Game semantics for an object-oriented language

Nicholas Wolverson

Doctor of Philosophy
Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
2008

Abstract

This thesis investigates the relationship between object-oriented programming languages and game models of computation. These are intuitively well matched: an object encapsulates some internal state and presents some behaviour to the world via its publicly visible methods, while a strategy for some game represents the possible interactions of a program with its environment.

We work with a simple and well-understood game model. Rather than tailoring our model to match some existing programming language, we view the simplicity of our semantic setting as a virtue, and try to find the appropriate language corresponding to the model.

We define a class-based, stateful object-oriented language, and give a heap-based operational semantics and an interpretation in our game model. At the heart of this interpretation lies a novel semantic treatment of the phenomenon of data abstraction. The model closely guides the design of our language, which enjoys an intermediate level of expressivity between that of first-order and general higher-order store.

The agreement between the operational and game interpretations is verified by a soundness proof. This involves the development of specialised techniques and a detailed analysis of the relationship between the concrete and abstract views. We also show that definability and full abstraction hold at certain types of arbitrary rank, but are problematic at other types.

We conclude by briefly discussing an extended language with a control operator, along with other extensions leading to a possible core for a more realistic programming language.

Table of Contents

List of Figures

Acknowledgements

Firstly I would like to thank my supervisor, John Longley, for his patience, support and guidance, and particularly his great enthusiasm towards my work. Without his conviction this thesis could not have come to be.

I am grateful to Gordon Plotkin and Ian Stark for serving on my progress review panels, and for their useful comments and discussion. I would like to thank everyone at the Laboratory for the Foundations of Computer Science for providing a pleasant, knowledgeable and friendly environment to work in, and in particular my occasional office-mates and badminton partners Ben Kavanagh and Laura Korte.

I would also like to thank my parents for their support. In particular I thank my mother for giving me some opportunities to truly unwind, and to my father I look forward to answering the question "so what will that let him do?". Many thanks are due to Andrew Irish for putting up with me for the last few years, particularly in stressful times, and for believing in me. I would also like to thank Liz Ashton, and various friends for forcing me to relax and have some fun.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Nicholas Wolverson*)

Chapter 1

Introduction and Motivation

A high-level programming language provides an abstraction from the realities of machine code running on some particular hardware. Of course, a programmer needs to know how a given program will behave, and the behaviour of a reference implementation on some particular hardware is less than useful. Traditionally a programming language is defined by describing in natural language the intended behaviour of the various constructs; as the language grows more powerful a language definition often becomes a tome specifying the language in a kind of legalese English—and even compiler writers may not interpret the definition correctly or consistently.

An operational semantics precisely specifies program behaviour, whether as a reference implementation on an abstract machine or more directly by structural syntactic manipulation of programs. This removes the imprecision while retaining a clear link to practical implementations, but does not provide a convincing account of what a program *means*. A denotational semantics defines the meaning of programs more directly by interpreting them in some mathematical structure. As well as a better understanding of a language one is then able to bring mathematical tools to bear on it, for example to develop program logics and verify their correctness. Those programmers not interested in formal program correctness proofs can benefit from type systems and other tools arising from the semantic analysis, and gain some confidence that their programs are correct or at least well-behaved in some sense.

A mathematical study of semantics can also suggest new avenues in programming language design, or areas in which better choices can be made. A programmer will appreciate the addition of powerful or well-chosen language features, while the theory benefits from the elimination of undesirable “rough edges” and general cleanliness of design. Standard ML, for example, grew out of work in denotational semantics, although the language is defined operationally [65], and many popular languages have taken inspiration from the field. The Eriskay project aims to design a language inspired by recent ideas in game semantics [59], and this thesis forms part of that work.

To be useful a model must be both easy to reason about and a good match for the operational semantics of the language in question. At the very least one wants the model to be *computationally adequate*, meaning that when the model equates two programs, they are operationally indistinguishable. Ideally the converse property of *full abstraction* also holds, so that the model does not draw undue distinctions between programs.

Much of the early work in this area focused on the prototypical, typed functional language PCF. Plotkin showed in [69] that the continuous model of PCF is not fully abstract, but is instead fully abstract for PCF extended with a “parallel or” operator. Later work aimed at PCF produced the *sequential algorithms* model [18], which is instead fully abstract for PCF extended with the control operator `catch` [27, 28]. In a similar spirit, *game models* succeeded in giving a fully abstract semantics for PCF [11, 43], retaining the notion of sequential computation but imposing restrictions to exclude the control features.

Two approaches to achieving a fully abstract semantics are described in [33]:

- Vary the language to fit the desired model, or
- Vary the model to fit the desired language

The common situation is to have an over-precise model, in which case these options are to add distinguishing operations to the language, or remove distinctions from the model. Abramsky and Ong investigate both approaches in the context of the lazy lambda calculus [14]—there they describe these as *expansive* and *restrictive* respectively.

Game semantics has enjoyed success in giving full abstraction results largely because it gives a rather precise model of sequential computation which is amenable to the imposition of various restrictions. As well as the result for PCF, fully abstract game semantics have been given for languages with continuations [52], Idealised Algol [7], and a language with ML-style references [10]. However, the definitions of the appropriate game models are frequently quite technical.

In this thesis we take a rather more model-driven approach. Given a simple model of games, we look for a suitable language. We feel our game model is very natural and carries a persuasive intuition. We do start out with a general idea of the kind of language we want—namely "object-oriented"—but the particular language we are led to is perhaps not an obvious choice. However, programs of our language can be quite expressive, while obeying certain desirable semantic properties. We view this as a virtue of a semantically driven approach, and of its ability to guide and inform language design.

1.1 Objects and strategies

In this thesis we study a small object-oriented calculus. Object-oriented programming has a surprisingly long history: techniques first used in the early 1960's [50] were incorporated into the programming language Simula 67 [35], and further developed in Smalltalk [38, 50]. The popularity of C++[71] brought OOP to an even wider audience, and this trend has continued more recently with languages such as Java[41]. Aside from the practical popularity or methodological merits of the OO paradigm, there is considerable intrinsic theoretic interest in various aspects of OOP—not least, the challenge of giving a suitable account of the semantics.

An *object* is an entity which accepts and responds to a number of *messages* from its environment. These messages are drawn from a fixed set (as according to the object's type) and may include various data in query or response, either primitive values (such as natural numbers) or other objects (again constrained by the object's type). Crucially an object need not maintain a fixed behaviour over time, but may vary its response to repeated messages as according to some internal *state*. This state is not externally visible, but instead only manifests itself in the object's behaviour—two objects with identical external behaviour can be considered to be equivalent, regardless of implementation details. We refer to this idea as *data abstraction*.

Figure 1.1 represents the behaviour of an object with a single `int` $\Rightarrow$ `int` method. Labelled edges represent messages received by the object from its environment (incoming method calls), while nodes represent the response (return value). Clearly any object with this external interface can be represented by such a tree, with no reference to how its behaviour was generated.

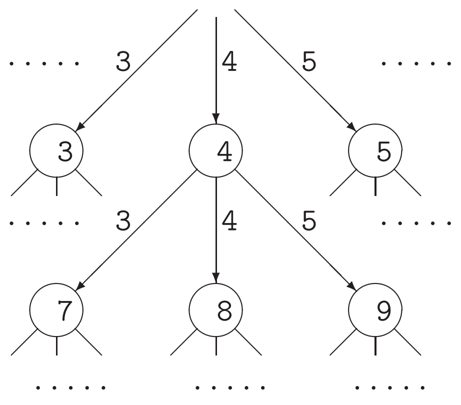


Figure 1.1: Interaction with an object with `int` $\Rightarrow$ `int` method

This decision tree can also be regarded as a *strategy* for a *game*. This "number swapping game" is lacking in rules, other than that the moves are integers—in particular there is no notion of winning or losing. However, it is clear that a strategy for this game can represent an object, while the game itself represents the type of that object. More complicated types correspond to more complicated games, where a method call and response are no longer simply represented by two successive moves but instead involve a longer sequence of interaction. The essential property remains, however, that a strategy for this game represents the externally observable behaviour of some object, abstracting from any implementation details. The interaction with this strategy represents a *reactive* notion of computation as a process which, rather than being a mathematical object fixed in time, evolves according to input received, and varies its output accordingly. This is a good fit for the object-oriented way of thinking, where objects are similarly viewed as reactive entities.

1.2 The language

In this thesis we present an object-oriented language inspired by this correspondence with games and strategies. We start with a particularly simple game model, which has been known of for some time. With this model in mind, we define a small language drawing on the functional programming and class-based object-oriented programming traditions. We include simple functions in the style of the λ-calculus, and objects which are viewed simply as a collection of functions; we allow both functions and objects to be defined recursively. We do not include

$$\begin{array}{lll}
\textbf{class} & \{ & \mathit{set} = \lambda\langle s,p\rangle\colon (\iota\otimes\iota)\otimes(\iota\otimes\iota).\ \langle p,p\rangle, \\
& & \mathit{getx} = \lambda\langle s,z\rangle\colon (\iota\otimes\iota)\otimes\iota.\ \textbf{let}\ \langle x,y\rangle\ \textbf{be}\ s\ \textbf{in}\ \langle s,x\rangle, \\
& & \mathit{gety} = \lambda\langle s,z\rangle\colon (\iota\otimes\iota)\otimes\iota.\ \textbf{let}\ \langle x,y\rangle\ \textbf{be}\ s\ \textbf{in}\ \langle s,y\rangle \\
& \} & :\textbf{Class}\ \langle\iota\otimes\iota;\ \mathit{set}\colon(\iota\otimes\iota)\to(\iota\otimes\iota),\ \mathit{getx}\colon\iota\to\iota,\ \mathit{gety}\colon\iota\to\iota\rangle
\end{array}$$

Figure 1.2: Sample class definition—a point class

a notion of classes and associated operations in our core language, but find it more convenient to present these as derived constructs. Instead, at the core of our language is an operation embodying the idea of data abstraction as discussed above. The **constr** operation constructs a stateful object from a functional implementation plus an initial state—in the case of an object with a single method of type $\tau \Rightarrow \tau'$, and a state of type σ, **constr** has a type of the following form:

$$\textbf{constr} : (\sigma \times \tau \Rightarrow \sigma \times \tau') \times \sigma \Rightarrow (\tau \Rightarrow \tau')$$

Figure 1.2 presents a small example of our language, making use of the derived constructs. The displayed expression defines a class representing a 2-dimensional point, and is annotated with its type—note that ι is the type of integers (we also use ι as a dummy type in place of a true *unit* type). Methods accept their initial state as an additional parameter s, and return the updated state as the first component of their result.

As well as the interpretation in our game model, we give an operational semantics. Here we make explicit the notion of the *heap*. Proving our game model sound with respect to this operational interpretation requires some considerable work, the difficulty essentially being to reconcile the representation of objects as reactive strategies with that of a graph-structured heap with objects explicitly represented as their implementation and state. Once we identify the property relating these two views, the soundness of our semantics comes down to the correctness of the data abstraction operation.

We show that strategies at a certain class of types are definable in our language, also giving a limited full abstraction result. Unusually, the types in question are not limited in rank, but by their structure. We also explain why it does not seem to be the case that definability holds for all types in our language. Finally, we introduce a control operator, the addition of which we suggest would extend our definability result to the more permissive (unbracketed) setting.

In the rest of this chapter, we discuss our interpretation of object-oriented programming, some approaches to semantics and existing work on semantics for OO languages. We cover related work in game semantics before concluding with an overview of the rest of the thesis.

1.3 Characterisation of language

To give a general idea of the relation of our language to other work we shall state our response to certain stylistic choices (these points are discussed in greater detail by Bruce [23], with some similar conclusions). We characterise our language as:

- Class-based rather than object-based. As in C++, Java and C# we consider objects to be created from classes and do not include features such as method update from object-based languages in the style of Abadi and Cardelli [2]. Partly this is guided by our intended semantics, but largely we feel that class-based languages are of wider interest. The class-based nature of the language presented in this thesis is slightly blurred by the fact that for (our) convenience we do not include classes as primitive. However, a more powerful and practically oriented language building on these ideas would include native class constructs (as in Eriskay [59]).

- Stateful rather than functional. Objects in our language are stateful entities as in most Object-Oriented languages, in contrast to pure functional languages. This stateful behaviour is pervasive in nature, and can be thought of in terms of a global heap, although our game semantics gives a different interpretation.

- Functional or type-theoretic style rather than procedural. Our language promotes a higher-order style of programming as enjoyed in the functional programming community. While we draw a distinction between classes and object, we give classes a first-class status, allowing them to appear in arbitrary expressions rather than as a list of definitions (of course the latter style is possible too). The syntax **class** $\{\ldots\}$ in Figure 1.2 denotes an expression representing the defined class, which can then be passed to functions etc. as well as simply instantiated to create an object of that class.

Particularly once a more fully featured type system is added, this supports a highly expressive factorised style of programming.

- Static rather than dynamic typing. We will work in a setting of strong static typing. For the purposes of this thesis we are not generally interested in questions of type inference—formally we will use a simple explicit type system, but we will often be more lax where types are not relevant.

- Interface types rather than classes as types. We consider the type of an object as specifying its external interface, rather than the class of its origin. For example an object of the class defined in Figure 1.2 would have type

$$\mathbf{Obj}\ \{set\colon (\iota \otimes \iota) \to \iota,\ \ getx\colon \iota \to \iota,\ \ gety\colon \iota \to \iota\}$$

  This decouples the notions of inheritance and subtyping (cf. [30]): while in our setting a subclass is always a subtype, the converse need not be true. It is also more in keeping with a behavioural view (and thus abstraction). If two objects constructed in different ways should exhibit the same behaviour, then we should not distinguish between them, and in particular they should have the same type. However, this does preclude the inclusion of *strong binary methods* [24], where methods are granted privileged access to the internals of arguments of the same class.

- Structural rather than nominal types. This rather goes hand in hand with the above. Regarding types as giving the public interface for an object (rather than a named type describing it, as interface types do in Java) means that there is no artificial type distinction between any two objects which support the same means of interaction, and thus have the same potential behaviour. In any case the implicit existence of types for all interfaces allows for the proper operation of subtyping (since the intersection and union of two object types exist).

There are some other concerns more particular to our language than object oriented languages in general.

Firstly, we use a *linear* type system. In particular our function types are linear, while we consider ground types and object types to be reusable. This presentation is to some extent a matter of convenience, but it is useful when we consider the extension of the language with a continuation operator, where it is important that the continuation is linearly used.

$$
\begin{array}{l}
\textbf{extend } c \textbf{ with } (\varsigma)\ \{ \\
\qquad\qquad add = \lambda\langle s,p\rangle. \\
\qquad\qquad\quad \textbf{let } \langle s,x\rangle \textbf{ be } \varsigma\cdot getx\langle s,0\rangle \\
\qquad\qquad\quad \textbf{in let } \langle s,y\rangle \textbf{ be } \varsigma\cdot gety\langle s,0\rangle \\
\qquad\qquad\quad \textbf{in } \langle\langle x+p\cdot getx\ 0, y+p\cdot gety\ 0\rangle,\ 0\rangle \quad \}
\end{array}
$$

Figure 1.3: Extending the class c of Figure 1.2

The interpretation of the **constr** operation as a strategy in our game model dictates a type system which restricts the possible object implementations to certain well-behaved ones. Acceptable method implementations are those which behave in an *argument safe* fashion. Roughly speaking, this means they do not store a pointer to an object received as an argument in the object's state, although they are free to interact with such objects before returning a value, and store ground-type values in the state.

An immediate consequence of this restriction is that the heap implicitly created during program execution never contains cycles. A more interesting consequence is that our language permits the definition of ground-type reference cells, but not reference cells of any higher type. In fact the level of expressive power comes strictly between these two, since certain *local* uses of higher-type state are supported.

Lastly, with respect to control features we take something of a mixed stance. We introduce games suited to both languages with and without such features, but for the majority of this thesis we work in the more restricted setting lacking such operations. However, we conclude by introducing the continuation operator mentioned above, and discussing the extension of our work to that setting.

Finally, we give a brief overview of some features supported by our language. The language supports inheritance using an **extend** expression (a derived form). An example of this is shown in Figure 1.3, where the metavariable c should be replaced by the class of Figure 1.2. This class extends c by adding an *add* method to add another object of the same type (thought of as a vector).[1] Here we omit required type annotations for brevity; also note that 0 is used as a dummy value as our language omits a unit type.

[1]Of course the type is not precisely the same—the supplied argument need not have a *set* method, and indeed may not be derived from the same class.

$$\begin{array}{ll}\textbf{extend } c' \textbf{ with } (\varsigma) & \{ \\ & \quad \mathit{gety} = \lambda\langle s, z\rangle : (\iota \otimes \iota) \otimes \iota.\ \langle s, 0\rangle \\ & \}\end{array}$$

Figure 1.4: Extending the class c' of Figure 1.3, overriding two methods.

As mentioned above inheritance creates subtypes, but is not the only way to do so. It should be noted that the language of this thesis does not support the addition of fields in subclasses, but as described in Chapter 7 the addition of this feature is unproblematic.

The language also enables (mutual) recursion via a *self* parameter (ς in Figure 1.3). As in the example above, such recursive calls are made via the internal functional interface of the class, while normal calls from the environment are not, as in the case of the method call $p \cdot \mathit{getx}\ 0$ on the argument object.

Finally, the language supports *virtual methods*. In fact all methods are taken to be virtual methods, unlike C++ where functions must be explicity declared to be `virtual` or Java where methods are by default virtual but may be declared to be `final`. A (somewhat artificial) example is given in Figure 1.4, where we extend the *getx* and *gety* methods to effectively make the y coordinate constant. The important thing to note here is that the *add* method uses *getx* and *gety* rather than inspecting the state directly, and so when we override the *gety* method *add* uses the old rather than the new version, and consequently the y coordinate is simply set to that of its argument (in the case that the argument is of the new class, this will mean set to 0). This example is terribly contrived, but virtual methods are in fact a key component of object-oriented programming.

1.4 Style of semantics

For a given language one can take a variety of approaches in order to build a semantic model. We shall characterise our approach by discussing a number of properties a model may have before discussing the relation to other work. These properties are somewhat interdependent, in that a random selection would probably make little sense, but they represent qualities we consider important about the work presented here.

- Imperative rather than functional. We do not attempt a functional coding of object update (as in e.g. [68]), instead modelling imperative updates directly. This allows us to stay within a simple typing framework without too much restriction on behaviour. There is a little complication in that the stateful behaviour of our objects is *specified* in a functional manner, but the external view is of imperative update.

- Semantic rather than syntactic. We give our semantics directly rather than by translation into some other language or type theory. There is a conceptual benefit to this directness: the interpretation of a term can be more readily understood to give it meaning, and our model provides a more satisfying explanation of our language. There are benefits to translation into some well-understood theory, but in our case the direct analysis proves interesting. In any case one would wish to give a fully abstract translation into a language with a fully abstract semantics, and we are not aware of a suitable target for the language described here.
- Compositional/structural rather than whole-program. Our semantics is given in a compositional way according to program structure, rather than for a whole program at once. This allows for the use of familiar "mathematical" reasoning principles, such as substitutivity.
- External rather than internal view of object behaviour. As discussed above we shall model objects according to their external behaviour. Any state owned by the object (i.e. updateable fields) gives rise to a certain behaviour of the object; the denotation of the object will reflect this behaviour, but will not expose the object's internal state (or even the type or existence of such).
- Intensional rather than extensional. Rather than attempting to give a direct account of the extensional behaviour of programs, perhaps at some complicated functional type as in the monadic approach [67], we give a slightly more intensional semantics. As is well known the intensional nature of game semantics paves the way to full abstraction results.

1.5 OO semantics survey

There has been much work already on the theoretical foundations for object-oriented languages. Some has concentrated on type systems and their safety (for example Featherweight Java [46], Middleweight Java [19], Classic Java [36]), while other work gives semantics to various object and class-based calculi, either directly (and often operationally) or by translation. Bruce gives a good summary in [23]; here we concentrate on some work which seems especially relevant to our own.

1.5.1 Object encodings

Various encodings of objects have been proposed—a good summary is given in [25]. In that paper four existing encodings are described in the context of System $F^{\omega}_{<:}$ with existential types, recursively defined types, recursive functions and records. The first and simplest, as introduced by Cardelli [26] and widely studied [66, 49, 70, 29], represents objects by *recursive records*—an object is simply a record with an identifier (usually called *self*) bound recursively within it. The second simple encoding uses existential types to hide an object's internal state [68], while the other two encodings combine recursion and (bounded) existentials. While these encodings are informative, and the first is relevant to this work, much work goes into understanding *functional* encodings, and in any case we prefer a more direct approach (based on recursive records).

The recursive record approach can be split into two camps, *early self binding* where a fixed point is taken at the point of object creation, and *late self binding* where a fixed point is taken at the point of method invocation. Abadi and Cardelli show that either approach fails to correctly implement method update as found in object calculi [1], where an existing object is extended with a new method implementation; recursive records are thus more often used in class-based systems. Here a class is modelled by a term $\lambda \mathit{self}.\ \{\ldots\}$, from which objects are created by taking the fixed point. *Open recursion* is implemented by taking advantage of the indirection via the *self* parameter. If when extending a class one replaces a method m_1, any method m_2 which refers to $\mathit{self} \cdot m_1$ will refer to the new version of m_1 when the fix-point is taken.

A class-based calculus is presented in [22], from which we take some inspiration. This language has functions, ML-style references and classes (as first class expressions); objects are recursive records with early self binding. The authors suggest that objects with imperative update together with a simple type system "achieve a reasonable trade-off between expressivity and simplicity". The semantics given is essentially via a translation into a fragment of ML with references (Reference ML), which has an operational semantics based on *heaps*. Work on game semantics for references as discussed below is somewhat relevant, in that one could compose an encoding of classes and objects with the game semantics of the calculus of references. However, as we have discussed we are interested in a more direct approach.

1.5.2 Object calculi

In their book [2], Abadi and Cardelli introduced their influential object calculus. They give a small and elegant object calculus, intended to be to object-based computation what the λ-calculus is to functional computation; the calculus is given in immutable and imperative untyped forms, and to these increasingly complex and powerful type systems are added. In brief, the calculus allows the definition of objects as a collection of methods with self-binding, with the operations of method update (fields being considered a special form of methods) and application. They give their calculus a primitive semantics based on *self-application*, where a method call is expanded to the method body with the object substituted for the method parameter.

In [39] Gordon and Rees investigate full abstraction via bisimilarity for the first-order stateless object calculus from the above. They derive a labelled transition system (LTS) from the reduction rules of the calculus, and show that bisimilarity according to this LTS coincides with contextual equivalence.

Gordon and Hankin introduce in [40] a concurrent extension of the mutable object calculus, with concurrency operators derived from the π-calculus, and mutex-based synchronisation. They give a structural congruence based reduction semantics in the style of those for the π-calculus, and show this is equivalent to a structural operational semantics defined using stores, threads and configurations.

1.5.3 Trace semantics

A variant of the concurrent object calculus is further studied by Jeffrey and Rathke in [47]. They give a *trace semantics* which they show is fully abstract with respect to contextual equivalence (in this concurrent setting the notion of *may testing* is used). Figure 1.1 can be interpreted as showing the set of *traces* of the program depicted there. Here a trace is the interaction observed at the boundary between a program (or component) and its environment (essentially a "game play"). Given the basic set of reduction rules for the calculus, an LTS is constructed containing named reductions for the observable actions of incoming and outgoing method calls and returns; the denotation of a component is the set of traces it admits. While in the case of the simple object above this is very similar to the strategy given by the game approach, at more complex types the two views diverge. This work is extended to a core Java language in [48].

Moving away from object-based languages, in [4, 3, 5] this concurrent calculus is modified to include the notion of *classes*, and again a fully abstract trace semantics is given. Here classes are named entities regarded as the generators for objects, and method update is removed. A large part of this work relates to the notion of observation in the class-based setting. A program's environment may include both objects and classes; observable behaviour now includes the creation of objects by the program from environment classes (and dually by the environment from program classes). The environment can observe all interaction with an object created from an environment class, but if these objects are created from different classes, they will initially have no way of referring to each other. For this reason the potential connectivity of these objects must be tracked, since there are certain observations which the environment cannot legitimately make (e.g. the order of events observed by two disconnected objects).

The notion of connectivity here might suggest a deficiency of our approach, that perhaps we do not take classes seriously enough. However, in a sense our setting corresponds to the reality of languages such as Java, where classes are not just object-generators but can contain their own (non-instance) state, meaning that objects created from classes in the program environment are always potentially interconnected. In this aspect our language is perhaps more expressive (and consequently harder to reason about than that of [4]); on the other hand, we cannot model true concurrency.

Unfortunately, the trace semantics is not a *compositional* one—the denotation of a composition of two program fragments is not a function of their individual denotations. This is problematic for reasoning about programs, as well as understanding them. On the other hand, our game semantics is compositional, but as yet we cannot handle the languages presented in [4, 3, 5].

1.6 Game semantics

The use of games in programming language semantics have arisen from work in logic and from other work in semantics. We give an incomplete summary and brief introduction here—for more detail the reader should consult [31].

1.6.1 Logic

The connection between games or debates and logic has in a way been implicit since logic has been studied, but became formalised in the study of constructive mathematics. Lorenzen [62] gave a semantics for the intuitionistic predicate calculus in terms of dialogue games. The proponent (or player) (P) wishes to verify a formula he has proposed as valid, while the opponent (O) wishes to refute it. Propositions are interpreted as games, and connectives as operations on games; a given dialogue consists of a sequence of moves (moves being e.g. to attack or defend a chosen sub-formula), and a dialogue is won by one player if they play a move to which the other cannot respond. A strategy is a function determining the next move to be made, and then a formula is said to be valid if there exists a winning strategy for proponent.

While there is a good match between intuitionistic logic and computation, a key influence on game semantics has been Girard's linear logic [37]. Linear logic introduces explicit structural rules controlling reuse, so that propositions can be thought of as resources which cannot in general be copied or discarded. Then a distinction is drawn e.g. between two products $A \otimes B$ and $A \& B$, the first of which representing both A and B, and the second representing a choice of A or B ("I have both, but I'm only going to give you one"). The exponential $!A$ indicates a reusable version of A, that is a proposition which can be used multiple times. Therefore the contraction rule applies to $!A$, allowing it to be copied. It is then possible to translate intuitionistic logic into linear logic, where we replace $A \Rightarrow B$ with $!A \multimap B$, with $\multimap$ being the linear logic implication.

Blass gave a game semantics for linear logic [21], opening up a correspondence between games and linear logic which appears often in later work. Abramsky and Jagadeesan gave a game semantics for linear logic with the "mix" rule [6] which improved on Blass's games, making them form a category. Here the notion of games are a little different; a game A is defined to include plays in which either Opponent or Player start, and $A^\top$ negates A by switching the O/P labelling. The tensor $A \otimes B$ then *imposes* the condition that only Opponent may switch between A and B, unlike our games in which this condition arises automatically out of the interleaving of play in A and B. Then the multiplicative disjunction (or *par*) operation $A \parr B$ (which is less easy to understand intuitively, but introduces a kind of dependency between A and B) is defined as $(A^\top \otimes B^\top)^\top$, $A \multimap B$ is defined as $A^\top \parr B$, and *history-free* strategies are defined to be as in Section 2.1.1. The category of such games and history-free strategies gives *full completeness*[2] for the interpretation of multiplicative linear logic with the "mix" rule (Hyland and Ong subsequently gave a fully complete semantics without this rule [44]).

1.6.2 Sequential Algorithms

A second area of work influential in the development of game semantics was that of sequential algorithms on concrete data structures [18]. Lamarche reformulated the sequential algorithm model of PCF in terms of games [55] (as described in [64, 32]). The fundamental idea here, and in all later work in game semantics, is to model a program by a strategy describing its interaction with the environment, as described in Section 1.1. This model is fully abstract for PCF+`catch` (or SPCF) rather than plain PCF [27, 28], but in many ways is closer to our simple game model than the later ones described below. Games are played "on trees"—there is a set of moves partitioned into Player and Opponent moves, with *plays* consisting of alternated sequences of moves starting with Opponent, and this can be viewed as a forest with layers alternately consisting of Player and Opponent nodes. A *strategy* gives the Player response for any Opponent move, or in other words consists of a sub-tree of the game tree branching when it is opponent's turn to move. This model also possesses products and function spaces as we describe in Chapter 2; these simple games are also described by Abramsky [8] and Hyland [45]. The significant difference occurs regarding reuse; the exponential described by Lamarche is a *backtracking*, *non-repetitive* one. When playing in $!A$ it is always possible to "back up" to an earlier position in the game tree of A, and explore a different branch, but there is no sense in which a question can be asked twice, as

[2]Full completeness is to logics as definability is to programming languages.

an answer is given once and for all.

In contrast, the exponential introduced by Hyland [45] was a *repetitive* one, so that $!A$ represents essentially as many copies of the game A as required. Furthermore, it is *non-uniform* in the sense that a strategy for $!A$ need not behave the same in each copy of A. We introduce this exponential in Chapter 2, and use it throughout this thesis. The idea that the same category of games could be endowed with more than one reasonable exponential was to become significant in later work: Melliès gives a detailed comparison of the various choices possible [63].

