, 1989.

[GTWW77] J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, January 1977.

[HCW08] Will Harwood, Ana Cavalcanti, and Jim Woodcock. A theory of pointers for the UTP. In John S. Fitzgerald, Anne Elisabeth Haxthausen, and Hüsnü Yenigün, editors, *ICTAC 2008 – 5th International Colloquium on Theoretical Aspects of Computing*, volume 5160 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2008.

[He08] Jifeng He. Transaction Calculus. In Andrew Butterfield, editor, *UTP 2008 – Second International Unifying Theories of Programming Symposium*, 2008. To appear in volume 5713 of Springer-Verlag's Lecture Notes in Computer Science.

[Heh93] 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.

[HH98] C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Computer Science. Prentice Hall, 1998.

[HH99] C.A.R. Hoare and Jifeng He. A trace model for pointers and objects. In *ECOOP 1999 – 13th European Conference on Object-Oriented Programming*, pages 1–17, 1999.

[HHJT98] Ulrich Hensel, Marieke Huisman, Bart Jacobs, and Hendrik Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In Chris Hankin, editor, *European Symposium on Programming*, pages 105–121. LNCS 1381, 1998.

[HL02] Tom Hirschowitz and Xavier Leroy. Mixin modules in a call-by-value setting. In Daniel Le Métayer, editor, *European Symposium on Programming*, volume 2305 of *Lecture Notes in Computer Science*, pages 6–20. Springer, 2002.

[HLL02a] Jifeng He, Zhiming Liu, and Xiaoshan Li. A relational model for specification of object-oriented systems. Technical Report 262, UNU/IIST, P.O. Box 3058, Macau, October 2002.

[HLL02b] Jifeng He, Zhiming Liu, and Xiaoshan Li. Towards a refinement calculus for object systems. In *ICCI 2002 – 1st IEEE International Conference on Cognitive Informatics*, pages 69–76. IEEE Computer Society, 2002.

[HLL06] Jifeng He, Xiaoshan Li, and Zhiming Liu. rcos: A refinement calculus of object systems. *Theoretical Computer Science*, 365(1-2):109–142, 2006.

[HLXS04] Jifeng He, Zhiming Liu, Li Xiaoshan, and Qin Shengchao. A relational model for object-oriented designs. In Wei-Ngan Chin, editor, *APLAS 2004 – Second Asian Symposium on Programming Languages and Systems*, volume 3302 of *Lecture Notes in Computer Science*, pages 415–436. Springer, 2004.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Computer Science. Prentice Hall, 1985.

[Hoa99] C.A.R. Hoare. A theory of programming: Denotational, algebraic and operational semantics. Technical report (in proceedings), Microsoft Research, November 1999.

[Hog91] John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA 1991 – 6th Conference on Object Oriented Programming Systems Languages and Applications*, volume 26 of *SIGPLAN Notices*, pages 271–285, 1991.

[ISO02] ISO. *Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics*, 2002. ISO/IEC 13568.

[Jac96] Bart Jacobs. Objects and classes, co-algebraically. In B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, editors, *Object-Orientation with Parallelism and Persistence*, chapter 6, pages 83–103. Kluwer Academic Publishers, 1996.

[Jac98] Bart Jacobs. Coalgebraic reasoning about classes in object-oriented languages. In Horst Reichel Bart Jacobs, Larry Moss and Jan Rutten, editors, *International Workshop on Coalgebraic Methods in Computer Science*, volume 11 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.

[Jon93] C. B. Jones. Process-algebraic foundations for an object-based design notation. Technical Report UMCS-93-10-1, University of Manchester, October 1993.

[Kas04] Ioannis T. Kassios. Objects as predicates. Technical report, Computer Systems Research Group, University of Toronto, 2004.

[Kas06] Ioannis T. Kassios. *A Theory of Object Oriented Refinement.* PhD thesis, University of Toronto, 2006.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 2^{nd} edition, 1988.

[LA03] Mark Lutz and David Ascher. *Learning Python.* O'Reilly & Associates, 2^{nd} edition, December 2003.

[LBE00] K. Lano, J. Bicarregui, and A. Evans. Structured axiomatic semantics for UML models. In *Rigorous Object-Oriented Methods*, Workshops in Computing. BCS, 2000.

[Lem00] Lemma 1 Ltd. *ProofPower Tutorial Manual*, 2000. Available from www.lemma-one.com/ProofPower/doc/doc.html.

[Lew95] Simon Lewis. *The Art and Science of Smalltalk.* Prentice-Hall, 1995.

[Lib05] Jesse Liberty. *Programming C#.* O'Reilly Media, 4^{th} edition, 2005.

[Lum99] Mark Lumpe. *A Pi-Calculus Based Approach for Sotware Composition.* PhD thesis, Universitat Bern, 1999.

[LW01] Barbara H. Liskov and Jeannette M. Wing. Behavioral subtyping using invariants and constraints. In Howard Bowman and John Derrick, editors, *FMDP 2001 – Formal Methods for Distributed Processing, an Object Oriented Approach*, pages 254–280. Cambridge University Press, 2001. Also available as a Carnegie Mellon University technical report CMU-CS-99-156.

[Mer00] Massimo Merro. *Locality in the Pi calculus and applications to distributed objects.* PhD thesis, l'Ecole des Mines de Paris, 2000.

[Mey96] Bertrand Meyer. The many faces of inheritance: a taxonomy of taxonomy. *Computer*, 29(5):105–108, May 1996.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2^{nd} edition, 1997.

[Mil80] Robin Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.

[Mor94] Carroll Morgan. *Programming from Specifications.* Computer Science. Prentice Hall, 2^{nd} edition, 1994.

[MT92] K. Meinke and J.V. Tucker. Universal algebra. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I: Mathematical Structures, pages 189–411. Oxford University Press, 1992.

[NCP99] James Noble, Daev Clarke, and John Potter. Object ownership for dynamic alias protection. In *TOOLS 1999 – 32nd International Conference on Technology of Object-Oriented Languages and Systems*, pages 176–187. IEEE Computer Society, 1999.

[Nor92] Else Nordhagen. Pi-calculus semantics of a Smalltalk-like language. In *Norsk Informatikk Konferanse*, 1992. Available from http://folk.uio.no/lc/lit-LC.html.

[Nor98] Else K. Nordhagen. *A Computational Framework for Verifying Object Component Substitutability*. PhD thesis, University of Oslo, November 1998.

[NVP98] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP 1998 – 12th European Conference on Object Oriented Programming*, pages 158–185, 1998. volume 1445 of Lecture Notes in Computer Science.

[OCRZ03] Martin Odersky, Vincent Cremet, Christine Rockl, and Matthias Zenger. A nominal theory of objects with dependent types. In *ECOOP 2003 – 17^{th} European Conference on Object Oriented Programming*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224, 2003.

[OCW07] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, 21(1):3–32, 2007.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A comprehensive step-by-step guide*. Artima, 2008.

[Par05] Matthew Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.

[PB08] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In George C. Necula and Philip Wadler, editors, *POPL 2008 – 35^{th} ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 75–86. ACM, 2008.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PNCB06] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for Generic Java. In Peri L. Tarr and William R. Cook, editors, *Proceedings of the 21^{st} Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 311–324. ACM, October 2006.

[PO04] R.F. Paige and J.S. Ostroff. ERC: an object-oriented refinement calculus for Eiffel. *Formal Aspects of Computing*, 16(1):51–79, April 2004.

[Pol01] Erik Poll. A coalgebraic semantics of subtyping. *Theoretical Informatics and Applications*, 35(1):61–82, 2001.

[PT97] Benjamin Pierce and David Turner. Pict: A programming language based on the pi-calculus. Technical report, Indiana University, 1997.

[QDC03] Shengchao Qin, Jin Song Dong, and Wei-Ngan Chin. A semantic foundation for TCOZ in unifying theories of programming. In *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 321–340. Springer Berlin / Heidelberg, 2003.

[Qui40] Willard Van Orman Quine. *Mathematical Logic*. Harvard University Press, first edition, 1940. Revised edtion, first printed 1951, corrected inconssitencies in the axioms of class theory.

[Ree00] Dafydd L. L. Rees. *An Algebraic Theory of Software Interfaces*. PhD thesis, Swansea University, September 2000. Revised March 2001.

[Rei95] Horst Reichel. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science*, 5(2):129–152, 1995.

[Rey74] John C. Reynolds. Towards a theory of type structure. In *Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[Rey75] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schuman, editor, *New Directions in Algorithmic Languages*, pages 157–186, 1975. Reprinted in: David Gries, editor, *Programming Methodology, A Collection of Papers by Members of IFIP Working Group 2.3*, 1978, Springer-Verlag, pages 309-317; and also in Carl A. Gunter and John C. Mitchel, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 13-24, MIT Press, 1994.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency*. Computer Science. Prentice Hall, 1998.

[SB98] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In *12th European Conference on Object-Oriented Programming (ECOOP 98)*, number 1445 in Lecture Notes in Computer Science. Springer, 1998.

[SB02] Yannis Smaragdakis and Don Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, April 2002.

[Sca98] Bryan Scattergood. *The Semantics and Implementation of Machine-Readable CSP*. PhD thesis, University of Oxford, 1998.

[Sch04] Birgit Schieder. Pointer theory and weakest preconditions without addresses and heap. In Dexter Kozen and Carron Shankland, editors, *MPC 2004 – 7th International Conference on the Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 357–380. Springer, 2004.

[SCS06] T. Santos, A. L. C. Cavalcanti, and A. C. A. Sampaio. Object-Orientation in the UTP. In S. Dunne and B. Stoddart, editors, *UTP 2006 – First International Symposium on Unifying Theories of Programming*, volume 4010 of *LNCS*, pages 20–38. Springer-Verlag, 2006.

[Sek96] E. Sekerinski. A type-theoretic basis for an object-oriented refinement calculus. In S. J. Goldack and S. J. H. Kent (Eds.), editors, *Formal Methods and Object Technology*. Springer-Verlag, 1996.

[SG07] Michael Anthony Smith and Jeremy Gibbons. Unifying Theories of Objects. In Jim Davies and Jeremy Gibbons, editors, *IFM 2007 – Integrated Formal Methods Conference*, volume 4591 of *Lecture Notes in Computer Science*, pages 599–618. Springer-Verlag, July 2007.

[SG08] Michael Anthony Smith and Jeremy Gibbons. Unifying Theories of Locations. In Andrew Butterfield, editor, *UTP 2008 – Second International Unifying Theories of Programming Symposium*, 2008. To appear in volume 5713 of Springer-Verlag's Lecture Notes in Computer Science.

[Shi04] Jamie Shield. *Towards an Object-Oriented Refinement Calculus*. Doctor of philosophy, University of Queensland, 2004.

[Sim95] Anthony James Howard Simons. *A Language with Class: The Theory of Classification Exemplified in an Object-Oriented Programming Language*. PhD thesis, University of Sheffield, December 1995.

[Smi95] Graeme Smith. A logic for Object-Z (additional rules). Technical report 95-26, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane 4072. Australia, September 1995.

[Smi99] Michael Anthony Smith. Type-based assessment of Java's bytecode. Master's thesis, Oxford University, February 1999.

[SP94] Martin Steffen and Benjamin Pierce. Higher-order subtyping: Revised version with fun subtyping. Technical Report ECS-LFCS-94-280, University of Edinburgh, February 1994.

[Spi92] Mike Spivey. *The Z Notation: A Reference Manual*. Computer Science. Prentice Hall, 2^{nd} edition, 1992.

[Str91] Bjarne Stroustrup. *The C^{++} Programing language*. Addison Wesley, 2^{nd} edition, August 1991.

[SU95] Randall B. Smith and David Ungar. Programming as an experience: The inspiration for Self. In *ECOOP 1995 – 9^{th} European Conference on Object-Oriented Programming*, volume 952 of *Lecture Notes in Computer Science*, pages 303–330, 1995.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[Tur95] David Turner. *The Polymorphic Pi-calculus: Theory and Implementation*. PhD thesis, University of Edinburgh, 1995.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA 1987 – Object-Oriented Programming Proceedings of the 2^{nd} Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 227–241, 1987. Published as SIGPLAN Notices 22(12), December, 1987. Also published in Lisp and Symbolic Computation 4(3), Kluwer Academic Publishers, June, 1991.

[WC02] Jim Woodcock and Ana Cavalcanti. The semantics of Circus. In *2^{nd} Z and B Conference: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–208. Springer-Verlag, 2002.

[WC04] 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.

[WD96] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement and Proof*. Computer Science. Prentice Hall, 1996.

[Wei01] Zhuangjian Wei. *A framework for studying the semantics of object-oriented programs*. PhD thesis, Temple University, 2001.

[WO99] Jeannette M. Wing and John Ockerbloom. Respectful type converters for mutable types. Technical Report CMU-CS-99-142, Carnegie Mellon University, Pittsburgh PA 15213-389, June 1999.

Appendix A

Concepts and Terminology

This appendix provides a slightly more detailed presentation of our understanding of object-oriented (OO) concepts, semantics, and aliasing. The concept map in Figure A.1 illustrates our view of how the typical OO-concepts are related. Essentially, we group the OO-concepts into four groups that contain illustrative sub-concepts and have links to their nearest *neighbouring* concept groups.

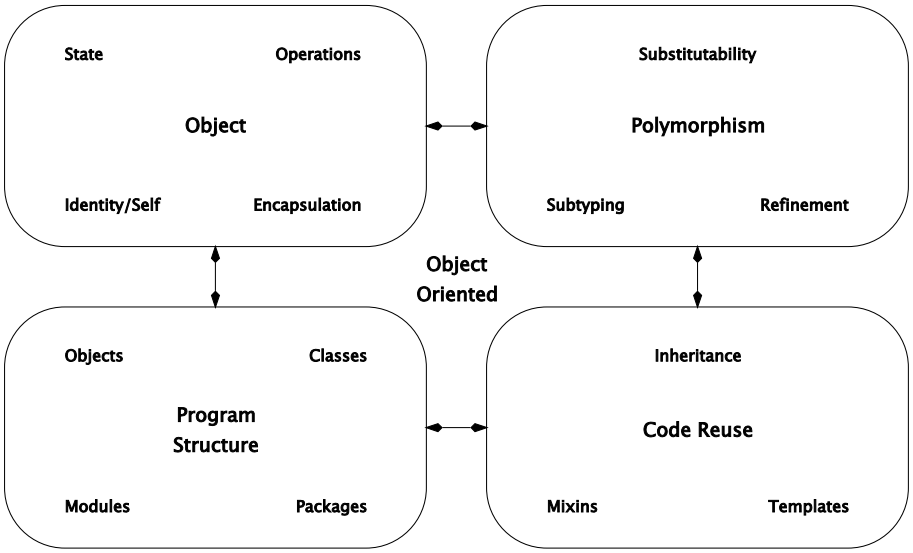


Figure A.1: Object-oriented concept map

These OO-concepts fit into a semantic framework, which may have a variety of characteristics, such as the *single threaded* assumption. In order to discuss these semantic issues, it is useful to organise them as illustrated in Figure A.2, where some of the entries in the diagram contain a brief context setting comment as represented by *slanted* text. This time, the ring presentation is more for convenience of explanation rather than demonstrating a genuine closeness of conceptual relationship. Having said this, the semantics of *concurrency* builds on that of the *state values* and *observable behaviour*.