1.6.3 PCF

One of the most celebrated early successes of game semantics was to give fully abstract models of PCF. Abramsky, Jagadeesan and Malacaria introduced a fully abstract game model for PCF [11]. Their games are defined similarly to the above, but tag each move as either Question or Answer, and impose the *bracketing condition* that each answer corresponds to the last unanswered question. Strategies are history-free. In contrast to the exponential described above, a *repetitive* ! is used so that a play of $!A$ is an interleaving of plays of A—$!A$ is essentially an infinite tensor product of copies of A. An equivalence relation is defined over strategies to make the exponential *uniform*, meaning a strategy for $!A$ must behave the same in each copy of A.

Independently of Abramsky et al., Hyland and Ong developed a fully abstract game semantics for PCF [43]. This paper introduced the influential notion of *arena games*, where games are no longer considered simply as trees. Instead, a game defines an arena, a tree specifying which moves *justify* other moves (i.e. enable them to be played), and the plays of the game are mechanically generated from this relation. Moves are thought of as carrying a *justification pointer*—a reference to the justifying move earlier in the play. The original presentation does not make use of a linear decomposition of $A \Rightarrow B$ as $!A \multimap B$, but there is an equivalent presentation in those terms [13]; the game $!A$ then simply consists of interleaved plays of A. The justification pointer structure allows different copies of A to be distinguished as in the AJM exponential. Rather than history-freeness and uniformity, the condition of *innocence* is imposed on strategies (as well as *well-bracketing*); a strategy may only act on information contained in a certain *view* of the play so far as determined by the justification-structure of the play.

1.6.4 Control and state

A program of subsequent work investigated the effects of removing the various constraints on this model, a task which the arena-based formulation seems particularly suited for. Laird investigated removal of the bracketing constraint, giving fully abstract models of languages with control features (such as `call/cc`) [52, 53]. Abramsky and McCusker gave a fully abstract semantics for Idealised Algol [7] by allowing strategies to be non-innocent (i.e. knowing). They introduce a knowing strategy $cell_X$ to represent a store cell with read and write operations holding a value of type X (where X is a basic datatype, i.e. a set). Its stateful behaviour is permitted because there is no constraint forcing such strategies to behave in a *uniform* way.

Abramsky, Honda and McCusker then gave a semantics for a language with ML-style higher-order references [10], which seems particularly relevant to our work. Their language is call-by-value, as ours is (but unlike Idealised Algol), and involves the creation of state of general type, as one might expect objects to have. The switch to a call-by-value setting used an existing technique [12] (and we shall do the same). However, the treatment of general references require a more liberal definition of games. We will discuss this in greater detail later, but in short the construction is such that a play of a game $A \otimes B$ when projected onto the component game A need not be a valid play of A. For the game $A \otimes B$ the usual Opponent/Player alternation property holds, but viewed at A it does not. To put it another way, type constructors are defined on arenas, and it does not make sense to think of them as operating on the generated game tree, unlike in the arena games described above. This relaxation of the rules is necessary to implement the higher-type reference cell, but makes the intuition behind these games a little less clear (and certainly adds complexity to the definition).

1.6.5 Names

Recent work has given game semantics for languages involving *names*, starting with the ν-calculus [9]. This work builds games on top of Fraenkel-Mostowski set theory, putting names into the heart of the construction. Laird gave a semantics of "local names and good variables" [54], the idea being to use nominal set theory to eliminate *bad variables*. These represent a defect in the semantics of general references in the previous section—there are strategies at reference type which do not represent actual reference cells, so to obtain full abstraction the language has to add some “junk” in the form of bad variables.

Some account of names must be made to interpret general object-oriented programming, in order to support circular references and object equality. However, we suggest that a direct behavioural interpretation of simple objects is interesting in its own right, and can provide a stepping stone to the later investigation of these more general features. We discuss this issue in Section 7.1.

1.6.6 Choice of game model

The game model of AHM [10] gives a good point of reference from which to discuss our approach and the game model we have chosen. There are two important differences to note, namely the approach to modelling stateful behaviour, and the game model in which this modelling takes place.

Stateful behaviour appears in [10] in the form of ML-style reference cells, implemented in the game model by a *cell* stratagy as also used in the earlier work on Idealised Algol. A simple translation of objects is given: fields are interpreted as reference cells, which are bound within a record of functions representing the methods of the object.

This approach does offer a convenient way to represent object-oriented behaviour. However, the AHM approach lacks a compositional account of data abstraction, in contrast to our use of the **constr** operation, and the corresponding *thread* morphism and characteristic properties. We contend that this process of data abstraction is one crucial aspect of object-oriented programming (there are of course other aspects which we do not other attempt to address), and deserves to be studied in its own right. While our approach relates the implementation of an object as state-transformer and the resulting stateful object, this is not visible in the AHM approach, the stateful behaviour instead residing in the *cell* strategy. Our approach can be carried out in the AHM model, and given this it is perhaps more natural to take this data abstraction operation rather than the store cells as primitive, being more closely tied to the object-oriented concepts being modelled.

While our interpretation could be given in the model of [10], it is interesting to discover that the data abstraction approach works in the "weaker" setting of our less powerful game model. Ultimately, one might wish for a more general axiomatic treatment of data abstraction, but for now we explore a particular weaker setting which gives "just enough" expressive power to investigate these ideas—our model has the advantage of being fairly minimal, in the sense that we could not present the interpretation of the **constr** operation in a less powerful model, leading to a more general result.

We will now briefly outline the technical differences between the two models. Some of the apparent differences have only minor significance or are presentational, for example the formulation in terms of Hyland-Ong style arena games with justification pointers, and here we concentrate on those which are more relevant. As mentioned above, the AHM games are "non-alternating", in the sense that a play of a compound game such as $A \otimes B$ need not be a valid play when projected to one of the component games A or B, as the requirement for plays to alternate between Opponent and Player is only imposed on the overall game and not the constituent components. This is achieved in [10] by defining games using an enabling relation which in turn generates valid game plays. We omit details here, but the key point is that constructions such as $\otimes$ operate on the enabling relation rather than the generated move-trees, and so for example plays in $A \otimes B$ need not arise from an interleaving of valid plays of A and B. The overall result of this is that a given game in the AHM model may admit more possible plays than a game in our model. The particular additional behaviour permitted is somewhat subtle, but it should be noted that the *cell* strategy mentioned above crucially uses plays of this form to model general higher-order store.

We believe our model is natural and inherently rather appealing. It is a simple and particularly intuitive model; while the AHM model is still relatively simple in technical terms, the intuition is rather more subtle. The fact that the AHM games are not just move trees does obscure the intuition behind the various constructions somewhat. In a sense our Lamarche-style games are more "extensional", in that the various type constructors operate on the games themselves rather than generators for these games.

While the category of Lamarche games is well known and the exponential we use has been studied previously (by Hyland [45]), questions about the expressive richness of this model have not been raised. In particular the stateful behaviour which can be expressed is rather subtle and unusual, as is shown in the remainder of this thesis, and seems worthy of study in its own right, regardless of any justification related to object-oriented programming.

In fact, although we have said that our setting is fairly minimal, we can support a surprising expressive power without having to resort to the additional power of the AHM model—many higher-order store phenomena arise even in our weaker setting. In fact an even larger proportion of the expressive power of [10] can be achieved in the world of Lamarche games with some more work, namely the use of a more powerful exponential and the encapsulation techniques of [61].

Although our setting allows considerable expressive power, it is of course less expressive than the AHM model. Not all stateful behaviour present in their model can be expressed in ours, and in particular the store cells of arbitrary type that form the basis of their language cannot in general be expressed in ours.[3] There is in fact a trade-off here: more expressive power means that there is more information in the denotation of an expression, and correspondingly the notion of observational equivalence is finer, and reasoning becomes more subtle. A strategy in our model should be easier to reason about than a strategy in the AHM model, and for this reason it seems desirable to give an interpretation in the simpler model where possible.

Finally, there is a retrospective justification for our approach in the interesting applications of our argument safety type system. The argument safety restriction itself is an interesting result deriving from our particular choice of model, and not something which we would otherwise have investigated, and for that reason alone the model seems worthy of consideration. Additionally, an application of argument safety to type-safe exceptions is outlined in Chapter 7, where we suggest that argument-safety captures the uses of higher-order store which allow for the static control of exceptions

1.7 Content and structure of thesis

This thesis makes three main contributions:

- A semantic treatment of *data abstraction*, in the form of an operation which takes an object implementation with explicit state and creates an object representing the corresponding externally observable behaviour.

[3]Ground type store cells can be written in our language, and other objects with higher-type state, but higher-type store cells can not.

- The identification of an object-oriented language with a natural level of computational power or expressivity, corresponding to a simple game model.
- The development of novel techniques required for a soundness proof relating the views of objects as reactive entities and as explicitly structured heap.

1.7.1 Overview

The remainder of this thesis is structured as follows. In **Chapter 2** we give definitions of our category of simple games and the basic structure available there. We construct call-by-value and well-bracketed variants, and finally introduce the technique of *memoisation* which shall later prove useful.

In **Chapter 3** we move on to the generation of interesting stateful behaviour in the setting of the previous chapter. We introduce the "data abstraction" operator *thread*, giving a definition and a series of properties for reasoning about the operator.

In **Chapter 4** we introduce our object-oriented language, and discuss the rationale for the various design decisions involved. We give a definition and static semantics, paying particular attention to the *argument safety* restriction we must impose. We then give an operational semantics, and a denotational semantics using the ideas from Chapters 2–3.

Chapter 5 concerns the proof of the soundness of the game semantics with respect to the operational semantics. We begin by discussing the property to be proved, taking the reader through a series of refinement steps, before presenting some auxiliary definitions and lemmas. We then prove soundness by induction on operational semantics derivations, a large part of which consists of the verification of the method invocation rule, which involves the *thread* operator. We briefly discuss further issues, and the other half of adequacy, which we do not prove.

In **Chapter 6** we turn to the issues of definability and full abstraction. The latter follows easily from the former, which forms the main part of the chapter. We give a series of programs which "interpret" an encoded strategy as a program, giving definability at a large class of (intuitionistic) types. However, we then show that at certain other (also intuitionistic) types strategies may exhibit some complex and problematic behaviour, and we conjecture that such strategies are not definable in our language.

Chapter 7 concerns some areas for future work, and possible extensions to our language. We discuss potential benefits of (and alternatives to) our argument safety restriction, then present a program which extracts the approximation operator from a class implementation. We then discuss some natural extensions of the adequacy and definability results presented in this thesis, followed by two potential language extensions which support object-oriented programming. We conclude by introducing the control operator **catchcont**, which we expect to lead to a fully abstract semantics for the non-well-bracketed version of our category of games, and we discuss the ramifications of adding this to our language.

We draw some conclusions in **Chapter 8**.

Chapter 2

Definition of categories of games

In this chapter we shall introduce and define our categories of games, and some of the structure that is present there.

We start by defining simple games $\mathbf{SG}$, and then extend these to $\mathbf{SG}^{\mathrm{V}}$ to give a setting to interpret values. Lastly we define categories of well-bracketed games $\mathbf{BG}$ (and $\mathbf{BG}^{V}$), and discuss their relationship to the unbracketed games.

2.1 Simple games

We shall start with the simplest notion of games we can get away with. As defined by Lamarche [55] (and described in [64, 32]), a game is simply a set of moves partitioned into opponent and player moves, together with the set of valid plays of that game—the *game tree*. These games are also described in [45, 8]. At this point there is no need for a notion of *question* and *answer*, but we shall introduce such a notion in Section 2.3.

The definition of a game as its collection of plays differs from the "arena games" of [43], which generate this from an enabling relation, and use this to associate moves via a "justification pointer" to the move which enables them. The games presented here are somewhat simpler as a result; here we are only interested in stateful computation, while the most obvious benefit of arena games is the identification of *innocent* strategies for state-free computation.[1]

Our definitions lead to a "linear" category of games, on which we define a linear exponential '!', and then further enlarge to give a setting for call-by-value computation.

[1]One other benefit concerns "non-alternating" game models as discussed in Section 1.6.4.

Define an *arena* A as a pair $\langle M_A, \ell_A^{OP} \rangle$, where

- M_A is a countable set of moves.
- $\ell_A^{OP} \colon M_A \to \{\mathsf{O}, \mathsf{P}\}$ is a labelling function distinguishing player and opponent moves. $\overline{\ell}_A^{OP}$ denotes the opposite labelling.

We write M_A^* for the set of sequences of zero or more moves from M_A, $\sqsubseteq$ for the prefix relation on M_A^*, and $s \sqsubseteq^{\text{even}} t$ for the prefix relation with s even. We write s_i for the $(i+1)$th element of the sequence s.

We define the *language* of an arena $L_A \subseteq M_A^*$ to be the set of finite plays in which moves alternate and opponent starts, i.e. L_A is the set of $s \in M_A^*$ satisfying

$$\begin{aligned} s = s_1 m n s_2 \quad &\to \quad \ell_A^{OP}(m) \neq \ell_A^{OP}(n) \\ s = m s' \quad &\to \quad \ell_A^{OP}(m) = O \end{aligned}$$

A *game* $A = \langle M_A, \ell_A^{OP}, P_A \rangle$ consists of an arena A plus a non-empty prefix-closed set of *valid* positions $P_A \subseteq L_A$. The subsets of P_A consisting of all even and odd length plays in P_A are denoted P^{even} and P^{odd} respectively.

For each game A we define the set of *strategies* for A as

$$\begin{aligned} R_A = \{\sigma \mid \sigma \subseteq P_A^{\text{even}},\ & \varepsilon \in \sigma, \\ & sab \in \sigma \to s \in \sigma, \\ & sab \in \sigma \wedge sac \in \sigma \to b = c\} \end{aligned}$$

That is, R_A consists of all non-empty, even-prefix-closed sets of even-length positions which are *deterministic*.

Given two sets of moves M_A and M_B, we write $M_A + M_B$ for their disjoint union. Given $s \in (M_A + M_B)^*$, the restriction of s to A, written $s\upharpoonright_A$, shall be the subsequence of s consisting of moves from M_A, so that $s\upharpoonright_A \in M_A^*$. More generally we write $s\upharpoonright_{A,B}$ to restrict from $M_A + M_B + M_C$ to $M_A + M_B$, and so on. For strategies, $\sigma\upharpoonright_A$ shall be the strategy consisting of those plays in σ only containing moves from M_A (as opposed to the pointwise restriction of plays $\{s\upharpoonright_A \mid s \in \sigma\}$).

Given games A and B we define the games $A \otimes B$, $A \multimap B$ as follows:

$$\begin{aligned} M_{A\otimes B} &= M_A + M_B \\ \ell^{OP}_{A\otimes B} &= [\ell^{OP}_A, \ell^{OP}_B] \\ P_{A\otimes B} &= \{s \in L_{A\otimes B} \mid s{\restriction}_A \in P_A,\ s{\restriction}_B \in P_B\} \end{aligned}$$

$$\begin{aligned} M_{A\multimap B} &= M_A + M_B \\ \ell^{OP}_{A\multimap B} &= [\overline{\ell}^{OP}_A, \ell^{OP}_B] \\ P_{A\multimap B} &= \{s \in L_{A\multimap B} \mid s{\restriction}_A \in P_A,\ s{\restriction}_B \in P_B\} \end{aligned}$$

Proposition 2.1 (Switching conditions)**.** *A player move in $A\otimes B$ must be in the same component as the previous (opponent) move. An opponent move in $A \multimap B$ must be in the same component as the previous (player) move. In other words, only opponent may switch components in $A \otimes B$, and only player may do so in $A \multimap B$.*

This is a standard result, given e.g. in [11], arising from the alternation conditions in the two constituent games.

2.1.1 Defining strategies

As well as giving a strategy σ for a game A explicitly as a subset of P_A^{even} as above, we can also give a partial function f from odd-length plays in P_A to the answering move (if any)

$$f\colon P_A^{\text{odd}} \rightharpoonup M_A$$

We shall say such an f is a strategy if whenever $f(s) = a$ then $\lambda(a) = P$ and whenever $f(sab) = c$ then $f(s) = a$. We are justified in calling f a strategy: we can construct σ_f as $\{\varepsilon\} \cup \{sab \mid f(sa) = b\}$, which is clearly a strategy as it contains ε, is even-prefix closed by the odd-prefix closure of f, and is deterministic by the fact that f is a function. Conversely, given a strategy σ we can construct a function f_σ obeying the above conditions simply by setting $f_\sigma(sa) = b \iff sab \in \sigma$. Furthermore, $\sigma_{f_\sigma} = \sigma$ and $f_{\sigma_f} = f$. We shall henceforth consider both set and function presentations as denoting strategies, and will use whichever is most convenient in a given situation.

In defining certain *history-free* strategies (in the sense of [6]), one can simply define a function $f : M_A \rightharpoonup M_A$; the intended strategy is that for $g\colon P_A^{\text{odd}} \rightharpoonup M_A$, where $g(sa) = f(a)$. We do not make any particular use of the history-freeness property, but shall sometimes define certain strategies this way for convenience and clarity.

2.1.2 The category SG

We define a category **SG** of *simple games*. Objects of **SG** are games, and morphisms $A \to B$ are strategies σ for $A \multimap B$. The identity morphisms are given by the usual copycat strategy (A_1 and A_2 distinguishing the two copies of A)

$$id_A = \{s \in P_{A_1 \multimap A_2} \mid \forall t \sqsubseteq^{\text{even}} s.\ t\upharpoonright_{A_1} = t\upharpoonright_{A_2}\ \}$$

Composition is the usual composition of strategies, a simple definition in the absence of justification pointers. If $\sigma \colon A \multimap B$ and $\tau \colon B \multimap C$ then we define the set of their interaction sequences as

$$\sigma \| \tau = \{u \in (M_A + M_B + M_C)^* \mid u\upharpoonright_{A,B} \in \sigma, u\upharpoonright_{B,C} \in \tau\}$$

and their composition as

$$\sigma;\tau = \{u\upharpoonright_{A,C} \mid u \in \sigma\|\tau\}$$

It is then easy to show that $\sigma;\tau$ is a strategy for $A \multimap C$, and composition is associative. We will also use the composition '$\circ$' in the other order ($\sigma;\tau = \tau \circ \sigma$) where convenient.

2.1.3 Copycat strategies

We mentioned that id_A is a *copycat* strategy. We will define many similar strategies, so it is worth expanding on that idea somewhat. The strategy id_A simply connects up the two copies of A, copying a move on the right to the left, a response on the left back to the right, and so on. No information is used about the particular game A, it is sufficient to know that the rules of each game A are the same.

Consider playing in the game

$$A \otimes B \to B \otimes A$$

Here we can "play the copycat" both in the game A and in the game B—when a move is played in A one side we play that move in A on the other, and when a move is played in B in one side we play that move in B in the other. It is easy to see that such play is always valid—again so long as Opponent abides by the rules of A and B, our moves will do too. A copycat strategy such as the one described can be thought of as "wiring together" each pair of games being copied between, matching a positive and negative copy.

Given a set of variables $A_1, \ldots, A_n$, define formal expressions F, G as obtained from the following grammar:

$$E ::= (E \otimes E) \mid 1 \mid A_1 \mid \ldots \mid A_n$$

Then write $\hat{F}$ and $\hat{G}$ for the obvious functors $\mathbf{SG}^n \to \mathbf{SG}$ corresponding to F and G respectively. Each game A_i may occur zero, one or many times in $F(\vec{A}), G(\vec{A})$ Write $\mathrm{Occ}(A_i)$ for the set of occurrences of A_i ($\mathrm{Occ}(\vec{A})$ for occurrences of any A_i in $\vec{A}$), and A_i^j for an occurrence.

Take C_R an injective map giving for each occurrence on the right A_i^j a matching occurrence A_i^k on the left, and set $C = C_R \cup C_R^{-1}$. C describes the list of connections of a copycat strategy. Given suitable F, G and a choice of C (there may be more than one option) define a collection of morphisms as follows:

$$f_{\vec{A}} = \{ s \in P_{\hat{F}(\vec{A}) \multimap \hat{G}(\vec{A})} \mid \forall t \sqsubseteq^{\text{even}} s. \forall A \in \mathrm{Occ}(\vec{A}).\ t\upharpoonright_A = t\upharpoonright_{C(A)} \}$$

In other words, define the history-free strategy (being informal about relabelling):

$$f_{\vec{A}}(m\colon A \in \mathrm{Occ}(\vec{A})) = m\colon C(A)$$

These strategies constitute a natural transformation $f\colon \hat{F} \to \hat{G}$.

It is very often the case that for given F, G only one choice of C is possible, and hence there is a unique copycat strategy. In this situation we do not need to spell out the strategy in question. Also note that the composition of two copycat strategies generated from C_1 and C_2 coincides with the copycat strategy generated from the composition of C_1 and C_2 (assuming they are compatible).

It would be possible to extend the above ideas to types involving $\multimap$, and other type constructors when we define them, connecting up positive and negative occurrences of each A_i, but it becomes less easy to say when F, G and C are of the correct form to generate a valid copycat strategy. We shall nevertheless use the term "copycat" informally in those situations too.

2.1.4 Symmetric monoidal closed structure

We extend the operation $\otimes$ to a bifunctor. Define a construction on strategies $\sigma : A \multimap C$ and $\tau : B \multimap D$ which interleaves them to form a strategy for $A \otimes B \multimap C \otimes D$ as follows:

$$\sigma \otimes \tau = \{ s \in P^{\text{even}}_{A\otimes B \multimap C \otimes D} \mid s{\restriction}_{A \multimap C} \in \sigma,\ s{\restriction}_{B\multimap D} \in \tau \}$$

Note that there exists an object 1 which is the terminal object of **SG**, and the unit for $\otimes$:

$$1 = \langle \emptyset, \_, \emptyset \rangle$$

We take the natural transformations with components

$$\begin{aligned}
\alpha_{A,B,C} &: (A \otimes B) \otimes C \to A \otimes (B \otimes C) \\
\lambda_A &: A \otimes 1 \to A \\
\rho_A &: 1 \otimes A \to A \\
\gamma_{A,B} &: A \otimes B \to B \otimes A
\end{aligned}$$

to be the appropriate (and unique) copycat strategies as described above. Note that in many contexts we will suppress the trivial morphisms λ, ρ and α as their presence can be deduced from the types, but we shall always be explicit in writing γ. These structural morphisms obey the following properties:

$$\begin{aligned}
(\alpha_{A,B,C} \otimes id_D); \alpha_{A,B\otimes C,D}; (id_A \otimes \alpha_{B,C,D}) &= \alpha_{A\otimes B,C,D}; \alpha_{A,B,C\otimes D} \\
\alpha_{A,B,C}; \gamma_{A\otimes B,C}; \alpha_{C,A,B} &= (id_A \otimes \gamma_{B,C}); \alpha_{A,C,B}; (\gamma_{A,C} \otimes id_B) \\
\gamma_{A,B}; \gamma_{B,A} &= id_{A\otimes B} \\
\gamma_{1,A}; \rho_A &= \ell^{OP}_A \\
\alpha_{A,1,C}; \rho_A \otimes id_C &= id_A \otimes \lambda_C
\end{aligned}$$

These properties hold simply from the copycat nature of the strategies, and can be viewed as simply "untangling" the corresponding wiring diagrams.

Before proceeding, we note that strategies for products in our category behave in an *interfering* fashion, in the sense that play in one component can affect future behaviour of the other. In other words, a strategy σ for $A \otimes B$ need not be a pair of strategies for A and B, and it is this which enables us to model stateful behaviour in **SG**. If interaction with σ in A affects the outcome of some later interaction in B (or vice versa), σ can be seen as representing a pair of objects of types A and B which share some internal state.

In fact **SG** is *affine*, possessing morphisms

$$\begin{aligned} 1_A &: A \to 1 \\ &= \{\varepsilon\} \end{aligned}$$

and so we may define projections

$$\begin{aligned} \Pi_L &: A \otimes B \to A & \qquad \Pi_R &: A \otimes B \to B \\ &= \rho_A \circ (id_A \otimes 1_B) & &= \lambda_A \circ (1_A \otimes id_B) \end{aligned}$$

Now the closed structure. Observe that a bijection

$$\mathbf{SG}(A \otimes B, C) \cong \mathbf{SG}(A, B \multimap C)$$

is induced simply by the bijection on move sets

$$(M_A + M_B) + M_C \cong M_A + (M_B + M_C)$$

In connection with this we use the notation

$$\frac{f\colon A \otimes B \to C}{\lambda_B(f)\colon A \to (B \multimap C)} \quad \frac{f\colon A \to (B \multimap C)}{f^*\colon (A \otimes B) \to C}$$

and take $eval\colon (A \multimap B) \otimes A \to B$ to be $(id_{A \multimap B})^*$.

We shall make use of the internal language of **SG**, writing morphisms as $(\lambda x^B.\ f)\colon A \to (B \multimap C)$ for $\lambda_B(f)$ where $f\colon A \otimes B \multimap C$, and $fx\colon A_1 \otimes A_2 \to C$ for $(f \otimes x); eval$ where $f\colon A_1 \to B \multimap C$ and $x\colon A_2 \to B$.

2.1.5 Additive product

For games A and B we can define the additive product $A\&B$ of linear logic. A play in the product is a play in *either* A or B—a strategy for $A\&B$ specifies Player's response to interaction in *either* A or B, as chosen by the Opponent's first move.

$$\begin{aligned} M_{A\&B} &= M_A + M_B \\ \ell^{OP}_{A\&B} &= [\ell^{OP}_A, \ell^{OP}_B] \\ P_{A\&B} &= \{s \in L_{A\&B} \mid (s\restriction_A \in P_A \wedge s\restriction_B = \varepsilon) \vee (s\restriction_B \in P_B \wedge s\restriction_A = \varepsilon)\} \end{aligned}$$

This additive product is in fact the Cartesian product in **SG**. We extend the operation & to morphisms by taking $f\&g$ as the strategy which behaves as either f or as g according to the component selected by the first opponent move. Unlike the tensor product $\otimes$, only one of f or g will be involved in a given play, so there is no need for two copies of A to be provided to $f\&g$.

$$\frac{f\colon A \to B \quad g\colon A \to C}{f\&g\colon A \to B\&C} \qquad f\&g = f \cup g$$

We abuse notation to label the projections Π_L and Π_R as for the product $- \otimes -$, which are the obvious copycat strategies:

$$\Pi_L \ : \ A\&B \to A \qquad \Pi_R \ : \ A\&B \to B$$

In fact we shall use a more general set-indexed version of the above product, $\&_{i\in I}A_i$ defined as follows:

$$\begin{array}{rcl} M_{\&_{i\in I}A_i} & = & \biguplus_{i\in I} A_i \\ \ell^{OP}_{\&_{i\in I}A_i}(\mathrm{in}_a(x)) & = & \ell^{OP}_{A_i}(x) \\ P_{\&_{i\in I}A_i} & = & \{s \in L_{\&_{i\in I}A_i} \mid \exists i \in I.\ s\upharpoonright_{A_i} \in P_{A_i} \wedge \\ & & \quad \forall j \in I.\ i \neq j \Rightarrow s\upharpoonright_{A_j} = \varepsilon\} \end{array}$$

Again we define the action on morphisms

$$\frac{f_i\colon Z \to A_i}{\&_{i\in I}f_i\colon Z \to \&_{i\in I}A_i} \qquad \&_{i\in I}f_i = \cup_{i\in I}f_i$$

and the evident projections

$$\Pi_j\colon \&_{i\in I}A_i \to A_j \qquad (j \in I)$$

There is also a distributivity morphism for $\otimes$ and $\&$:

$$\mathit{dist} = (\&_{i\in I}A_i) \otimes B \xrightarrow{\&_{i\in I}(\Pi_i \otimes id_B)} \&_{i\in I}(A_i \otimes B)$$

2.1.6 Coproducts

Given a set I and family of games A_i in **SG**, we define a weak coproduct game $\Sigma_{i\in I}A_i$ in **SG** which we shall use later in the construction of our call-by-value category $\mathbf{SG}^{\mathrm{V}}$ as in [12]. We take a fresh initial move q (which we may regard as a "question"), which can be followed by an "answer" $i \in I$, with play subsequently as for the game A_i.

$$\begin{array}{rcll} M_{\Sigma_{i\in I}A_i} & = & \{q\} + I + \bigsqcup_{i\in I} M_{A_i} & \\ \ell^{OP}_{\Sigma_{i\in I}A_i} & = & \ell^{OP}_1, \quad \text{where} & \\ & & & \\ \ell^{OP}_1(q) & = & O & \\ \ell^{OP}_1(i) & = & P & (i \in I) \\ \ell^{OP}_1(m) & = & \ell^{OP}_{A_i}(m) & (m \in M_{A_i}) \end{array}$$

$$P_{\Sigma_{i\in I}A_i} \quad = \quad \{\epsilon, q\} \cup \{qis \mid i \in I, s \in P_{A_i}\}$$

For each $i \in I$ there is an injection

$$\mathrm{in}_i \colon A_i \to \Sigma_{i\in I}A_i$$

which responds to the initial q with i, and thereafter acts as a copycat.

Call a game *pointed* if it starts with a unique initial move q. Given a pointed game B and a collection $\{f_i \mid i \in I\}$ with $f_i \colon A_i \to B$ we can define

$$[f_i]_{i\in I} \colon \Sigma_{i\in I}A_i \to B$$

as

$$[f_i]_{i\in I} = \{\varepsilon\} \cup \bigcup_{i\in I}\{qqis \mid qs \in f_i\}$$

It is easy to see that for any $j \in I$, $\mathrm{in}_j; [f_i]_{i\in I} = f_j$, and that $[f_i]_{i\in I}$ is the unique such strategy.

We will later use this construction with each A_i the same game A, and in that situation write simply $\Sigma_I A$ for $\Sigma_{i\in I}A$.

2.1.7 Skewed products

SG also has what we will refer to as a *skewed product*, $A \oslash B$. This is a game in which the first move *must* be in A, but thereafter moves can be in A or B as with $A \otimes B$. This is defined as follows:

$$\begin{array}{rcl} M_{A\oslash B} & = & M_A \uplus M_B \\ \ell^{OP}_{A\oslash B} & = & [\ell^{OP}_A, \ell^{OP}_B] \\ P_{A\oslash B} & = & \{s \in L_{A\oslash B} \mid s\upharpoonright_A \in P_A,\ s\upharpoonright_B \in P_B,\ s = \varepsilon \text{ or } s_0 \in M_A\} \end{array}$$

Our skewed product $A \oslash B$ is in fact the sequoidal product $B \oslash A$ of [51]: we say “A then B” while he says “B after A”.

The skewed product $A \oslash B$ is clearly related to the normal product $A \otimes B$, in that one can always take a strategy for $A \otimes B$ and restrict attention to plays beginning in A, giving the following inclusion morphism:

$$\mathit{skproj}_{A,B} : A \otimes B \to A \oslash B$$

More interesting is the morphism we can define in the other direction, when the paired types are identical:

$$\mathit{skew}_A \colon A \oslash A \multimap A \otimes A$$

This is a *dynamic copycat* strategy which identifies the component of $A \otimes A$ which *happens* to be accessed first with the first component of $A \oslash A$. Refer to the four games A using subscripts $A_{00}, A_{01}, A_{10}, A_{11}$ from left to right.

$$\mathit{skew}_A = \{t \in L_{A\oslash A \to A\otimes A} \mid \forall s \sqsubseteq^{\text{even}} t. \forall i \in \{0,1\}.\ s\restriction_{A_{1i}} = \text{pick}(i,s)\}$$

where

$$\text{pick}(i,s) = \begin{cases} s\restriction_{A_{00}} & \mathit{if}\ s_0 \in A_{1i} \\ s\restriction_{A_{01}} & \mathit{if}\ s_0 \notin A_{1i} \end{cases}$$

Thus we have a retraction

$$(\mathit{skew}_A, \mathit{skproj}_A) \colon A \oslash A \quad \lhd \quad A \otimes A$$

The key property of the morphism skew_A, however, is that the apparent ordering of the pair $A \otimes A$ is rendered irrelevant, that is

$$\begin{tikzcd} A \oslash A \arrow[d, "\mathit{skew}_A"'] \arrow[dr, "\mathit{skew}_A"] & \\ A \otimes A \arrow[r, "\gamma_{A,A}"'] & A \otimes A \end{tikzcd}$$

The skew product comes with a pseudo-associativity isomorphism

$$\mathit{passoc}_{A,B,C} \colon (A \oslash B) \oslash C \cong A \oslash (B \otimes C)$$

since, on either side, the valid plays are simply those of $A \otimes B \otimes C$ which start with an A move.

$- \oslash -$ is not a bifunctor on **SG**, since $f \oslash g : A \oslash B \to C \oslash D$ could result in the first move on the left of the arrow being played in B. However, we can consider the *strict* sub-category $\mathbf{SG}_s$ of **SG**. Every object A in **SG** admits a morphism $\bot_A \colon 1 \to A$ consisting of the trivial strategy $\{\epsilon\}$, and we say a morphism $f \colon A \to B$ is *strict* when $f \circ \bot_A = \bot_B$. $\mathbf{SG}_s$ has the same objects as **SG**, and as morphisms the strict morphisms of **SG**. Thus we have a functor $\oslash : \mathbf{SG}_s \times \mathbf{SG} \to \mathbf{SG}_s$, whose action of $\oslash$ on morphisms is simply that of $\otimes$. Strictness of f in $f \oslash g \colon A \oslash B \to C \oslash D$ ensures that the first move on the right in C is immediately followed by a move in A, satisfying the requirement on $A \oslash B$.

One could also define a lift monad on **SG** to go with this notion of strictness. Instead of doing so here, we will give a more useful one below when we introduce our category for call-by-value computation.

Returning to the evaluation morphism from Section 2.1.4, we note that this can be given a type involving $\oslash$:

$$eval \colon A \multimap B \oslash A \rightarrow B$$

This is naturally the type of *eval*, since the first move on the left is always in B, copied from the right.

2.1.8 Recursive types

We note that the allowable sets of positions which define a game are countable sets of plays, and form a CPO under set inclusion. We define $A \sqsubseteq B$ if $M_A \subseteq M_B$, $\ell_A^{OP}(m) = \ell_B^{OP}(m)$ when $m \in M_A$, and $P_A \subseteq P_B$. The least upper bound $\bigvee_{i \in I} A_i$ is the game $\langle \bigcup_{i \in I} M_{A_i}, \bigcup_{i \in I} \lambda_{A_i}, \bigcup_{i \in I} P_{A_i} \rangle$, and the empty game $1_{\mathbf{SG}}$ is a least element. So our games themselves form a big CPO.

For a continuous map F on games we define a fixed point operator μ

$$\begin{array}{rcl} F & : & |\mathbf{SG}| \rightarrow |\mathbf{SG}| \\ \mu X.F(X) & \in & |\mathbf{SG}| \\ & = & \bigvee_k (F^k \circ 1_{\mathbf{SG}}) \end{array}$$

We now note that all the operators $\otimes, \multimap, \&, \Sigma, \oslash$ introduced above are monotone and continuous in all arguments, since they are defined in a point-wise fashion in terms of positions, and those are finite. Hence μ gives least fixed points when applied to them, or operators built up from them by composition. Note that recursive types are not really used in this thesis, except rather informally in Section 2.2.2

2.2 The linear exponential

In this section we shall define a linear exponential comonad '!' which we use to interpret reusable objects. This is a standard concept, and we first review the requirements for such a thing, before giving the particular definition we use. We define the same exponential as that found in [51] and [45], but being somewhat more explicit. Our definition is notable for the general machinery we use to easily yet rigorously define the required data (various structural morphisms). We also indicate how this machinery can be used to verify many of the required properties (without giving full details).

2.2.1 Requirements of an exponential

To give an interpretation of linear logic (and the linear/affine lambda calculus) it is sufficient to have a symmetric monoidal closed category $(\mathbf{SG}, \otimes, 1)$ with a comonad ! with certain structure—such a ! is called a *linear exponential comonad*. Here we follow the presentation of Bierman, Benton et al. [16, 17, 20].

We must give some categorical definitions, generally specifying what it is for a comonad to be well behaved in an SMCC.

Let $\mathscr{C}, \mathscr{D}$ be symmetric monoidal categories. A functor $F \colon \mathscr{C} \to \mathscr{D}$ is symmetric monoidal if there is a morphism $mi \colon I \to F(I)$ and a natural transformation $m \colon F(-) \otimes F(-) \to F(- \otimes -)$ respecting the symmetric, associative and unit structures as follows:

$$\begin{array}{rcl} m_{A,B}; F(\gamma_{A,B}) & = & \gamma_{FA,FB}; m_{A,B} \\ \alpha_{FA,FB,FC}; m_{A,B} \otimes id_{FC}; m_{A\otimes B,C} & = & id_{FA} \otimes m_{B,C}; m_{A,B\otimes C}; F(\alpha_{A,B,C}) \\ m_{I,A}; F(\lambda_A) & = & mi \otimes id_{FA}; \lambda_{FA} \end{array}$$

Let (F, m, mi) and (G, m', mi') be two symmetric monoidal functors $\mathscr{C} \to \mathscr{D}$. A natural transformation $\tau \colon F \to G$ is monoidal if

$$\begin{array}{rcl} mi; \tau_I & = & mi' \\ m_{A,B}; \tau_{A\otimes B} & = & \tau_A \otimes \tau_B; m'_{A,B} \end{array}$$

A comonad on $\mathscr{C}$ consists of a functor $! \colon \mathscr{C} \to \mathscr{C}$ together with two natural transformations, the *counit* $\varepsilon \colon ! \to \mathrm{Id}_{\mathscr{C}}$ and *comultiplication* $\delta \colon ! \to !!$, such that

$$\delta_A; \varepsilon_{!A} = id_{!A} = \delta_A; !\varepsilon_A \quad \text{and} \quad \delta_A; \delta_{!A} = \delta_A; !\delta_A$$

The comonad $(!, \varepsilon, \delta)$ is monoidal if ! is a monoidal functor and ε and δ are monoidal natural transformations.

A *linear exponential comonad* is a monoidal comonad $(!, \varepsilon, \delta, m, mi)$ equipped with monoidal natural transformations $e \colon ! \to I$ and $d \colon !- \to !- \otimes !-$ such that each $(!A, e_A, d_A)$ form a commutative comonoid:

$$\begin{array}{rcl} d_A; \gamma_{!A,!A} & = & d_A \\ d_A; e_A \otimes id_{!A} & = & \lambda_{!A} \\ d_A; d \otimes id_{!A}; \alpha_{!A,!A,!A} & = & d_A; id_{!A} \otimes d_A \end{array}$$

and e_A, d_A are coalgebra morphisms (on the free coalgebra $(!A, \delta_A)$):

$$\begin{aligned} e_A; mi &= \delta_A; !e_A \\ \delta_A; !d_A &= d_A; \delta_A \otimes \delta_A; m_{!A,!A} \end{aligned}$$

and finally each coalgebra morphism between the free coalgebras $(!A, \delta_A)$ and $(!B, \delta_B)$, that is $f\colon !A \to !B$ such that

$$f; \delta_B = \delta_A; !f$$

is a comonoid morphism:

$$\begin{aligned} e_A &= f; e_B \\ d_A; f \otimes f &= f; d_B \end{aligned}$$

These conditions are given diagrammatically and slightly more explicitly in [20], along with a proof that an SMCC with linear exponential comonad provides a model for the multiplicative, exponential fragment of intuitionistic linear logic.

2.2.2 Linear exponential

We now define the particular exponential '!' that we shall use in this thesis. The game $!A$ should be thought of as a "reusable" version of A—the intention is that, for example, with the type $!(A \multimap B)$ we can model a function which can be called more than once. We shall think of this as countably many copies of A, so that the behaviour of a strategy for $!A$ may vary from use to reuse. This *non-uniformity* is an important feature of '!': we can model a function which behaves differently from one call to the next. Moreover, not only can the behaviour of a strategy for $!A$ be different in each copy of A, as with $\otimes$ and $\oslash$ different components of $!A$ may *interfere*, so that for example interaction in one function call can affect interaction in another.

We construct $!A$ as an infinitary version of the skew product $A \oslash A$:

$$!A = \mu B. A \oslash B$$

This definition in terms of $\oslash$ is also used by Laird [51], and the same ! operator is also defined directly by Hyland [45]. We also find it convenient to use the following direct definition. On objects, define:

$$\begin{aligned} M_{!A} &= \textstyle\bigsqcup_{i \in N} M_A^{(i)} \\ \lambda_{!A} &= \textstyle\bigsqcup_{i \in N} \ell_A^{OP} \\ P_{!A} &= \{t \in L_{!A} \mid \forall i. t\restriction_{M_A^{(i)}} \in P_A \wedge \\ &\qquad (\forall i \geq 0.\ t = sau \wedge a \in M_A^{(i+1)} \Rightarrow s\restriction_{M_A^{(i)}} \neq \varepsilon)\} \end{aligned}$$

We note that

$$!A \cong A \oslash !A$$

witnessed by the following morphisms

$$\begin{aligned} \mathit{unfold}_A &: \quad !A \to A \oslash !A \\ \mathit{fold}_A &: \quad A \oslash !A \to !A \end{aligned}$$

On morphisms we note that the "obvious" pointwise definition for $\sigma\colon A \to B$

$$!\sigma = \{t \in P_{!A \multimap !B} \mid \forall i.\ t\!\upharpoonright_{A_i \multimap B_i} \in \sigma\}$$

is not correct—as one can see from the definition in terms of $\oslash$, this only defines a functor in $\mathbf{SG}_s$. Instead we want to add to this the infinitary version of skew_A, so that components of A are dynamically associated with components of B according to when they are opened (we defer explanation until the next section). If σ is strict, this will coincide with the naïve definition. The function $\mathit{comp}(s, i)$ picks the component of $!B$ associated with the ith component of $!A$ in s, or a fresh one if there is none:

$$\mathit{comp}(s,i) = \begin{cases} j & \text{if } s = tb_i a_j t', \\ 1 + \max\{j \mid \exists k.\ s_k \in M_{A_j}\} & \text{otherwise} \end{cases}$$

Then define $!\sigma$ as follows:

$$!\sigma = \{\varepsilon\} \cup \{t \in P_{!A \multimap !B} \mid \forall smm' \sqsubseteq^{\text{even}} t.\ \forall i.\ (j = \mathit{comp}(s,i)) \Rightarrow smm'\!\upharpoonright_{A_j \multimap B_i} \in \sigma\}$$

Verifying that $!(\sigma \circ \tau) = !\sigma \circ !\tau$ involves a tedious "chasing" of moves through the two instances of *comp*, and we omit the proof here.

It should be noted that it is possible to define various '!' operators in **SG** (such issues are discussed in depth by Melliès [63]). The AJM exponential [11] is somewhat similar to that defined above—it is instead an infinitary version of $\otimes$. However, in that context *uniformity* is imposed so that each component must behave the same, rendering ordering irrelevant. In the non-uniform situation the ordering on components imposed by $\oslash$ cuts down on some redundancy in the representation, ensuring that ! forms a comonad.

In contrast, the "backtracking exponential" of Lamarche [55] interprets $!A$ as the game whose plays "explore" the game tree of A. Here each move of such an exploration represents a play in A, and an exploration can visit two plays st_1, st_2 representing a "fork" in the game tree, but ask for a response to a given play only once. This leads to a uniform behaviour, in the sense that the Opponent must remember a Player answer instead of repeating the question (and perhaps receiving a different answer), but unlike the AJM exponential this arises from the definition rather than as an imposed constraint.

It seems that there is another exponential present in $\mathbf{SG}$, which can be thought of as a combination of the one we define above and Lamarche's backtracking exponential, where in $!A$ we essentially permit repetition in the exploration of A, but this additional "power" is not required for the purposes of this thesis.

2.2.3 Dynamic copycat strategies

In association with our exponential, we wish to define operations such as the contraction $d_A \colon !A \to !A \otimes !A$. Since the intended meaning of this operation is to produce two "copies" of $!A$ from one, we should expect that d_A will be a strategy translating (possibly interleaved) interaction in the two copies to interaction in the original. We might consider defining a copycat strategy in the sense of Section 2.1.3, where we think of each component of $!A$ as a separate "occurrence" of A, but this does not work: it is not possible to define such an operation which ensures that the components of each $!A$ are opened in order. Instead of a static association between components of each $!A$, we must define a *dynamic* one as in [45], where each new A component opened in $!A \otimes !A$ is associated with the next available component of $!A$. In fact, at this type it is not hard to see that we have no choice in the matter.

We now generalise this idea. Define suitable formal expressions E over variables $A_1, \ldots, A_n$:

$$E \quad ::= \quad E \otimes E \mid !E \mid 1 \mid A_1 \mid \ldots \mid A_n$$

We identify an "occurrence" of some A_i in an expression E by the path in $(\{L, R\} \cup \mathbb{N})^*$ which navigates E to A_i by choosing the left or right branch of each $\otimes$ encountered and the given component of each ! encountered, and define $\mathrm{Occ}(E)$ as the set of all such valid paths.

Given expressions F, G, and taking the evident functors $\hat{F}, \hat{G} \colon \mathbf{SG}^n \to \mathbf{SG}$, define a *dynamic copycat play* in $\hat{F}(\vec{A}) \multimap \hat{G}(\vec{A})$ with respect to a injection

$$C \colon (\{L, R\} \cup \mathbb{N})^* \to (\{L, R\} \cup \mathbb{N})^*$$

as a play s such that

$$\forall t \sqsubseteq^{\text{even}} s. \quad (\forall p \in \text{Occ}(G).\ t\upharpoonleft_p = t\upharpoonleft_{C(p)})$$

where we write$\upharpoonleft_p$ for the restriction to the moves of the game specified by a path p. Then define a dynamic copycat strategy $f\colon \hat{F}(\vec{A}) \multimap \hat{G}(\vec{A})$ as a strategy in which each play in f is a dynamic copycat play with respect to some C, and if $t \in f$ and $t\upharpoonleft_p \quad a \in L_{A_i}$ where p designates an occurrence of A_i then $f(ta_p)$ is defined. We explicitly do *not* require that the choice of C coincides for each play.

Lemma 2.2 (Dynamic copycats)**.** *Any copycat strategy f is also a dynamic copycat strategy, and for any dynamic copycats f and g, $f;g$, $f \otimes g$ and $!f$ are both dynamic copycats.*

Proof. A copycat strategy $f\colon \hat{F}(\vec{A}) \to \hat{G}(\vec{A})$ is simply a dynamic copycat strategy $\hat{F}(\vec{A}) \to \hat{G}(\vec{A})$ which specifies the same map C for each play s. Assume f and g are dynamic copycat strategies $f\colon \hat{F}(\vec{A}) \to \hat{G}(\vec{A})$ and $g\colon \hat{G}(\vec{A}) \to \hat{H}(\vec{A})$. Each play s in $f;g$ arises from plays $t_1 \in f$, $t_2 \in g$ such that $t_1\upharpoonleft_{\hat{G}(\vec{A})} = t_2\upharpoonleft_{\hat{G}(\vec{A})}$. Where t_1 and t_2 specify maps C_1 and C_2 respectively, defining the map C as $C_1;C_2$ makes s a dynamic copycat play. Thus $f;g$ is a dynamic copycat $\hat{F}(\vec{A}) \to \hat{H}(\vec{A})$.

We omit the proof that $!f$ is a dynamic copycat here. □

Proposition 2.3 (Unique dynamic copycats)**.** *For any expressions F and G with respect to variables $A_1, \ldots, A_n$ where $F = !A_1 \otimes \ldots \otimes !A_n$, giving functors $\hat{F}, \hat{G}\colon \mathbf{SG}^n \to \mathbf{SG}$, for each choice of objects $\vec{A}$ there exists a unique dynamic copycat strategy $\hat{F}(\vec{A}) \to \hat{G}(\vec{A})$. Furthermore, these morphisms form a natural transformation $\hat{F} \to \hat{G}$.*

Proof. Observe that for any dynamic copycat strategy f, for any plays $s \sqsubseteq t$ in f, $\tilde{C}_s \subseteq \tilde{C}_t$ where $\tilde{C}_u$ is the restriction of C_u such that $p \in \text{dom}(\tilde{C}_u) \Rightarrow u\upharpoonleft_p \neq \varepsilon$. ε is a play of any game, and is a dynamic copycat play.

Given a dynamic copycat play t with C_t, and a move a_p in $\hat{G}(\vec{A})$ (such that ta_p is a valid play), there are two possibilities. Firstly, a_p is a move in the already opened game p, in which case $a_{C(p)}$ can and must be played. Secondly, a_p may be opening a new component p. Then we must choose a new p' in F, and set $C(p) = p'$. By the restricted format of F, there is only ever one component p' of a given A_i which may be opened in $\hat{F}(\vec{A})$, so we must take this component p' as $C(p)$.

It is also fairly straightforward to verify that the strategies σ_A so constructed are natural in $\vec{A}$ (again the only interesting part is with regards to non-strict strategies), but we omit the proof here. $\square$

2.2.4 Exponential structure

We now consider the required operations and properties of !. Since in our case $I = 1$ is the empty game, $!I$ and $mi \colon I \to !I$ are trivial and we shall not discuss them further. As I is also the terminal object 1, $e_A \colon !A \to 1$ comes from the affine structure, $e_A = 1_{!A}$. We must also define:

$$\begin{array}{lclclcl} \delta_A & : & !A \to !!A & \quad d_A & : & !A \to !A \otimes !A \\ \varepsilon_A & : & !A \to A & \quad m_A & : & !A \otimes !B \to !(A \otimes B) \end{array}$$

Here we are in the situation of Proposition 2.3, and so we take these morphisms to be the unique dynamic copycats of those types (giving the required natural transformations). Actually, the dereliction ε_A is not very dynamic, as it simply selects the first component of $!A$. The other morphisms are genuinely dynamic, and for illustration we give a more explicit definition in the case of δ_A:

$$\begin{aligned} \delta_A \colon & !A \to !!A \\ & = \{ t \in L_{!A \multimap !!A} \mid \forall n. \forall s \sqsubseteq^{\text{even}} t. s\restriction_{A_n} = s\restriction_{\text{component}(s,n)} \} \end{aligned}$$

component(s, n) is the nth right-side A-game opened up in the play s, i.e.

$$\begin{aligned} & \text{component}(s, n) = A_{p,q} \\ & \quad \textit{where } s_i \in M_{A_{p,q}}, i = \min\{ i \mid s_i \notin \{\text{component}(s, 0), \ldots, \text{component}(s, n-1)\}\} \end{aligned}$$

It is easy to verify that all but the last of the required properties of Section 2.2.1 are satisfied by application of Lemma 2.2 and Proposition 2.3. Observe that each equation involves the dynamic copycats just defined, the static copycats from the SMCC structure, use of the functors ! and $\otimes$, and composition By Lemma 2.2 both sides of each equation are dynamic copycats, and since their types all start in a game of the correct form Proposition 2.3 shows they must therefore be equal. The last property does not follow for these general reasons, but can be routinely verified.

We shall also make use of the Kleisli operator, which we define from the above structure in the normal way:

$$\begin{aligned} f &: \; !A \to B \\ f^{\ddagger} &: \; !A \to !B \\ &= \; \delta_A; !(f) \end{aligned}$$

This gives rise to a co-Kleisli category $\mathbf{SG}_!$ as usual, but we shall work directly in $\mathbf{SG}$—partly because we will define an operation $!A \to !B$ which is not a promoted morphism $f^{\ddagger}$.

2.2.5 CPO structure and fixpoints

We note that strategies for any given game are countable sets of plays, and thus form a ωCPO under set inclusion, with the empty strategy as least element $\perp$ and least upper bounds (lubs) of ω-chains being given by set-theoretic union. Composition of strategies is monotone, from the pointwise definition of composition, and continuous, following from plays being finite and hence in some finite element of the lub. Therefore we can view $\mathbf{SG}$ as a CPO-enriched category.

We then have a fixed point operator on $\mathbf{SG}$, (writing **ext** to suggest this is an *external* operator):

$$Y_A^{\mathbf{ext}} \colon \mathbf{SG}(A, A) \to \mathbf{SG}(1, A)$$

Concretely, where $f^k = f \circ f \circ \ldots \circ f$, and writing $\bigvee_k f_k$ to denote the lub of the sequence $\{f_k \mid k \in \mathbb{N}\}$, define

$$Y_A^{\mathbf{ext}}(f) = \bigvee_k (f^k \circ \perp_A)$$

We then have as usual that

$$f \circ Y_A^{\mathbf{ext}}(f) = Y_A^{\mathbf{ext}}(f)$$

The external fixpoint operator is standard, but it is worth noting the type of the internal version

$$Y_A \colon !(A \multimap A) \to A$$

An exponential appears because the function has to be used repeatedly to obtain the fixed point. The internal fixpoint operator may be defined from the external one via a standard trick from domain theory: Define

$$Y_A = [Y^{\mathbf{ext}}_{!(A \multimap A) \multimap A}(\lambda_{!(A \multimap A)}((id \otimes d); (eval \otimes \varepsilon); eval))]^*$$

This is more readable in internal language notation. Write $f \bullet x$ to denote dereliction plus application, $((f;\varepsilon)\otimes x); \mathit{eval}$, and recall that the juxtaposition $(f\ x)$ denotes ordinary application. Then the above definition is simply:

$$Y_A = Y^{\mathbf{ext}}(\lambda F.\lambda f.\ f \bullet (Ff))$$

The need for the exponential in the type of Y_A manifests in the double occurrence of f in the above expression.

We can then verify that

$$\begin{array}{rcl} Y_A\ f & = & Y^{\mathbf{ext}}(\lambda F.\lambda f.\ f \bullet (Ff))\ f \\ & = & [(\lambda F.\lambda f.\ f \bullet (Ff))\ Y_A]\ f \\ & = & (\lambda f.\ f \bullet (Y_A\ f))\ f \\ & = & f \bullet (Y_A\ f) \end{array}$$

2.2.6 Universal object

SG has a universal object U, where play simply consists of the player and opponent exchanging numbers $n \in \mathbb{N}$. We could define the universal game $U = \langle M_U, \lambda_U, P_U\rangle$ as follows:

$$\begin{array}{rcl} M_U & = & \mathbb{N} + \mathbb{N} \\ \lambda(\mathit{inl}(n)) & = & O \\ \lambda(\mathit{inr}(n)) & = & P \\ P_U & = & L_U \end{array}$$

Plays of this game are just any sequence of natural numbers, with the correct opponent/player alternation. However, it will be more convenient to use an equivalent formulation where moves are labelled with their position in the play in order to avoid the apparent repetition of moves:[2]

$$U = !\&_{\mathbb{N}}\Sigma_{\mathbb{N}}1$$

The moves of this game can be described as $\{q_i^n, a_i^n\}_{n,i\in\mathbb{N}}$, where the subscript represents the component of the exponential, and the superscript on q or a represents the choice of game for & and Σ respectively.[3] The plays of this game are then those which have the form $q_0^n a_0^m q_1^{n'} a_1^{m'} \ldots$, the order of these subscripts being constrained by the definition of the exponential.

[2]This revised formulation will coincide exactly with the denotation of a type in our language in Chapter 4.

[3]Strictly speaking we should also annotate each a with a &-index too, see Section 2.6.

We now show that U is a universal object. Take an object $A = \langle M_A, \ell_A^{OP}, P_A \rangle$ of **SG**: we will define $f_A : A \to U$ and $g_A : U \to A$ such that $g_A \circ f_A = id_A$ and $f_A \circ g_A \sqsubseteq id_U$.

Since M_A is countable, we may choose an injection $\iota \colon M_A \to \mathbb{N}+\mathbb{N}$, respecting the P/O labelling. Now define a function $t : (M_A + M_U) \rightharpoonup (M_A + M_U)$ to be the least partial function such that:

$$\begin{array}{rcll} t(a) & = & \iota(a) & (a \in M_A) \\ t(\iota(a)) & = & a & \end{array}$$

and the required morphisms:

$$\begin{array}{rcl} f_A & = & \{ s \mid \forall s'ab \sqsubseteq^{\text{even}} s.\ b = t(a) \} \\ g_A & = & \{ s \mid (\forall s'ab \sqsubseteq^{\text{even}} s.\ b = t(a)) \wedge s \upharpoonright_{M_A} \in P_A \} \end{array}$$

It is easy to see that (thanks to the restriction $s \upharpoonright_{M_A} \in P_A$) these are indeed strategies of the correct type, $g_A \circ f_A = id_A$ and $f_A \circ g_A \sqsubseteq id_U$.

Note that when we impose a notion of bracketing on strategies in Section 2.3, the retraction defined above may violate well-bracketing. Thus, U will no longer be a universal object in the category of well-bracketed strategies.

By the nature of the universal object, an alternative construction of **SG** is possible. There is a *linear λ-algebra* corresponding to U which can be defined relatively simply [60], giving rise via the Karoubi envelope construction to a *category of projections* equivalent to **SG**. This adds support to our belief that **SG** is mathematically a rather natural category of games to consider.

2.3 Well-bracketed games

We now define a subcategory of **SG** in which the strategies obey some bracketing discipline. This category **BG** will be the setting in which we shall work for the majority of this thesis—we shall return briefly to the non-well-bracketed setting in Chapter 7. We will essentially be excluding the possibility of methods terminating prematurely or out of order, or in other words we will rule out any kind of continuation or *catch* operator. This will help to ensure a close match with the language we shall introduce in Chapter 4, which lacks such control features.

Consider the following two plays of type $(1_\perp \multimap 1_\perp) \multimap 1_\perp$, where $1_\perp$ is the game consisting soley of a unique move q and response a.[4] The first rep-

[4]The $-_\perp$ construction is formally introduced in Section 2.4.

resents a "normal" interaction of this type, while the second represents interaction with a strategy which "terminates early"—the characteristic play of the Cartwright/Felleisen `catch` strategy [27] at this type.

$$\begin{array}{ccccc}
(1_\perp & \multimap & 1_\perp) & \multimap & 1_\perp \\
 & & & & q \\
 & & q & & \\
q & & & & \\
a & & & & \\
 & & a & & \\
 & & & & a
\end{array}
\qquad
\begin{array}{ccccc}
(1_\perp & \multimap & 1_\perp) & \multimap & 1_\perp \\
 & & & & q \\
 & & q & & \\
q & & & & \\
 & & & & a
\end{array}$$

The second strategy above is the prototypical strategy which we want to disallow. It violates the *well-bracketing* principle that questions should be answered in the correct order, and not early or otherwise out of turn (a kind of stack discipline). The following play violates this principle in a slightly different way:

$$\begin{array}{ccccccc}
((1_\perp & \multimap & 1_\perp) & \multimap & 1_\perp) & \multimap & 1_\perp \\
 & & & & & & q \\
 & & & & q & & \\
 & & q & & & & \\
q & & & & & & \\
 & & & & a & & \\
 & & & & & & a
\end{array}$$

This would arise as an interaction with an opponent playing as in the second case above. We also exclude this behaviour in **BG**, but when working in **SG** it is worth bearing in mind the difference between strategies which *commit* a bracketing violation and those such as the last strategy above which merely carry on in the face of opponent violation.