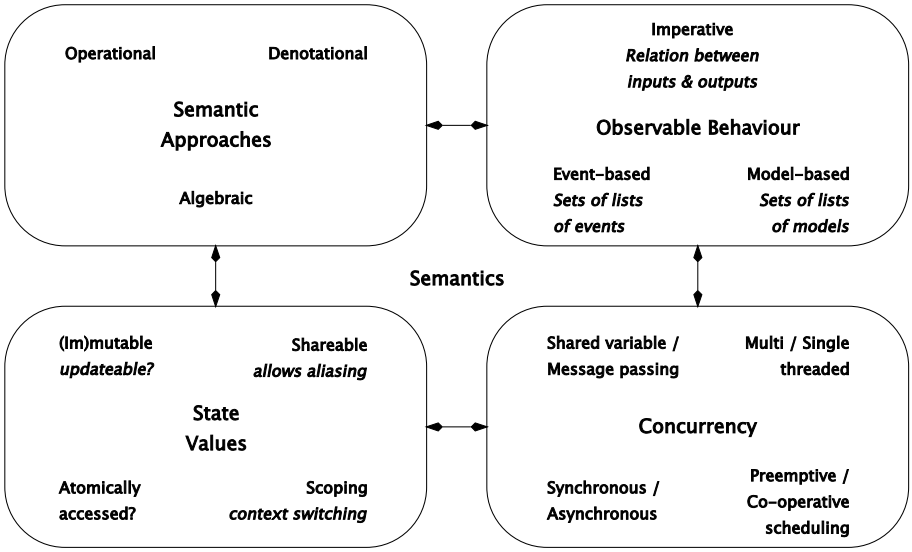


Figure A.2: Semantic concept map

The remainder of this appendix explores these ideas in more detail, and also briefly discusses aliasing.

A.1 Semantics

A language's *semantics* provides its constructs (*terms*), both individually and collectively, with a meaning; this can be anything from an informal description to a fully formal mathematical definition.

Example A.1: The JVM (Java virtual machine) has a semi-formal definition, where an informal description is augmented with some formal notation. Here, a simple list-based model of a method's operand stack is used to aid in the clarity of defining the effect of executing a bytecode instruction.
□

Frequently, the semantics of a language is given in terms of another language or model that is already understood, such as the untyped lambda calculus (λ-calculus). However, this need not be the case, as it is sometimes appropriate to describe the semantics of a language in terms of an abstract mathematical model. Such models can then be validated, experimentally, against the world that they are supposed to represent.

Example A.2: The semantics of the λ-calculus can be defined in terms of a set of operational rewrite (*reduction*) rules [Pie02]. This is augmented by what is known as an *evaluation scheme* which states when each rule can be applied.
□

Each of the concept-groups presented in Figure A.2 are now discusses in more detail.

A.1.1 Semantic approaches

There are three basic approaches to the formalisation of semantics [Pie02, Hoa99, Ros98]:

Operational semantics – Here the semantics is given in terms of how each operation affects the state of the system, which is sometimes referred to as the system's environment. Two common approaches are the *virtual machine*[1] and *reduction-rule* approaches. The first of

[1] Virtual machines are sometimes referred to as abstract (state) machines and vice-versa.

these approaches models the environment (or state) as a *virtual machine* and defines each operation as a partial mapping from current environment to the next (possibly updated) environment.[2] The second approach models the state of the system as a single – often compound – term, and its operations are defined in terms of a collection of *reduction rules* along with an evaluation scheme for indicating when each rule can be applied.

Denotational semantics – Here the semantics of the source language is given in terms of existing semantic models (domains), which may be captured by a target language. Hence, compilers essentially provide a denotational semantics for their source languages, though it may not be what is expected (or wanted). This is why denotational semantics are often targeted at well understood abstract machines or calculi, such as the λ-calculus. However, in general any compatible formal collection of semantic domains may be used. Note that the chosen target domains may immediately indicate whether certain source language properties are (or are not) available.

Algebraic semantics – Here the semantics is given by a collection of *laws* (equations) that specify when two constructs in the language are equivalent. Such semantics are very useful in the high level transformation of a design, whether this be for intuitive understanding, efficiency, modularisation, etc.

Before continuing, it is worth considering the following example, which illustrates the distinction between semantic models and approaches.

Example A.3: The CSP (Communicating Sequential Processes) language [Hoa85, Ros98], has three standard semantic interpretations (models), namely the *trace*, the *stable-failures*, and the *failure-divergence* models, which represent different granularities of observable behaviour. These models form a partial order, where valid refinements[3] in either the stable-failures or failure-divergence models are also valid refinements of the traces model.

The CSP *failure-divergence* model has three semantically equivalent presentations, which use the operational, denotational, and algebraic approaches respectively. Note that being able to prove (demonstrate) results in one approach and apply them in another has been invaluable in the efficient construction of the CSP FDR (Failures-Divergence and Refinement) model checking tool [Fse05, Sca98, Ros98, Gol03].
□

A.1.2 State values

Main stream OO-programming languages, such as Smalltalk [GR89], Eiffel [Mey97], Python [LA03], C^{++}[Str91], C$^{\#}$[Lib05], and Java [GJS96], are essentially defined in terms of, what is commonly known as, a *reference semantics*; that is a semantics which is based on a pair of mappings, one from labels to locations, and the other from locations to values. This differs from several formal treatments of OO, such as Abadi and Cardelli's untyped object calculus (ς-calculus)[AC96], System-F (a variant of the λ-calculus) [Pie02], and various object refinement calculi [CN02, HLL02b, Sek96], which are defined in terms of a *value semantics*; that is a single map from labels to values.

The semantic gap between the mainstream OO-programming languages and the value-based formal models is problematic, as value-semantics essentially prevents the sharing of an object, such as used in a typical implementation of a doubly-linked list.[4] Having said this, Chapter

[2] Partial mappings are important, as they define which operations are available only in a given environmental state.

[3] The notion of refinement in this case is essentially reverse containment; that is the behaviours of the implementation are contained within the behaviours of the specification.

[4] Suppose $A - B - C$ represents a three element doubly linked list, then both A and C share object B.

13 of [Pie02] presents a mechanism for adding references to the simply typed λ-calculus ($\lambda_{\rightarrow}$-calculus) with *Unit* (a type containing a single value *unit*). In principle, similar extensions can be made to: other varieties of the λ-calculus, such as System-F; and various object calculi, such as the ς-calculus. An example of an object calculus with this type of extension is given in Section 2.2.

Clarifying Terms: Unfortunately, the term *reference semantics* is in itself ambiguous, as it can either be interpreted as a semantics of references, or a base-line semantics for discussing other semantics. Therefore, within this document the term *location semantics* will be used instead, as this is less likely to cause confusion. It also pattern matches into the usual abstract way of representing references that cannot be arbitrarily calculated via pointer arithmetic [Pie02]. Further, the term *value semantics*, could be similarly replaced by the term *copy semantics*, to more accurately capture its nature; however, it is felt that the term *value semantics* is sufficiently clear, and provides a more natural conceptual correspondence to location semantics, so it is retained.

A.1.3 Observable behaviour

The concept of observable behaviour is key to the semantics of software, as two software entities are considered equivalent if in *all contexts* no distinction can be made between their patterns of observable behaviour. Ironically, the notion of *all contexts* depends upon its context, i.e. its underlying modelling assumptions.

Example A.4: The assumptions underpinning the CSP modelling language include the constraint that observable events are atomic and instantaneous. These assumptions (constraints) ensure that it is impossible to have two distinct observable events happening at the same time. Hence, the notion of *all contexts* that is associated with the CSP language does not include the possibility of either independent simultaneous events, or a single event happening over a period of time. Having said this, it is straightforward to model events with durations, by having named start and stop events. In the author's experience the highlighted CSP assumptions have not caused significant problems when modelling real systems.
□

All programming languages have an associated model of expected behaviour, which is broken down, at least conceptually, into a collection of related sub-models, such as the arithmetic model, the (sub)routine invocation model, the memory model, and the exception handling model. These sub-models describe (or define) various aspects of the programming language at varying degrees of depth and completeness. In particular, some aspects of a programming language may be deliberately undefined, in order to enable a variety of context dependent (e.g. operating system specific) interpretations.

Example A.5: In the 'C' programming language [KR88] the size of both the integer and pointer types may vary, where the pointer type's size is set to reflect that of the architecture of the target computer's Central Processing Unit (CPU).
□

What impact does such undefinedness have on the observable behaviour of a program? At best, it has the potential of adding well defined context dependent aspects to the values and types of behaviour that can be observed. At worst, it may introduce subtle non-deterministic ambiguity into the behaviour of a language, whose affects may be very hard to predict and reason about. Having said this, it does allow for optimisation of code for a particular computer's hardware architecture and operating systems infrastructure.

Some of the undefinedness in a programming language can be mitigated by adopting programming practices that are designed to ensure that however a semantic ambiguity is resolved, the resulting program semantics is stable; that is given a suitably chosen granularity of observation, all potential implementations of the program are still guaranteed to meet their specification. For example, one could ensure that regardless of whether preemptive or cooperative multi-threading is used by the implementation, the program would still not deadlock.

The notion of granularity of observation is important. If it is too fine (detailed) then it distinguishes software components that are actually equivalent to each other in all relevant contexts; this is analogous to a specification stating 'how something is to be implemented', rather than 'what is to be implemented'. Conversely, if the granularity of observation is too coarse, then software components that should be distinguished as having significantly different behaviours are not distinguished; this is analogous to an ambiguous specification. The correct granularity of observation can be dependent on the property that is being checked.

Example A.6: Being able to observe the internal state of an encapsulated entity, such as the private members (internal state and actions) of an object, when considering how that entity interacts with its peers, provides too fine a level of granularity. In general, it prevents the replacement of the entity with an entity that has the same behaviour from the peers' perspective but a different internal behaviour (e.g. different sequence of internal actions). However, being able to observe the internal behaviour of an entity may be useful, when the implementation of that entity is updated, so as to treat that entity's internal behaviour as the new system context. Alternatively, it may be useful to observe the internal state of an entity for checking properties such as type-correctness of the program; though in this case, it could be argued that such type correctness checks should be done in a layered manner, thus in any individual layer there is no need to check the internal type correctness of its components.
□

In summary, establishing the correct level of visibility of observable behaviour is critical when determining the semantic effects of a program. In particular, it governs when one software entity can be replaced by another software entity without changing the semantics of the program as a whole.

A.1.4 Concurrency

Modern operating systems, such as Microsoft Windows and Linux, provide their users with an inherently concurrent environment; in the sense that it is possible for the computer to be performing several independent jobs for their user at the *same time*. For example, a user can be playing a CD, whilst browsing the Internet and background printing some interesting articles. From the user's perspective, all these events appear to be happening simultaneously (e.g. at the *same time*), but in practise these jobs are likely to be sharing the same CPU core. Hence, some form of job management is being provided – behind the scenes – by the operating system[5] in order to maintain this illusion. The clever, or tricky, bit is ensuring that the independent jobs do not adversely interfere with each other.

In general, avoiding adverse interference between various processes (jobs), or sub-processes, is a non-trivial task, particularly when these processes are supposed to interact with each other. This leads to the study of the semantics of concurrency; with such semantics it is possible to demonstrate that an application's design and implementation will be well behaved. For example, the application is both deadlock and livelock free; that is all processes are eventually able to make progress and no process can infinitely loop without user (environmental) intervention respectively.

[5] The operating system may be supported by various run-time environments, such as a Java virtual machine.

Essentially, the semantics behind concurrency can be summarised by the semantics of interaction; that is when and how are processes allowed to interact. This suggests that a natural model of concurrent behaviour is a monitoring of these interactions or communication events. It is possible that such observations led to the design of process algebraic languages, such as CSP [Hoa85, Ros98] and CCS (Calculus of Communicating Systems) [Mil80]. More recently these ideas have been enhanced to include more modern notions of portability, or reconfigurability, of the channels of communication, such as in the pi-calculus (π-calculus) [Mil99, SW01] and the ambient calculus [CG98]. The latter ideas have been suggested as a natural basis for modelling OO-languages [SW01, PT97, Lum99, Mer00, Nor92, Nor98]. However, we believe that the notions of concurrency and object orientation, though related, are sufficiently independent to be studied in their own rights, and then combined. In particular, studying them in a concurrent context may mean that the notion of an object is not as general as it could be.

A.2 Object-Oriented Infrastructure

For convenience we reprint the OO concept map in Figure A.3. The concepts in this map are now discussed in the following four subsections on objects, polymorphism, program structure and code reuse.

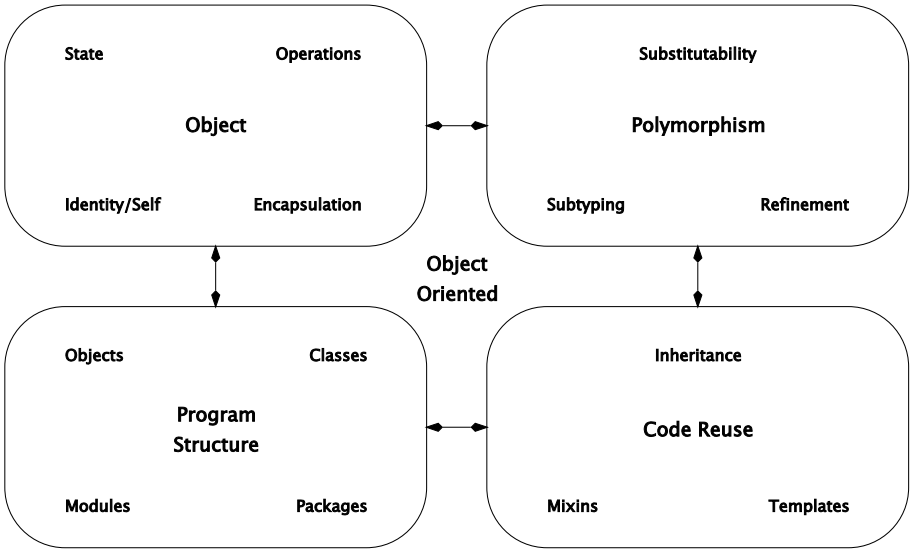


Figure A.3: Object-oriented concept map (reprint)

A.2.1 Objects

For the purposes of this dissertation the key concepts that an object possesses are:

State – An object contains data, which is typically represented by a collection of named fields, much like a record (or form). This data may be hidden from external view by the object's, possibly context dependent, *encapsulation* mechanism(s). For example, in Java [GJS96] an object's field could be marked as package visible; that is visible to operations (methods) that are defined in the same package as the field.

Operations – An object contains named operations for accessing and modifying its state. These operations are typically referred to as the methods of an object. Similarly to fields, methods can be hidden from external view, by the object's, possibly context dependent, *encapsulation* mechanism(s).