2.3.1 The category BG

Consider a game A in **SG**. A is equipped with a number of *moves* which are not the same as *positions* (or *plays*) of A, so that a move m may be present in two distinct plays sm and tm. By the rules of **SG**, m may occur multiple times in a single play s_1ms_2m, but none of the definitions we have given introduce such a possibility. In particular, the definition of '$!A$' explicitly distinguishes between each instance of a given move in A.

We can therefore partition the moves of each game into questions and answers, with each answer move justified by a question move, and with this justification being built into the moves of the game, rather than being data associated with each play as in some models. We do not consider any justification of questions, since this is not required for the definition of well-bracketing.

We shall equip a game A of **SG** with a Q/A labelling $\ell_A^{QA}\colon M_A \to \{\mathsf{Q}, \mathsf{A}\}$, and a justification function $J_A\colon M_A \to M_A$ satisfying

$$\begin{gathered}\ell_A^{QA}(J(a)) \neq \ell_A^{QA}(a)\\ \ell_A^{OP}(J(a)) \neq \ell_A^{OP}(a)\end{gathered}$$

So for example, for the game $N_\perp$ we would choose to set $\ell_{N_\perp}^{QA}(q) = Q$, and for each $n \in \mathbb{N}$, $\ell_{N_\perp}^{QA}(n) = A$ and $J_{N_\perp}(n) = q$. The justification function serves only to associate questions and their corresponding answers, so that we can ensure that questions are answered in the correct order according to a bracketing discipline (the relation to control features is discussed in [53]).

We now define the notion of a well-bracketed play in A. Firstly, a play s is *fully bracketed* if it has no unanswered questions. Define FB as the least relation such that

$$\mathrm{FB}(\varepsilon) \qquad \mathrm{FB}(s) \wedge \mathrm{FB}(t) \Rightarrow \mathrm{FB}(J(a)sat)$$

A play s is well-bracketed if it contains no prematurely answered questions. Define

$$\mathrm{WB}(s) \iff \begin{array}{c} s = s_1 a s_2 \wedge \ell^{QA}(a) = \mathsf{A} \Rightarrow \mathsf{J(a)} \in \mathsf{s_1} \\ \text{and} \quad s = s_1 J(a) s_2 a s_3 \Rightarrow \mathrm{FB}(s_2) \end{array}$$

We define the category of well-bracketed games **BG** as follows. For each arena $A = \langle M_A, \ell_A^{OP}\rangle$ of **SG**, Q/A labelling ℓ_A^{QA} and a suitable justification function J_A, an arena of **BG** is a tuple $\langle M_A, \ell_A^{OP}, \ell_A^{QA}, J_A\rangle$. The language L_A of an arena is that of **SG** with the additional restriction that for each play s in L_A, $\mathrm{WB}(s)$. Given this new definition of L_A, a game is specified by the addition of a set of plays P_A as in **SG**. The resulting set of strategies R_A for A are then defined as in **SG**, and as before a morphism $A \to B$ is a strategy for $A \multimap B$.

Given games A and B, we revisit the definitions of $\otimes$ and $\multimap$: define the following games

$$\begin{aligned}
M_{A\otimes B} &= M_A + M_B \\
\ell^{OP}_{A\otimes B} &= [\ell^{OP}_A, \ell^{OP}_B] \\
\ell^{QA}_{A\otimes B} &= [\ell^{QA}_A, \ell^{QA}_B] \\
J_{A\otimes B} &= [J_A, J_B] \\
P_{A\otimes B} &= \{s \in L_{A\otimes B} \mid s\upharpoonright_A \in P_A,\ s\upharpoonright_B \in P_B\}
\end{aligned}$$

$$\begin{aligned}
M_{A\multimap B} &= M_A + M_B \\
\ell^{OP}_{A\multimap B} &= [\overline{\ell}^{OP}_A, \ell^{OP}_B] \\
\ell^{QA}_{A\otimes B} &= [\ell^{QA}_A, \ell^{QA}_B] \\
J_{A\otimes B} &= [J_A, J_B] \\
P_{A\multimap B} &= \{s \in L_{A\multimap B} \mid s\upharpoonright_A \in P_A,\ s\upharpoonright_B \in P_B\}
\end{aligned}$$

Notice that the only changes we have made are the addition of the Q/A labelling and the justification function J—the definition of the set of plays remains the same, because now the constraint $s \in L_{A\otimes B}$ etc. restricts to well-bracketed plays.

Proposition 2.4. BG *is a category.*

The definition of the identity strategy carries over and can be seen to be the identity (see the discussion below regarding copycat strategies), so the content of this proposition is that composition respects the bracketing condition. Firstly, observe that restriction of a fully bracketed sequence s on $A \multimap B$ to A or B must be a fully bracketed sequence. Since questions and answers in A and B are unrelated, if a question was closed early in A in $s\upharpoonright_A$ the pending question at that time could not be closed in s by a question in B, so the question was closed early in s too.

Here we show that WB is preserved by composition (a similar proof is given by Laird [53]). Assume for $\sigma\colon A \to B$ and $\tau\colon B \to C$ that WB(σ) and $WB(\tau)$ yet it is not the case that WB($\sigma;\tau$). Then there is some $sa \in \sigma\|\tau$ such that a answers a question q while there is some later question q' remaining unanswered, i.e. $s = s_1 q s_2 a$ with $q' \in s_2$.

Note by the switching condition on $\multimap$, an odd number of moves have to be played in B between playing a move in A and being able to play a move in C (and vice versa). Consequently, an even number of moves in B have to be played between two successive moves in A (or in C).

This means $s_2\upharpoonright_B$ must be of even length, and by well-bracketing of σ must be fully bracketed. So s_2 is of the form $q_1 t_1 a_1 q_2 t_2 a_2 \ldots q_n t_n a_n$ with q_i,a_i in B

(note that t_n can still contain moves in B, i.e. nested brackets). For any $q_i t a_i$, since $q_i t_i \upharpoonright_{B,C}\ a_i$ is in τ then t_i must be fully bracketed, and hence leave no pending questions. Since there are no pending questions from C, q cannot have been answered prematurely.

The same reasoning holds when q is in C, so it must be the case that $\mathrm{WB}(\sigma;\tau)$ after all.

2.3.2 Structure of BG

We now show that the structure defined for **SG** in Sections 2.1–2.2 can also be understood in the context of **BG**.

Proposition 2.5 (**BG** is nice)**.** *The object* 1*, functors* $\otimes, \multimap, \oslash, !, \&, \Sigma_X$*, and all the relevant morphisms defined earlier in* **SG** *restrict to* **BG***, making* **BG** *a SMCC with linear exponential comonad. The* Y *operator also restricts to* **BG***, while* U *is not a universal object in* **BG***.*

As for the definition of $A \multimap B$ above, we endow each of our type constructors with a Q/A labelling and a justification function, and reinterpret the definition given earlier in **BG**:

- For $1_{\mathbf{BG}}$ we must take the empty labelling and justification function.
- For $!A$ take $\ell^{OP}_{!A} = \bigsqcup_{i\in\mathbb{N}} \ell^{OP}_{A}$ and $J_{!A} = \bigsqcup_{i\in\mathbb{N}} J_A$.
- For $\&_{x\in X} A_x$ take $\ell^{OP}_{\&_{x\in X}A_x} = \bigsqcup_{x\in X} \ell^{OP}_{A_x}$ and $J_{\&_{x\in X}A_x} = \bigsqcup_{x\in X} J_{A_x}$.
- For $\Sigma_X A$ take $\ell^{OP}_{\Sigma_X A}(q) = \mathsf{Q}$ where q is the unique initial move, and for each $x \in X$ take $\ell^{OP}_{\Sigma_X A}(x) = \mathsf{A}$ and $J_{\Sigma_X A}(x) = q$. Furthermore, for any move $z \in M_A$, take $\ell^{OP}_{\Sigma_X A}(z) = \ell^{OP}_{A}(z)$ and $J_{\Sigma_X A}(z) = J_A(z)$ (where defined).

The action of the functors $\otimes, \oslash, !$ on morphisms was defined in terms of restriction of valid plays, and this carries over to **BG**, as the set of valid plays already imposes the bracketing restriction. For example for products, the following definition stands:

$$\sigma \otimes \tau = \{s \in P^{\text{even}}_{A\otimes B\multimap C\otimes D} \mid s\upharpoonright_{A\multimap C} \in \sigma,\ s\upharpoonright_{B\multimap D} \in \tau\}$$

In the case of $\&$, the morphism $\&_{i\in I} f_i$ is simply the union of the f_i which is clearly still correct, while for coproducts $[f_i]_{i\in I}$ the more explicit definition

$$[f_i]_{i\in I} = \{\varepsilon\} \cup \bigcup_{i\in I} \{qqis \mid qs \in f_i\}$$

can be seen to obey the bracketing condition for $\Sigma_{i\in I} A_i \to B$ as defined above.

Recall that we defined the bijection $\mathbf{SG}(A \otimes B, C) \cong \mathbf{SG}(A, B \multimap C)$ simply from the bijection on move sets, and this extends to $\mathbf{BG}(A \otimes B, C) \cong \mathbf{BG}(A, B \multimap C)$ as it agrees with the above definition of Q/A labelling and justification function as disjoint unions, giving the *eval* morphism and $\lambda(f)$ construction.

Informally, the morphisms defined in **SG** are well-behaved in **BG** because they never introduce a bracketing violation. Here we show that dynamic copycat strategies (and hence also static copycats[5]) are well-behaved. Consider a proposed dynamic copycat $f\colon \hat{F}(\vec{A}) \to \hat{G}(\vec{A})$, and a play $s \in f$. Consider an even subsequence qta of s, where $q = J(a)$: if a is a P-move, $J(a)$ must be an O-move, and the sequence must have the form $qq't'a'a$ where q' and a' are relabellings of q and a' via C_s. Since $\hat{F}$ and $\hat{G}$ are constructed from $\otimes$ and $!$, from the definition above each occurrence of an A_i in F and G inherits the same labelling and justification function as A_i. Therefore $q' = J(a')$, and by assumption there are no unanswered questions in t'.

Finally, note that the universal object in **SG** $U = !\&_N \Sigma_N 1$ has every move either an opponent question or the player answer to the immediately preceding opponent question, flattening the justification structure. The retractions $A \lhd U$ do not respect bracketing, so U is not a universal object in the **BG**.

2.4 Call by value games

We construct from **SG** a category $\mathbf{SG}^{\mathrm{V}}$ to model call-by-value computation by a simplification of the $\mathbf{Fam}(-)$ construction of [12], essentially considering just the subcategory of $\mathbf{Fam}(\mathbf{SG})$ where games are "constant families" of one repeated object $\{A \mid i \in I\}$. Our category thus lacks general coproducts (having only coproducts of a repeated object), and for these we should move to the richer setting of [12], but for our purposes this simpler construction suffices.[6]

Objects of $\mathbf{SG}^{\mathrm{V}}$ are pairs (A, X) of a countable set A and an object X of **SG**, and morphisms $f = \langle \bar{f}, \hat{f} \rangle\colon (A, X) \to (B, Y)$ are pairs of functions $\bar{f}\colon A \to B$ and $\hat{f}\colon A \to \mathbf{SG}(X, Y)$. The identity $id_{(A,X)}$ is $\langle id_A, \Lambda a.id_X \rangle$, and composition

[5]And those morphisms such as *skew*, *skproj*, *passoc* which are essentially copycats but which do not fit in our formal definition.

[6]An alternative choice would have been the technique of [42].

is defined by

$$\langle \bar{g}, \hat{g} \rangle \circ \langle \bar{f}, \hat{f} \rangle = \langle \ \bar{g} \circ \bar{f},\ \Lambda a.\ \hat{g}(\bar{f}(a)) \circ \hat{f}(a)\ \rangle$$

We can define a full inclusion $\mathbf{SG} \to \mathbf{SG}^{\mathrm{V}}$ by $X \mapsto (\{*\}, X)$, $f\colon X \to Y \mapsto \langle id, \hat{f} \rangle$ where $\hat{f}(*) = f$, and a full inclusion $\mathbf{Set} \to \mathbf{SG}^{\mathrm{V}}$ by $A \mapsto (A, 1_{\mathbf{SG}})$, $f\colon A \to B \mapsto \langle f, \Lambda a.id_1 \rangle$. Subsequently we will notationally confuse $\mathbf{SG}$ with the corresponding full subcategory of $\mathbf{SG}^{\mathrm{V}}$.

Similarly, we construct $\mathbf{BG}^{V}$ from $\mathbf{BG}$ by the same process. Much of the structure available in $\mathbf{SG}$ lifts straightforwardly to $\mathbf{SG}^{\mathrm{V}}$, and that of $\mathbf{BG}$ to $\mathbf{BG}^{V}$. We shall now review this and other important structure. We shall explicitly discuss $\mathbf{SG}$, but the following holds in $\mathbf{BG}$ also.

2.4.1 Symmetric monoidal closed structure

We define products

$$\begin{aligned} (A, X) \otimes (B, Y) &= (A \times B, X \otimes Y) \\ (A, X) \& (B, Y) &= (A \times B, X \& Y) \end{aligned}$$

The tensor product has unit $1 = (\{*\}, 1_{\mathbf{SG}})$, and both products lift to bifunctors by taking

$$f \otimes g = \langle \bar{h}, \hat{h}_1 \rangle \qquad f \& g = \langle \bar{h}, \hat{h}_2 \rangle$$

where

$$\begin{aligned} \bar{h}(a, b) &= (\bar{f}(a), \bar{g}(b)) \\ \hat{h}_1(a, b) &= \hat{f}(a) \otimes \hat{g}(b) \\ \hat{h}_2(a, b) &= \hat{f}(a) \& \hat{g}(b) \end{aligned}$$

Now we shall define $\multimap\colon (\mathbf{SG}^{\mathrm{V}})^{op} \times \mathbf{SG} \to \mathbf{SG}$—we do not give the full version of $\multimap$ which strictly speaking is required to give the symmetric closed structure. While it is possible to do so, we give the following restricted definition for simplicity, since we are only interested in objects of the form $(A, X) \multimap (B, Y)_\perp$ as used to interpret our call-by-value language.

Writing $\&_{a \in A} Z_a$ for the (countable) additive product over A in $\mathbf{SG}$, simply take

$$(A, X) \multimap (1, Y) = (1, X \multimap \&_A Y)$$

and note that

$$X \multimap \&_A Y \cong \&_A (X \multimap Y)$$

Now given a morphism

$$f = \langle \bar{f}, \hat{f} \rangle \colon (A, X) \otimes (B, Y) \to (1, Z)$$

we define

$$\begin{array}{rcl} \tilde{f} & : & A \to \mathscr{C}(X, \&_B(Y \multimap Z)) \\ & = & \Lambda a.\ \&_{b \in B}\ \lambda(\hat{f}(a, b)) \\ \lambda(f) & : & (A, X) \to (B, Y) \multimap (1, Z) \\ & = & \langle \lambda(\bar{f}), \tilde{f} \rangle \end{array}$$

Also define

$$\begin{array}{rcl} \mathit{eval}_{(A,X),(1,Y)} & : & (A, X) \multimap (1, Y) \otimes (A, X) \to (1, Y) \\ & : & (A, \&_A(X \multimap Y) \otimes X) \to (1, Y) \\ & = & \langle \Lambda a.*,\ \Lambda a.\ (\Pi_a \otimes \mathit{id}_X); \mathit{eval}_{X,Y} \rangle \end{array}$$

It is easy to verify that $\mathit{eval} \circ \lambda(f) \otimes \mathit{id} = f$ (cf. [12]).

We can lift the skew product, taking

$$(A, X) \oslash (B, Y) = (A \times B, X \oslash Y)$$

and corresponding definitions of $\mathit{skproj}_{A,B}$ and $\mathit{passoc}_{A,B,C}$. However, there is no sensible definition of

$$\mathit{skew}_{A,X} \colon (A, X) \oslash (A, X) \to (A, X) \otimes (A, X)$$

since the A-part must be given "up front", while skew must delay the choice whether to behave like γ or id until the first move is played.

2.4.2 Lift monad

We shall now define a lifting functor $\bot \colon \mathbf{SG}^{\mathrm{V}} \to \mathbf{SG}$. Simply take

$$\begin{array}{rcl} \bot(A, X) & = & \Sigma_A X \\ \bot(\langle \bar{f}, \hat{f} \rangle \colon (A, X) \to (B, Y)) & : & \Sigma_A X \to \Sigma_B Y \\ & = & \{\varepsilon\} \cup \{q_B q_A\} \cup \{q_B q_A a b s \mid \bar{f}(a) = b \wedge s \in \hat{f}(a)\} \end{array}$$

where q_A, q_B are the initial moves of $\Sigma_A X, \Sigma_B Y$ as in Section 2.1.6. Note that $\bot$ is an endofunctor on $\mathbf{SG}^{\mathrm{V}}$, as we can view $\mathbf{SG}$ as a subcategory of $\mathbf{SG}^{\mathrm{V}}$. We can easily define monoidal natural transformations $\mu \colon \bot\bot \to \bot$, $\eta \colon \mathrm{Id} \to \bot$, $\psi \colon \bot(-) \otimes \bot(-) \to \bot(- \otimes -)$ to make $\bot$ a (non-symmetric) monoidal monad.

$$\begin{array}{rcl}
\mu_{(A,X)} & : & \Sigma_1\Sigma_A X \to \Sigma_A X \\
& = & \{\varepsilon, q^r_A q_1, q^r_A q_1 a_1 q^l_A, q^r_A q_1 a_1 q^l_A a^l_A a^r_A, \ldots\} \\
\eta_{(A,X)} & : & (A,X) \to \Sigma_A X \\
& = & \Lambda a.\{\varepsilon, q_A a, \ldots\} \\
\psi_{(A,X),(B,Y)} & : & \Sigma_A X \oslash \Sigma_B Y \to \Sigma_{A\times B}(X \otimes Y) \\
& = & \{\varepsilon, q_{AB} q_A, q_{AB} q_A a q_B, q_{AB} q_A a q_B b(a,b), \ldots\}
\end{array}$$

Notice the type we have given ψ of $A_\perp \oslash B_\perp \to (A \otimes B)_\perp$;[7] this reflects the fact that the left component is always accessed first by ψ. We can get the usual type $A_\perp \otimes B_\perp \to (A \otimes B)_\perp$ by pre-composing with $\mathit{skproj}_{A_\perp, B_\perp} : A_\perp \otimes B_\perp \to A_\perp \oslash B_\perp$, and we can also get an "all skewed" version $\psi \colon A_\perp \oslash B_\perp \to (A \oslash B)_\perp$ by post-composing with $\perp(\mathit{skproj}_{A,B})$. We shall not distinguish these notationally since the meaning is clear from the types.

We shall also make use of the Kleisli operator, which we define from the above structure in the normal way:

$$\begin{array}{rcl}
f & : & A \to B_\perp \\
f^\dagger & : & A_\perp \to B_\perp \\
& = & \perp(f); \mu_B
\end{array}$$

Together with the skew product we introduced the notion of *strictness*. This is particularly relevant for types $A_\perp \to B_\perp$. Note that μ is strict, and for any f, $!f$ is strict, and consequently $f^\dagger$ is strict. Lastly, ψ is strict in both of its arguments. On the other hand, $g; \eta$ is *not* strict for any g.

There is also another connection with the skew product:

$$\begin{array}{rcl}
\Sigma_A X & \cong & \Sigma_A 1 \oslash X \\
(A,X)_\perp & \cong & (A,1)_\perp \oslash (1,X)
\end{array}$$

and hence note that

$$(A,X)_\perp \oslash (1,Y) \cong (A,1)_\perp \oslash (1, X \otimes Y)$$

i.e.

$$(A,X)_\perp \oslash Y \cong A_\perp \oslash (X \otimes Y)$$

Finally, there is a morphism

$$\mu_{\multimap} \colon (X \multimap Y_\perp)_\perp \to (X \multimap Y_\perp)$$

[7]We shall write $-_\perp$ for $\perp(-)$ on objects (but not morphisms).

which responds to the initial question q^x on the right-hand $Y_\perp$ by the initial question q on the left, and to the unique response a to that question by playing q^x on the left, and behaving as a copycat thereafter. Trivially for any $f\colon (X \multimap Y_\perp)$ it is the case that

$$\mu_\multimap \circ \eta \circ f = f$$

justifying the notation $\mu_\multimap$ by analogy to μ.

2.4.3 Linear exponential

We extend the ! operator of **SG** to $\mathbf{SG}^{\mathrm{V}}$. Since the set-part in $\mathbf{SG}^{\mathrm{V}}$ is intuitively copyable, we define

$$\begin{array}{rcl} !(A, X) & = & (A, !X) \\ !\langle \bar{f}, \hat{f} \rangle & = & \langle \bar{f}, \Lambda a.!\hat{f}(a) \rangle \end{array}$$

and lift δ, ε, d and m accordingly.

We note that it is no longer the case in general that $!A \cong A \oslash !A$, because the morphism $\mathit{fold}_A\colon A \oslash !A \rightarrow !A$ has no sensible action to take on the set part of A—but where the set part is trivial, as in the game $A \multimap B_\perp$, the morphism is still defined.

Since ! has a meaningful action on **SG**, we cannot expect to have

$$!(X_\perp) = (!X)_\perp$$

since considering the case of the object $(1, \mathbb{N})$, the second game is the usual game $N_\perp$ in **SG**, whereas the first is $!(N_\perp)$. Clearly there is more information in a strategy for $!(X_\perp)$ than $(!X)_\perp$, since any given component can go undefined or not independently of another. However, we can define a natural transformation

$$\mathit{dist}^{!\perp}\colon !\perp \rightarrow \perp!$$

which simply discards this additional information; this is still strong enough to be a *distributive law* for ! over $\perp$.

Being more explicit, we define a morphism

$$f\colon !\Sigma_A X \rightarrow \Sigma_A !X$$

where f responds to the initial q on the right with q in the first component on the left, and then plays copycat on the first components; when another component is opened on the right, f asks the initial q on that component on the left, ignores the answer and plays copycat thereafter. Note that this definition makes use of the fact that components of the exponential are opened in order.

We can also define a morphism

$$g\colon \Sigma_A !X \to !\Sigma_A X$$

which responds to the first opening q on the right by asking the opening q on the left and playing copycat thereafter, excepting that the opening q in subsequent components on the right receive the same response. We call the resulting natural transformation

$$\mathit{dist}^{\perp !}\colon \perp ! \to ! \perp$$

In summary, we have that $\perp ! \lhd !\perp$.

We shall later use $\mathit{dist}^{!\perp}$ with the co-Kleisli operator to go from $!X \to Y_\perp$ to $!X \to (!Y)_\perp$, so we define $f^\sharp = \mathit{dist}^{!\perp} \circ f^\ddagger$. Note that $\mathit{dist}^{!\perp} \circ !\eta_X = \eta_{!X}$ and $\mathit{dist}^{\perp !} \circ \eta_{!X} = !\eta_X$, and consequently for $f\colon !X \to Y$, $(\eta_Y \circ f)^\sharp = \eta_{!Y} \circ f^\ddagger$

2.4.4 Fixpoint Operator

Rather than defining a fixpoint operator on $\mathbf{SG}^{\mathrm{V}}$ we shall be able to use the operator from **SG**

$$Y_X\colon !(X \multimap X) \to X$$

Where $X = Y \multimap Z_\perp$ there is a morphism $\mu_\multimap\colon X_\perp \to X$, giving

$$!(id_X \multimap \mu_\multimap); Y_X\colon !(X \multimap X_\perp) \to X$$

2.4.5 Natural numbers and conditional

To model natural numbers, we take $N = (\mathbb{N}, 1_{\mathbf{SG}})$, and then for a function $\varphi\colon \mathbb{N}^k \rightharpoonup \mathbb{N}$ take $\bar{\varphi}\colon \otimes_{1,\ldots,k} N \to N_\perp$ the obvious morphism.

We define a conditional

$$\mathit{ifz}_A\colon N_\perp \otimes (A_\perp \& A_\perp) \to A_\perp$$

In fact the essence of this is

$$\mathit{ifz}^v_A\colon N \to (A_\perp \& A_\perp) \multimap A_\perp$$

where $\mathit{ifz}^v_A = \langle \Lambda n.*, f\rangle$ with

$$f = \Lambda n. \begin{cases} \Pi_L & \text{if } n = 0, \\ \Pi_R & \text{otherwise.} \end{cases}$$

We then define

$$\mathit{ifz}_A = (\bot(\mathit{ifz}^v_A); \mu_{\multimap})^*$$

and note that $\mathit{ifz}_A \circ (\bar{0} \otimes (f \& g)) = f$ and $\mathit{ifz}_A \circ (\bar{n} \otimes (f \& g)) = g$ for $n \neq 0$.

Note that we could have chosen a conditional of type

$$\mathit{ifz}_A \colon N_\bot \otimes (A_\bot \otimes A_\bot) \to A_\bot$$

but this is a restriction of that defined above, requiring us to be able to supply two $A_\bot$ arguments even though it only ever evaluates one. Worse, we could define a conditional of the above type which evaluates both arguments, while it is clear from the type of our *ifz* that only one $A_\bot$ argument is ever evaluated.

2.5 Memoisation

Here we introduce an operation on strategies designed to isolate certain portions of interest. This will be of use later to “memoise” a computation, in order to construct from a strategy σ and play s a strategy which behaves as σ would had it already undergone the interaction s. In Chapter 5 we will use this operation to extract the behaviour of a strategy after a certain heap interaction, in the course of our soundness proof. The name *memoisation* refers to the fact that the resulting strategy in this case starts with the same initial question as the original strategy, but has no need to repeat the heap interaction, and can instead return an answer immediately. The definitions and results in this section appear to be new.

Firstly we define a method of extracting the portion of a strategy starting after a given play. Informally, for a strategy σ, and an even-length play $s \in \sigma$, we construct the strategy σ_s by stripping out everything apart from play following s, as depicted in Figure 2.1. Observe that σ_s is generally not a *substrategy* of σ, since the prefix s has been removed from all plays. Indeed, σ_s may be a strategy for a different game entirely, namely the one which allows play to start at the correct point. To that end, our first definition is an operation on types. Given a game $X = \langle M_X, \lambda_X, P_X \rangle$ and a play $s \in P_X$ we give a new game X_s which "starts after s" as follows

$$\begin{aligned} X_s &= \langle M_X, \lambda_X, P_{X_s} \rangle \\ P_{X_s} &= \{t \mid st \in P_X\} \end{aligned}$$

Now we can define the operation on strategies as follows. For σ a strategy of type X, define a strategy of type X_s as $\sigma_s = \{t \mid st \in \sigma\}$. It is easy to see this is a legitimate strategy of that type, since prefix-closure is maintained.

Now we note that when $s \in \sigma \colon X \multimap Y$ has first and last moves in Y, then

$$(X \multimap Y)_s = (X_{s\restriction_X} \multimap Y_{s\restriction_Y})$$

since the only additional constraint on the right hand side is that play must start in Y. For $\sigma \colon X \to Y$ we then have $\sigma_s \colon X_{s\restriction_X} \to Y_{s\restriction_Y}$. There is a unique interleaving of $s\restriction_X$ and $s\restriction_Y$ in σ (namely s), since by the switching condition (Proposition 2.1) only Player may switch between X and Y in $X \multimap Y$, and the positions at which to switch are specified by σ. Therefore we use the convenient notation $\sigma^t_{t'}$ for σ_s such that $s\restriction_X = t$ and $s\restriction_Y = t'$ (notice that $\sigma^\varepsilon_s = \sigma_s$).

Now given strategies $X \xrightarrow{\sigma} Y \xrightarrow{\tau} Z$ and $s \in \sigma \| \tau$ we have

$$(\sigma;\tau)_{s\restriction_{X \multimap Z}} = X_{s\restriction_X} \xrightarrow{\sigma_{s\restriction_{X \multimap Y}}} Y_{s\restriction_Y} \xrightarrow{\tau_{s\restriction_{Y \multimap Z}}} Z_{s\restriction_Z}$$

This equality is easily seen from the definition of composition.

We shall now discuss the application of the above definition in the situation in which it will be most useful, namely with a pair of strategies $1 \xrightarrow{\sigma} Y \xrightarrow{\tau} Z_\perp$. In this case, we shall be interested in the situation after the initial play qa in $Z_\perp$. Suppose qsa is a play in $\sigma\|\tau$ (see Section 2.1.2) with q and a in $Z_\perp$ and s in Y.

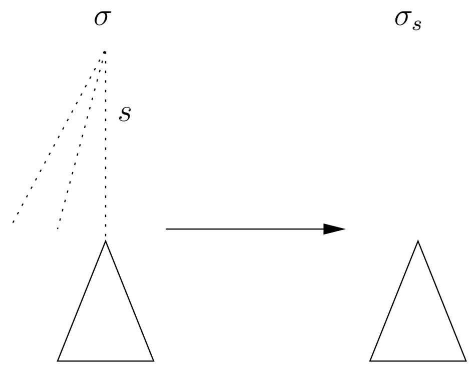


Figure 2.1: Memoisation

We note that $\sigma_{qsa{\upharpoonright}_{1\to Y}} = \sigma_s$. For τ, we wish to retain the qa, while still removing s, leaving us a strategy consisting of qa followed by τ^s. We shall denote this τ^s, defined simply as $\tau^s = \{qat \mid t \in \tau_{qsa}\}$. We note that this can be defined directly as $\tau^s = \{qat \mid qsat \in \tau\}$. The benefit of this construction is that the result type of τ^s is of the same type as that of τ:

$$\begin{aligned} \tau &: Y \to Z_\perp \\ \tau^s &: Y_s \to Z_\perp \end{aligned}$$

Then we have the following as (a trivial consequence of) a special case of the above, that given strategies $1 \xrightarrow{\sigma} Y \xrightarrow{\tau} Z_\perp$ and $qsa \in \sigma \| \tau$ we have

$$\sigma;\tau = 1 \xrightarrow{\sigma_s} Y_s \xrightarrow{\tau^s} Z_\perp$$

We shall in fact require an apparently stronger fact that, given play in some composition $\sigma;\tau;\upsilon$, we can equivalently memoise at the interaction between $(\sigma;\tau)$ and υ, or σ and $(\tau;\upsilon)$, or indeed at the interaction between each of σ, τ, and υ.