Note that in some OO-languages methods and fields are treated uniformly, as updateable members. Conceptually, this can be presented from two distinct points of view. First, a method can be treated as a special sort of data value, which defines an operation; this is sometimes referred to as *higher-order* programming. Second, fields can be treated as constant operations which return the field's value; thus writing to a field is represented by the updating (replacement) of the constant operation. This latter view is adopted in this dissertation.

Identity/Self – An object has a unique identity that can be used to distinguish it from other objects that otherwise share the same state (i.e. have the same value). In practise, this is often simply the value of the location at which the object is stored in the *heap* (i.e. the memory address). However several significant formal models of objects ignore this point. For example the simplest of the Abadi–Cardelli ς-calculi does not have this property [AC96], as two objects that differ only in the *self* variable are considered to be equivalent, in all contexts. This is an example, of a value-based semantics of objects, where objects are: passed by value, updated by replacement, and considered equal whenever they have the *same* state. In general, the unique identity of an object is less important in a value-based semantics as it is impossible for two variables to share the same object, the best they can do is have a duplicate (copy) of each other's object. Having said this, the notion of *self* (limited ownership) is still important and does not require unique identity, and this is supported by the Abadi–Cardelli ς-calculi.

Encapsulation – An object provides a container that ought to be used to group related data and operations. Such a grouping is sometimes considered to be a form of encapsulation. However, object encapsulation usually refers to a combination of hiding and access control, where strictly internal members are completely hidden and visible members are given context dependent access control markings, such as *read-only* and *write-once*. The intention is that an object can only be accessed via operations on its published interfaces.

Note that the visibility of these interfaces may be context dependent (as in Java). Further, aspects of an object's internal representation may be compromised, by *reference exposure*; for example, where a reference to an internal object is passed to the object's environment. In this case, the object's environment can update the inner-object directly. At first sight this may appear to be a poor idea (as proponents of a value-based object semantics might suggest), but in practise this feature provides an efficient way of manipulating structured data. Some commentators believe that sharing objects via references (*aliasing*) is "fundamental to good object-oriented design" [Cla01]; it is certainly required for the implementation of a doubly linked list, or any other cyclic data structure. In such contexts the problem is how to bound the range and scope of object references (i.e. control reference exposure).

A.2.2 Polymorphism

Before the topic of polymorphism can be discussed, the standard notion of a *type* needs to be introduced. Essentially operations such as addition and multiplication only make sense, when applied to certain *types* of values, such as the integers, but not when applied to other types of values such as the booleans. At the lowest level a type describes the interpretation of bit patterns. For example, both the ASCII character 'A' and the 8-bit representation of the number 65 have the same bit pattern, namely 01000001. At more abstract levels of reasoning, a type can be used to characterise a class of functions or objects, by defining their interface (or signature). Further, types can be placed in a *subtype* hierarchy,[6] where "A is a subtype of B"

[6] The subtype relationship forms a pre-order over object types, i.e. it is reflexive and transitive. Note that the reason it is not necessarily a partial order, is that two distinct object types may be subtypes of each other.

informally means that a term of type A can be used anywhere a term of type B is expected; this is known as subsumption.

"The term polymorphism refers to a range of language mechanisms that allow a single part of a program to be used with different types in different contexts" [Pie02, Section 22.7]. One example, is operation (or method) overloading, which enables the same operation to be applied to different types of data. For example, a single addition or multiplication operation that can be applied to different sizes of integer. Within the OO-context, the ability to use one object in place of another in a *controlled manner* is a form of polymorphism. The precise definition of what is an adequate measure of control is dependent on the context, as discussed below.

From a compiler writer's point of view, the rules governing when one object can validly replace another, would ideally be statically determinable, by a *straightforward and efficient algorithm*. This leads to the standard type-based sanity checking algorithms, which focus on an entity's syntactic description (*signature*), rather than its *behaviour*. In this context, a type correct program is guaranteed to be able to apply each of the operations to the supplied operands, but there is no guarantee that the subtyped operators actually behave appropriately. In general this amounts to the use of subsumption.

From a program verifier's (semantic) point of view, it is not sufficient for an object to be replaced by an object with a compatible signature. What is required is that the replacing object is *semantically equivalent*. Precisely what is meant by semantically equivalent is a matter for interpretation, but essentially amounts to ensuring that the specified behaviour of the original object is implemented by the replacement object. Hence, this style of subtyping is commonly known as *behavioural subtyping*. The notions of behavioural equivalence and refinement are typical in this area of research.

From now on the notion of subtyping refers to standard subtyping, rather than behavioural subtyping, unless explicitly indicated otherwise.

In untyped languages the concept of polymorphism is not particularly useful, as there is no mechanism for classifying different families of values. Here, any value can be passed as an argument to any operation. In general, being able to statically check that every operation within a program will always receive sensible operands requires a full program analysis. For example, ensuring that the divisor of every division operation is non-zero can require a full program analysis.

Aside A.7: The concepts of polymorphism and subtyping are distinct from those of inheritance, mixins and other code reuse mechanisms. Some real OO-programming languages, such as Java, merge these concepts; in this case inheritance implies subtyping (i.e. A inherits-from $B \Rightarrow A$ subtype-of B). This strictly limits what inheritance can be *safely* used for.
□

Polymorphism and subtyping is, and has been, a popular field of academic research [CW85, CM92, SP94, Sim95, Bru96, WO99, LW01, BMvW00, BPG00, FR00, Pol01]. Pierce's text book on "Types and Programming Languages" [Pie02] summarises a significant portion of this work, though it is essentially limited to the compatibility, rather than behavioural, style of subtyping. Behavioural subtyping is normally discussed in the context of program equivalence, program refinement and module composition [Jac96, BMvW00, Pol01, CN00].

A.2.3 Program structure (composability)

Most programming languages include some *structural* mechanisms, such as modules, packages, classes, procedures, and objects, which enable the code to be organised into a collection of components and *links* between components. Such components ought to provide access to their resources via, and only via, a published (e.g. documented) interface.

Interfaces are the key to the *structural* mechanisms, as they provide the ability to describe services (or resources) at a higher level of abstraction than that of the implementing code. In

particular, they enable the hiding of implementation details, which can then be changed to improve efficiency, fault tolerance, security, etc.

The ability to update or replace an implementation of a component without performing a complete program analysis, assumes: that the published component interface accurately describes what the component provides; and that the clients of this component only use published features of the interface (i.e. the client respects the components interface).

Aside A.8: There are many techniques for demonstrating that a component conforms to its interface, such as bisimulation or refinement. Hoare and He's Unifying Theories of Programming (UTP) is constructed in terms of the logical implication ordering, which can also be used as a refinement order. Hence, refinement is the natural way of checking interface descriptions within the UTP.
□

Aside A.9: A new *sort* of object can be introduced into a multi-method program that affects the resolution of existing objects, even if no object of that *sort* is ever instantiated. Essentially, the new object definition introduces a method which *better fits* (pattern matches) an existing method call within the program, and hence is used instead of the existing method call.
□

A.2.4 Code reuse

There are several motivating reasons for supporting the ability to reuse code, both on global and local scales. On the global scale, the idea is to build a resource of library components that can be used in the construction of an application. Modern programming languages, such as Java and C$^{\#}$, come with a large library of utility routines, which provide facilities for things like network access (e.g. sockets and remote method calls), data storage (e.g. linked lists, trees, and hash tables), and user interface design (e.g. windows, buttons and text boxes). In general, however, there are some significant difficulties to overcome, such as: how to find suitable components for a given context; who owns the *rights* to those components and the limitations on their use; who is responsible for failures of a component to meet its design; etc.

On the local scale, the idea is to factor out common aspects of a design, and only write the code for implementing a common aspect once. This has the benefit of reducing the size of the source code, and aiding in the construction of a logical program design. However, there are sometimes significant run-time overheads with such a design methodology, as it may introduce some level of indirection, such as additional method calls, and method lookups. These can be worked around to some extent, by *in-lining* the code during compilation process (e.g. macros rather than procedures or methods). Further, the very mechanisms that introduce the ability to factor out common aspects of a design can complicate the formal model of a language.

Example A.10: Within the class-based OO-context a new class can typically declare that it *extends* (inherits from) a selection of existing classes (and interfaces), which are called its parent classes, or simply its parents. In such a context, the new *child* class inherits the visible members of its parent's class, which are promoted to the child class's interface. These visible members can then be augmented (and sometimes overridden) by the child class definition, in order to provide the necessary functionality.

There are many subtleties in the precise semantics of inheritance, particularly when the effects of scoping and aliasing are taken into account, see [Coo89, CW85, Cla02], [Smi99, Appendix D] and [GM94, Part 5] for more details.
□

Inheritance mechanisms vary significantly from language to language; but they essentially amount to some form of entity specialisation or extension. These ideas can be generalised by

the notion of a *mixin*, as illustrated by Bracha and Cook in [BC90] and extended by Bracha's PhD. thesis [Bra92]. Here *mixin* is informally defined to be an abstract subclass that may be used to specialise the behaviour of parent classes [BC90, Section 3.2]. In other words, a mixin can be thought of as a form of class generator function, which takes a class and generates a new class. Mixins have since, been used by other researchers in their module and inheritance systems [Sim95, AZ96, AZ98, AZ02, BPS99, FF98, FKF98, HL02, SB98, SB02, OCRZ03, OSV08].

A.3 Aliasing

Sharing (*aliasing*) objects is a natural concept in OO-software design because of the location semantics in mainstream OO-languages, such as C^{++} and Java. In practise, the notion of object sharing is linked with that of object identity. "Object identity is the foundation of object-oriented programming" [NVP98]. Once created an object retains its unique identity for duration of its existence; in other words the object's identity is immutable.

Aside A.11: The existence of such identities provides a platform, whereby the operation performed on the object contained in a typed-variable can be dependent on the run-time value of the variable's object, rather than compile- or load-time type of the variable itself. This is one of the ways in which an object is distinguished from the concept of a *record*; a record is a compound data-type that essentially maps labels, known as fields or attributes, onto values (or typed-values, for a typed-language).
□

Aliasing is a well known software engineering issue [Hog91, NVP98]. It can be usefully examined at various levels of abstraction, which may depend on *strong* assumptions about the nature of the context.

Aside A.12: The assumption that a location in memory can only be obtained via *creation* or *duplication* requires a check that no *pointer arithmetic* is used; that is memory locations are not obtained by direct *numeric address* conversion, whether this be absolute or relative to an existing memory location (i.e. an address). This assumption, though *strong*, is not sufficient to guarantee that a memory location is valid, as it is still possible to *free* the contents of that location before its last use.
□

In general, it is useful to make several *strong* assumptions about the way in which memory locations, and their contents, can be used. These assumptions then need to be checked, via appropriate analyses for the context in question, where the context includes all issues introduced by languages and *design patterns* [GHJV94] that are used.

Example A.13: Some languages, such as *C* [KR88], provide almost no memory protection. Hence, analysis concerning the *valid* use of memory is potentially very difficult, as it is hard to provide *safe* abstractions for high level reasoning. Having said this, the use of well understood *design patterns* and *safe* coding practises can significantly simplify matters.

Other languages, such as *pure* Java [GJS96][7], provide certain guarantees about the way in which memory locations and their contents can be used. In the case of *pure* Java these guarantees, ensure that memory access is always *valid*; that is the contents of an available memory location always exist and are either *type correct*, or will throw a run-time exception, such as the array storage exception[8].
□

[7] *Pure* Java refers to Java which does not embed (interface to) other languages.
[8] Java's `ArrayStorageException` is *thrown* whenever an attempt is made to store an element of an incompatible type in an array.

Aliasing can provide a means for exposing the internal representation of an object and dynamically updating it, as illustrated by the following example.

Example A.14: Consider a simple object that has a private inner-object O of type T, and a public method M that returns an object reference to an object of type T. If M returns a reference to O, it could be argued that this has broken the encapsulation of the simple object. However, it could also be argued, that the simple object's private data was actually only a reference to another external object, and thus the encapsulation was not broken. Either way, if M had returned a reference to a deep copy of object O there would not have been an issue; note that a shallow copy may expose the contained elements within object o.
□

The problem of bounding or controlling the exposure of object references is well known. A significant amount of work has been done in this area, such as [NVP98, NCP99, Cla01, Cla02, CD02].

Appendix B

Classes in the Object Calculus

A class essentially provides a means for generating objects of the same form (e.g. type). Here, objects of the same class would typically have the same members. Further, the *genuine* methods of these objects (rather than methods that are used to represent fields) typically never change, so can legitimately be shared.

This appendix presents seven different ways of sharing methods, from simple *static* copy based approaches, to *dynamic* approaches that make use of object and function references. Here, the point is to demonstrate that the flexible primitives we have defined (e.g. method update and heap access) allow us to explicitly model and compare a number of different implementation techniques for sharing of methods.

The *static* approaches to modelling the sharing of methods constrain the use of method labels. Specifically, the *label* used to identify which of an object's methods is to be invoked or update cannot be calculated. In other words, within the *static* approach labels are not considered to be values, therefore cannot be passed into or returned out of a method invocation or function call.

The *dynamic* approaches to modelling the sharing of methods allow the value of a method's label to be calculated. Having said this, such label calculations may provide no means for modifying a literal label value, which essentially limits the dynamism to a selection process between existing label values.

The remainder of this chapter is split into two sections, one describing the *static* approaches and the other the *dynamic*.

B.1 Static Approaches to Sharing Methods

B.1.1 Prototype approach

The prototype approach simply shares method definitions, by copying an existing object. This is illustrated by the following simple cell example.

Example B.1: A simple cell has a `value` field, a `get` method, and a `set` method. It also has a `new` method for generating new instances of itself; this is strictly unnecessary, but helps mirror the class-definitions that follow later. Further, the `new` operation does more than simply copy the current cell; it resets the value of the copy.

$$
\begin{array}{l}
Cell \mathrel{\widehat=} [\\
\quad \texttt{value} = InitVal, \\
\quad \texttt{new} = InitVal, \\
\quad \texttt{get} = \varsigma(s)\ GetCell, \\
\quad \texttt{set} = \varsigma(s)\ SetCell \\
]
\end{array}
$$

where:

$$InitVal \equiv \varsigma(s)\,(s.\texttt{value} \Leftarrow \varsigma(z)\,z.\texttt{value})$$

$$GetCell \equiv s.\texttt{value}$$

$$SetCell \equiv \lambda(v)\,s.\texttt{value} := v$$

Note that the definitions of the initial-value and the methods to get and set the cell value are abstracted, so that they can be used to highlight the differences between this approach and the class-based approaches that follow. Further, note that the values in the cells are initialised to a method that infinitely invokes itself. An alternative would have been to initialise to the unset value ¿. It is unclear which definition is best: the former does not require a special value; whereas the latter is simpler (at least in an untyped environment).
□

B.1.2 Naive class-based approach

One approach to modelling classes, with objects, is to have a special *class* object, which can be used to generate instance objects of that class. The naive model of a class contains a function to model each method within one of its instances, where the function's argument is set to the instance's self parameter. The class also contains a special member for generating new instance objects, denoted by `new`, which creates members for both the instance's fields and methods.