Lemma 2.6 (Resplitting). *For any strategies σ, τ and υ of the appropriate types, and for some interaction sequence $qsa \in \sigma \| \tau \| \upsilon$, the following diagram commutes:*

$$\begin{tikzcd}
 & X \arrow[rr, "\tau"] & & Y \arrow[dr, "\upsilon"] & \\
1 \arrow[ur, "\sigma"] \arrow[rr, "\sigma_{s{\upharpoonright}_X}"] \arrow[drr, "(\sigma;\tau)_{s{\upharpoonright}_Y}"'] & & X_{s{\upharpoonright}_X} \arrow[rr, "(\tau;\upsilon)^{s{\upharpoonright}_X}"] \arrow[d, "\tau_s"] & & Z_\perp \\
 & & Y_{s{\upharpoonright}_Y} \arrow[urr, "\upsilon^{s{\upharpoonright}_Y}"'] & &
\end{tikzcd}$$

Again this is from the definition of memoisation and composition (and its associativity).

Lemma 2.7 (Notation). *The following equalities hold for types and plays such that the memoisations involved are defined:*

$$\begin{array}{rclrclcl}
(X_{s_1})_{s_2} & = & X_{s_1 s_2} & X_\varepsilon & = & X & & \\
(\sigma_{s_1})_{s_2} & = & \sigma_{s_1 s_2} & \sigma_\varepsilon & = & \sigma & = & \sigma^\varepsilon \\
\tau_s^\varepsilon & = & \tau_s & (\tau^s)_{qa} & = & \tau^s_{qa} & = & \tau_{qsa}
\end{array}$$

Lemma 2.8 (Memoisation of the identity). *For any s, t, and any type X, if $(id_X)^s_t$ is well defined then $s = t$ and*

$$(id_X)^s_t = id_{X_t}$$

Proof. If $(id_X)^s_t \colon X_s \to X_t$ is indeed well-defined, it must be the case that $s = t$, since the interleaving of s and t is in id_X. Removing this initial portion of the identity strategy, one is clearly left with the identity on the remaining game X_t. □

Lemma 2.9 (Memoisation preserves isomorphism). *If $X \cong Y$ then for a play t in X there is a play $\tilde{t}$ in Y such that $X_t \cong Y_{\tilde{t}}$.*

Since there is a play t in X, there is a morphism $(id_X)^t_t \colon X_t \to X_t$. If the isomorphism comprises $f : X \to Y$ and $g \colon Y \to X$, then $(id_X)^t_t = (f; g)^t_t$, and by Lemma 2.6 there is $\tilde{t}$ to split this as $(f; g)^t_t = f^t_{\tilde{t}}; g^{\tilde{t}}_t$. By Lemma 2.8 $(id_X)^t_t = id_{X_t}$, so $f^t_{\tilde{t}}; g^{\tilde{t}}_t = id_{X_t}$. Similarly $g^{\tilde{t}}_t; f^t_{\tilde{t}} = id_{Y_{\tilde{t}}}$, hence $X_t \cong Y_{\tilde{t}}$.

Lemma 2.10 (Memoisation of pairs). *For any game $X \otimes X'$ and even play t admitted by that game, there exist plays t_1, t_2 along with a canonical isomorphism*

$$(X \otimes X')_t \cong X_{t_1} \otimes X'_{t_2}$$

Furthermore, for any morphisms f, g and plays t, u with $(f \otimes g)^t_u \colon (X \otimes X')_t \to (Y \otimes Y')_u$, and t_1, t_2, u_1, u_2 as given above, the following diagram commutes

$$\begin{tikzcd}
(X \otimes X')_t \arrow[r, "(f \otimes g)^t_u"] \arrow[d, phantom, "\cong"] & (Y \otimes Y')_u \arrow[d, phantom, "\cong"] \\
X_{t_1} \otimes X'_{t_2} \arrow[r, "f^{t_1}_{u_1} \otimes g^{t_2}_{u_2}"] & Y_{u_1} \otimes Y'_{u_2}
\end{tikzcd}$$

Proof. From the definition of pairing, t is an interleaving of moves in X and X', so the restriction to each of X, X' is a play in that game. Thus take $t_1 = t\restriction_X$ and $t_2 = t\restriction_{X'}$. Then note that continued play $tt' \in X \otimes X'$ arises from interleaving of some $t_1 t'_1 \in X$, $t_2 t'_2 \in X'$, so the above memoisations coincide. In the case of morphisms, again pairing is simply interleaving, and the same reasoning applies. □

Lemma 2.11 (Memoisation of skewed products). *For any skewed product $X \oslash X'$ and even play $t \neq \varepsilon$ admitted by that game, there exist plays t_1, t_2 along with a canonical isomorphism as above*

$$(X \oslash X')_t \cong X_{t_1} \otimes X'_{t_2}$$

Furthermore, for any pair of morphisms f, g and plays $t \neq \varepsilon$, $u \neq \varepsilon$ such that $(f \oslash g)^t_u \colon (X \oslash X')_t \to (Y \oslash Y')_u$, and t_1, t_2, u_1, u_2 as given above, the following diagram commutes

$$\begin{tikzcd}
(X \oslash X')_t \arrow[r, "(f \oslash g)^t_u"] \arrow[d, phantom, "\cong" rotate=90] & (Y \oslash Y')_u \arrow[d, phantom, "\cong" rotate=90] \\
X_{t_1} \otimes X'_{t_2} \arrow[r, "f^{t_1}_{u_1} \otimes g^{t_2}_{u_2}"] & Y_{u_1} \otimes Y'_{u_2}
\end{tikzcd}$$

This is a trivial consequence of Lemma 2.10, since $(X \oslash Y)_t = (X \otimes Y)_t$ where $t \neq \varepsilon$, the first move in t having satisfied the "left first" requirement of $\oslash$.

The memoisation of a reusable object has one rather useful characteristic: the result can be viewed as an object of the same type for future use, plus a (non-reusable) "everything else" game for continued interaction with those components which have already been opened. In the case of an object with methods, this will correspond to the updated object and continued interaction with the argument or results of any earlier method invocation.

Lemma 2.12 (Memoisation residue)**.** *For any even play t in any reusable object $!X$, there exists an object Z_t along with a canonical isomorphism*

$$(!X)_t \cong Z_t \otimes !X$$

where Z_t consists of any continuing play in components of $!X$ occurring in t, and the right-hand $!X$ consists of all untouched components. For some n, Z_t has the following form

$$Z_t = \bigotimes_{0 \leq i < n} (X)_{t_i}$$

where $t_i = t\restriction_{X_i}$

Proof. The play t must open some number n of components of $!X$ (possibly 0), i.e. t will contain moves in each of components $1, \ldots, n$. By repeated unfolding

$$!X \cong X \oslash \cdots \oslash X \oslash !X$$

and then if $t_i = t\restriction_{X_i}$, by Lemma 2.11 we have

$$(!X)_t \cong (X \oslash \cdots \oslash X \oslash !X)_t = (X)_{t_1} \otimes \cdots \otimes (X)_{t_n} \otimes !X$$

□

In fact we can lift the above lemma to products.

Lemma 2.13 (Product-memoisation residue)**.** *For any even play t in any product of reusable objects $\bigotimes_{i \in I} !X_i$, there exists an object Z_t along with a canonical isomorphism*

$$\left(\bigotimes_{i \in I} !X_i \right)_t \cong Z_t \otimes \bigotimes_{i \in I} !X_i$$

and

$$Z_t = \bigotimes_{\substack{i \in I \\ 0 \leq j < n_i}} (X_i)_{t_{i,j}}$$

where $t_{i,j} = t \upharpoonright_{(X_i)_j}$

Proof. For each component i there is a play $t_i = t \upharpoonright_{!(X_i)}$ to which the above reasoning holds. Now apply Lemma 2.10. □

2.5.1 Memoisation in BG

We observe that a restricted version of the memoisation operation carries over to **BG**. Define X_s as above, but only where s is fully bracketed (FB(s)). This ensures that plays of X_s are well-bracketed. Similarly for $\sigma \colon X \to Y$ we can define $\sigma_s \colon X_{s\upharpoonright_X} \to Y_{s\upharpoonright_Y}$ as before since for a fully bracketed s, $s \upharpoonright_X$ and $s \upharpoonright_Y$ are fully bracketed. For $\sigma \colon X \to Y_\perp$ we define σ^s as before for FB(s) since of course FB(qsa). The above Lemmas can then be seen to hold in **BG** where the requirement "even play" is replaced by "fully-bracketed play" throughout.

2.6 Notation

We shall fix some helpful notation for use in later chapters.

We write z_n for the move z in the n^{th} Z component of an exponential $!Z$. We write q^z for the initial question move q in the z-component of a game

$$(X, A)_\perp = \&_Z \Sigma_X A$$

where $z \in Z$ (thinking of this as the question supplying the value z). If $X = 1$ we simply write q for q^*. We take all moves after q^z to implicitly be in the z-component without further indication (by the definition of the game there is no other possibility, but formally every later move is labelled with the same z). For $x \in X$ we write a^x for the corresponding answer in $\&_Z \Sigma_X A$.

Definition 2.14 (Termination)**.** Call a play qsa^x of $X \multimap Z_\perp$ *terminating* if qa^x is an initial play in $Z_\perp$. Say the morphism $f\colon X \to Z_\perp$ terminates if it admits such a terminating play.

Note that the terminating plays of f are just the minimal non-ε fully-bracketed ones. If a morphism $f\colon 1 \to Z_\perp$ terminates then it admits a unique terminating play qa^x. Lastly, $f;\eta$ terminates for any f.

Chapter 3

A strategy for data abstraction

In this chapter we shall explore the issue of modelling stateful behaviour in the category $\mathbf{BG}^V$ of well-bracketed games defined in the previous chapter.

We define an operation which creates a stateful object from a "functional" implementation of that object, giving a semantic treatment of *data abstraction*. We identify an important restriction on the notion of "implementation", then prove some characteristic properties of the operation.

3.1 Modelling stateful behaviour

We intend to model expressions with state, and in particular our language will include stateful objects. Consider for instance the type

$$!(X \multimap Y_\perp)$$

This is the type of a function which can be used multiple times, but the nature of our ! means that the different uses need not behave in the same manner, and indeed may alter their behaviour based on the history of the other uses, and so the function can behave as if it has some internal *state*.

One can also consider this behaviour to be a property of our product $\otimes$, in that it allows *interfering* behaviour between its two components. Since play in one component can affect future behaviour of the other component, the two components of the product can be seen to share some internal state, as in the contraction map $!X \to !X \otimes !X$.

How does one construct a strategy of the above type which exploits this behaviour? There are no interesting such strategies given in the previous chapter, and the structure described there is not sufficient to construct one.[1]

One approach would be to choose some strategy (or family of strategies) which have stateful behaviour, and introduce these as constants when we define our calculus. For example, one could imagine defining a strategy which implements some kind of ML-style reference cell, and then consider a language with references—this is the approach taken in [10]. Such a strategy might have the following type:

$$\mathit{cell}_X \colon 1 \to !(!X \multimap 1_\perp \,\&\, 1 \multimap (!X)_\perp)$$

We could only hope to construct reference cells for types of the form $!X$, since a value may be read many times before it is replaced. In fact we cannot even do that—with our notion of games, it is only possible to define such a strategy for *ground* type reference cells (such as natural numbers). We will discuss this issue later, as it is still relevant in the context of our eventual solution.

Rather than contenting ourselves with ground-type reference cells, we choose instead to define an operation on strategies, which will internalise some specified stateful behaviour. Consider a strategy

$$\sigma \colon !(S \otimes X \multimap (S \otimes Y)_\perp)$$

where σ is a reusable operation which given an initial state along with an argument returns an updated state together with a result. Given some initial state $s \colon S$ we shall construct a new strategy

$$\hat{\sigma} \colon !(X \multimap Y_\perp)$$

The state-transforming behaviour of σ will be hidden within $\hat{\sigma}$, so that $\hat{\sigma}$ exhibits stateful behaviour as defined by σ.

This operation will in fact allow us to construct ground type reference cells as described above, as well as other interesting strategies (but not general reference cells) at higher types.

3.2 A state-threading operator

We implement the operation discussed above as a family of morphisms

$$\mathit{thread} \colon S \otimes !(S \otimes X \multimap (S \otimes Y)_\perp) \to !(X \multimap Y_\perp)$$

[1]The *fold* morphism goes some way towards this by allowing construction of non-uniform strategies, but cannot be used to establish the interdependence between components which would be required to model a stateful object.

The name *thread* comes from the desired behaviour, which is that the state is *threaded* through successive invocations of a method implementation (we postpone for now discussion of objects with multiple methods). The initial state is passed as an argument to the first method invocation, and the state returned is then passed as argument to the second, and so on.

The notion of a method implementation described here is subtly different from the programmer's usual notion, as we are really talking about the behaviour of a single invocation on the object's external interface. Any recursive calls (including mutually recursive calls) will be "compiled in" by the time we wish to perform the above threading procedure, with state being passed around internally in a functional manner.[2] Therefore for the remainder of this chapter we shall not have to refer to these recursive calls again, but it should be understood that the interpretation of Chapter 4 handles these as one would expect.

On the other hand, the issue of *re-entrant* methods is more subtle. In this case a second method invocation may genuinely occur at the point of performing the threading operation. In particular, what happens if the second invocation starts before the first finishes? In the case of such *nested* invocations there will be no result state available to supply to the second invocation. There are two possible responses to this: either the original state is duplicated (requiring that S be a reusable object of the form $!S'$) or *thread* simply makes no response. We shall deal with the former possibility in the next section and for the remainder of the chapter, but first we discuss the latter "avoidance" strategy.

In order to avoid dealing with nested method calls, one would need either to establish that they would not occur simply from context—we are after all dealing with a sequential language—or to impose some restriction to ensure this. Unfortunately, nesting can indeed occur. Consider possible play in the game $!(X \multimap Y_\perp)$. After the initial question in the first component $X \multimap Y_\perp$ is asked by Opponent in $Y_\perp$, there is some play in the argument X ending in a Player move there. Opponent may then opt to ask the initial question in the second component—not only is this a valid move in the game, but this strategy is a perfectly sensible one. Such a strategy corresponds to a dependence of the supplied

[2]More precisely by "compiled in" we are referring to the process of taking the fixed point of the approximation operator corresponding to the class implementation, as discussed in Section 4.1.

argument in X on the object in question, such as in an expression of the form

$$o \cdot m(\lambda x.\ o \cdot m(\ldots))$$

We hope the reader will understand this syntax informally, but a concrete example of the above scheme will be discussed in Section 4.2.1, in the context of our language.

Fortunately, the above discussion suggests a restriction which shall suffice to prohibit nesting. If X contains no interesting computational behaviour (i.e. is of *ground type*, being simply a set of values together with the empty game) then the above scenario cannot occur. A little thought will reveal that such an indirect interaction via the method argument is in fact the only circumstance in which which nesting can occur. This restriction also appears in the language context in Section 4.2.1.

Given this restriction, we now show how to implement this *linear thread* operation. By linear, we mean that there is only ever one invocation in progress at a time, with no nesting permitted. Consequently, the state S need never be duplicated by *thread* itself, and so is permitted to be of linear type rather than of the form $!S$.

Rather than giving a direct definition in terms of game plays, we define the operation recursively as a composition of simpler morphisms, including a special morphism *pproj* not definable from the structure defined in Chapter 2, which we define here for this purpose. This structured definition is both easier to understand and to reason about than one given directly on game plays. The more general non-linear *thread* operation to be introduced in Section 3.3 will build upon this definition, including that of *pproj*, so this is a useful stepping-stone and should help to explain the more complex definition later.

3.2.1 Partial projection

We define a "partial projection" morphism

$$\mathit{pproj} \colon X \multimap (Y \otimes Z)_\perp \to (X \multimap Y_\perp) \oslash Z_\perp$$

We shall use this to access the state resulting from a method invocation.

The $X \multimap Y_\perp$ part of *pproj* is essentially a projection, with

$$\Pi_{X \multimap Y_\perp} \circ \mathit{pproj} = \mathit{id}_X \multimap (\Pi_Y)_\perp$$

However, the other part ($Z_\perp$) is not a genuine projection, not always being very well behaved; nevertheless if the following two conditions hold, the Z component can be thought of as a projection:

- The result is accessed in a fashion one could deem "sequential" (that is when the second component is only accessed *after* a value is returned in $Y_\perp$).
- The Z component of the argument does not depend on (or otherwise cause a move in) the X argument after the initial answer is given.

If the two conditions do not hold, at some point *pproj* will go undefined. One could imagine replacing $Z_\perp$ with $(1+Z)_\perp$ so that *pproj* could explicitly give a "no result" answer if the first of these conditions does not hold, but for our purposes the simple solution suffices.

The requirements are automatically satisfied when X is a "basic game" of the form $(A, 1)_\perp$, since in that case there is no play between question and response in X (notice that the $\oslash$ in the type is important). When X is a computationally interesting game the situation is more complex—in the next section we shall ensure that the requirements are satisfied when we use *pproj*.

The definition of *pproj* is given as a function on odd-length plays as follows. The set part of both $X \multimap (Y \otimes Z)_\perp$ and $(X \multimap Y_\perp) \oslash Z_\perp$ is trivial, so *pproj* is really just a strategy in **BG**, and we give it directly as such here. We label a move z in the instances of Z on the left and right as z^L and z^R respectively (and write z_1, z_2 for two arbitrary Z-moves); on the other hand, we confuse the two instances of the games X and Y, where we simply define a copycat behaviour.

Recall that we write q^x for an initial question in $X \multimap W_\perp$ carrying the ground-type datum x (where x is in the set part of X, and W is either $Y \otimes Z$ or Y). Similarly we write a^w for the corresponding answer in $W_\perp$ carrying ground-type datum $w \in W$.

$$\begin{array}{rcll}
pproj(q^x) & = & q^x & \\
pproj(sa^{y,z}) & = & a^y & \\
pproj(q^x q^x sa^{y,z} tq) & = & a^z & \\
pproj(sx) & = & x & \\
pproj(sy) & = & y & \\
pproj(sz^R) & = & z^L & \\
pproj(sz_1^R z_1^L z_2^L) & = & z_2^R & \\
pproj(q^x q^x sq) & = & \perp & (\nexists a^{y,z} \in s) \\
pproj(sz_1^R z_1^L x) & = & \perp & (x \in X)
\end{array}$$

The two non-response ($\perp$) cases correspond to the two requirements above, in the same order. The first case represents a request for the result in Z "too early", i.e. before an answer has been given. The second is less obvious; this represents a dependency of Z on X. It might seem that we could simply copy the X-move on the left back over to the right, but this is not a valid move of the game $X \multimap Y_\perp$. It is a violation of the switching condition—the game $X \multimap Y_\perp$ does not allow Opponent to switch play into X. Such a move would represent a kind of spontaneous interaction in a function argument not initiated by the function.

This is the fundamental reason why a strategy for a higher-order state cell cannot be defined in **SG**. As we discussed in Section 1.6.4, there are more liberal game models which allow such behaviour. This does come at the cost of added complexity; here instead we stay with our more restricted notion of games, and explore how much can be done without making such "bad" moves.

We define the the abbreviation $pproj^\gamma$ as $(id \multimap \gamma); pproj$, since $pproj$ projects the wrong component for use in our definitions.

3.2.2 Partial application

We shall give another auxiliary definition for use in the definition of *linthread* itself. Define the "left partial application" morphism

$$eval^L_{X,Y,Z} = (X \otimes Y \multimap Z) \otimes X \xrightarrow{\lambda_Y(eval_{X\otimes Y,Z})} Y \multimap Z$$

And note that

$$[f \otimes (x \otimes y)]; eval = [(f \otimes x) \otimes y]; [eval^L \otimes id]; eval$$

Which is to say that the following diagram commutes:

$$\begin{tikzcd}
{[(X \otimes Y) \multimap Z] \otimes X \otimes Y} \arrow[d, "eval^L_{X,Y,Z} \otimes id_Y"'] \arrow[dr, "eval_{X \otimes Y, Z}"] & \\
{[Y \multimap Z] \otimes Y} \arrow[r, "eval_{Y,Z}"] & Z
\end{tikzcd}$$

As for *eval*, we can give $eval^L$ a skewed type

$$eval^L_{X,Y,Z} \colon (X \otimes Y \multimap Z) \oslash X \to Y \multimap Z$$

since the first move on the left must always be in Z, copied from the Z on the right side. Or in terms of the skew-typed *eval*:

$$eval^L_{X,Y,Z} = \lambda_Y(skproj; passoc^{-1}; eval_{X \otimes Y, Z})$$

3.2.3 Pseudopromotion

Before giving the definition of *linthread*, we deal with one final technical detail. Given

$$linthread \colon S \otimes !(S \otimes X \multimap (S \otimes Y)_\perp) \to !(X \multimap Y_\perp)$$

and a state $s \colon S$, we can construct an object of type $!(X \multimap Y_\perp)$. Given a state $s' \colon S_\perp$ (as would arise from a use of *pproj*), one could promote *thread* to get an object of type $[!(X \multimap Y_\perp)]_\perp$. However, it will be convenient to have a non-$\perp$ result object, giving the following type:

$$linthread \colon S_\perp \otimes !(S \otimes X \multimap (S \otimes Y)_\perp) \to !(X \multimap Y_\perp)$$

In general, we can refrain from evaluating the state until the first method is called on the result object, and we produce a definition which does so below. In the cases we are interested in, the state will always be defined.

We give the definition more generally. For $f \colon X \otimes Y \to !(Z \multimap Z'_\perp)$ we define $f^\star \colon X_\perp \otimes Y \to !(Z \multimap Z'_\perp)$ as:

$$f^\star = (id \otimes \eta); \psi; \perp(f); \lambda^{-1}; !(\mu_\multimap)$$

Note that

$$([g; \eta] \otimes h); f^\star = (g \otimes h); f$$

since $(\eta \otimes \eta); \psi = \eta$, $\eta; \perp(f) = f; \eta$, $\eta; \lambda^{-1} = \eta$ and $\eta; \mu_\multimap = id$.

$$linthread\colon !S \otimes !(S \otimes X \multimap (S \otimes Y)_\perp) \to !(X \multimap Y_\perp)$$

$$\begin{tikzcd}
S \otimes !(S \otimes X \multimap (S \otimes Y)_\perp) \arrow[d, "id \otimes unfold"'] \\
S \otimes [(S \otimes X \multimap (S \otimes Y)_\perp) \oslash !(S \otimes X \multimap (S \otimes Y)_\perp)] \arrow[d, "reassoc; passoc^{-1}; (eval^L \oslash id)"'] \\
(X \multimap (S \otimes Y)_\perp) \oslash !(S \otimes X \multimap (S \otimes Y)_\perp) \arrow[d, "pproj^\gamma \oslash id"'] \\
((X \multimap Y_\perp) \oslash S_\perp) \oslash !(S \otimes X \multimap (S \otimes Y)_\perp) \arrow[d, "passoc"'] \\
(X \multimap Y_\perp) \oslash [S_\perp \otimes !(S \otimes X \multimap (S \otimes Y)_\perp)] \arrow[d, "id \oslash linthread^\star"'] \\
(X \multimap Y_\perp) \oslash !(X \multimap Y_\perp) \arrow[d, "fold"'] \\
!(X \multimap Y_\perp)
\end{tikzcd}$$

Figure 3.1: Linear thread definition

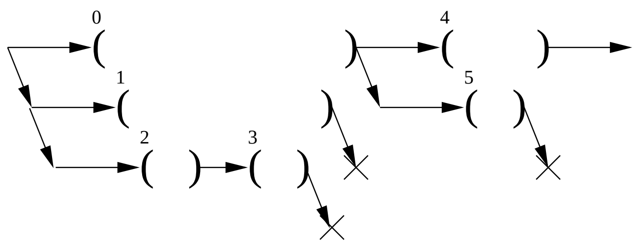


Figure 3.2: Behaviour of nested methods

3.2.4 The linear "thread" operation

We may now define *linthread* recursively as the least fixed-point of the circular definition given by Figure 3.1, as per Section 2.2.5.

The general structure of *linthread* is to peel off the first component of the source exponential, which will give rise to the first component of the destination exponential, and recursively handle the other components. The first component is used for the first invocation of the function in question; the state resulting from this is then (recursively) supplied as the initial state to subsequent invocations.

The first component is extracted with *unfold*, and the initial state is supplied via the appropriate (partial) evaluation map. The result of this partial evaluation is then split by *pproj*; the function part becomes the first component of the destination exponential, while the state becomes the initial state for the recursive use of *linthread*.

We make use of the following reassociation morphism:

$$reassoc\colon A \otimes (B \oslash C) \xrightarrow{\gamma; skproj; passoc; (id \oslash \gamma)} B \oslash (A \otimes C)$$

and note that since the morphisms involved are strict, the morphisms $pproj \oslash id$ and $\gamma; eval^L \oslash id$ are well-defined. Then Figure 3.1 defines *linthread*.

3.3 Dealing with nesting

As suggested above, nested method calls are perfectly reasonable behaviour, which we shall handle correctly, but there is some subtlety and we must be careful in our definitions. Figure 3.2 shows the required behaviour of *thread* in the presence of a particular sequence of nested invocations. Pairs of opening and closing parentheses "(   )" on the same line represent the initial question/answer pair of a method, i.e. its call and return, and the calls are numbered chronologically.

Consider first the calls 0 and 4; the arrow going into the start of 0 represents the initial state, and the arrow exiting 0 and entering 4 represents the returned state from call 0 being fed into call 4. However, at the point of call 1 the previous call has not returned, and so no updated state is available; the only choice allowing progress to be made is to duplicate the initial state and feed that into call 1 also. Again, call 2 occurs before 1 returns, and the initial state is again duplicated and fed into call 2, however this returns before call 3 occurs, and so the updated state from call 2 is given to call 3.

As mentioned earlier, this duplication of state is the reason we must restrict *thread* to states with types of the form $!S$. In particular, the game N which we will use to interpret natural numbers is of this form, as it may be useful to think of states as simply natural numbers in order to understand the behaviour of *thread*.[3]

Figure 3.2 illustrates another important aspect of nested calls, which is that state updates from nested method calls do not propagate to their containing calls. This occurs with the state returned from calls 1, 3, and 5 in this diagram. Consider the most deeply nested, call 3—there is no way for this call to return its result state to call 1, since 1 has already received its input state. There is no other useful thing to do with this result state, since it is going to be overwritten by the parent invocation (in this case, by call 0). We are left with no other choice but to discard the state.

The fact that the state updates from nested calls are discarded is an unfortunate consequence of the decision to model the concrete implementation of a method as a state-transforming operation of the form $!S \otimes X \to (!S \otimes Y)_\perp$. This contrasts with conventional OO languages such as Java, where a method may read or update the instance variables of its object at any time. The restriction described here corresponds to only permitting methods which copy the instance variables into a local variable at the start of a method, and copy the modified version back at the end. Coping with more general behaviour would entail a more fine-grained modelling of method implementations, which in turn would complicate reasoning about program behaviour. Note that for ground types, it is possible to define a reference cell via *thread*, with which one can achieve the effect of Java-style state updates—it is only at higher types that this is a genuine

[3]In most cases one would wish the state to be copyable, in order to both retain the state and perform some computation based upon it, but one could imagine that there might be exceptions to this general usage.

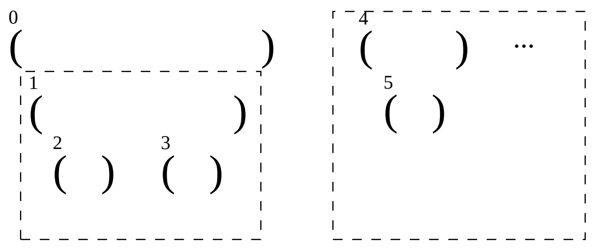


Figure 3.3: Categorisation of method calls

restriction.

On the other hand, the view of methods as explicitly state-transforming functions is shared by encodings of objects using existential types. Here the type of an object ($\exists S.\ S \times (S \to F(S))$ in the simplest encoding) explicitly includes a state, encapsulated by an existential type, and method implementations are state transformers for this hidden internal state. This suffers the issues with regards to nesting we have just described. However, in contrast to the existentially quantified state type in these encodings, the objects that we construct do not even reveal that their implementation uses some (unspecified) internal state. Not only do the types not mention such a thing, a strategy for the game $!(X \multimap Y_\perp)$ contains no inherent state—we may make use of a state to construct such a strategy, but this is not apparent in the behaviour of the resulting object.

In Chapter 6 we shall discuss a language extension which permits the definition of more expressive method implementations. In brief, we can relax the restriction that methods may only read the state at the start, allowing instead for this to happen at any point. This means that there the result states which were discarded in Figure 3.2 may instead be read by the enclosing method.

3.3.1 Branch

We give one more specialised morphism *branch* not definable using the structure from the previous chapter. Our new definition of *thread* will involve two recursive calls: one to handle the nested methods corresponding to 1–3 in Figure 3.3, and one to handle the remaining methods corresponding to 4–5 and beyond. The allocation of a given method to one of these recursive calls will be performed by *branch*.

Given the pair of recursively defined objects $!Z \otimes !Z$, *branch* will construct an object $!Z$ by associating Z components on the right with Z components on the left, and behaving in a *dynamic copycat* fashion. For a newly opened component of $!Z$, there are always two choices in $!Z\otimes!Z$; *branch* will allocate each component of $!Z$ to the left side of $!Z\otimes!Z$ until the main method call has returned an answer, and then switch to allocating each component to the right side.

We give *branch* the following type. The *branching* will be solely on whether the first argument (of type $X \multimap Y_\perp$) has answered the initial question in $Y_\perp$. This first argument is only present so *branch* can spy on it watching for termination—*branch* will be a copycat there.

$$\mathit{branch} \colon (X \multimap Y_\perp) \oslash (!Z \otimes !Z) \to (X \multimap Y_\perp) \oslash !Z$$

We label a move m in the left or right-side $X \multimap Y_\perp$ as m^L, m^R respectively, and a move z in the nth component of one of the three $!Z$ instances as as z_n^{LL}, z_n^{LR}, or z_n^R in turn. We assume that Z has a trivial set-part, and again give a definition in **SG**:

$$\begin{array}{rcll}
\mathit{branch}(sm^R) & = & m^L & \\
\mathit{branch}(sm^L)) & = & m^R & \\
\mathit{branch}(sz_n^R) & = & z_n^{LL} & ((a^y)^R \notin s) \vee \exists z'.{z'}_n^{LL} \in s \\
\mathit{branch}(sz_n^{LL}) & = & z_n^R & \exists z'.{z'}_n^{LL} \in s \\
\mathit{branch}(s(a^y)^R t z_n^R) & = & z_{n-k}^{LR} & k = B(s), n \geq k \\
\mathit{branch}(s(a^y)^R t z_{n-k}^{LR}) & = & z_n^R & k = B(s), n \geq k
\end{array}$$

where

$$B(s) = \max\left(\{m+1 \mid \exists z.z_m^{LL} \in s\} \cup \{0\}\right)$$

In the definition of *thread* we shall in fact only make use of *branch* with $Z = (X \multimap Y_\perp)$.

3.3.2 The non-linear "thread" operation

We implement the behaviour of *thread* as described above by a family of morphisms

$$\mathit{thread} : !S \otimes !(!S \otimes X \multimap (!S \otimes Y)_\perp) \to !(X \multimap Y_\perp)$$

The language defined in Chapter 4 will include objects with multiple methods, to be interpreted using *thread*. For this reason, we extend *thread* to an operation

$$\begin{aligned}
Z &= \&_{m\in M}(!S\otimes X_m \multimap (!S\otimes Y_m)_\perp)\\
W &= \&_{m\in M}X_m \multimap (Y_m)_\perp\\
\mathit{thread} &: \ !S\otimes !Z \rightarrow !W
\end{aligned}$$

$$\begin{tikzcd}
!S\otimes !Z \arrow[d, "d_S \otimes \mathit{unfold}"'] \\
!S\otimes !S \otimes [\&_{m\in M}(!S\otimes X_m \multimap (!S\otimes Y_m)_\perp)\oslash !Z] \arrow[d, "id_{!S} \otimes [\mathit{reassoc}; \mathit{passoc}^{-1}; (\mathit{dist} \oslash id)]"'] \\
!S \otimes (\&_{m\in M}[(!S\otimes X_m \multimap (!S\otimes Y_m)_\perp)\oslash !S]\oslash !Z) \arrow[d, "id_{!S} \otimes (\&_{m\in M}(\mathit{eval}^L \circ \Pi_m) \oslash id)"'] \\
!S \otimes [\&_{m\in M}(X_m \multimap (!S\otimes Y_m)_\perp)\oslash !Z] \arrow[d, "id \otimes (id \oslash d)"'] \\
!S \otimes ([\&_{m\in M}(X_m \multimap (!S\otimes Y_m)_\perp)] \oslash [!Z\otimes !Z]) \arrow[d, "\mathit{reassoc}"'] \\
\&_{m\in M}(X_m \multimap (!S\otimes Y_m)_\perp) \oslash [!S\otimes !Z\otimes !Z] \arrow[d, "id \oslash (\mathit{thread} \otimes id)"'] \\
\&_{m\in M}(X_m \multimap (!S\otimes Y_m)_\perp) \oslash [!W\otimes !Z] \arrow[d, "(\&_{m\in M}(id_{X_m} \multimap \gamma_{!S,Y_m}); \mathit{pproj}^{\gamma}) \oslash id"'] \\
[\&_{m\in M}(X_m \multimap (Y_m)_\perp) \oslash (!S)_\perp] \oslash [!W\otimes !Z] \arrow[d, "\mathit{passoc}; [id \oslash (\gamma \otimes id)]"'] \\
\&_{m\in M}(X_m \multimap (Y_m)_\perp) \oslash [!W \otimes (!S)_\perp\otimes !Z] \arrow[d, "id \oslash (id \otimes \mathit{thread}^\star)"'] \\
W \oslash [!W\otimes !W] \arrow[d, "\mathit{branch}"'] \\
W\oslash !W \arrow[d, "\mathit{fold}"'] \\
!W
\end{tikzcd}$$

Figure 3.4: Thread definition

on records, interpreted with the additive product &, with the following type:

$$\mathit{thread} : !S \otimes !\&_{m\in X}(!S \otimes X_m \multimap (!S \otimes Y_m)_\perp) \to !\&_{m\in X}(X_m \multimap (Y_m)_\perp)$$

No new techniques are required to handle multiple methods, but we must extend the operations defined earlier. We enrich the type of *pproj* to include a labelling:

$$\mathit{pproj} \colon \&_{m\in M}(X_m \multimap (Y_m \otimes Z)_\perp) \to \&_{m\in M}(X_m \multimap (Y_m)_\perp) \oslash Z_\perp$$

There is in fact no modification required to the above definition in this case, other than the new type; as well as containing the value-part of the X_m component, the initial question q^x_m now selects the record component m required, but this is just copied across as before.

We enrich the type of *branch* records as follows:

$$\mathit{branch} \colon \&_{m\in M}(X_m \multimap (Y_m)_\perp) \oslash (!Z \otimes !Z) \to \&_{m\in M}(X_m \multimap (Y_m)_\perp) \oslash !Z$$

As with *pproj* no modification is required other than the type. We also need skew-typed distributivity morphisms:[4]

$$\mathit{dist} = (\&_{m\in M}X_m) \oslash Y \xrightarrow{\&_{m\in M}(\Pi_m \oslash id_Y)} \&_{m\in M}(X_m \oslash Y)$$

and we understand the abbreviation

$$f^\star = (id \otimes \eta); \psi; \perp(f); \lambda^{-1}; !(\mu_\multimap)$$

as involving

$$\mu_\multimap \colon [\&_{m\in M}(X \multimap Y_\perp)]_\perp \to \&_{m\in M}(X \multimap Y_\perp)$$

since $\&_{m\in M}(X \multimap Y_\perp) \cong X \multimap (\&_{m\in M}Y)_\perp$.

Now we can define *thread*. The following definition contains a large number of structural morphisms which can be mostly be ignored, and should be apparent from the types at each stage. The general structure is as follows, and corresponds to the three-fold categorisation of method calls discussed earlier and depicted in Figure 3.3. Using *unfold*, a copy of the method body is peeled off, and a copy of the start state is supplied to this via *eval* (corresponding to call 0). The remaining copies of the method body are split into two, and these two sets are selected between by the *branch* near the bottom of the diagram. For the left

[4]This distributivity is not particular to $\oslash$ or the ordering of $\&_{m\in M}X_m$ and Y, it is just a property of the Cartesian product &.

set (corresponding to the nested calls 1–3), we simply supply the other copy of the start state to a recursive use of *thread*. For the right (corresponding to calls 4–5 and more), we use *pproj* to feed the resulting state from the main method invocation to a second recursive use of *thread* as in *linthread*. Here, rather than by a property of the types as in Section 3.2.4, the requirements ensuring that *pproj* never goes undefined are enforced by the use of *branch*.

3.4 Properties of thread

Here we shall prove some general properties of the *thread* morphism which shall be used later in our soundness proof (see Chapter 5). We simplify to the single-method case for simplicity and clarity; multiple methods do not add any essential difficulty.

We first introduce the restriction to *disciplined* strategies which insures that the use of *pproj* in the definition of *thread* satisfies the two conditions identified earlier, ensuring that *pproj* is well-behaved. We then show that for strategies obeying this restriction, certain characteristic properties hold of *thread*. Furthermore, we show that for strategies obeying a more restrictive property of being *pair-like*, two stronger properties hold of *thread*.

3.4.1 Disciplined strategies

In the case of the linear thread operator, we mentioned that the two requirements of *pproj* (from Section 3.2.1) are satisfied simply by virtue of the types involved. In the non-linear case things are not so simple. Here we give a property on strategies (having more general types) of being *disciplined* which ensures that they satisfy the *pproj* requirements.

The restriction to disciplined strategies is one of the key ingredients of this thesis. It will appear again in a more syntactic form in the next chapter; in Section 4.2.1 we justify the restriction with some operational intuitions. This idea may at first appear rather ad hoc, but it will emerge later that this is just what is required to obtain the language matching our model.

A disciplined strategy for the type $S_1 \otimes X \multimap (S_2 \otimes Y)_\perp$ is one which makes no move in X in response to a S_2 move after answering the initial question of $(S_2 \otimes Y)_\perp$.

Definition 3.1 (Disciplined strategies)**.** For any types S_1, S_2, X and Y and any morphism $k\colon 1 \to (S_1 \otimes X \multimap (S_2 \otimes Y)_\perp)$, k is disciplined if

- There is no play $qsatbc \in k$, where a answers q, b is an O-move in S_2 and c is a P-move in X.

A morphism $k\colon \Delta \to (S_1 \otimes X \multimap (S_2 \otimes Y)_\perp)$ is disciplined if for every $e\colon 1 \to \Delta$, $k \circ e$ is disciplined.

Here and below, we think of k as modelling some method implementation, and of the object Δ as corresponding to a 'context' consisting of any free variables by which this method implementation might be parametrised.

The condition above ensures that k does not cause *pproj* to go undefined in order to avoid making an illegal move in X. Any information about the X component required by the computation must therefore be extracted before the point of returning from the function call, and we can consider the state at this point as being solely a function of the previous state and information already gleaned from X. This is what excludes the behaviour of a higher-order reference cell as mentioned earlier.

Certain types only permit disciplined strategies. If $!S$ is of ground type, no b exists, while if X is of ground type, no c exists, so in either case *every* k of the relevant type is disciplined.

It is easy to see that for a disciplined $k\colon \Delta \to (S_1 \otimes X \multimap (S_2 \otimes Y)_\perp)$ and any $s\colon \Delta \to S_1$, the morphism $(k \otimes s); eval^L$ is disciplined when considered at type $\Delta \to (1 \otimes X \multimap (S_2 \otimes Y)_\perp)$.

We now introduce a technical property that characterises *pproj*. The property is stated using the memoization machinery of Section 2.5, allowing us to examine the behaviour of *pproj* from the point of method return. Given a disciplined k, and a terminating play $qu'a$ of the application of k to an s and v, there is a certain induced interaction u'' with k and s. The property states (in terms of memoization) that further interaction with this application in S_2 after $qu'a$ coincides with interaction with *pproj* in S_2 after u''. It is important to note that there is no v on the bottom line of the diagram below—because k is disciplined, all the relevant information is contained in the play u'' involving any initial pre-return interaction with v.

Lemma 3.2 (*pproj* property)**.** *Let* $K = (S_1 \otimes X \multimap (S_2 \otimes Y)_\perp)$*, and suppose* $k\colon \Delta \to K$ *is disciplined and there are* $s\colon \Delta \to S_1$ *and* $v\colon \Delta \to X$ *such that there* *is a terminating play $qu'a$ of $(k \otimes (s \otimes v)); eval$ Then there are plays $u'', \bar{u}$ such that*

$$\begin{tikzcd}
 & & (S_2 \otimes Y)_\perp \arrow[dr, "\perp(\Pi_{S_2})"] & \\
(\Delta^3)_{u'} \arrow[urr, "{[k \otimes (s \otimes v); eval]_{u''}}"] \arrow[r, swap, "{[(k \otimes s; eval^L) \otimes id_\Delta]^{u'}_{u''}}"] & {[X \multimap (S_2 \otimes Y)_\perp \otimes \Delta]_{u''}} \arrow[ur, "{[id \otimes v; eval]_{u''}}"] \arrow[dr, "(\Pi; pproj)^{u''}_{\bar{u}}"] & & S_{2\perp} \\
 & & (X \multimap Y_\perp)_{\bar{u}} \otimes S_{2\perp} \arrow[ur, swap, "\Pi_{S_{2\perp}}"] &
\end{tikzcd}$$

Proof. Note that

$$(k \otimes (s \otimes v)); eval = [(k \otimes s; eval^L) \otimes id_\Delta]; [id \otimes v; eval]; \perp(\Pi_S)$$

and the memoized morphism splits at $[X \multimap (S_2 \otimes Y)_\perp \otimes \Delta]$ as

$$[(k \otimes s; eval^L) \otimes id_\Delta]^{u'}_{u''}; [id \otimes v; eval]_{u''}; \perp(\Pi_S)$$

Then $\bar{u}$ is the relabelling of the moves of u'' via *pproj*; the diagram above commutes since the morphism on the left is disciplined, u'' is a terminating play in $X \multimap (S_2 \otimes Y)_\perp$, and by examination of their definitions, both *eval* and *pproj* therefore act as a copycat on S_2. □

3.4.2 Branch Property

Abbreviate $E = X \multimap Y_\perp$, giving

$$branch\colon E \oslash (!E \otimes !E) \rightarrow E \oslash !E$$

Lemma 3.3 (*branch* property)**.** *Suppose u is a terminating play in $E \oslash !E$. Then there is a terminating play $\bar{u}$ in $E \oslash (!E \otimes !E)$ such that (a) $u \in f; branch$ iff $u \in f; (id \otimes \Pi_L)$ and if $u \in f; branch$ then $\bar{u} \in f$ and (b) the following diagram commutes:*

$$\begin{tikzcd}
{[E \oslash (!E \otimes !E)]_{\bar{u}}} \arrow[r, "branch^{\bar{u}}_{u}"] \arrow[d, phantom, "\cong"] & (E \oslash !E)_u \arrow[d, phantom, "\cong"] \\
E_{u_0} \otimes (!E)_{u_L} \otimes !E \arrow[d, phantom, "\cong"] & E_{u_0} \otimes (!E)_{u_L} \arrow[d, phantom, "\cong"] \\
E_{u_0} \otimes (Z_{u_2} \otimes !E) \otimes !E \arrow[r, "id \otimes \Pi_{Z_{u_2}} \otimes id"] & E_{u_0} \otimes (Z_{u_2} \otimes !E)
\end{tikzcd}$$

for the evident isomorphisms from Lemmas 2.10—2.12.

Proof. The above holds by examination of the definition of *branch*—since u is a terminating play, *branch* selects the right-hand $!E$ for components opened after u (the third pair of clauses in the definition), but still copies further play in previously opened components appropriately (the second pair of clauses). Since *branch* is always a copycat on the initial E (the first pair of clauses in the definition), E_{u_0} is untouched. □

3.4.3 Thread Properties

We show three main properties, corresponding to the three classes of method calls depicted in Figure 3.2. We show (1) that a method invocation gives the correct result, (2) that it leaves the correct object for future use, and (3) that nested calls are handled properly. These correspond to calls 0, 4–5 and 1–3 in that figure respectively.

Thread Property 1

Here we show that the abstract behaviour of a single method invocation given by *thread* results in the same *value* as the concrete implementation specifies, assuming we start with the correct state. More precisely, the result of invoking a method m with argument v_1 upon the results of $\mathit{thread} \circ (k^\dagger \otimes s_1^\dagger)$ agrees with the non-state part of the result of invoking m on $k^\dagger$ with argument $\langle s_1^\dagger, v_1\rangle$:

Lemma 3.4 (Thread Property 1)**.** *For any k, s and x the following diagram commutes (in the sense that the first morphism equalises the two ways around the square):*

$$\begin{tikzcd}
(!\Delta)^3 \arrow[d, "k^\dagger \otimes s^\dagger \otimes x"'] & \\
!(!S \otimes X \multimap (!S \otimes Y)_\perp) \otimes !S \otimes X \arrow[r, "(\gamma; \mathit{thread}) \otimes id_X"] \arrow[d, "(\varepsilon \otimes id_{!S\otimes X}); \mathit{eval}"'] & !(X \multimap Y_\perp) \otimes X \arrow[d, "(\varepsilon \otimes id_X); \mathit{eval}"'] \\
(!S \otimes Y)_\perp \arrow[r, "\perp(\Pi_Y)"] & Y_\perp
\end{tikzcd}$$

Note that we do not claim that the bottom square on the above diagram commutes—the property depends on the fact that $k^\dagger$ and $s^\dagger$ are promoted morphisms.

Proof. Write $\Pi_{\bar{S}}$ for $id_X \multimap \bot(\Pi_Y)$. Since it is equivalent to supply a value at X then project with $\bot(\Pi_Y)$ or project with $\Pi_{\bar{S}}$ then supply a value at X, and we can split $eval_{!S\otimes X, !S\otimes Y}$ into $eval^L_{!S,X,!S\otimes Y}$ and $eval_{X,!S\otimes Y}$, it is enough to show that the following diagram commutes:

$$\begin{tikzcd}
(!\Delta)^2 \arrow[d, "k^\dagger \otimes s^\dagger"'] & \\
!(!S \otimes X \multimap (!S \otimes Y)_\bot) \otimes !S \arrow[r, "\gamma; thread"] \arrow[d, "(\varepsilon \otimes id_{!S}); eval^L"'] & !(X \multimap Y_\bot) \arrow[d, "\varepsilon"] \\
X \multimap (!S \otimes Y)_\bot \arrow[r, "\Pi_{\bar{S}}"] & X \multimap Y_\bot
\end{tikzcd}$$

We proceed by simplifying the composition $(k^\dagger \otimes s^\dagger); \gamma; thread; \varepsilon$ using the definition in Figure 3.4. We split *thread* in two for convenience as $(thread_1; thread_2)$, where $thread_1$ is the composition in the figure up to and including the *pproj* line and $thread_2$ is the rest. We simplify the first portion (omitting some $d^n_{!\Delta}$):

$$\begin{aligned}
& (k^\dagger \otimes s^\dagger); \gamma; thread_1 \\
= \; & (s^\dagger \otimes k^\dagger); (d \otimes unfold); [id \otimes reassoc; passoc^{-1}; (eval^L \oslash d)]; \\
& \quad reassoc; [id \oslash (thread \otimes id)]; (pproj^\gamma \oslash id) \\
= \; & ((k \otimes s^\dagger); eval^L \oslash [(s^\dagger \otimes k^\dagger); thread \otimes k^\dagger]); (pproj^\gamma \oslash id) \\
& \qquad\qquad (\text{Since } k^\dagger; unfold = d_{!\Delta}; (k \otimes k^\dagger) \text{ and } s^\dagger; d_{!S} = d_{!\Delta}; [s^\dagger \otimes s^\dagger]) \\
= \; & (k \otimes s^\dagger); eval^L; pproj^\gamma \oslash [(s^\dagger \otimes k^\dagger); thread \otimes k^\dagger]
\end{aligned}$$

Now simplify the second portion of the composition:

$$\begin{aligned}
& thread_2; \varepsilon \\
= \; & passoc; (id \oslash (\gamma \otimes id)); [id \oslash (id \otimes thread^\star)]; branch; fold; \varepsilon \\
= \; & passoc; [(id \oslash (\gamma \otimes id)); [id \otimes thread^\star)]; branch; \Pi_L \\
& \qquad\qquad (\text{Since } \varepsilon = unfold; \Pi_L \text{ and } fold; unfold = id) \\
= \; & passoc; [(id \oslash (\gamma \otimes id)); [id \otimes thread^\star)]; \Pi_L \\
& \qquad\qquad (\text{Since } branch; \Pi_L = \Pi_L) \\
= \; & \Pi_L; \Pi_L \\
& \qquad\qquad (\text{Since } (id \oslash f); \Pi_L = \Pi_L \text{ and } passoc; \Pi_L = passoc; \Pi_L; \Pi_L)
\end{aligned}$$

Putting these together:

$$\begin{array}{rl}
 & (k^\dagger \otimes s^\dagger); \gamma; \mathit{thread}; \varepsilon \\
= & [(k \otimes s^\dagger); \mathit{eval}^L; \mathit{pproj}^\gamma \oslash ((s^\dagger \otimes k^\dagger); \mathit{thread} \otimes k^\dagger)]; \Pi_L; \Pi_L \\
= & (k \otimes s^\dagger); \mathit{eval}^L; \mathit{pproj}^\gamma; \Pi_L \\
= & (k \otimes s^\dagger); \mathit{eval}^L; \Pi_{\overline{S}} \\
 & \qquad\qquad (\text{Since } \mathit{pproj}^\gamma; \Pi_L = \Pi_{\overline{S}}) \\
= & (k^\dagger \otimes s^\dagger); (\varepsilon \otimes \mathit{id}); \mathit{eval}^L; \Pi_{\overline{S}}
\end{array}$$

Thus above diagram commutes, completing the proof of Thread Property 1. □

Thread Property 2

Here we show that further interaction with the object after a method invocation behaves like interaction with a fresh object created from the resulting state. More precisely, if invoking m on $k^\dagger$ as above would result in a state-part s_2, then the memoization of *thread* $\circ\, (k^\dagger \otimes s_1^\dagger)$ with respect to the above play is equivalent to its residue paired with *thread* $\circ\, (k^\dagger \otimes s_2)$, the result of rethreading with the result state s_2.

It should be noted that nowhere in the following does s_2 represent some syntactically obtained result state—it is merely an abbreviation for the state-part result of an evaluation as defined below. However, in Chapter 5 s_2 will indeed be related to such a syntactically obtained state. It should also be noted that s_2 contains mention of the argument v_1, but the property below relates that to an expression containing no mention of v_1, since the permitted interaction with v_1 has already occurred (see the earlier discussion of the *pproj* property).

Lemma 3.5 (Thread Property 2). *Suppose* $k\colon (!S \otimes X \multimap (!S \otimes Y)_\perp$ *is disciplined and for some* $s_1\colon \Delta \to !S$, $v_1\colon \Delta \to X$ *we abbreviate* $s_2\colon \Delta \to !S$ *as*

$$s_2 = (k \otimes (s_1^\dagger \otimes v_1)); \mathit{eval}; \perp(\Pi_S)$$

Suppose furthermore there is a terminating play $qu'a \in s_2$. *Then there exists a terminating play* $t \in (s_1^\dagger \otimes k^\dagger); \mathit{thread}$ *such that*

$$\begin{tikzcd}
\Delta_{u'} \arrow[r, "{[(s_1^\dagger \otimes k^\dagger); \mathit{thread}]_u^{u'}}"] \arrow[d, "{([s_2]^{u'} \otimes k^\dagger); \mathit{thread}^\star}"'] & {[!(X \multimap Y_\perp)]_u} \arrow[d, phantom, "\cong"] \\
!(X \multimap Y_\perp) & Z_u \otimes !(X \multimap Y_\perp) \arrow[l, "\Pi_R"']
\end{tikzcd}$$

where $t\upharpoonright_{\Delta} = u'$ *and* $t\upharpoonright_{!(X\multimap Y_\perp)} = u$, *and where the isomorphism is that given by Lemma 2.12.*

Proof.

$$\begin{array}{rcl}
qu'a^s & \in & (k\otimes(s_1^\dagger\otimes v_1)); \mathit{eval}; \perp(\Pi_S)\\
qu'a^{s,y} & \in & (k\otimes(s_1^\dagger\otimes v_1)); \mathit{eval}\\
q^x\bar{u}a^y & \in & (k\otimes s_1^\dagger); \mathit{eval}^L; \mathit{pproj}\\
 & & \quad\text{(By the } \mathit{pproj} \text{ property. Where } \bar{u}\upharpoonright_\Delta = u',\ \bar{u}\upharpoonright_{X\multimap Y_\perp} = \hat{u}).\\
q^x\bar{u}a^y & \in & (k\otimes s_1^\dagger); \mathit{eval}^L; \mathit{pproj}^\gamma \oslash [(s_1^\dagger\otimes k^\dagger); \mathit{thread}\otimes k^\dagger]\\
 & = & (s_1^\dagger\otimes k^\dagger); \mathit{thread}_1\\
 & & \quad\text{(As for Property 1)}\\
q^x\bar{u}a^y & \in & (s_1^\dagger\otimes k^\dagger); \mathit{thread}; \varepsilon\\
 & & \quad\text{(Ignoring trivial relabelling of plays. Recall } \mathit{thread}_2; \varepsilon = \Pi_L; \Pi_L.)
\end{array}$$

So we may take t to be a relabelling of $\bar{u}$ via ε (and therefore u bears the same relationship to $\hat{u}$), giving the first desired property that there exists a terminating play $t \in (s_1^\dagger\otimes k^\dagger); \mathit{thread}$.

For the commutativity of the square,

$$\begin{array}{rl}
 & ([s_2]^{u'}\otimes k^\dagger); \mathit{thread}^\star\\
= & ([(k\otimes s_1^\dagger); \mathit{eval}^L; \mathit{pproj}]^{u'}_{q^x\hat{u}a^y}; \Pi_{S_\perp}\otimes k^\dagger); \mathit{thread}^\star\\
 & \quad\text{(By the } \mathit{pproj} \text{ property.)}\\
= & [(s_1^\dagger\otimes k^\dagger); \mathit{thread}_1]^{u'}_{q^x\hat{u}a^y}; (\Pi_{S_\perp}\otimes\Pi_R); \mathit{thread}^\star\\
 & \quad\text{(Earlier simplification of } \mathit{thread}_1\text{, Lemma 2.11)}\\
= & [(s_1^\dagger\otimes k^\dagger); \mathit{thread}_1; \mathit{passoc}; \mathit{id}\oslash\gamma\otimes \mathit{id}]^{u'}_{q^x\hat{u}a^y}; \Pi_R; \Pi_R; \mathit{thread}^\star\\
 & \quad\text{(Memoization of copycats)}\\
= & [(s_1^\dagger\otimes k^\dagger); \mathit{thread}_1; \mathit{passoc}; (\mathit{id}\oslash\gamma\otimes \mathit{id}); (\mathit{id}\oslash \mathit{id}\otimes \mathit{thread}^\star); \mathit{branch}]^{u'}_{q^x\hat{u}a^y}; \Pi_R\\
 & \quad\text{(Branch right after terminating play } q^x\hat{u}a^y)\\
= & [(s_1^\dagger\otimes k^\dagger); \mathit{thread}_1; \mathit{passoc}; (\mathit{id}\otimes\gamma\otimes \mathit{id}); (\mathit{id}\oslash \mathit{id}\otimes \mathit{thread}^\star); \mathit{branch}; \mathit{fold}]^{u'}_{q^x\hat{u}a^y}; \Pi_E\\
 & \quad\text{(Lemma 2.12, where play is in component 0.)}\\
= & [(s_1^\dagger\otimes k^\dagger); \mathit{thread}]^{u'}_{q^x\hat{u}a^y}; \Pi_E
\end{array}$$

Here we write Π_E for

$$[!(X\multimap Y_\perp)]_{qua} \cong (Z_{qua}\otimes !(X\multimap Y_\perp)) \xrightarrow{\Pi_R} !(X\multimap Y_\perp)$$

This completes the proof of Property 2. □

Thread Property 3

Here we show that nested method calls on an object o produce the same behaviour as method calls on a duplicate o' of o. The duplicate o' starts with the same state o at the start of the containing method call, but any updates to the state of o' do not take effect on o.[5] Similarly, the state resulting from nested method calls on o is discarded when the containing call returns a value, leaving o in the same condition in either situation. We must of course discard o' at this point, or the correspondence no longer holds.

We shall compare two copies $\langle o, o\rangle$ with two uses of one copy in the form $o; d$, so that a play in one is literally a play in the other. The left side shall host the containing method invocation, while the right side will have the nested invocations. It is important that the left side does not also have nested invocations. Since it has the initial (enclosing) method invocation, the left side will be the object we keep, while the right side is thrown away. The only lasting effect of performing the nested invocations resides in their interaction with objects in Δ.

If we did not throw the right side away, we would notice that in the case of $\langle o, o\rangle$, the right side has been updated to the state of the last nested method invocation, while in the case of $o; d$ the result from this nested invocation has been overwritten.