Example B.2: The naive model of a simple cell class has three members, a `new` method, a `get` field, and a `set` field. The two fields contain functions for getting and setting the `value` field of an instance object. The `new` method generates an instance object, with a circular `value` field, a `get` method, and a `set` method. Note that an instance object's `get` and `set` methods both contain a complete copy of the underlying class, along with an appropriate field selection label.

$$\begin{array}{l} Cell \mathrel{\widehat=} [\\ \quad \texttt{new} = \varsigma(c)\,[\\ \qquad \texttt{value} = InitVal, \\ \qquad \texttt{get} = \varsigma(s)\,c.\texttt{get}(s), \\ \qquad \texttt{set} = \varsigma(s)\,c.\texttt{set}(s) \\ \quad], \\ \quad \texttt{get} = \lambda(s)\,GetCell, \\ \quad \texttt{set} = \lambda(s)\,SetCell \\ ] \end{array}$$

□

In general, the instance objects created in this approach contain multiple copies of the creating class, one for each instance method. This duplication can be avoided by pre-processing the class field selection during object instance generation, as we discuss next.

B.1.3 Pre-processed class-based approach

In order to avoid unnecessary duplication of the class, its is possible to pre-process the class-field selection, as illustrated by the following example.

Example B.3: The pre-processing of the class-field selections is achieved via the use of the

let-in construct.

$$
\begin{array}{l}
\mathit{Cell} \mathrel{\widehat{=}} [\\
\quad \mathtt{new} = \varsigma(c)\ \mathsf{let}\, f_1 \mathrel{\widehat{=}} c.\mathtt{get}, f_2 \mathrel{\widehat{=}} c.\mathtt{set}\ \mathsf{in}\ [\\
\quad\quad \mathtt{value} = \mathit{InitVal}, \\
\quad\quad \mathtt{get} = \varsigma(s)\, f_1(s), \\
\quad\quad \mathtt{set} = \varsigma(s)\, f_2(s) \\
\quad], \\
\quad \mathtt{get} = \lambda(s)\ \mathit{GetCell}, \\
\quad \mathtt{set} = \lambda(s)\ \mathit{SetCell} \\
]
\end{array}
$$

□

In this approach, each instance object has its own copy of the instance methods. This duplication can be avoided via the use of references to objects or functions (or both).

B.1.4 Sharing method instances

So far the sharing of methods has been achieved via duplication. The introduction of object references allows method instances to be shared. A class is no longer represented directly by an object; instead, it is represented by a reference to a class-object. Here, each object-method selection dereferences its base class, then selects and invokes the appropriate class method, which takes the object's self parameter as its argument.

Example B.4: This variant of the simple cell is defined as follows, where r represents the unique heap location of the class, which is freshly generated on its first usage.

$$
\begin{array}{l}
\mathit{Cell} \mathrel{\widehat{=}} \mathsf{let}\ r \mathrel{\widehat{=}} \mathsf{fresh}\ \mathsf{in}\ r \mathrel{*\!=} [\\
\quad \mathtt{new} = [\\
\quad\quad \mathtt{value} = \mathit{InitVal}, \\
\quad\quad \mathtt{get} = \varsigma(s)\, {*r}.\mathtt{get}(s), \\
\quad\quad \mathtt{set} = \varsigma(s)\, {*r}.\mathtt{set}(s) \\
\quad], \\
\quad \mathtt{get} = \lambda(s)\ \mathit{GetCell}, \\
\quad \mathtt{set} = \lambda(s)\ \mathit{SetCell} \\
]
\end{array}
$$

□

One issue with this representation of a class is that its instances dereference the whole of the class on each method invocation (selection). Having done that the appropriate class method needs to be selected, and then invoked with the instance's self parameter.

B.1.5 Caching shared methods

One way of reducing the number of steps required to access a shared method is to use a cache. The idea here is that the class contains a collection of references to methods, which are then directly used by its instances.

Before looking at a full example, it is worth considering the format of caching the location of function f in a member n of an object o, where o is stored in heap location ℓ_i.

$$o \equiv [\mathtt{n} = \varsigma(c)\, \ell_i \mathrel{*\!=} c.n := \mathsf{fresh} \mathrel{*\!=} f]$$

Given that all update operations are of equal precedence and right associative, the above member definition is equivalent to:

$$o \equiv [\mathtt{n} = \varsigma(c)\, \ell_i \mathrel{*\!=} (c.n := (\mathsf{fresh} \mathrel{*\!=} f))]$$

The stepwise evaluation of $o.n$ is:

1. create a fresh (new) heap location ℓ_j;
2. store function f in location ℓ_j;
3. update member n of object o with location value ℓ_j;
4. store the updated object in location ℓ_i.

$$o.n \equiv [\mathtt{n} = \ell_j]$$

Example B.5: The caching method instances variant of the simple cell is defined as follows, where r represents the unique heap location of the class, and f_i represents a pointer to an appropriate function.

$$
\begin{array}{l}
\mathit{Cell} \mathrel{\widehat=} \mathsf{let}\ r \mathrel{\widehat=} \mathsf{fresh}\ \mathsf{in}\ r \mathrel{*=} [\\
\quad \mathtt{new} = \varsigma(c) *(\mathsf{let}\ f_1 \mathrel{\widehat=} c.\mathtt{get}\ \mathsf{in}\ \mathsf{let}\ f_2 \mathrel{\widehat=} c.\mathtt{set}\ \mathsf{in}\ r \mathrel{*=} c.\mathtt{new} := [\\
\qquad \mathtt{value} = \mathit{InitVal}, \\
\qquad \mathtt{get} = \varsigma(s) *f_1(s), \\
\qquad \mathtt{set} = \varsigma(s) *f_2(s) \\
\quad]).\mathtt{new}, \\
\quad \mathtt{get} = \varsigma(c) *(r \mathrel{*=} c.\mathtt{get} := (\mathsf{fresh} \mathrel{*=} (\lambda(s)\ \mathit{GetCell}))).\mathtt{get}, \\
\quad \mathtt{set} = \varsigma(c) *(r \mathrel{*=} c.\mathtt{set} := (\mathsf{fresh} \mathrel{*=} (\lambda(s)\ \mathit{SetCell}))).\mathtt{set} \\
]
\end{array}
$$

□

B.2 Dynamic Approaches to Sharing Methods

B.2.1 Simulating dynamic method sharing

So far all the models of sharing methods between a *class* of objects have required each object to contain an explicit entry for every shared method. One way of avoiding this duplication is to enable dynamic method resolution, where the *name* of the method is passed as an argument to the invocation process. However, as labels are not values within the object calculi presented so far, they cannot be passed as arguments to an invocation process. This leaves us with the problem of how to represent the name of a method. One technique would be to represent each method by an enumerated value, which could be modelled as either an integer or a string literal. Such enumerated names could then be used to construct a dynamic dispatch lookup table (function) that mapped the name onto the corresponding method. Here the result of a lookup is a function that expects the *self* object as its argument.

These lookup tables can then be used to set the generic apply method of an object: $\mathtt{apply} = \varsigma(s)\ \lambda(n)\ \mathit{lookup}(n)(s)$, where n represents the name of the method to be applied.

Example B.6: In the case of the simple cell, the lookup table that contains the two shared methods, get and set, can be modelled as follows:

$$
\begin{array}{l}
\mathit{GetSetLookup} \mathrel{\widehat=} \mathsf{fresh} \mathrel{*=} \lambda(n) \\
\quad \mathsf{if}\ \mathtt{n} = \texttt{"}\mathit{get}\texttt{"}\ \mathsf{then} \\
\qquad \lambda(s)\ \mathit{GetCell} \\
\quad \mathsf{else\ if}\ \mathtt{n} = \texttt{"}\mathit{set}\texttt{"}\ \mathsf{then} \\
\qquad \lambda(s)\ \mathit{SetCell} \\
\quad \mathsf{else} \\
\qquad \mathsf{let}\ \mathtt{v} = [\mathtt{l} = \varsigma(z)\ z.l].l\ \mathsf{in}\ v
\end{array}
$$

where the *GetSetLookup* table is stored in a fresh location on the heap. Note that the issue of handling invalid method names is resolved in this case by ensuring that the evaluation infinitely loops. An alternative would have been to set this to the undeclared value, so that the rule interpreter could detect the error.

It is now possible to define the simple cell object in terms of the *GetSetLookup* table, where the get and set methods are replaced by a generic apply method, which takes the names "*get*" or "*set*" as its arguments.

$$
\begin{array}{l}
Cell \mathrel{\widehat{=}} [\\
\quad \texttt{new} = [\\
\quad\quad \texttt{value} = InitValue, \\
\quad\quad \texttt{apply} = \varsigma(s)\ \lambda(n) * GetSetLookup(n)(s) \\
\quad] \\
]
\end{array}
$$

Note that the lookup function could have been defined within the simple Cell class object. There is some merit in this approach as it would co-locate data with functionality (increasing the cohesiveness of the modelling of a class).
□

The lookup scheme presented in Example B.6 does not provide a dynamic means for determining what *method names* are available. In this case, it is up to the *programmer* or *compiler* to ensure that only valid names are requested. If dynamic determination of what method names are available is required, such as for modelling some reflective language features, then this functionality could be provided by a companion *what names are defined* lookup function.

B.2.2 Directly supporting dynamic method sharing

It is also possible for dynamic method sharing to be directly supported by the object calculus. One approach would be to model an object's label as a string literal, as method names are represented by identifier strings. A side effect of this approach, is that a label's value could be calculated (generated) by the application of general string operations, such as string concatenation. An alternative approach is to classify the existing method labels as literal values, thus enabling them to be stored in an object's state and passed as the argument to a function. Semantically, this is all that is required as parameters are substituted for their values as part of the method and function invocation process. Hence there is no need to update the existing $\mathcal{O}$-calculus reduction rules.

Aside B.7: Having said that there is no need to update the existing rules, it might be worth extending them so that they treat the label arguments as evaluable expressions, rather than literal values (or identifiers that are substituted with literal values). Without the suggested rule extensions such expressions would have to be pre-evaluated, in a local definition block, before use within the context of either a method invocation or update operation.

$$\frac{\Gamma \bullet e_1 \longrightarrow \Gamma' \bullet e_1'}{\Gamma \bullet e_1.e_2 \longrightarrow \Gamma' \bullet e_1'.e_2}\ \textsc{InvM-1} \qquad \frac{\Gamma \bullet e \longrightarrow \Gamma' \bullet e'}{\Gamma \bullet o.e \longrightarrow \Gamma' \bullet o.e'}\ \textsc{InvM-2}$$

$$\frac{\Gamma \bullet e_1 \longrightarrow \Gamma' \bullet e_1'}{\Gamma \bullet e_1.e_2 \Leftarrow m \longrightarrow \Gamma' \bullet e_1'.e_2 \Leftarrow m}\ \textsc{UpdM-1} \qquad \frac{\Gamma \bullet e \longrightarrow \Gamma' \bullet e'}{\Gamma \bullet o.e \Leftarrow m \longrightarrow \Gamma' \bullet o.e' \Leftarrow m}\ \textsc{UpdM-2}$$

$$\frac{\Gamma \bullet e_1 \longrightarrow \Gamma' \bullet e_1'}{\Gamma \bullet e_1.e_2 := e_3 \longrightarrow \Gamma' \bullet e_1'.e_2 := e_3}\ \textsc{FldA-1} \qquad \frac{\Gamma \bullet e_1 \longrightarrow \Gamma' \bullet e_1'}{\Gamma \bullet o.e_1 := e_2 \longrightarrow \Gamma' \bullet o.e_1 := e_2}\ \textsc{FldA-2}$$

$$\frac{\Gamma \bullet e_1 \longrightarrow \Gamma' \bullet e_1'}{\Gamma \bullet e_1?e_2 \longrightarrow \Gamma' \bullet e_1'?e_2}\ \textsc{HasM-1} \qquad \frac{\Gamma \bullet e \longrightarrow \Gamma' \bullet e'}{\Gamma \bullet o?e \longrightarrow \Gamma' \bullet o'?e}\ \textsc{HasM-2}$$

□

Example B.8: Direct support for dynamic method binding enables the simple *Cell* class to define objects that have both: a single application function; and a single reference to their defining class.

$$
\begin{array}{l}
Cell \mathrel{\widehat=} [\\
\quad \mathtt{new} = [\\
\quad\quad \mathtt{value} = InitValue, \\
\quad\quad \mathtt{apply} = \varsigma(s)\ \lambda(n) * GetSetLookup(n)(s) \\
\quad] \\
]
\end{array}
$$

Note that this direct model of the dynamic method application provides no support for handling invalid method names. If an invalid name was passed to the apply function, none of the existing rules would pattern match with a non-existent method selection, so the term evaluation as a whole would become blocked. Fundamentally this is not a new type of failure, because it has previously been possible to ask for a non-existent method of an object; it is just that doing this dynamically has highlighted the issue.
□

The previous example highlights the issue that the non-existent method invocation on an object causes all previously discussed models of the object calculus to block, and thus fail to produce a result. This is not necessarily a problem, as the *code* is invalid.

Aside B.9: When operations are being called outside their domains, it might be convenient to produce an *invalid* result, instead of blocking, in order to support error reporting and/or exception handling. In this case, providing additional rules for addressing method selection and update when invalid member names are encountered. The topic of error and exception handling is beyond the scope of the work being presented here.
□

Instead of waiting for an error to occur, it might be better to prevent an error occurring in the first place. This can be achieved in a number of ways, from arguments concerning correctness by construction, through static analysis to ensure that each operation is always supplied with arguments of the correct form (type), to dynamic testing of values *just* prior to their use. The last of these approaches is already supported by a combination of our $\mathcal{O}$-calculus's membership test and conditional evaluation rules.

Example B.10: The previous dynamic dispatch example has been updated to return an unset (¿) value whenever the class c does not contain a method with name n.

$$
\begin{array}{l}
Cell \mathrel{\widehat=} \mathsf{let}\ \mathtt{c} = \mathsf{fresh}\ \mathsf{in}\ c \mathrel{*=} [\\
\quad \mathtt{new} = [\\
\quad\quad \mathtt{value} = InitValue, \\
\quad\quad \mathtt{apply} = \varsigma(s)\ \lambda(n)\,(\mathsf{if}\ c?n\ \mathsf{then}\ *c.n(s)\ \mathsf{else}\ ¿) \\
\quad], \\
\quad \mathtt{get} = \lambda(s)\ GetCell, \\
\quad \mathtt{set} = \lambda(s)\ SetCell \\
]
\end{array}
$$

□

Appendix C

UTP Design Laws

This appendix provides some basic laws concerning designs within Unifying Theories of Programming (UTP). It is split into three sections. The first section demonstrates that our theory of extended UTP designs is closed; that is each of our UTP design commands generates a design, whenever their subprogram arguments are designs (or constructed from a UTP commands that generates a design). The second section presents some general laws for manipulating UTP these designs. The third section presents a specific law for manipulating our UTP models' of objects.

C.1 Closed Design Laws

In this section we demonstrate that the conditional evaluation, sequential composition, variable introduction, variable elimination, variable hiding, and block scoped variable declaration commands return UTP designs whenever their subprogram arguments are designs. Note that the iteration command also has a subprogram argument. This awkward case has already been considered within the literature [HH98, WC04] and has been shown to generate a design so long as the other UTP commands generate designs (via Tarski weakest fix-point theorem on lattices [Tar55]). We use this result. Informally, the iteration command produces a design because it either: generates a UTP program which infinitely loops, in which case it is equivalent to the chaos design; or it sequentially composes a finite number of – non-iterative – designs, which is a design.

C.1.1 Conditional binding

The following law states that a conditional choice between two designs is a design.

Law C.1 – Conditional design:

$$(p \vdash P) \lhd b \rhd (q \vdash Q) \quad = \quad (p \lhd b \rhd q) \vdash (P \lhd b \rhd Q)$$

□

Proof

$$\begin{array}{ll} & (p \vdash P) \lhd b \rhd (q \vdash Q) \\ = & \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \quad \text{Defn. of conditional} \\ & b \wedge (p \vdash P) \vee \neg b \wedge (q \vdash Q) \\ = & \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots \quad \text{Defn. of design} \\ & b \wedge (\Pi_{OK} \wedge p \Rightarrow \Pi_{OK}' \wedge P) \\ & \vee \\ & \neg b \wedge (\Pi_{OK} \wedge q \Rightarrow \Pi_{OK}' \wedge Q) \end{array}$$

$=$ .... Disjunctive normal form

$\neg\Pi_{\text{OK}} \vee b \wedge \neg p \vee \neg b \wedge \neg q$

$\vee$

$\Pi_{\text{OK}}' \wedge b \wedge P \vee \Pi_{\text{OK}}' \wedge \neg b \wedge Q$

$=$ .... Introduce implication

$\Pi_{\text{OK}} \wedge (\neg b \vee p) \wedge (b \vee q)$

$\Rightarrow$

$\Pi_{\text{OK}}' \wedge (b \wedge P \vee \neg b \wedge Q)$

$=$ .... Propositional Logic

$\Pi_{\text{OK}} \wedge (b \wedge p \vee \neg b \wedge q)$

$\Rightarrow$

$\Pi_{\text{OK}}' \wedge (b \wedge P \vee \neg b \wedge Q)$

$=$ .... Defn. of conditional

$\Pi_{\text{OK}} \wedge (p \lhd b \rhd q)$

$\Rightarrow$

$\Pi_{\text{OK}}' \wedge (P \lhd b \rhd Q)$

$=$ .... Defn. of design

$(p \lhd b \rhd q) \vdash (P \lhd b \rhd Q)$

□

C.1.2 Sequential composition

The following law states that the sequential composition of two designs is a design.

Law C.2 – Composition design:

$$p \vdash P \mathbin{;} q \vdash Q \quad = \quad p \wedge \neg(P \mathbin{;} \neg q) \vdash P \mathbin{;} Q$$

□

Proof

$p \vdash P \mathbin{;} q \vdash Q$

$=$ .... Defn. of design

$(\Pi_{\text{OK}} \wedge p \Rightarrow \Pi_{\text{OK}}' \wedge P) \mathbin{;} (\Pi_{\text{OK}} \wedge q \Rightarrow \Pi_{\text{OK}}' \wedge Q)$

$=$ .... Defn. of $;$

$\exists\, \Pi_{\text{OK}0}, w_0 \bullet (\Pi_{\text{OK}} \wedge p \Rightarrow \Pi_{\text{OK}0} \wedge P[w_0/w'])$

$\wedge$

$(\Pi_{\text{OK}0} \wedge q[w_0/w] \Rightarrow \Pi_{\text{OK}}' \wedge Q[w_0/w])$

$=$ .... Let $P_0 = P[w_0/w']$, $q_0 = q[w_0/w]$, $Q_0 = Q[w_0/w]$

$\exists\, \Pi_{\text{OK}0}, w_0 \bullet (\Pi_{\text{OK}} \wedge p \Rightarrow \Pi_{\text{OK}0} \wedge P_0)$

$\wedge$

$(\Pi_{\text{OK}0} \wedge q_0 \Rightarrow \Pi_{\text{OK}}' \wedge Q_0)$

$=$ .... Case analysis of $\Pi_{\text{OK}0}$

$\exists\, w_0 \bullet (\Pi_{\text{OK}} \wedge p \Rightarrow \mathsf{false} \wedge P_0) \wedge (\mathsf{false} \wedge q_0 \Rightarrow \Pi_{\text{OK}}' \wedge Q_0)$

$\vee$

$(\Pi_{\text{OK}} \wedge p \Rightarrow \mathsf{true} \wedge P_0) \wedge (\mathsf{true} \wedge q_0 \Rightarrow \Pi_{\text{OK}}' \wedge Q_0)$

$=$ .... Simplification

$\exists\, w_0 \bullet ((\Pi_{\text{OK}} \wedge p) \Rightarrow \mathsf{false})$

$\vee$

$(\Pi_{\text{OK}} \wedge p \Rightarrow P_0) \wedge (q_0 \Rightarrow \Pi_{\text{OK}}' \wedge Q_0)$

$=$ .... Removal of implication

$\exists\, w_0 \bullet \neg(\Pi_{\text{OK}} \wedge p)$

$\vee$

$(\neg(\Pi_{\text{OK}} \wedge p) \vee P_0) \wedge (\neg q_0 \vee \Pi_{\text{OK}}' \wedge Q_0)$

$$
\begin{array}{ll}
= & \cdots\cdots\cdots\cdots \text{ Disjunctive normal form} \\
& \exists\, w_0 \bullet \neg \Pi_{OK} \lor \neg p \lor (P_0 \land \neg q_0) \\
& \qquad \lor \\
& \qquad \Pi_{OK}' \land P_0 \land Q_0 \\
= & \cdots\cdots\cdots\cdots \text{ Predicate logic} \\
& \neg \Pi_{OK} \lor \neg p \lor (\exists\, w_0 \bullet P_0 \land \neg q_0) \\
& \lor \\
& \Pi_{OK}' \land (\exists\, w_0 \bullet P_0 \land Q_0) \\
= & \cdots\cdots\cdots\cdots \text{ Defn. of } ;, P_0, q_0, \& \ Q_0 \\
& \neg \Pi_{OK} \lor \neg p \lor (P \mathbin{;} \neg q) \lor \Pi_{OK}' \land (P \mathbin{;} Q) \\
= & \cdots\cdots\cdots\cdots \text{ Introduce implication} \\
& \Pi_{OK} \land p \land \neg(P \mathbin{;} \neg q) \Rightarrow \Pi_{OK}' \land P \mathbin{;} Q \\
= & \cdots\cdots\cdots\cdots \text{ Defn. of design} \\
& p \land \neg(P \mathbin{;} \neg q) \vdash P \mathbin{;} Q
\end{array}
$$

□

There are at least two useful corollaries, for the special cases where: the second design has a true precondition; and the first design's post condition explicitly sets the output value of each variable in the alphabet.

Corollary C.3 – Pre-true composition:

$$p \vdash P \mathbin{;} \mathsf{true} \vdash Q \quad = \quad p \vdash P \mathbin{;} Q$$

□

Proof

$$
\begin{array}{ll}
& p \vdash P \mathbin{;} \mathsf{true} \vdash Q \\
= & \cdots\cdots\cdots\cdots \text{ Design comp. law} \\
& p \land \neg(P \mathbin{;} \neg\mathsf{true}) \vdash P \mathbin{;} Q \\
= & \cdots\cdots\cdots\cdots \mathsf{false} \text{ – zero of } ; \\
& p \land \neg(\mathsf{false}) \vdash P \mathbin{;} Q \\
= & \cdots\cdots\cdots\cdots \text{ Propositional logic} \\
& p \vdash P \mathbin{;} Q
\end{array}
$$

□

Corollary C.4 – Known value composition:

$$p \vdash w' = V \mathbin{;} q \vdash Q \quad = \quad p \land q[V/w] \vdash Q[V/w]$$

□

Proof

$$
\begin{array}{ll}
& p \vdash w' = V \mathbin{;} q \vdash Q \\
= & \cdots\cdots\cdots\cdots \text{ Design comp. law} \\
& p \land \neg(w' = V \mathbin{;} \neg q) \vdash w' = V \mathbin{;} Q \\
= & \cdots\cdots\cdots\cdots \text{ Defn. of } ; \\
& p \land \neg(\exists\, w_0 \bullet w_0 = V \mathbin{;} \neg q[w_0/w]) \\
& \vdash \\
& \exists\, w_0 \bullet w_0 = V \mathbin{;} Q[w_0/w] \\
= & \cdots\cdots\cdots\cdots \text{ Predicate Logic} \\
& p \land \neg(\neg q[V/w]) \vdash Q[V/w] \\
= & \cdots\cdots\cdots\cdots \text{ Propositional Logic} \\
& p \land q[V/w] \vdash Q[V/w]
\end{array}
$$

□

C.1.3 Variable introduction and elimination

The following law says that the variable introduction command can be rewritten as a design.

Law C.5:

$$\mathsf{var}\,x \quad = \quad \mathsf{true} \vdash \exists\, x \bullet w' = w \wedge x' = x$$

where $w \equiv x_{i=1}^{k}$, $x \notin \{w\}$ and $\alpha(\mathsf{var}\,x) = \{w, w', x'\}$.

□

Proof

$\mathsf{var}\,x$
$=$ .... Defn. of $\mathsf{var}\,\_$
$\exists\, x \bullet \mathsf{skip}^{\mathrm{D}}$
$=$ .... Defn. of $\mathsf{skip}^{\mathrm{D}}$
$\exists\, x \bullet \mathit{true} \vdash w' = w \wedge x' = x$
$=$ .... Defn. of $\_ \vdash \_$
$\exists\, x \bullet \Pi_{\text{OK}} \Rightarrow (\Pi_{\text{OK}}' \wedge w' = w \wedge x' = x)$
$=$ .... Predicate Logic
$\Pi_{\text{OK}} \Rightarrow (\Pi_{\text{OK}}' \wedge \exists\, x \bullet w' = w \wedge x' = x)$
$=$ .... Defn. of $\_ \vdash \_$
$\mathsf{true} \vdash \exists\, x \bullet w' = w \wedge x' = x$

□

Note that the last step can be further simplified to remove the existential quantification altogether, as we can use the one-point rule to set the variable x equal to x'. In this case, we end up with the design $\mathsf{true} \vdash w' = w$.

The following law says that the variable elimination command can be rewritten as a design.

Law C.6:

$$\mathsf{end}\,x \quad = \quad \mathsf{true} \vdash \exists\, x' \bullet w' = w \wedge x' = x$$

where $w \equiv x_{i=1}^{k}$, $x \notin \{w\}$ and $\alpha(\mathsf{end}\,x) = \{w, w', x\}$.

□

Proof

$\mathsf{end}\,x$
$=$ .... Defn. of $\mathsf{var}\,\_$
$\exists\, x' \bullet \mathsf{skip}^{\mathrm{D}}$
$=$ .... Defn. of $\mathsf{skip}^{\mathrm{D}}$
$\exists\, x' \bullet \mathit{true} \vdash w' = w \wedge x' = x$
$=$ .... Defn. of $\_ \vdash \_$
$\exists\, x' \bullet \Pi_{\text{OK}} \Rightarrow (\Pi_{\text{OK}}' \wedge w' = w \wedge x' = x)$
$=$ .... Predicate Logic
$\Pi_{\text{OK}} \Rightarrow (\Pi_{\text{OK}}' \wedge \exists\, x' \bullet w' = w \wedge x' = x)$
$=$ .... Defn. of $\_ \vdash \_$
$\mathsf{true} \vdash \exists\, x' \bullet w' = w \wedge x' = x$

□

C.1.4 Hiding

The following law states that hiding some variables from a design returns a design.

Law C.7 – Hide Design:

$$\mathsf{hide}\ u\ \mathsf{from}\ p[w] \vdash P[w, w'] \quad = \quad p[w] \vdash P[w, w'] \wedge u' = u$$

□

Proof

Let P_1, P_2, and P_3 denote $(\exists\, u' \bullet u' = u \wedge w' = w)$, $(\exists\, u \bullet u' = u \wedge w' = w)$, and $(u' = u_0 \wedge w' = w)$ respectively.

$$\begin{array}{ll}
& \mathsf{hide}\ u\ \mathsf{from}\ p[w] \vdash P[w, w'] \\
= & \cdots\cdots \text{Defn. of } \mathsf{hide}\ _\ \mathsf{from}\ _ \\
& \exists\, u_0 \mid u_0 = u \bullet \\
& \quad (\mathsf{end}\ u) \fatsemi (p[w] \vdash P[w, w']) \fatsemi (\mathsf{var}\ u) \fatsemi (u := u_0) \\
= & \cdots\cdots \text{Design variants of } \mathsf{end}\ _,\ \mathsf{var}\ _, \text{ and } _ := _ \\
& \exists\, u_0 \mid u_0 = u \bullet \\
& \quad (\mathsf{true} \vdash P_1) \fatsemi (p[w] \vdash P[w, w']) \fatsemi \\
& \quad (\mathsf{true} \vdash P_2) \fatsemi (\mathsf{true} \vdash P_3) \\
= & \cdots\cdots \text{Law C.2 comp. design} \\
& \exists\, u_0 \mid u_0 = u \bullet \\
& \quad \mathsf{true} \wedge \neg(P_1 \fatsemi \neg p[w]) \wedge \\
& \quad \neg(P_1 \fatsemi P[w, w'] \fatsemi \neg\mathsf{true}) \wedge \neg(P_1 \fatsemi P[w, w'] \fatsemi P_2 \fatsemi \neg\mathsf{true}) \\
& \quad \vdash \\
& \quad P_1 \fatsemi P \fatsemi P_2 \fatsemi P_3 \\
= & \cdots\cdots \text{false is the right zero of } _ \fatsemi _ \\
& \exists\, u_0 \mid u_0 = u \bullet \\
& \quad \neg(P_1 \fatsemi \neg p[w]) \\
& \quad \vdash \\
& \quad P_1 \fatsemi P[w, w'] \fatsemi P_2 \fatsemi P_3 \\
= & \cdots\cdots \text{Defn. of } _ \fatsemi _ \\
& \exists\, u_0 \mid u_0 = u \bullet \\
& \quad \neg(\exists\, w_0 \bullet P_1[w_0/w'] \wedge \neg p[w_0/w]) \\
& \quad \vdash \\
& \quad P_1 \fatsemi P[w, w'] \fatsemi P_2 \fatsemi P_3 \\
= & \cdots\cdots \text{Defn. of } P_1 \\
& \exists\, u_0 \mid u_0 = u \bullet \\
& \quad \neg(\exists\, w_0 \bullet (\exists\, u' \bullet u' = u \wedge w_0 = w) \wedge \neg p[w_0/w]) \\
& \quad \vdash \\
& \quad P_1 \fatsemi P \fatsemi P_2 \fatsemi P_3 \\
= & \cdots\cdots u' = u,\ w_0 = w, \text{ and} \\
& \exists\, u_0 \mid u_0 = u \bullet \qquad \text{simplify} \\
& \quad p[w] \vdash P_1 \fatsemi P[w, w'] \fatsemi P_2 \fatsemi P_3 \\
= & \cdots\cdots \text{Defn. of } _ \fatsemi _, \\
& \exists\, u_0 \mid u_0 = u \bullet \qquad \text{used 3 times} \\
& \quad p[w] \vdash \exists\, w_0, w_1, w_2 \bullet \\
& \qquad P_1[w_0/w'] \wedge P[w_0, w_1/w, w'] \wedge \\
& \qquad P_2[w_1, w_2/w, w'] \wedge P_3[w_2/w]
\end{array}$$