Lemma 3.6 (Thread Property 3)**.** *Suppose k is disciplined, $o = \langle s_1^\dagger, k^\dagger\rangle; thread$ with $o\colon\, !\Delta \to !E$, and u is a terminating play in $!\Delta \to E \oslash !E$.*

Then (a):

$$u \in d_\Delta; ((o;\varepsilon) \otimes o) \Leftrightarrow u \in o; d_E; (\varepsilon \otimes id)$$

and (b) if u' is the evident injection of $u\restriction_{E\oslash !E}$ into $!E \otimes !E$, and $t = u\restriction_\Delta$:

$$\langle o, o\rangle^t_{u'}; \Pi = (o; d)^t_{u'}; \Pi$$

where $\Pi = (!E \otimes !E)_{u'} \cong Z_{u'} \otimes !E \otimes !E \xrightarrow{id_{Z_{u'} \otimes !E} \otimes !_{!E}} Z_{u'} \otimes !E \cong (!E)_{u'}$

Proof. We abbreviate

$$\begin{aligned} T &= (s_1^\dagger \otimes k^\dagger); thread \\ T_1 &= (s_1^\dagger \otimes k^\dagger); thread_1 \\ T_0 &= (k \otimes s_1^\dagger); eval^L; pproj \end{aligned}$$

[5] Note that any interaction with other objects in Δ will be the same in either case, here we are only discussing the explicit state of o.

Then the following reasoning gives the property for plays:

$$\begin{array}{rcl}
u & \in & T; d; (\varepsilon \otimes id) \\
 & = & T; \mathit{unfold} \\
 & = & T_1; \mathit{passoc}; (id \oslash (\gamma \otimes id)); (id \oslash (id \otimes \mathit{thread}^\star)); \mathit{branch} \\
u & \in & T_1; (\Pi_L \otimes \Pi_L) \\
 & & \qquad (\mathit{branch} \text{ property (a).)} \\
 & = & (T_0; \Pi_L) \oslash T \\
 & & \qquad \text{(Simplification from Property 1)} \\
u & \in & (T; \varepsilon) \otimes T \\
 & & \qquad (T; \varepsilon = T_1; \Pi_L; \Pi_L \text{ from P1, and } T_1; \Pi_L; \Pi_L = T_0)
\end{array}$$

This completes the proof of part (a). For part (b), we first obtain the required plays. Since $u \in T; d; (\varepsilon \otimes id)$, there is $\hat{u} \in T; d$ such that $\hat{u} \upharpoonright_{!E \otimes E!} = u'$ and $\hat{u} \upharpoonright_{!\Delta} = t$. Then the following reasoning shows that $\langle o, o \rangle^t_{u'}; \Pi = (o; d)^t_{u'}; \Pi$.

$$\begin{array}{rl}
 & (T; d)^t_{u'}; \Pi \\
= & [T_1; \mathit{passoc}; (id \oslash (\gamma \otimes id)); [id \oslash (id \otimes \mathit{thread}^\star)]; \mathit{branch}]^t_{\tilde{u}}; [\mathit{fold}; d]^{\bar{u}}_{u'}; \Pi \\
 & \qquad \text{(Where } \bar{u} \text{ is the relabelling of } u' \text{ via } \mathit{fold}; d.\text{)} \\
\cong & [T_1; \mathit{passoc}; (id \oslash (\gamma \otimes id)); [id \oslash (id \otimes \mathit{thread}^\star)]; \mathit{branch}]^t_{\tilde{u}} \\
 & \qquad \text{(Definition of } \Pi.\text{)} \\
\cong & [T_1; \mathit{passoc}; (id \oslash (\gamma \otimes id)); [id \oslash (id \otimes \mathit{thread}^\star)]]^t_{\tilde{u}}; \Pi' \\
 & \qquad \text{(Where } \Pi' \text{ is from } \mathit{branch} \text{ property (b).)} \\
\cong & [(T_0 \otimes k^\dagger); (id \otimes \mathit{thread}^\star)]^{t_0}_{u_0} \otimes T^{t_L}_{u_L}; \Pi_Z \\
 & \qquad \text{(Expanding } T_1 \text{ and simplifying)} \\
\cong & [T_1; \mathit{passoc}; (id \oslash (\gamma \otimes id)); [id \oslash (id \otimes \mathit{thread}^\star)]]^{t_0}_{u_0}; \Pi' \otimes T^{t_L}_{u_L}; \Pi_Z \\
 & \qquad \text{(Adding an unused copy of T.)} \\
\cong & [T_1; \mathit{passoc}; (id \oslash (\gamma \otimes id)); [id \oslash (id \otimes \mathit{thread}^\star)]; \mathit{branch}]^{t_0}_{u_0} \otimes T^{t_L}_{u_L}; \Pi_Z \\
 & \qquad \text{(Again by } \mathit{branch} \text{ property (b).)} \\
= & T^{t_0}_{u'_0} \otimes T^{t_L}_{u_L}; \Pi_Z) \\
 & \qquad \text{(Composing with } \mathit{fold}^{u_0}_{u'_0}.\text{)} \\
\cong & (T \otimes T)^t_{u'}; \Pi \\
 & \qquad \text{(Definition of } \Pi.\text{)}
\end{array}$$

This completes the proof of part (b). □

3.4.4 Pair-like methods

In Chapter 5, we shall have to use a stronger property of strategies representing method implementations than the above, namely that they are defined to "return a pair" in a certain sense. Here we shall set up some definitions and show the required properties hold. It should be noted that the requirements here represent a genuine restriction, disallowing methods which create new objects and both store them in their state and return them (but not methods which simply return objects from their state). This section is therefore rather particular to the property we prove in Chapter 5.

Definition 3.7 (Pair-like strategies)**.** Suppose there are games $!\Delta, X_1, X_2, Y_1, Y_2$, and a morphism

$$\kappa\colon !\Delta \to (X_1 \otimes X_2) \multimap (Y_1 \otimes Y_2)_\perp$$

κ is a pair-like morphism if for all plays in κ such that $\kappa^u_{q^{v_1}ta^{v_2}}$ is defined there is an isomorphism $\imath\colon (Z)_u \cong (Z)_{u_1} \otimes (Z)_{u_2}$ (where Z_u, Z_{u_1}, and Z_{u_2} relate to $(!\Delta)_u$, $(!\Delta)_{u_1}$ and $(!\Delta)_{u_2}$ respectively via the notation of Lemma 2.12) and a play $\hat{u}$ so that $d^u_{\hat{u}}\colon (!\Delta)_u \to (!\Delta)_{u_1} \otimes (!\Delta)_{u_2}$, and there is a pair of morphisms

$$\begin{aligned} f &: (!\Delta)_{u_1} \otimes (X_1)_{t_{11}} \to (Y_1)_\perp \\ g &: (!\Delta)_{u_2} \otimes (X_1)_{t_{12}} \otimes (X_2)_{t_2} \to (Y_2)_\perp \end{aligned}$$

where $(X_1 \otimes X_2)_t = (X_1)_{t_1} \otimes (X_2)_{t_2}$ and $(X_1)_{t_1} \xrightarrow{d^{t_1}_{\hat{t}_1}} (X_1 \otimes X_1)_{\hat{t}_1} = (X_1)_{t_{11}} \otimes (X_1)_{t_{12}}$, such that

$$\begin{tikzcd}
 & (!\Delta)_u \otimes (X_1)_{t_1} \otimes (X_2)_{t_2} \arrow[d, "(d_{!\Delta\otimes X_1} \otimes id_{X_2})^{\cdots}_{\cdots}"'] \\
(\kappa^u_{q^{v_1}ta^{v_2}})^* = & ((!\Delta)_{u_1} \otimes (X_1)_{t_{11}}) \otimes ((!\Delta)_{u_2} \otimes (X_1)_{t_{12}} \otimes (X_2)_{t_2}) \arrow[d, "(f \otimes g);\psi"'] \\
 & (Y_1 \otimes Y_2)_\perp
\end{tikzcd}$$

Note that every pair-like strategy is automatically disciplined. By simple calculation, writing for the above construction $f * g$,

$$[s \otimes (f * g)]; eval^L = \lambda_{X_2}((\lambda(f)@s_1) \otimes ((s_2 \otimes \lambda(g)); eval^L)^*)$$

where $s\colon \Delta_{u'} \to (X_1)_{t_1}$ and $s; d^{t_1}_{\hat{t}_1} = d^{u'}_{\hat{u}'}; (s_1 \otimes s_2)$.

As a consequence we have the following, writing $f@g$ for $(f \otimes g); eval$:

Lemma 3.8 (pproj)**.** *For any sequences $q^{v_1}\bar{t}a^{v_2}$, u such that $(pproj \circ eval^L)^u_{q^{v_1}\bar{t}a^{v_2}}$ is defined and morphisms f, g, s with*

$$f * g \otimes s\colon \Delta_{u_1} \otimes \Delta_{u_2} \otimes \Delta_{u_3} \to \kappa_{v_1} \otimes (!\sigma)_{v_2}$$

where $s = (z^\dagger)^{u_3}_{v_2}$, if u_3', u_3'', v_2', v_2'' are those plays such that, abbreviating $s_1 = (z^\dagger)^{u_3'}_{v_2'}$ and $s_2 = (z^\dagger)^{u_3''}_{v_2''}$,

$$(z^\dagger)^{u_3}_{v_2}; d^{v_2}_{\hat{v}_2} = d^{u_3}_{\hat{u}_3}; (s_1 \otimes s_2)$$

then

$$\begin{aligned} & (pproj \circ eval^L)^u_{q^{v_1}\bar{t}a^{v_2}} \circ (f * g \otimes s) \\ = \ & \lambda_{X_2}((\lambda(f)@s_1) \otimes ((s_2 \otimes \lambda(g)); eval^L)^*) \circ (id_{\Delta_1,\Delta_2} \otimes d^{u_3}_{\hat{u}_3}) \end{aligned}$$

Thread Property 4

We introduce a property which shows that, when given a state s and a pair-like strategy k representing the method implementation, a terminating play t of *thread* results in a pair of morphisms f' and g', with f' giving the further behaviour of the method call represented by t, and g' giving the behaviour of future method calls.

This property is of a rather different character to the previous 3 properties, namely that if the strategies involved satisfy the property of being pair-like, *thread* maintains this property. What this means is that if the method implementation contains no essential post-return interdependency between state and return value, then after a terminating play on the threaded object, there is no interdependency between the updated object and the returned result.

Lemma 3.9 (Thread Property 4)**.** *For any s and pair-like k, and a terminating play $t \in (s^\dagger \otimes k^\dagger); thread$, there exist f', g' such that*

$$f' \otimes g' = \Delta_{u'} \xrightarrow{[(s^\dagger \otimes k^\dagger); thread]^{u'}_{u}} (!E)_u \cong Z_u \otimes !E$$

where $u' = t\upharpoonright_\Delta, u' = t\upharpoonright_{!E}$, and the isomorphism is that induced by Lemma 2.12.

Thread Property 4 holds simply by the construction of *thread*—by the *pproj* property, the function and result state given by *pproj* are a pair, which are then simply manipulated as such.

Proof. As in Property 1, the first half of *thread* simplifies to

$$((k^\dagger \otimes s^\dagger); eval^L; pproj \oslash [(s^\dagger \otimes k^\dagger); thread \otimes k^\dagger]$$

which for some $u'', \bar{u}$ memoizes as

$$\begin{aligned} & ((k^\dagger \otimes s^\dagger)^{u'}_{u''}; (eval^L; pproj)^{u''}_{\bar{u}} \otimes [(s^\dagger \otimes k^\dagger); thread \otimes k^\dagger] \\ = \quad & (f * g \otimes (s^\dagger)^{u'_R}_{u''_R}; (eval^L; pproj)^{u''}_{\bar{u}} \otimes [(s^\dagger \otimes k^\dagger); thread \otimes k^\dagger] \end{aligned}$$

where the pair-like property of $k^\dagger$ comes into play because u' is a terminating play there. By the above *pproj* property there are some f_1, g_2 making this

$$(f_1 \otimes g_1) \otimes [(s^\dagger \otimes k^\dagger); thread \otimes k^\dagger]$$

then adding the bottom half of *thread*:

$$thread^{u'}_{u} = (f_1 \otimes ((T_1 \otimes T_2); branch)); fold$$

where $T_1 = (s^\dagger \otimes k^\dagger); thread$ and $T_2 = g_1 \otimes k^\dagger; \psi; thread^\dagger$. Thus taking $f' = f_1$ and $g' = ((T_1 \otimes T_2); branch)); fold$ we have completed the proof of Thread Property 4. □

3.5 Thread and bracketing

In defining *thread* here we have clearly assumed well-bracketed behaviour, as can be seen for example in Figure 3.3. In support of this, *thread* is itself well-bracketed.

This can be seen from the constituent morphisms in the definition of *thread*. Note that both *pproj* and *branch* obey the bracketing condition: they are both just somewhat context-dependent copycat strategies. In the case of *pproj*, we note that in the intended usage questions and answer match up, and otherwise *pproj* goes undefined (which is permitted), while for *branch* there is no difference in terms of justification from a simple copycat with $!Z$ replacing $!Z \otimes !Z$. Since the other morphisms and operators involved obey the bracketing condition, so does *thread*.

In a non-well-bracketed setting, the *thread* defined here will only operate as expected under well-bracketed behaviour. If methods are used in a less restricted fashion, the decomposition into nested and non-nested methods, and thus the use of *branch*, fails. It would appear that we can define a more general *thread* operator which coincides with the one given here under well-bracketed play, but also gives the expected result with unrestricted play. However, the generalisation necessitates a more global and unstructured definition. We discuss these issues again in Chapter 7.

Chapter 4

An object-oriented language

In this chapter we shall motivate and define our object-oriented calculus, and give operational and denotational semantics.

We begin by describing our interpretation of object-oriented programming—which is to say by defining the fragment of interest for the purpose of this thesis. We then describe a *base calculus* suitable for study, which is sufficient to implement these ideas as derived forms, and give typing rules for this language.

Toy OO languages abound in the literature (e.g. [2, 22, 46, 23]), but in brief, what is distinctive about our language is that it is designed to closely match what it is possible to model in $\mathbf{BG}^V$. Several details of the language design have proved rather subtle, and have in fact required multiple attempts to "get right", in the sense of matching the game models in expressivity. The precise definition of our language is thus one of the main contributions of this thesis, and the insights gained are significantly influencing the Eriskay project.

We follow our static semantics by giving a heap-based operational semantics, and discuss some properties of this. We conclude by giving a denotational semantics using the ideas from Chapters 2–3.

4.1 Introducing the language

We start by introducing informally some of the main ingredients of our language, in preparation for the formal definition in Section 4.2.

We shall work with a linear (or rather affine) call-by-value lambda calculus, where $\sigma \to \tau$ is the type of functions which "consume their argument" of type σ to produce a result of type τ. We make the contraction rule available only for certain *reusable* types (see the predicate $re(-)$ defined below); semantically these will be modelled by objects of !-type. The language presented here essentially uses a linear type system for technical convenience, but an extended language might make more essential use of non-reusable types, for example in connection with continuations as discussed in Chapter 7.

We shall view an *object* as a collection of methods which may be invoked repeatedly with some argument, approximately a reusable record of functions which may behave in a stateful manner. The calculus will be class-based, so we take objects to be created from *classes* via the **new** operator. We restrict attention to classes with a single updateable field, since we can consider multiple fields to be a single field of tuple type. On the other hand, we explicitly allow multiple methods, partly because method names play a role in overriding.

In Java terminology, we consider all fields to be `protected` and all methods `public`. Public fields can be simulated using accessor methods, while private methods can be simulated using ordinary functions let-bound in any method in which they occur. Private fields are more problematic, and we do not permit these—we discuss this further in Chapter 7. We also disallow the addition of fields during class extension; this is by nature of a simplification, and we suggest some solutions in Section 7.4.1.1.

We shall not consider Java-style constructors, but instead take the expression **new** $e\ c$ to create an instance of class c by implicitly using the constructor which initialises all fields to the provided values e. Handling the more general case of user-defined constructors is not problematic, but just adds complexity to the definition.

As in e.g. [22], we take classes to be first-class expressions rather than a top level construct, both for simplicity and with a view to defining a language with a higher-order flavour. We create a class via an *expression* of the form **class** $\{\ldots\}$, or if c is an existing class we subclass it with an expression **extend** c **with** $\{\ldots\}$.

In general, to define a class one must give a collection of named method implementations in a fashion allowing for recursion. A key principle of object-oriented programming is that of open recursion, via method overriding; methods are defined in a context with a *self* object, standing for an instance of the class presently being defined. Recursive method invocations via *self* refer not to their currently defined implementations, but to the potentially redefined implementations in a future subclass. We thus define a class with an expression

$$\mathbf{class}\ (\varsigma)\ \{m_1 = e_1, \ldots, m_n = e_n\}$$

or

$$\mathbf{extend}\ c\ \mathbf{with}\ (\varsigma)\ \{m_1 = e_1, \ldots, m_n = e_n\}$$

as a collection of functions $e_1 \ldots e_n$ labelled with method names $m_1 \ldots m_n$, where the variable ς is the *self*-binding which allows recursive reference to the methods of the object under construction. We might use this class in an expression such as

$$(\mathbf{new}\ s\ (\mathbf{extend}\ c\ \mathbf{with}\ (\varsigma)\ \{m = e_m\})) \cdot m\ e$$

where we are subclassing c, creating a new object with state s, and then invoking method m with argument e. It will commonly be useful to bind such a class to a name for creation of objects at a later point:

$$\mathbf{let}\ c\ \mathbf{be}\ \mathbf{class}\ (\varsigma)\ \{m_1 = e_1,\ \ldots\}\ \mathbf{in}\ e$$

Equally there are situations such as singleton classes or some higher-order expressions in which this will not be necessary.

As classes define stateful objects, a method implementation for a method m of type $\tau \to \tau'$ for an object with state type σ must give a behaviour dependent on the existing value of that state, and also give the resultant state. As discussed in Section 3.1, we choose to take a method implementation to be an operation which does this explicitly, having type:

$$m\colon \sigma \otimes \tau \to \sigma \otimes \tau'$$

This interpretation can be considered as a first step to a more general treatment of objects. Viewing a method as a state-transforming function can be thought of as allowing the state to be read at the start of a method's execution (taking a private copy) and written at the end. As mentioned in Section 3.3, we might wish to extend this to allow the state to be read from or written to at arbitrary points, since this affects the semantics of nested calls. We discuss this issue further in Chapter 6 and Section 7.3.3, but here we simply note that for ground-type state there is no loss of expressivity.

There are some further subtleties in our precise notion of method implementations, but we will come back to these shortly after a discussion of our base calculus.

4.1.1 Base Calculus

While the constructs presented above represent our conception of classes, we shall regard these as derived constructs implemented in more basic operations on objects (see Figures 4.7 and 4.8). We could define a language with built-in classes as primitive, but interpretation of these in our operational and denotational semantics would involve a duplication of effort (and require longer proofs). For the purpose of our operational semantics, we must split class instantiation into two steps in any case,[1] so we work in a simpler setting with objects, a fixed point operator and a state-internalisation operation. Together these are sufficient to implement classes as described above.

Firstly, we have "objects" which can be created directly, simply as records:

$$\mathbf{obj}\ \{m_1 = e_1, \ldots, m_n = e_n\}$$

These are reusable in the sense that they may be copied and reused repeatedly. Therefore they must be defined in a reusable context, that is any free variables appearing in e_i must themselves be reusable. We freely use a shorthand

$$\mathbf{obj}\ \{m_i = e_i\}_{i \in X}$$

for a set X of indices, treating the components as unordered (this is justified by our subtyping relation). Here we give an example object of this kind:[2]

$$\mathbf{obj}\ \{\mathit{inc} = \lambda n.n+1, \mathit{twice} = \lambda n.\langle n, n\rangle\} \colon \mathbf{Obj}\ \{\mathit{inc} \colon \iota \to \iota, \mathit{twice} \colon \iota \to \iota \otimes \iota\}$$

It might seem here that we are confusing the two distinct concepts of records and of "real" objects, and we should perhaps not give them both types of the form $\mathbf{Obj}\ X$. We do so with a view to defining as small a language as possible; since we need both concepts, their types will receive the same semantics, and they will admit the same operations, we choose to avoid the duplication of effort which would otherwise ensue.

Secondly, objects may be created as instances of classes. We shall interpret a class as an *approximation operator*, which takes an implementation of *self* and returns a refined one:

$$\mathbf{Class}\ X = \mathbf{Obj}\ X \to \mathbf{Obj}\ X$$

[1]This is because it is neither a class nor the resulting object which will naturally live in the heap, but rather a "pre-object" constructed as the fixed point of the approximation operator defining the class.

[2]We freely use simple arithmetic in these examples, formally we have in mind the use of suitable function constants c_φ (see below).

We shall take the fixed point of this approximation operator when constructing an object, but leave it "open" until then to facilitate inheritance. One can extend a class by constructing a new approximation operator from the old, adding fields to the returned object to add new methods, or replacing fields to override existing ones. Crucially, the fact that this extension operation acts upon the approximation operator rather than the fixed point of this means that any overriding methods are correctly used by methods of the original class in place of the overridden version.

This interpretation of classes and inheritance is as described by Wand in [73], and frequently used elsewhere (including [22]). As discussed in [2], method update[3] is not permitted, and field update must be handled separately. However, we are happy to consider objects as being created from classes, rather than using method update, and wish to explicitly consider object state (i.e. fields) in any case.

The following simple example defines a class which would generate a counter object, with an "increment by m" method, and a "get current value" method.

$$\lambda\varsigma.\ \mathbf{obj}\ \{\ \mathit{inc} = \lambda\langle n, m\rangle.\langle n+m, n+m\rangle,\ \ \mathit{get} = \lambda\langle n, z\rangle.\langle n, n\rangle\ \}$$

If $\kappa = \mathbf{Obj}\ \{\mathit{inc}\colon \iota\otimes\iota \to \iota\otimes\iota,\ \mathit{get}\colon \iota\otimes\iota\to\iota\otimes\iota\}$ then the above class has type $\kappa\to\kappa$, and generates objects of type $\mathbf{Obj}\ \{\mathit{inc}\colon \iota\to\iota,\ \mathit{get}\colon \iota\to\iota\}$.

This class does not have any recursively defined methods, so ς does not appear in any method body. The following class is more interesting:

$$\begin{aligned}
\mathit{Fib} = \lambda\varsigma.\ \mathbf{obj}\ \{\mathit{init} &= \lambda\langle \_, p\rangle.\ \langle p, p\rangle,\\
\mathit{op} &= \lambda\langle s, p\rangle.\ \mathbf{let}\ \langle x, y\rangle\ \mathbf{be}\ p\ \mathbf{in}\ \langle s, x+y\rangle,\\
\mathit{fib} &= \lambda\langle s, n\rangle.\ \mathbf{let}\ \langle f_0, f_1\rangle\ \mathbf{be}\ s\ \mathbf{in}\\
&\qquad\quad \mathbf{let}\ f_n\ \mathbf{be}\ \big(\mathbf{ifz}\ n\ \mathbf{then}\ f_0\\
&\qquad\qquad\quad \mathbf{else\ ifz}\ n-1\ \mathbf{then}\ f_1\\
&\qquad\qquad\quad \mathbf{else}\ \varsigma\cdot\mathit{op}\langle s, \langle \varsigma\cdot\mathit{fib}\langle s, n-1\rangle, \varsigma\cdot\mathit{fib}\langle s, n-2\rangle\rangle\rangle\big)\\
&\qquad\quad \mathbf{in}\ \langle s, f_n\rangle\ \}
\end{aligned}$$

This class computes the nth number in a generalised Fibonacci sequence starting from the pair of numbers in the state, in the naïve recursive fashion. The *init* method can be used to update the state to start from a new pair of numbers—without this method, these could just be set at class creation, but then we would have an unchanging state.

[3]That is, replacing the methods of an existing object.

We can *extend* the above class as follows (where the above class *Fib* is in scope):

$$\begin{aligned} C = \lambda\varsigma.\ \mathbf{obj}\ \{ init &= (Fib\ \varsigma) \cdot init, \\ fib &= (Fib\ \varsigma) \cdot fib, \\ op &= \lambda\langle s, \langle x, y\rangle\rangle.\ \langle s, x * y\rangle\ \} \end{aligned}$$

This new class replaces *op* by a function which multiplies rather than adds; since ς is supplied to *Fib* the inherited *fib* method refers to the self object in the same way that a new method would. In the resulting object, *fib* will thus use the new version of *op* rather than the one in *Fib*.

The last element required to implement classes internalises the stateful behaviour of an object. Given an explicitly state-transforming object *obj* (as arising from the fixed point of an approximation operator), and an initial state s, the expression

$$\mathbf{constr}\ s\ obj$$

gives an object where the state is hidden, incorporated into the behaviour of the object. The usual **new** operation is then just **constr** combined with the fixed point operator. The order of the s and *obj* arguments here may seem counter-intuitive at first, but we choose to give the state first to agree with the semantics—this order will also prove more convenient for the programs defined in later chapters.

To continue our example, we create an object representing the usual Fibonacci sequence as

$$\mathbf{constr}\ \langle 0, 1\rangle\ (\mathbf{Y}\,Fib)$$

or equivalently **new** $\langle 0, 1\rangle$ *Fib*, where *Fib* is bound to the above class definition.

We feel that the derived nature of our classes makes our language more modular, and will make it easier to study possible language extensions. For example, one can easily extend the class syntax shown to provide a **super** facility, for invoking overridden methods of the superclass, without modifying the core language as presented.

4.2 Syntax and typing rules

We now define the syntax for our *core* language, then give some *derived* syntactic constructs. Define types:

$$\begin{array}{rcl}\tau, \sigma & ::= & \iota \mid \tau_1 \otimes \tau_2 \mid \tau_1 \to \tau_2 \mid \mathbf{Obj}\ \{m_1 \colon \tau_1, \ldots, m_n \colon \tau_n\} \\ & & \mathbf{CObj}\ \{m_1 \colon \tau_1, \ldots, m_n \colon \tau_n\} \mid \mathbf{CMeth}\ \tau\end{array}$$

Here ι is the type of natural numbers, $\tau_1 \otimes \tau_2$ of pairs, and $\tau_1 \to \tau_2$ is a linear function from τ_1 to τ_2. $\mathbf{Obj}\ X$[4] is the type of objects as discussed above, while we postpone discussion of $\mathbf{CObj}\ X$ and $\mathbf{CMeth}\ \tau$ until Section 4.2.1. The notions of *basic* and *reusable* types are defined inductively as follows:

$$\begin{array}{cc} & re(\mathbf{Obj}\ X) \\ basic(\iota) & re(\mathbf{CObj}\ X) \\ basic(\sigma) \wedge basic(\tau) \Rightarrow basic(\sigma \otimes \tau) & basic(\tau) \Rightarrow re(\tau) \\ & re(\sigma) \wedge re(\tau) \Rightarrow re(\sigma \otimes \tau)\end{array}$$

We extend $re(-)$ to a predicate on contexts $\Gamma = x_1 \colon \tau_1, \ldots, x_n \colon \tau_n$ via

$$re(x_1 \colon \tau_1, \ldots, x_n \colon \tau_n) \Leftrightarrow \forall i. re(\tau_i)$$

Given for each $k \in \mathbb{N}$ some set Φ_k of k-ary functions $\phi \colon \mathbb{N}^k \to \mathbb{N}$, we take for each $\phi \in \Phi_k$ a constant c_ϕ. In particular we shall use n, m to range over constants $0, 1, \ldots \in \Phi_0$. We define terms:

$$\begin{array}{rcl} e & ::= & x \mid c_\phi \mid \mathbf{ifz}_\tau\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \\ & & \langle e_1, e_2 \rangle \mid \mathbf{let}\ \langle x, y \rangle \colon \sigma \otimes \tau\ \mathbf{be}\ e_1\ \mathbf{in}\ e_2 \mid \lambda x \colon \tau.\ e \mid e_1\ e_2 \mid \\ & & \mathbf{obj}\ \{m_1 = e_1, \ldots, m_n = e_n\} \mid e \cdot m \mid \mathbf{Y}_\tau(e) \mid \mathbf{constr}\ e_1\ e_2 \end{array}$$

Typing rules for our core language are given in Figures 4.1–4.6. Typing judgements have the form $\Gamma \vdash e \colon \tau$, where Γ is of the form $x_1 \colon \tau_1, \ldots x_n \colon \tau_n$ for distinct x_i. We define three languages of succesively greater expressive power: $\mathcal{L}_{\mathrm{pair}}$, $\mathcal{L}_{\mathrm{ret}}$, and $\mathcal{L}_{\mathrm{arg}}$. Figures 4.1 and 4.6 contain the typing rules common to all three languages. The addition of Figure 4.2 plus either Figure 4.3 or Figure 4.4 generate $\mathcal{L}_{\mathrm{arg}}$ or $\mathcal{L}_{\mathrm{pair}}$ respectively; the addition of Figure 4.5 instead generates $\mathcal{L}_{\mathrm{ret}}$. Note that in Figure 4.1, where the subscript (Y) appears, this should be taken to be ε in $\mathcal{L}_{\mathrm{arg}}$ and $\mathcal{L}_{\mathrm{pair}}$ to match their respective typing rules, and the set Y should be

[4] We use the metavariable X to range over "chunks of syntax" which are not themselves types, such as the list of methods here.

taken to be empty, while in the case of $\mathcal{L}_{\text{ret}}$ (Y) should simply read Y. The main language of interest is $\mathcal{L}_{\text{arg}}$, but our soundness proof in the next chapter is with respect to the more restrictive $\mathcal{L}_{\text{ret}}$.