$$
\begin{array}{ll}
= & \cdots\cdots\cdots\cdots \text{Defn. of } P_1,\ P_2, \text{ and } P_3, \\
& \exists\, u_0 \mid u_0 = u \bullet \quad \text{where bound variables are} \\
& \quad p[w] \vdash \exists\, w_0, w_1, w_2 \bullet \quad \text{renamed} \\
& \qquad (\exists\, u_1 \bullet u_1 = u \land w_0 = w) \land P[w_0, w_1/w, w'] \land \\
& \qquad (\exists\, u_2 \bullet u' = u_2 \land w_2 = w_1) \land \\
& \qquad (u' = u_0 \land w' = w_2)[w_2/w] \\
= & \cdots\cdots\cdots\cdots u_1 = u,\ u_2 = u, \text{ and} \\
& \exists\, u_0 \mid u_0 = u \bullet \quad \text{simplify} \\
& \quad p[w] \vdash \exists\, w_0, w_1, w_2 \bullet \\
& \qquad w_0 = w \land P[w_0, w_1/w, w'] \land \\
& \qquad w_2 = w_1 \land (u' = u_0 \land w' = w_2)[w_2/w] \\
= & \cdots\cdots\cdots\cdots w_0 = w,\ w_1 = w',\ w_2 = w', \\
& \exists\, u_0 \mid u_0 = u \bullet \quad \text{and simplify} \\
& \quad p[w] \vdash P[w, w'] \land u' = u_0 \\
= & \cdots\cdots\cdots\cdots \text{Defn. of } \_ \vdash \_ \text{ and} \\
& \exists\, u_0 \bullet u_0 = u \land (\quad \text{Defn. of } \exists \_ \mid \_ \bullet \_ \\
& \quad \Pi\text{ок} \land p[w] \Rightarrow \Pi\text{ок}' \land P[w, w'] \land u' = u_0 \\
&) \\
= & \cdots\cdots\cdots\cdots \text{Predicate Logic} \\
& \exists\, u_0 \bullet \\
& \quad (\Pi\text{ок} \land p[w]) \lor u_0 \neq u \\
& \quad \Rightarrow \\
& \quad \Pi\text{ок}' \land (P[w, w'] \land u' = u_0 \land u_0 = u \\
= & \cdots\cdots\cdots\cdots \text{Predicate Logic} \\
& (\forall\, u_0 \bullet (\Pi\text{ок} \land p[w]) \lor u_0 \neq u) \\
& \Rightarrow \\
& (\exists\, u_0 \bullet (\Pi\text{ок}' \land P[w, w'] \land u' = u_0 \land u_0 = u) \\
= & \cdots\cdots\cdots\cdots u_0 \text{ not free in } p[w] \text{ or} \\
& (\Pi\text{ок} \land p[w]) \lor (\forall\, u_0 \bullet u_0 \neq u) \quad P[w, w'] \\
& \Rightarrow \\
& (\Pi\text{ок}' \land P[w, w'] \land (\exists\, u_0 \bullet u' = u_0 \land u_0 = u) \\
= & \cdots\cdots\cdots\cdots \text{Predicate Logic} \\
& \Pi\text{ок} \land p[w] \Rightarrow \Pi\text{ок}' \land P[w, w'] \land u' = u \\
= & \cdots\cdots\cdots\cdots \text{Defn. of } \_ \vdash \_ \\
& p[w] \vdash P[w, w'] \land u' = u
\end{array}
$$

□

C.1.5 Block scoped variable declaration

The following law states that when a block scoped declaration command, which introduces an uninitialised vector of variables for use within a subprogram, is provided with a design it returns a design.

Law C.8 – Declaration Design:

$$\mathsf{decl}\, u \,\mathsf{in}\, p[u] \vdash P[u, u'] \quad = \quad (\forall\, u \bullet p[u]) \vdash (\exists\, u, u' \bullet P[u, u'])$$

□

Proof

$$
\begin{array}{ll}
& \mathsf{decl}\, u \,\mathsf{in}\, p[u] \vdash P[u, u'] \\
= & \cdots\cdots\cdots\cdots \text{Defn. of } \mathsf{decl} \_ \,\mathsf{in}\, \_ \\
& (\mathsf{var}\, u) \fatsemi (p[u] \vdash P[u, u']) \fatsemi (\mathsf{end}\, u)
\end{array}
$$

$$
\begin{array}{ll}
= & \dots\dots\dots\dots \text{Law C.11 var scoping} \\
& \exists\, u \bullet (p[u] \vdash P[u,u']) \,;\, (\mathsf{end}\; u) \\
= & \dots\dots\dots\dots \text{Law C.12 end scoping} \\
& \exists\, u, u' \bullet p[u] \vdash P[u,u'] \\
= & \dots\dots\dots\dots \text{Defn. of \_} \vdash \text{\_} \\
& \exists\, u, u' \bullet \Pi_{OK} \wedge p[u] \Rightarrow \Pi_{OK}' \wedge P[u,u'] \\
= & \dots\dots\dots\dots \text{Predicate Logic} \\
& (\forall\, u, u' \bullet \Pi_{OK} \wedge p[u]) \Rightarrow (\exists\, u, u' \bullet \Pi_{OK}' \wedge P[u,u']) \\
= & \dots\dots\dots\dots u' \text{ not free in } \Pi_{OK} \wedge p[u]. \\
& (\forall\, u \bullet \Pi_{OK} \wedge p[u]) \Rightarrow (\exists\, u, u' \bullet \Pi_{OK}' \wedge P[u,u']) \\
= & \dots\dots\dots\dots \text{Predicate Logic} \\
& \Pi_{OK} \wedge (\forall\, u \bullet p[u])) \Rightarrow \Pi_{OK}' \wedge (\exists\, u, u' \bullet P[u,u']) \\
= & \dots\dots\dots\dots \text{Defn. of \_} \vdash \text{\_} \\
& (\forall\, u \bullet p[u])) \vdash (\exists\, u, u' \bullet P[u,u'])
\end{array}
$$

□

The following law states that when a block scoped declaration command, which uses a partial map of variables to values to introduce an initialised collection of variables to a sub-program, is provided with a design it returns a design.

Law C.9 – Declaration Map Design: Let u denote the list of distinct undashed variables $x_{i=1}^{k}$. Let V denote the list of values $v_{i=1}^{j}$.

$$\mathsf{decl}\{_{i=1}^{k} x_i \mapsto v_i\} \;\mathsf{in}\; p \vdash P \quad = \quad p[V/u] \vdash (\exists\, u' \bullet P[V/u])$$

□

Proof

$$
\begin{array}{ll}
& \mathsf{decl}\; \{_{i=1}^{k} x_i \mapsto v_i\} \;\mathsf{in}\; p \vdash P \\
= & \dots\dots\dots\dots \text{Defn. of } \mathsf{decl}\,\_\;\mathsf{in}\,\_ \\
& \mathsf{decl}\; u \;\mathsf{in}\; (u := V) \,;\, (p \vdash P) \\
= & \dots\dots\dots\dots \text{Defn. of \_} := \text{\_} \\
& \mathsf{decl}\; u \;\mathsf{in}\; (\mathsf{true} \vdash u' = V) \,;\, (p \vdash P) \\
= & \dots\dots\dots\dots \text{Law C.2 comp. design} \\
& \mathsf{decl}\; u \;\mathsf{in} \\
& \quad \mathsf{true} \wedge \neg((u' = V) \,;\, \neg p) \\
& \quad \vdash \\
& \quad (u' = V) \,;\, P \\
= & \dots\dots\dots\dots \text{Defn. of \_} ; \text{\_} \\
& \mathsf{decl}\; u \;\mathsf{in} \\
& \quad \neg(\exists\, u_0 \bullet u_0 = V \wedge \neg p[u_0/u]) \\
& \quad \vdash \\
& \quad (\exists\, u_0 \bullet (u_0 = V) \wedge P[u_0/u] \\
= & \dots\dots\dots\dots \text{One point rule} \\
& \mathsf{decl}\; u \;\mathsf{in}\; \neg(\neg p[V/u]) \vdash P[V/u] \\
= & \dots\dots\dots\dots \text{Law C.8 decl. design} \\
& (\forall\, u \bullet p[V/u])) \vdash (\exists\, u, u' \bullet P[V/u]) \\
= & \dots\dots\dots\dots u \text{ not free in \_[\_}/u] \\
& p[V/u] \vdash (\exists\, u' \bullet P[V/u])
\end{array}
$$

□

C.2 General Design Laws

In this section we present (and prove) a collection of basic laws for the theory of UTP designs. Several of these are variants on existing work [HH98, WC04]; the differences in the proofs result from our use of [HLL02a]'s updated variant of the design definition.

C.2.1 Skip the unit of sequential composition

We have two distinct notions of skip, one for the basic relational unifying theory (i.e. skip) and one for the design-based unifying theory (i.e. $\mathsf{skip}^{\mathrm{D}}$). They are both considered to be the unit of sequential composition. Note the design-based definition of skip is not strictly necessary; as it can be shown to be equal to that of the basic one, when applied to designs, however, it has the convenient form of a design.

We now state and prove the law that the design-based version of skip is the unit of composition for designs. The proof that the basic relational version of skip is also a unit of composition can be proved in a similar manner (it is actually easier to do).

Law C.10 – Design-skip composition unit:

$$\mathsf{skip}^{\mathrm{D}} \fatsemi (p \vdash P) \;=\; p \vdash P \;=\; (p \vdash P) \fatsemi \mathsf{skip}^{\mathrm{D}}$$

□

This is proved in two parts, one where skip is on the left hand side (LHS) of the sequential composition, and the other where it is on the right hand side (RHS).

Proof of LHS

$$\begin{array}{ll} & \mathsf{skip}^{\mathrm{D}} \fatsemi (p \vdash P) \\ = & \quad \text{Defn. of } \mathsf{skip}^{\mathrm{D}} \\ & (\mathsf{true} \vdash w' = w) \fatsemi (p \vdash P) \\ = & \quad \text{Known val. Corol. C.4} \\ & \mathsf{true} \wedge p[w/w] \vdash P[w/w] \\ = & \quad \text{Simplification} \\ & p \vdash P \end{array}$$

□

Proof of RHS

$$\begin{array}{ll} & (p \vdash P) \fatsemi \mathsf{skip}^{\mathrm{D}} \\ = & \quad \text{Defn. of } \mathsf{skip}^{\mathrm{D}} \\ & (p \vdash P) \fatsemi (\mathsf{true} \vdash w' = w) \\ = & \quad \text{Pre-true Corol. C.3} \\ & p \vdash P \fatsemi w' = w \\ = & \quad \text{Defn. of } \fatsemi \\ & p \vdash \exists\, w_0 \bullet P[w_0/w'] \wedge w' = w_0 \\ = & \quad \text{1-point rule } (w_0 = w') \\ & p \vdash P \end{array}$$

□

C.2.2 Scoping variables

The following law states that the scope of an introduced variable extends to the RHS of a sequential composition of designs.

Law C.11 – Variable scoping law:

$$\mathsf{var}\, x \fatsemi (q \vdash Q) = \exists\, x \bullet (q \vdash Q)$$

□

Proof

$$
\begin{array}{ll}
& \mathsf{var}\, x \fatsemi (q \vdash Q) \\
= & \cdots\cdots \text{Defn. of } \mathsf{var} \\
& (\exists\, x \bullet \mathsf{skip}^{\mathrm{D}}) \fatsemi (q \vdash Q) \\
= & \cdots\cdots \text{Defn. of } \fatsemi \\
& \exists\, w_0 \bullet (\exists\, x \bullet \mathsf{skip}^{\mathrm{D}})[w_0/w'] \wedge (q \vdash Q)[w_0/w] \\
= & \cdots\cdots \text{Pred. calc. } (x \notin w_0, w') \\
& \exists\, w_0 \bullet (\exists\, x \bullet \mathsf{skip}^{\mathrm{D}}[w_0/w'] \wedge (q \vdash Q)[w_0/w]) \\
= & \cdots\cdots \text{Pred. calc.} \\
& \exists\, x \bullet (\exists\, w_0 \bullet \mathsf{skip}^{\mathrm{D}}[w_0/w'] \wedge (q \vdash Q)[w_0/w]) \\
= & \cdots\cdots \text{Defn. of } \fatsemi \\
& \exists\, x \bullet \mathsf{skip}^{\mathrm{D}} \fatsemi (q \vdash Q) \\
= & \cdots\cdots \mathsf{skip}^{\mathrm{D}} \text{ is a unit } \fatsemi \\
& \exists\, x \bullet (q \vdash Q)
\end{array}
$$

□

The following law states that the scope of an eliminated variable extends to the LHS of a sequential composition of designs.

Law C.12 – Variable completion law:

$$(q \vdash Q) \fatsemi \mathsf{end}\, x = \exists\, x' \bullet (q \vdash Q)$$

□

Proof

$$
\begin{array}{ll}
& (q \vdash Q) \fatsemi \mathsf{end}\, x \\
= & \cdots\cdots \text{Defn. of } \mathsf{end} \\
& (q \vdash Q) \fatsemi (\exists\, x' \bullet \mathsf{skip}^{\mathrm{D}}) \\
= & \cdots\cdots \text{Defn. of } \fatsemi \\
& \exists\, w_0 \bullet (q \vdash Q)[w_0/w'] \wedge (\exists\, x' \bullet \mathsf{skip}^{\mathrm{D}})[w_0/w] \\
= & \cdots\cdots \text{Pred. calc. } (x' \notin w_0, w) \\
& \exists\, w_0 \bullet (\exists\, x' \bullet (q \vdash Q)[w_0/w'] \wedge \mathsf{skip}^{\mathrm{D}}[w_0/w]) \\
= & \cdots\cdots \text{Pred. calc.} \\
& \exists\, x' \bullet (\exists\, w_0 \bullet (q \vdash Q)[w_0/w'] \wedge \mathsf{skip}^{\mathrm{D}}[w_0/w]) \\
= & \cdots\cdots \text{Defn. of } \fatsemi \\
& \exists\, x' \bullet (q \vdash Q) \fatsemi \mathsf{skip}^{\mathrm{D}} \\
= & \cdots\cdots \mathsf{skip}^{\mathrm{D}} \text{ is a unit } \fatsemi \\
& \exists\, x' \bullet (q \vdash Q)
\end{array}
$$

□

C.2.3 Assignment composition

We now introduce four laws associated with applying assignment within a sequential composition context. The first of these laws is the one that is most used, as it states how to apply the affects of an assignment operation to the next sequentially composed subprogram.

Left assignment composition: The following law states that the affect of applying an assignment of the value v to the variable x to the next sequentially composed subprogram, is to perform a semantic replacement of the composed subprogram's variable x with the value v. Note that this generalises to multiple assignment case in a straightforward manner.

Law C.13 – Left Assignment Composition:

$$x := v \fatsemi (p \vdash P) \quad = \quad (p \vdash P)[v/x]$$

□

Proof

$$
\begin{array}{ll}
& x := v \mathbin{;} (p \vdash P) \\
= & \dotfill \text{ Defn. } \_ := \_ \\
& (\mathsf{true} \vdash x' = v \land w' = w) \mathbin{;} (p \vdash P) \\
= & \dotfill \text{ Law C.2 comp. design} \\
& \neg((x' = v \land w' = w) \mathbin{;} \neg p) \\
& \vdash \\
& (x' = v \land w' = w) \mathbin{;} P \\
= & \dotfill \text{ Defn. } \_ \mathbin{;} \_ \\
& \neg(\exists\, w_0, x_0 \bullet (x_0 = v \land w_0 = w) \land \neg p[w_0, x_0/w, x]) \\
& \vdash \\
& \exists\, w_0, x0 \bullet x_0 = v \land w_0 = w \land P[w_0, x_0/w, x] \\
= & \dotfill \text{ 1-point rule} \\
& \neg(\neg p[v/x]) \vdash P[v/x] \\
= & \dotfill \text{ Defn. } \_[\_/\_] \\
& (p \vdash P)[v/x]
\end{array}
$$

□

Double assignment composition: The following law states that assigning two values to the same variable in succession, is the same as assigning the second value to the variable. Note this relies on the UTP constraint that expressions cannot have side-effects (or at least side-effects that affect the program's state).

Law C.14 – Double Assignment Composition:

$$x := e_1 \mathbin{;} x := e_2 \quad = \quad x := e_2$$

□

Proof

$$
\begin{array}{ll}
& x := e_1 \mathbin{;} x := e_2 \\
= & \dotfill \text{ Defn. } \_ := \_ \\
& (\mathsf{true} \vdash x' = e_1 \land w' = w) \mathbin{;} (\mathsf{true} \vdash x' = e_2 \land w' = w) \\
= & \dotfill \text{ Corol. C.3 comp. pre-true} \\
& \mathsf{true} \vdash (x' = e_1 \land w' = w) \mathbin{;} (x' = e_2 \land w' = w) \\
= & \dotfill \text{ Defn. } \_ \mathbin{;} \_ \\
& \mathsf{true} \vdash \exists\, x_0, w_0 \bullet x_0 = e_1 \land w_0 = w \land x' = e_2 \land w' = w_0 \\
= & \dotfill \text{ 1 point rule} \\
& \mathsf{true} \vdash x' = e_2 \land w' = w \\
= & \dotfill \text{ Defn. } \_ := \_ \\
& x := e_2
\end{array}
$$

□

Right assignment elimination: The following two laws demonstrate how to eliminate an assignment that does not change the state of a program. The first law states that the affect of assigning a value to a variable can be pushed back into the preceding sequentially composed subprogram, which does not otherwise alter the variable, by updating this subprogram so that it also sets the variable to the assigned value.

Law C.15 – Right Assignment Composition:

$$(p \vdash P) \mathbin{;} x := v \quad = \quad p \vdash P[x/x'] \land x' = v \qquad \text{where } P = (P \land x' = x)$$

□

Proof

$$
\begin{array}{ll}
& (p \vdash P) \fatsemi x := v \\
= & \cdots\cdots \text{Defn. } \_ := \_ \\
& (p \vdash P) \fatsemi (\mathsf{true} \vdash x' = v \wedge w' = w) \\
= & \cdots\cdots \text{Corol. C.3 comp. pre-true} \\
& p \vdash P \fatsemi (x' = v \wedge w' = w) \\
= & \cdots\cdots \text{Defn. } \_ \fatsemi \_ \\
& p \vdash \exists\, w_0, x_0 \bullet P[w_0, x_0/w', x'] \wedge (x' = v \wedge w' = w_0) \\
= & \cdots\cdots \text{1 point rule} \\
& p \vdash \exists\, x_0 \bullet P[x_0/x'] \wedge x' = v \\
= & \cdots\cdots \text{A1 i.e. } P = (P \wedge x' = x) \\
& p \vdash \exists\, x_0 \bullet (P \wedge x' = x)[x_0/x'] \wedge x' = v \\
= & \cdots\cdots \text{Defn. } \_[\_/\_] \text{ and Pred. L.} \\
& p \vdash (\exists\, x_0 \bullet P[x_0/x'] \wedge x_0 = x) \wedge x' = v \\
= & \cdots\cdots \text{1 point rule} \\
& p \vdash P[x/x'] \wedge x' = v
\end{array}
$$

□

The second law states that a program that the reassigns of the same value to a variable can be eliminated.

Law C.16 – Special Assignment Composition:

$$x := v \fatsemi (p \vdash P[x/x'] \wedge x' = v) \quad = \quad x := v \fatsemi (p \vdash P) \qquad \text{where } P = (P \wedge x' = x)$$

□

Proof

$$
\begin{array}{ll}
& x := v \fatsemi (p \vdash P[x/x'] \wedge x' = v) \\
= & \cdots\cdots \text{Law C.13 assign comp} \\
& (p \vdash P[x/x'] \wedge x' = v)[v/x] \\
= & \cdots\cdots \text{Defn. } \_[\_/\_] \\
& (p[v/x] \vdash P[x/x'][v/x] \wedge x' = v) \\
= & \cdots\cdots \text{Defn. } \_[\_/\_] \\
& (p[v/x] \vdash P[v/x'][v/x] \wedge x' = v) \\
= & \cdots\cdots \text{Defn. } \_[\_/\_] \\
& (p \vdash P[v/x'] \wedge x' = v)[v/x] \\
= & \cdots\cdots v = x' \\
& (p \vdash P \wedge x' = v)[v/x] \\
= & \cdots\cdots \text{A1 i.e. } P = (P \wedge x = x') \\
& (p \vdash P \wedge x' = x \wedge x' = v)[v/x] \\
= & \cdots\cdots x = x' \\
& (p \vdash P \wedge x' = x \wedge x = v)[v/x] \\
= & \cdots\cdots \text{A1 i.e. } P = (P \wedge x = x') \\
& (p \vdash P \wedge x = v)[v/x] \\
= & \cdots\cdots \text{Similar rewriting} \\
& (p \vdash P)[v/x] \\
= & \cdots\cdots \text{Law C.13 assign comp} \\
& x := v \fatsemi (p \vdash P)
\end{array}
$$

□

C.2.4 Conjunctive implication

The following law demonstrates that the a conjunctive predicate involving a design can be weakened, by apply the other argument to the post-condition within the design, so long as this other argument does not involve the design specific observational variables.

Law C.17:

$$(p \vdash P) \wedge R \Rightarrow (p \vdash P \wedge R) \qquad \text{provided } \Pi_{\text{OK}}, \Pi_{\text{OK}}' \notin R.$$

□

Proof

$$(p \vdash P) \wedge R$$
$= \ldots\ldots\ldots\ldots\ldots\ldots$ Defn. of $\_ \vdash \_$
$$(\Pi_{\text{OK}} \wedge p \Rightarrow \Pi_{\text{OK}}' \wedge P) \wedge R$$
$= \ldots\ldots\ldots\ldots\ldots\ldots$ Propositional Logic
$$(\Pi_{\text{OK}} \wedge p) \vee \neg R \Rightarrow (\Pi_{\text{OK}}' \wedge P) \wedge R$$
$\Rightarrow \ldots\ldots\ldots\ldots\ldots\ldots$ Propositional Logic
$$(\Pi_{\text{OK}} \wedge p) \Rightarrow (\Pi_{\text{OK}}' \wedge P) \wedge R$$
$= \ldots\ldots\ldots\ldots\ldots\ldots$ Defn. of $\_ \vdash \_$
$$p \vdash P \wedge R$$
□

C.3 Object Model Design Laws and Example Proofs

In this section we present (and prove) a law that is specific to the UTP models of the $\mathcal{O}$-calculus, and proofs of the 'factorial example' lemmas of Section 4.6.3. Most of the proofs concerning the UTP object models are contained in Chapter 4.

C.3.1 Extract identity

The extract identity law states that substituting the UTP program call_{J} for the call_{J} program text within the body of a method has no effect on that method. This special case occurs when one iteration of the fix-point definition of procedure invocation is unwound.

Law C.18 – Extract identity:

$$\mathsf{ext}_{\text{J}}(pt, \mathsf{call}_{\text{J}}) \quad = \quad [\![\, pt \,]\!]_{\text{J}}$$

□

Proof

$$\mathsf{ext}_{\text{J}}(pt, \mathsf{call}_{\text{J}})$$
$= \ldots\ldots\ldots\ldots\ldots\ldots$ Defn. ext_{J}
$$[\![\, \mathsf{fix}_{\text{J}}(pt, \mathsf{text}_{\text{J}}(\mathsf{call}_{\text{J}})) \,]\!]$$
$= \ldots\ldots\ldots\ldots\ldots\ldots$ Defn. text_{J}
$$[\![\, \mathsf{fix}_{\text{J}}(pt, \mathsf{call}_{\text{J}}) \,]\!]$$
$= \ldots\ldots\ldots\ldots\ldots\ldots$ Defn. fix_{J}
$$[\![\, pt \,]\!]$$
□

C.3.2 Factorial example UTP correspondence proofs

We now present the correspondence proof for each of the four steps in the factorial example (Section 4.6.3).

Step 1 – Method invocation to function invocation:

$$[\![sFact.\texttt{fact}(n)]\!]_{\mathrm{S}} = [\![sFactFn(n)]\!]_{\mathrm{S}}$$

Proof

$$
\begin{array}{ll}
& [\![sFact.\texttt{fact}(n)]\!]_{\mathrm{S}} \\
= & \cdots\cdots \text{Compile } sFact.\texttt{fact}(n) \\
& [\![\ SF(OF(O), n)\ ]\!] \\
= & \cdots\cdots \text{Defn. of } SF(OF(O), n) \\
& [\![\ vfPair^{\mathrm{CE}}_{\mathrm{S}}(\\
& \quad OF(O), \\
& \quad \mathcal{E}_{\mathrm{S}}\, n \\
&)\ ⨟\ \mathsf{call}_{\mathrm{S}}\ ]\!] \\
= & \cdots\cdots \text{Defn. of } OF(O) \text{ and } [\![\ \_\ ]\!] \\
& vfPair^{\mathrm{CE}}_{\mathrm{S}}(\\
& \quad omPair^{\mathrm{CE}}_{\mathrm{S}}(\mathcal{E}_{\mathrm{S}}\, x, \mathcal{E}_{\mathrm{S}}\, \texttt{fact})\ ⨟\ \mathsf{call}_{\mathrm{S}}, \\
& \quad \mathcal{E}_{\mathrm{S}}\, n \\
&)\ ⨟\ \mathsf{call}_{\mathrm{S}} \\
= & \cdots\cdots \text{Defn. of } omPair^{\mathrm{CE}}_{\mathrm{S}} \\
& vfPair^{\mathrm{CE}}_{\mathrm{S}}(\\
& \quad \mathsf{cmdExp}_{\mathrm{S}}(\\
& \quad\quad \mathsf{trans}_{\mathrm{S}}(omPair, 2), \\
& \quad\quad (\mathcal{E}_{\mathrm{S}}\, O, \mathcal{E}_{\mathrm{S}}\, \texttt{fact}) \\
& \quad)\ ⨟\ \mathsf{call}_{\mathrm{S}}, \\
& \quad \mathcal{E}_{\mathrm{S}}\, n \\
&)\ ⨟\ \mathsf{call}_{\mathrm{S}} \\
= & \cdots\cdots \text{Lemma 4.1 cmd. exp., where } (O, \texttt{fact}) \in omPair \\
& vfPair^{\mathrm{CE}}_{\mathrm{S}}(\\
& \quad \mathcal{E}_{\mathrm{S}}\, omPair(O, \texttt{fact})\ ⨟\ \mathsf{call}_{\mathrm{S}}, \\
& \quad \mathcal{E}_{\mathrm{S}}\, n \\
&)\ ⨟\ \mathsf{call}_{\mathrm{S}} \\
= & \cdots\cdots \text{Defn. of } omPair^{\mathrm{CE}}_{\mathrm{S}} \\
& vfPair^{\mathrm{CE}}_{\mathrm{S}}(\\
& \quad \mathcal{E}_{\mathrm{S}}(O, (\!|\ x, F(x)\ |\!))\ ⨟\ \mathsf{call}_{\mathrm{S}}, \\
& \quad \mathcal{E}_{\mathrm{S}}\, n \\
&)\ ⨟\ \mathsf{call}_{\mathrm{S}} \\
= & \cdots\cdots \text{Lemma 4.3 proc. call} \\
& vfPair^{\mathrm{CE}}_{\mathrm{S}}(\\
& \quad \mathsf{ext}_{\mathrm{S}}(F(x)\{\!|x \mapsto O|\!\}_{\mathrm{S}}, \mathsf{call}_{\mathrm{S}}), \\
& \quad \mathcal{E}_{\mathrm{S}}\, n \\
&)\ ⨟\ \mathsf{call}_{\mathrm{S}} \\
= & \cdots\cdots \text{Defn. of } \{\!|\_ \mapsto \_|\!\}_{\mathrm{S}} \\
& vfPair^{\mathrm{CE}}_{\mathrm{S}}(\\
& \quad \mathsf{ext}_{\mathrm{S}}(F(O), \mathsf{call}_{\mathrm{S}}), \\
& \quad \mathcal{E}_{\mathrm{S}}\, n \\
&)\ ⨟\ \mathsf{call}_{\mathrm{S}} \\
= & \cdots\cdots \text{Law C.18 extract id.} \\
& vfPair^{\mathrm{CE}}_{\mathrm{S}}(\\
& \quad [\![\ F(O)\ ]\!], \\
& \quad \mathcal{E}_{\mathrm{S}}\, n \\
&)\ ⨟\ \mathsf{call}_{\mathrm{S}}
\end{array}
$$

$= \dots$ Defn. of $[\![\_]\!]$

$[\![\, vfPair^{CE}_{S}(F(O), \mathcal{E}_S\, n) \mathbin{;} \mathsf{call}_S \,]\!]$

$= \dots$ Compile $sFactFn(n)$

$[\![\langle sFactFn(n) \rangle]\!]_S$

□

Step 2 – Function invocation to conditional expression:

$$[\![\langle sFactFn(n) \rangle]\!]_S = [\![\langle sFactCond1 \rangle]\!]_S$$

Proof

$[\![\langle sFactFn(n) \rangle]\!]_S$

$= \dots$ Compile $sFactFn(n)$

$[\![\, SF(F(O), n) \,]\!]$

$= \dots$ Defn. of $SF(F(O), n)$

$[\![\, vfPair^{CE}_{S}($
$\quad F(O), \mathcal{E}_S\, n$
$) \mathbin{;} \mathsf{call}_S \,]\!]$

$= \dots$ Defn. of $F(O)$ and $[\![\_]\!]$

$vfPair^{CE}_{S}($
$\quad \mathcal{E}_S(\![\, i, E1(O, i) \,]\!), \mathcal{E}_S\, n$
$) \mathbin{;} \mathsf{call}_S$

$= \dots$ Defn. of $vfPair^{CE}_{S}$

$\mathsf{cmdExp}_S($
$\quad \mathsf{trans}_S(vfPair, 2),$
$\quad (\mathcal{E}_S(\![\, i, E1(O, i) \,]\!), \mathcal{E}_S\, n)$
$) \mathbin{;} \mathsf{call}_S$

$= \dots$ Lemma 4.1 cmd. exp., where $((\![\, i, E1(O, i) \,]\!), n) \in vfPair$

$\mathcal{E}_S\, vfPair((\![\, i, E1(O, i) \,]\!), n) \mathbin{;}$
call_S

$= \dots$ Defn. of $vfPair$

$\mathcal{E}_S(n, (\![\, i, E1(O, i) \,]\!)) \mathbin{;}$
call_S

$= \dots$ Lemma 4.3 proc. call

$\mathsf{ext}_S(E1(O, i)\{\!| i \leftarrow n |\!\}_S, \mathsf{call}_S)$

$= \dots$ Defn. of $\{\!| \_ \leftarrow \_ |\!\}_S$

$\mathsf{ext}_S(E1(O, n), \mathsf{call}_S)$

$= \dots$ Law C.18 extract id.

$[\![\, E1(O, n) \,]\!]$

$= \dots$ Compile $sFactCond1$

$[\![\langle sFactCond1 \rangle]\!]_S$

□

Step 3 – Conditional expression decision evaluation:

$$[\![\langle sFactCond1 \rangle]\!]_S = [\![\langle sFactCond2 \rangle]\!]_S$$

Proof

$[\![\langle sFactCond1 \rangle]\!]_S$

$= \dots$ Compile $sFactCond1$

$[\![\, E1(O, n) \,]\!]$

$$
\begin{array}{ll}
= & \cdots\cdots\cdots\cdots \text{Defn. of } E1(O,n) \text{ and } [\![\_]\!] \\
& (\mathcal{E}_{\mathrm{S}}\, n <^{\mathrm{CE}}_{\mathrm{S}} \mathcal{E}_{\mathrm{S}}\, 2) \fatsemi [\![SW(O,n)]\!] \\
= & \cdots\cdots\cdots\cdots \text{Defn. of } (\_ <^{\mathrm{CE}}_{\mathrm{S}} \_) \\
& \mathsf{cmdExp}_{\mathrm{S}}(\mathsf{trans}_{\mathrm{S}}(\_ < \_, 2), (\mathcal{E}_{\mathrm{S}}\, n, \mathcal{E}_{\mathrm{S}}\, 2)) \fatsemi \\
& [\![SW(O,n)]\!] \\
= & \cdots\cdots\cdots\cdots \text{Lemma 4.1 cmd. exp., where } (n,2) \notin (\_ < \_) \\
& \mathcal{E}_{\mathrm{S}}(\_ < \_)(n,2) \fatsemi \\
& [\![SW(O,n)]\!] \\
= & \cdots\cdots\cdots\cdots \text{Assumption that } n > 1 \\
& \mathcal{E}_{\mathrm{S}}\, \mathsf{false} \fatsemi [\![SW(O,n)]\!] \\
= & \cdots\cdots\cdots\cdots \text{Defn. of } [\![\_]\!] \\
& [\![\mathcal{E}_{\mathrm{S}}\, \mathsf{false} \fatsemi SW(O,n)]\!] \\
= & \cdots\cdots\cdots\cdots \text{Compile } sFactCond2 \\
& [\![sFactCond2]\!]_{\mathrm{S}}
\end{array}
$$

□

Step 4 – Selection of else conditional expression branch:

$$[\![sFactCond2]\!]_{\mathrm{S}} = [\![sFactIter]\!]_{\mathrm{S}}$$

Proof

$$
\begin{array}{ll}
& [\![sFactCond2]\!]_{\mathrm{S}} \\
= & \cdots\cdots\cdots\cdots \text{Compile } sFactCond2 \\
& [\![E2(O,n,\mathsf{false})]\!] \\
= & \cdots\cdots\cdots\cdots \text{Defn. of } E2(O,n,false) \\
& [\![\mathcal{E}_{\mathrm{S}}\, \mathsf{false} \fatsemi SW(O,n)]\!] \\
= & \cdots\cdots\cdots\cdots \text{Defn. of } SW(O,n) \\
& [\![\mathcal{E}_{\mathrm{S}}\, \mathsf{false} \fatsemi \mathsf{switch}_{\mathrm{S}}(\mathcal{E}_{\mathrm{S}}\, 1, ITER(O,n))]\!] \\
= & \cdots\cdots\cdots\cdots \text{Defn. of } [\![\_]\!] \\
& \mathcal{E}_{\mathrm{S}}\, \mathsf{false} \fatsemi \mathsf{switch}_{\mathrm{S}}(\mathcal{E}_{\mathrm{S}}\, 1, [\![ITER(O,n)]\!]) \\
= & \cdots\cdots\cdots\cdots \text{Lemma 4.9 cond. false} \\
& [\![ITER(O,n)]\!] \\
= & \cdots\cdots\cdots\cdots \text{Compile } sFactIter \\
& [\![sFactIter]\!]_{\mathrm{S}}
\end{array}
$$

□

Appendix D

Combined model extended example

This appendix contains the detailed working of the evaluation of the first object-method pair in Section 6.2.3. Here the *ExCel* method invocation starts by constructing an object-method pair. This takes thirteen steps from the initial state, which is illustrated in Figure D.1.

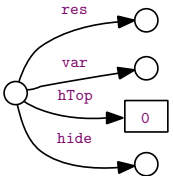


Figure D.1: Evaluate pair – Step 00 – Initial state

For convenience we now present another copy of the thirteen step program:

(decl x_1, x_2 in	(1)
(hide x_1, x_2 from	(2)
$\mathcal{E}_{\text{L}}$ *ExCel*	(3)
) ⨟	(4)
$x_1 :=_{\text{L}} \Pi_{\text{RES}}$ ⨟	(5)
(hide x_1, x_2 from	(6)
$\text{getMethod}_{\text{L}}$ `get`	(7)
) ⨟	(8)
$x_2 :=_{\text{L}} \Pi_{\text{RES}}$	(9)
$\Pi_{\text{RES}} :=_{\text{L}} (¿, ¿)$	(10)
$\Pi_{\text{RES}}.1 :=_{\text{L}} x_1$	(11)
$\Pi_{\text{RES}}.2 :=_{\text{L}} x_2$	(12)
)	(13)

The first step declares two local variables (x_1 and x_2), which are used to store the contents of the pair. This is illustrated in Figure D.2.

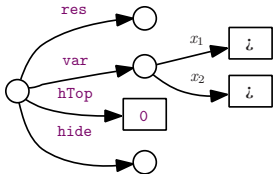


Figure D.2: Evaluate pair – Step 01 – Declare local variables

The second step hides the introduced local variables from the calculation of the first element within the pair. This is illustrated in Figure D.3.

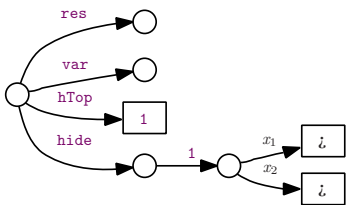


Figure D.3: Evaluate pair – Step 02 – Hide local variables

The third step evaluates the first element of the pair, which is the target object of the method invocation. This is illustrated in Figure D.4.

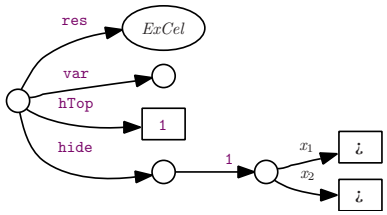


Figure D.4: Evaluate pair – Step 03 – Evaluated object

The fourth step restores the local variables following the evaluation of the first element of the pair. This is illustrated in Figure D.5.

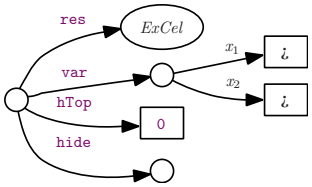


Figure D.5: Evaluate pair – Step 04 – Restore local variables

The fifth step stores the first element of the pair (i.e. the target object) in local variable x_1. This is illustrated in Figure D.6.

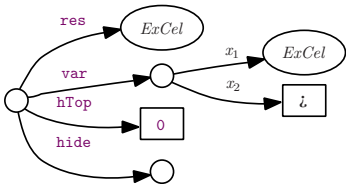


Figure D.6: Evaluate pair – Step 05 – Store object in first local variable

The sixth step hides the introduced local variables from the calculation of the second element within the pair. This is illustrated in Figure D.7.

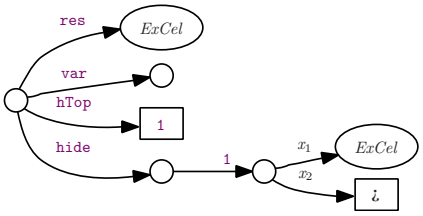


Figure D.7: Evaluate pair – Step 06 – Hide local variables

The seventh step evaluates the second element of the pair by looking up the method that is associated with the given label (`get`) within the target object. This is illustrated in Figure D.8.

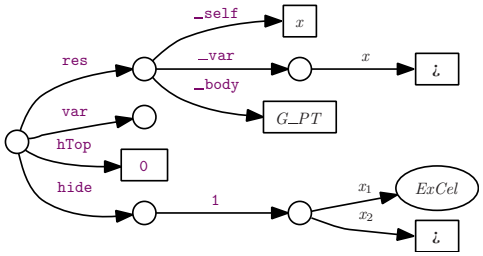


Figure D.8: Evaluate pair – Step 07 – Get method with label `get`

The eighth step restores the local variables following the evaluation of the first element of the pair. This is illustrated in Figure D.9.

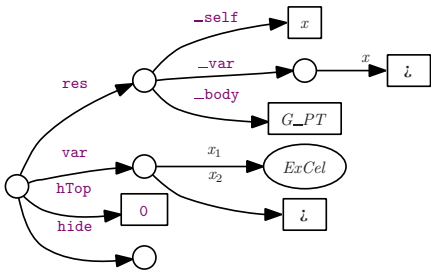


Figure D.9: Evaluate pair – Step 08 – Restore local variables

The ninth step stores the second element of the pair (i.e. the method to be invoked) in local variable x_2. This is illustrated in Figure D.10.

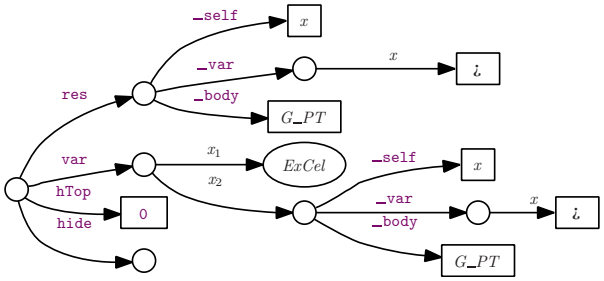


Figure D.10: Evaluate pair – Step 09 – Store method in second local variable

The tenth step creates an empty output pair, which will eventually contain the result of the object-method pair calculation that is being performed. This is illustrated in Figure D.11.

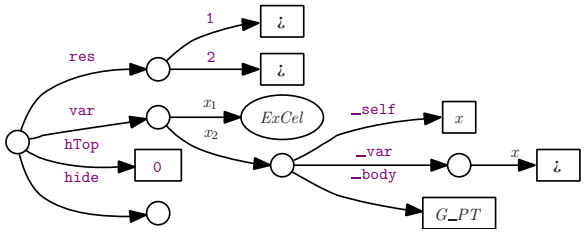


Figure D.11: Evaluate pair – Step 10 – Create an empty output pair

The eleventh step copies the target object to the first element in the output pair. This is illustrated in Figure D.12.

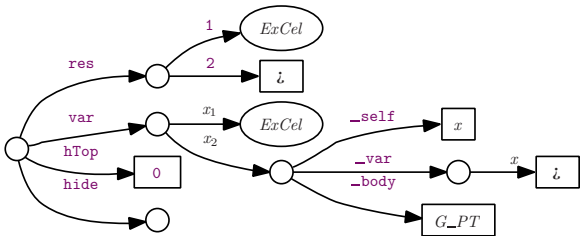


Figure D.12: Evaluate pair – Step 11 – Copy object to output pair

The twelfth step copies the method to be invoked to the second element in the output pair. This is illustrated in Figure D.13.

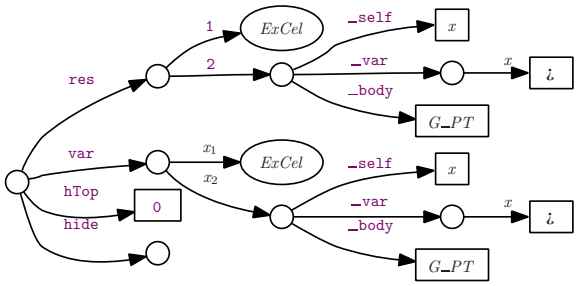


Figure D.13: Evaluate pair – Step 12 – Copy method to output pair

The thirteenth step removes the local variables from scope. This is illustrated in Figure D.14.

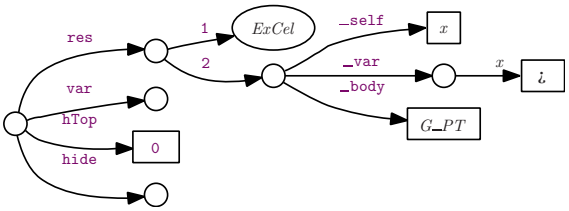


Figure D.14: Evaluate pair – Step 13 – Remove local variables from scope

List of Abbreviations

The entries in this list of abbreviations are either *general* or *product-specific*, where: the text for a *general* abbreviation is in lower-case words; and the text for a *product-specific* abbreviation is in capitalised words. The only exception to this rule is when an abbreviation contains another abbreviation, in which case the inner abbreviation is formatted as an upper case word.

ALTG Abstract Location Trace Graph

CCS Calculus of Communicating Systems

CLOS Common Lisp Object System

CORBA Common Object Request Broker Architecture

CPU Central Processing Unit

CSP Communicating Sequential Processes

FDR Failures-Divergence and Refinement

JVM Java virtual machine

λ-calculus untyped lambda calculus

$\lambda_{\rightarrow}$-calculus simply typed λ-calculus

LHS left hand side

OO object-oriented

π-calculus pi-calculus

rCOS Refinement Calculus of Object Systems

RHS right hand side

RMI Remote Method Invocation

SA Systems Assurance

ς-calculus untyped object calculus

TCOZ Timed Communicating Object-Z

$\mathcal{U}_{\mathrm{M}}$-design UTP constant–map-design

$\mathcal{U}_{\mathrm{R}}$-design UTP result–value-design

$\mathcal{U}_{\mathrm{S}}$-design UTP operand–stack-design

$\mathcal{U}_{\mathrm{L}}$-design UTP location–graph-design

UML Unified Modelling Language

UTP Unifying Theories of Programming