We now briefly review the intended semantics of these types and terms, with some discussion of the associated typing rules. Firstly for types, ι shall be the type of natural numbers, $\tau_1 \otimes \tau_2$ shall be the *multiplicative product* (in the sense of linear logic) of τ_1 and τ_2, and $\tau_1 \to \tau_2$ shall be a *linear* function type, i.e. that of a function which takes a value of type τ_1 to be used linearly and (if it terminates) produces a result of type τ_2. **Obj** $\{m_1 \colon \tau_1, \ldots, m_n \colon \tau_n\}$ is the type of reusable records with components of types $\tau_1, \ldots, \tau_n$—we shall call these records *objects* as that is their main intended use.

Moving on to the meaning of the terms defined above, $c_\varphi \in \Phi_k$ stands for the function φ at an appropriate type $(\overbrace{\iota \to \iota \to \ldots \to \iota \to}^{k+1 \text{ copies}}\iota)$. Call this type $\iota^k \to \iota$, in particular where $k = 0$ this is just ι.

The term $\textbf{ifz}_\tau\ e$ **then** e_1 **else** e_2 is our conditional, which shall evaluate $e \colon \iota$, and *if zero* behave as e_1, otherwise as e_2. This avoids the need for a Boolean type, although it would not be problematic to add one. The typing rule is as follows:

$$\frac{\Gamma \vdash e \colon \iota \quad \Delta \vdash e_1 \colon \tau \quad \Delta \vdash e_2 \colon \tau}{\Gamma, \Delta \vdash \textbf{ifz}_\tau\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \colon \tau}$$

Notice that both e_1 and e_2 are typed in the same non-reusable context Δ, since only one will ever be evaluated. Certain other typing rules demand a reusable context, since they result in a reusable expression, while the remainder split the context in two, with one portion going to one subexpression and the other portion to another. Any reusable variables which are to be used in both subexpressions must first be copied by an instance of the contraction rule:

$$\frac{\Gamma, x \colon \sigma, y \colon \sigma, \Delta \vdash e \colon \tau}{\Gamma, z \colon \sigma, \Delta \vdash e[z/x, z/y] \colon \tau}\ re(\sigma)$$

This localises such copying, and permits us to avoid reasoning about it much of the time. A system with a split reusable and non-reusable context along the lines of DILL [15] could omit the contraction rule at the expense of having "contraction everywhere"; this might be more convenient from some points of view, particularly in relation to our operational semantics, but ultimately this is a presentational issue.

The terms $\langle e_1, e_2 \rangle$ and **let** $\langle x, y \rangle$ **be** e_1 **in** e_2 are pairing and unpairing respectively—the latter uses a binding operation rather than supplying projection functions so that even in a linear setting we shall be allowed to use *both* halves of a pair.

Function creation and application $\lambda x.e$ and $e_1\ e_2$ have the usual meaning, again bearing in mind the above remarks on linearity.

We now come to the *object* part of our calculus. The term

$$\mathbf{obj}\ \{m_1 = e_1, \ldots, m_n = e_n\}$$

directly constructs a record with the given components. Here $e_1, \ldots, e_n$ must be copyable (i.e. of a *reusable* type), since the resultant object may be copied as often as required, and thus a given m_i may be selected multiple times. The term $e \cdot m$ selects the m component of the object defined by e.

Our fixed point operator takes the form $\mathbf{Y}(e)$, where e is a function of type $\rho \to \rho$ for ρ either an object or function type. We allow both options even though one might suffice for the purpose of expressivity, because in practice one will want recursive definitions for both function and object types. Recursive functions may often be convenient, while the recursive construction of an object (in particular, allowing for mutually recursive methods) is the basis for our class system, as we have discussed.

We include a subtyping judgement $\tau <: \tau'$, which includes the trivial subtype $\tau <: \tau$ and the standard rules for pair and function types. For object types, we give a single rule incorporating both depth and width subtyping, and permitting permutation. We also make **CObj** X a subtype of **Obj** X, and similarly **CMeth** a subtype of the corresponding plain function type. On the other hand, we do not allow subtyping on the structure of **CObj** or **CMeth**—this could perhaps be added, but would only cause additional complication here. We do not include a transitivity rule, so that a given subtyping judgement has a unique derivation, but such a rule is admissible. Subtyping judgements then appear in a typing derivation via the subsumption rule.

4.2.1 Typing constr

Before attempting to describe the typing rules for the object construction operator **constr** $e\ c$, we must now come back to the subtle question referred to earlier—of all possible method implementations, which ones are permissible? Are there any we could write which are semantically unsound, or undefined? The potential issues here concern linearity, circular references and the *disciplined* restriction of Chapter 3.

Most simply, one can certainly say that any class state of ground type (γ) is acceptable. The typing rule here would be:

$$\frac{\Gamma \vdash c\colon \mathbf{Obj}\ \{m\colon \gamma \otimes \tau_m \to \gamma \otimes \tau'_m\}_{m \in X} \quad \Delta \vdash e\colon \gamma}{\Gamma, \Delta \vdash \mathbf{constr}\ e\ c\colon \mathbf{Obj}\ \{m\colon \tau_m \to \tau'_m\}_{m \in X}}\ basic(\gamma)$$

This simple rule is sufficient to define a ground-type reference cell, but also gives rise to other interesting behaviour. This may include interaction with other objects in Γ which are referenced in e—such objects can be considered to reside in an implicit non-updateable (or `final`) field of the constructed object.

What happens when we lift the restriction to ground-type state? It is in fact *not* possible to define a **constr** operation which works as one would expect for the full range of types. We explicitly give the hypothetical typing rule, which we could *not* add to the calculus:

$$\frac{\Gamma \vdash c\colon \mathbf{Obj}\ \{m\colon \sigma \otimes \tau_m \to \sigma \otimes \tau'_m\}_{m \in X} \quad \Delta \vdash e\colon \sigma}{\Gamma, \Delta \vdash \mathbf{constr}\ e\ c\colon \mathbf{Obj}\ \{m\colon \tau_m \to \tau'_m\}_{m \in X}}$$

The first issue is to do with reusability. Even if a method implementation treats the state linearly,[5] the possibility of nested method calls as discussed in Chapter 3 means that the state may be reused. Consider the following program:

$$\begin{aligned}&\mathbf{let}\ o\ \mathbf{be\ constr}\ (\lambda x.\ x+1)\\ &\qquad\qquad\quad \mathbf{obj}\ \{\ m = \lambda\langle f\colon \iota \to \iota, g\colon \iota \to \iota\rangle.\ \langle f, f(g\ 1)\rangle\ \}\\ &\mathbf{in}\ o \cdot m(\lambda x.\ o \cdot m(\lambda y.\ 0))\end{aligned}$$

The nested call to m caused by the forcing of the supplied thunk which calls m—the externally mediated recursive call of m, if you will—causes a non-linear use of the state. One has to ask what value is supplied to the nested call at f? In this case both the original and nested call should surely receive the original value ($\lambda x.\ x+1$), and yet this is not a copyable expression.

As an aside, the form of nested method call via the method argument as illustrated above shall prove important, and we will often refer to this situation.

As discussed in Chapter 3, nested method calls can be avoided if method arguments are restricted to ground type, since only a higher-type argument to $o \cdot m$ could conceal a reference to o. Corresponding to the *linear thread* operation,

[5] Note that this means that "reading" the state is a destructive operation.

we can permit *linear classes* with the following rule:

$$\frac{\Gamma \vdash c\colon \mathbf{Obj}\ \{m\colon \sigma \otimes \gamma_m \to \sigma \otimes \tau'_m\}_{m \in X} \quad \Delta \vdash e\colon \sigma}{\Gamma, \Delta \vdash \mathbf{constr}\ e\ c\colon \mathbf{Obj}\ \{m\colon \gamma_m \to \tau'_m\}_{m \in X}}\ basic(\gamma_m)$$

One might alternatively impose a restricted type system at the other end, ensuring that a linear class cannot be supplied a self-reference in a method invocation, but this seems to be complexity for little gain. We do not actually add the above rule to our language, but we discuss the issue further in Chapter 7.

We now move on to the more exciting case of classes with reusable state (σ) and arbitrary argument (τ) and result types (τ'), giving the following typing rule:

$$\frac{\Gamma \vdash c\colon \mathbf{Obj}\ \{m\colon \sigma \otimes \tau_m \to \sigma \otimes \tau'_m\}_{m \in X} \quad \Delta \vdash e\colon \sigma}{\Gamma, \Delta \vdash \mathbf{constr}\ e\ c\colon \mathbf{Obj}\ \{m\colon \tau_m \to \tau'_m\}_{m \in X}}\ re(\sigma)$$

In this situation nested method calls are unproblematic, as the state is reusable and so can be supplied as input to each call, although it should be noted that the result state of any nested calls[6] are discarded and play no further part in the computation (cf. Section 3.3).

It might seem that this version of **constr** is the most natural of those presented, and is just what is desired, but unfortunately it is semantically problematic, being unsuitable for interpretation in our game model. The most obvious problem is with regard to circularity. Consider the following program:

$$\begin{array}{l}\mathbf{let}\ o\ \mathbf{be}\ \mathbf{constr}\ \ (\mathbf{obj}\ \{\})\ \ \mathbf{obj}\ \{m = \lambda\langle s, f\rangle.\ \langle f(), 1\rangle\}\\ \mathbf{in}\ o \cdot m(\lambda x.\ o)\end{array}$$

Here an object is created with a state initialised to a dummy object. The method m is invoked with a function argument which when applied returns its parent object, which m returns as the updated object state. The result, then, is an object which points to itself. The operational semantics presented in the next section does exactly this, but our *behavioural* game model cannot cope with circular reference. We discuss this issue further in Section 7.1, but for now we note that our type system must prevent such circularity.

There is another issue related to the storage of “pointers” from method arguments. *While our language has no such concept, it is helpful to think in terms pointers or references.* The interpretation of **constr** in our game model cannot account for methods which retain or “capture” a pointer to their argument—more precisely, methods which result in an updated state containing a reference to any

[6]More accurately, the result of the last method call nested within any given call.

part of their argument. This issue was discussed in Section 3.4.1; the essence of the problem is that a subsequent method invocation may make use of that stored pointer, which would cause what appears to be spontaneous interaction in the original method argument.

The solution to this problem (and happily also that of circularity) is to impose a restriction on method implementations corresponding to the *disciplined* property of Chapter 3. Rather than an arbitrary function $\sigma \otimes \tau \to \sigma \otimes \tau'$, we only consider those *argument safe* functions which contain no post-return dependency of the result state on the τ-argument. We still allow full interaction with the argument before the function returns a value, but then only the τ'-result may interact with the τ-argument. Argument safety may at first look like an awkward and unpleasant restriction, but it turns out to imply a surprising number of pleasant properties, such as acyclicity (see below) and exception safety (as we discuss in Section 7.1).

We implement this restriction by means of a *ground-type funnelling* operation. After an initial interaction, some value of ground type must be produced, which is then made available for separate computations resulting in the updated state and method result. Since a ground-type value cannot contain any reference to any objects which may be involved in its construction (or alternatively permits no interaction), the result state has no further dependency on the method argument. In the present calculus to ensure this ground-type funnelling occurs we require method implementations to have a fixed syntactic form, but it is possible to give a more flexible type system having the same property, at the expense of complexity.

The following rule assigns a **CMeth** (or "certified method") type to functions which make suitable method implementations:

$$\frac{\Gamma, s\colon \sigma, x\colon \tau \vdash e\colon \gamma \qquad \begin{array}{ll}\Gamma, y\colon \gamma, s\colon \sigma, x\colon \tau & \vdash e_1\colon \sigma \\ \Gamma, y\colon \gamma, x\colon \tau & \vdash e_2\colon \tau'\end{array}}{\Gamma \vdash \lambda\langle s, x\rangle.\ \mathbf{let}\ y\colon \gamma\ \mathbf{be}\ e\ \mathbf{in}\ \langle e_1, e_2\rangle\colon \mathbf{CMeth}\ (\sigma \otimes \tau \to \sigma \otimes \tau')}\ \mathit{re}(\Gamma, \sigma), \mathit{basic}(\gamma)$$

While we are happy to impose a fixed form on method implementations for simplicity, the above is rather restrictive, as it does not permit the (perfectly legitimate) use of other **CMeth** methods in Γ, for example when giving a recursively defined method. We modify the rule to end in a "tail call" $e\ \langle e_1, e_2\rangle$ for e of **CMeth** type, and give the identity function **CMeth** type in order to recover the above rule. Since the self parameter ς appearing in method implementations is declared to have **CObj** type (see Figures 4.7, 4.8), tail calls to other methods are thereby admitted. With more complex typing machinery, the idea can be extended to allow non tail calls (as we discuss in Chapter 7) but tail calls are in fact enough for the programs we write in Chapter 6.

We give three versions of this rule, in Figures 4.3–4.5. The version for $\mathcal{L}_{\text{arg}}$ is more generous, while that for $\mathcal{L}_{\text{ret}}$ is a restricted version for our proof in Chapter 5. The more restricted version for $\mathcal{L}_{\text{pair}}$ is included for completeness, and to aid understanding of the $\mathcal{L}_{\text{ret}}$ rule. The $\mathcal{L}_{\text{arg}}$ rule is the one we have just discussed, while the $\mathcal{L}_{\text{pair}}$ rule not only restricts dependence of e_1 on x, but symmetrically restricts dependence of e_2 on s. The $\mathcal{L}_{\text{ret}}$ rule allows e_2 to depend on s, but does not allow this to occur in the body of any tail-call—allowing one initial "non-pair-like" dependence on the initial state s, followed by any number of tail-calls as in the $\mathcal{L}_{\text{pair}}$ typing discipline. The subscript which appears in **CObj** types tracks the set of methods with a pair-like typing for this purpose.

Given our **CMeth** rule, we add a rule to construct a **CObj** - simply an **Obj** consisting entirely of functions typed as **CMeth**. Another rule goes in the other direction to give field selection for methods, and then we have our "final" typing rule:

$$\frac{\Gamma \vdash c\colon \mathbf{CObj}\ \{m\colon \sigma \otimes \tau_m \to \sigma \otimes \tau'_m\}_{m \in X} \quad \Delta \vdash e\colon \sigma}{\Gamma, \Delta \vdash \mathbf{constr}\ e\ c\colon \mathbf{Obj}\ \{m\colon \tau_m \to \tau'_m\}_{m \in X}}\ re(\sigma)$$

4.2.2 Values

We define a subset of terms which we consider to be *values* as follows

$$\begin{array}{rcl} v & ::= & x \mid c_\varphi \mid \lambda x.e \mid \langle v_1, v_2 \rangle \mid \mathbf{obj}\ \{m_1 = v_1, \ldots, m_n = v_n\} \\ & & \mathbf{Y}(v) \mid \mathbf{Y}(v) \cdot m \end{array}$$

The concept of values will mostly be used in connection with our operational semantics, but in fact it is also used in one of our typing rules—we restrict object creation expressions to have the form $\mathbf{obj}\ \{m_1 = v_1, \ldots, m_n = v_n\}$. In practice this is not much of a restriction, partly because objects will usually be constructed with the form

$$\mathbf{obj}\ \{m_1 = \lambda x.e_1, \ldots, m_n = \lambda x.e_n\}$$

but also because if required one can more explicitly write

$$\mathbf{let}\ x_1\ \mathbf{be}\ e_1\ \mathbf{in}\ \ldots \mathbf{let}\ x_n\ \mathbf{be}\ e_n\ \mathbf{in}\ \mathbf{obj}\ \{m_1 = x_1, \ldots, m_n = x_n\}$$

We impose this restriction to avoid a subtle difficulty in our soundness proof which would give rise to significant duplication of effort.

4.2.3 Derived constructs

For our full language, we extend the base calculus with constructs related to classes as well as a collection of generally useful abbreviations.

Extend the grammar for types to include

$$\mathbf{class}\ \langle \sigma;\ m_1 \colon \tau_1 \to \tau'_1, \ldots, m_n \colon \tau_n \to \tau'_n \rangle$$

and the grammar for terms to include

$$\begin{aligned}
&\mathbf{new}\ e_1\ e_2 \mid \mathbf{class}\ \{m_1 = e_1, \ldots, m_n = e_n\} \mid \\
&\mathbf{extend}\ e_1\ \mathbf{with}\ (\varsigma)\ \{m_1 = e_1, \ldots, m_n = e_n\} \mid \\
&\mathbf{let}\ x\ \mathbf{be}\ e_1\ \mathbf{in}\ e_2 \mid \lambda\langle x_1, x_2\rangle.\ e \mid \mathbf{letrec}\ f(x) = e_1\ \mathbf{in}\ e_2 \mid e \circ e'
\end{aligned}$$

where we use ς, ς' to range over variables of class type. We introduce **class** , **extend** and **new** to show how familiar OO concepts can be represented, while the others are just convenient forms to facilitate the writing of programs.

Typing rules for the derived forms are given in Figure 4.7, and the translation of derived types and terms into the core language is given in Figure 4.8.

4.3 Operational semantics

We define a big-step evaluation semantics for an augmented language with *heaps*, where an expression in some heap evaluates to a value and an updated heap. We take a set L of special variables to represent *locations*, take l and decorated variants to range over these locations ($l \in L$), and consider open expressions with $\mathrm{FV}(e) \subset L$ to be given in some heap containing entries at those locations. We restrict locations from appearing as λ or **let** bindings, but permit them to appear in the context as usual. Furthermore, we extend the set of values to include l and $l \cdot m$ (but not $x \cdot m$ for a binding variable x). Locations play a crucial auxiliary rôle in our operational semantics: they do not appear in complete programs, but may appear at intermediate stages in the evaluation of such programs.

In Figure 4.9 we give an evaluation relation $\Downarrow \subseteq (H \times E) \times (H \times V)$ from expressions with heaps to values with heaps, writing $h, e \Downarrow h', v$. We interpret a heap as an element of $H = L \rightharpoonup V \times V$, a partial function mapping locations to heap cells, where a heap cell is the actual object state paired with its class definition. So for $l \in \mathrm{dom}(h)$, $h(l) = \langle s, c\rangle$ with s and c values representing the

$$\frac{}{\Gamma, x\colon \tau, \Delta \vdash x\colon \tau} \qquad \frac{\Gamma, x\colon \sigma, y\colon \sigma, \Delta \vdash e\colon \tau}{\Gamma, z\colon \sigma, \Delta \vdash e[z/x, z/y]\colon \tau}\ re(\sigma)$$

$$\frac{}{\Gamma \vdash c_\varphi\colon \iota^k \to \iota}\ \varphi \in \Phi_k \qquad \frac{\Gamma \vdash e\colon \tau}{\Gamma \vdash e\colon \tau'}\ \tau <: \tau'$$

$$\frac{\Gamma \vdash e\colon \iota \quad \Delta \vdash e_1\colon \tau \quad \Delta \vdash e_2\colon \tau}{\Gamma, \Delta \vdash \mathbf{ifz}_\tau\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\colon \tau}$$

$$\frac{\Gamma_1 \vdash e_1\colon \tau_1 \quad \Gamma_2 \vdash e_2\colon \tau_2}{\Gamma_1, \Gamma_2 \vdash \langle e_1, e_2\rangle\colon \tau_1 \otimes \tau_2} \qquad \frac{\Gamma \vdash e\colon \tau_1 \otimes \tau_2 \quad \Delta, x : \tau_1, y : \tau_2 \vdash e' : \tau}{\Gamma, \Delta \vdash \mathbf{let}\ \langle x, y\rangle\colon \tau_1 \otimes \tau_2\ \mathbf{be}\ e\ \mathbf{in}\ \ e'\colon \tau}\ x, y \notin \Gamma, \Delta$$

$$\frac{\Gamma, x\colon \tau \vdash e\colon \tau'}{\Gamma \vdash \lambda x\colon \tau.\ e\colon \tau \to \tau'}\ x \notin \Gamma \qquad \frac{\Gamma \vdash e\colon \tau \to \tau' \quad \Delta \vdash e'\colon \tau}{\Gamma, \Delta \vdash e\ e'\colon \tau'}$$

$$\frac{\Gamma \vdash v_1\colon \tau_1\ \cdots\ \Gamma \vdash v_n\colon \tau_n}{\Gamma \vdash \mathbf{obj}\ \{m_1 = v_1, \ldots, m_n = v_n\}\colon \mathbf{Obj}\ \{m_1\colon \tau_1, \ldots, m_n\colon \tau_n\}}\ re(\Gamma)$$

$$\frac{\Gamma \vdash e\colon \mathbf{Obj}\ \{m\colon \tau\}}{\Gamma \vdash e \cdot m\colon \tau}$$

$$\frac{\Gamma \vdash e\colon \rho \to \rho}{\Gamma \vdash \mathbf{Y}_\rho(e)\colon \rho}\ \begin{array}{l} re(\Gamma), \\ \rho = \tau \to \tau' \text{ or } \rho = \mathbf{Obj}\ \{m\colon \tau_m \to \tau'_m\}_{m \in X} \text{ or} \\ \rho = \mathbf{CObj}_{(Y)}\ \{m\colon \tau_m \to \tau'_m\}_{m \in X \cup Y} \end{array}$$

$$\frac{\Gamma \vdash c\colon \mathbf{CObj}_{(Y)}\ \{m\colon \sigma \otimes \tau_m \to \sigma \otimes \tau'_m\}_{m \in X \cup Y} \quad \Delta \vdash e\colon \sigma}{\Gamma, \Delta \vdash \mathbf{constr}\ e\ c\colon \mathbf{Obj}\ \{m\colon \tau_m \to \tau'_m\}_{m \in X \cup Y}}\ re(\sigma)$$

Figure 4.1: Core Language

$$\frac{\big(\ \Gamma \vdash e_m \colon \mathbf{CMeth}\ (\sigma \otimes \tau_m \to \sigma \otimes \tau'_m)\ \big)_{m \in X}}{\Gamma \vdash \mathbf{obj}\ \{m = e_m\}_{m \in X} \colon \mathbf{CObj}\ \{m \colon \sigma \otimes \tau_m \to \sigma \otimes \tau'_m\}_{m \in X}}\ re(\Gamma)$$

$$\frac{\Gamma \vdash e \colon \mathbf{CObj}\ \{m \colon \sigma \otimes \tau_m \to \sigma \otimes \tau'_m\}_{m \in X}}{\Gamma \vdash e \cdot m \colon \mathbf{CMeth}\ (\sigma \otimes \tau_m \to \sigma \otimes \tau'_m)}\ m \in X$$

$$\frac{}{\vdash \lambda\langle s, x\rangle.\ \langle s, x\rangle \colon \mathbf{CMeth}\ (\sigma \otimes \tau \to \sigma \otimes \tau)}$$

Figure 4.2: Common $\mathcal{L}_{\text{arg}}$ and $\mathcal{L}_{\text{pair}}$ $\mathbf{CObj}$ and $\mathbf{CMeth}$ rules

$$\frac{\begin{array}{c} \Gamma \vdash e_m \colon \mathbf{CMeth}\ (\sigma \otimes \tau_1 \to \sigma \otimes \tau') \\ \Gamma, s \colon \sigma, x \colon \tau \vdash e \colon \gamma \quad \begin{array}{l} \Gamma, y \colon \gamma, s \colon \sigma \qquad \vdash e_1 \colon \sigma \\ \Gamma, y \colon \gamma, s \colon \sigma, x \colon \tau \vdash e_2 \colon \tau_1 \end{array} \end{array}}{\Gamma \vdash \lambda\langle s, x\rangle.\ \mathbf{let}\ y \colon \gamma\ \mathbf{be}\ e\ \mathbf{in}\ e_m\ \langle e_1, e_2\rangle \colon \mathbf{CMeth}\ (\sigma \otimes \tau \to \sigma \otimes \tau')}\ re(\Gamma, \sigma), basic(\gamma)$$

Figure 4.3: $\mathcal{L}_{\text{arg}}$ $\mathbf{CMeth}$ rule

$$\frac{\begin{array}{c} \Gamma \vdash e_m \colon \mathbf{CMeth}\ (\sigma \otimes \tau_1 \to \sigma \otimes \tau') \\ \Gamma, s \colon \sigma, x \colon \tau \vdash e \colon \gamma \quad \begin{array}{l} \Gamma, y \colon \gamma, s \colon \sigma \vdash e_1 \colon \sigma \\ \Gamma, y \colon \gamma, x \colon \tau \vdash e_2 \colon \tau_1 \end{array} \end{array}}{\Gamma \vdash \lambda\langle s, x\rangle.\ \mathbf{let}\ y \colon \gamma\ \mathbf{be}\ e\ \mathbf{in}\ e_m\ \langle e_1, e_2\rangle \colon \mathbf{CMeth}\ (\sigma \otimes \tau \to \sigma \otimes \tau')}\ re(\Gamma, \sigma), basic(\gamma)$$

Figure 4.4: $\mathcal{L}_{\text{pair}}$ $\mathbf{CMeth}$ rule

$$\frac{\begin{array}{c}\big(\,\Gamma \vdash e_m \colon \mathbf{CMeth}\ (\sigma \otimes \tau_m \to \sigma \otimes \tau'_m)\ \big)_{m \in X} \\ \big(\,\Gamma \vdash e_m \colon \mathbf{CMeth}_p\ (\sigma \otimes \tau_m \to \sigma \otimes \tau'_m)\ \big)_{m \in Y}\end{array}}{\Gamma \vdash \mathbf{obj}\ \{m = e_m\}_{m \in X \cup Y} \colon \mathbf{CObj}_Y\ \{m \colon \sigma \otimes \tau_m \to \sigma \otimes \tau'_m\}_{m \in X \cup Y}}\ \mathit{re}(\Gamma)$$

$$\frac{\Gamma \vdash e \colon \mathbf{CObj}_Y\ \{m \colon \sigma \otimes \tau_m \to \sigma \otimes \tau'_m\}_{m \in X \cup Y}}{\Gamma \vdash e \cdot m \colon \mathbf{CMeth}\ (\sigma \otimes \tau_m \to \sigma \otimes \tau'_m)}\ m \in X$$

$$\frac{\Gamma \vdash e \colon \mathbf{CObj}_Y\ \{m \colon \sigma \otimes \tau_m \to \sigma \otimes \tau'_m\}_{m \in X \cup Y}}{\Gamma \vdash e \cdot m \colon \mathbf{CMeth}_p\ (\sigma \otimes \tau_m \to \sigma \otimes \tau'_m)}\ m \in Y$$

$$\frac{}{\vdash \lambda\langle s, x\rangle.\ \langle s, x\rangle \colon \mathbf{CMeth}_p\ (\sigma \otimes \tau \to \sigma \otimes \tau)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_m \colon \mathbf{CMeth}_p\ (\sigma \otimes \tau_1 \to \sigma \otimes \tau') \\ \Gamma, s \colon \sigma, x \colon \tau \vdash e \colon \gamma \quad \begin{array}{l}\Gamma, y \colon \gamma, s \colon \sigma \vdash e_1 \colon \sigma \\ \Gamma, y \colon \gamma, x \colon \tau \vdash e_2 \colon \tau_1\end{array}\end{array}}{\Gamma \vdash \lambda\langle s, x\rangle.\ \mathbf{let}\ y \colon \gamma\ \mathbf{be}\ e\ \mathbf{in}\ e_m\ \langle e_1, e_2\rangle \colon \mathbf{CMeth}_p\ (\sigma \otimes \tau \to \sigma \otimes \tau')}\ \mathit{re}(\Gamma, \sigma), \mathit{basic}(\gamma)$$

$$\frac{\begin{array}{c}\Gamma \vdash e_m \colon \mathbf{CMeth}_p\ (\sigma \otimes \tau_1 \to \sigma \otimes \tau') \\ \Gamma, s \colon \sigma, x \colon \tau \vdash e \colon \gamma \quad \begin{array}{l}\Gamma, y \colon \gamma, s \colon \sigma \qquad \vdash e_1 \colon \sigma \\ \Gamma, y \colon \gamma, s \colon \sigma, x \colon \tau \vdash e_2 \colon \tau_1\end{array}\end{array}}{\Gamma \vdash \lambda\langle s, x\rangle.\ \mathbf{let}\ y \colon \gamma\ \mathbf{be}\ e\ \mathbf{in}\ e_m\ \langle e_1, e_2\rangle \colon \mathbf{CMeth}\ (\sigma \otimes \tau \to \sigma \otimes \tau')}\ \mathit{re}(\Gamma, \sigma), \mathit{basic}(\gamma)$$

Figure 4.5: $\mathcal{L}_{\mathrm{ret}}$ **CObj** and **CMeth** rules

state and class of the object l.[7]

It might seem that the heap should only be used to store the state of an object, since that is all that may be updated,[8] but there is good reason to consider the class body as existing there. Locations present in the defining context of a class body behave just like fields which are never updated, and so a similar treatment makes sense as well as being convenient.

We use the state convention that a rule not mentioning heaps

$$\frac{e_1 \Downarrow v_1 \ \cdots \ e_n \Downarrow v_n}{e \Downarrow v}$$

[7] We shall abuse notation to write $\langle -, - \rangle$ for this meta-level pairing as well as pairing in the language itself.

[8] One might wish to allow method update, but this is not supported by our game model.

$$\frac{}{\tau <: \tau} \qquad \frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{\tau_1 \otimes \tau_2 <: \tau_1' \otimes \tau_2'} \qquad \frac{\tau_1' <: \tau_1 \quad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}$$

$$\frac{\tau_1 <: \tau_1' \quad \cdots \quad \tau_n <: \tau_n'}{\mathbf{Obj}\ \{m_{\pi 1} \colon \tau_{\pi 1}', \ldots, m_{\pi m} \colon \tau_{\pi m}'\} <: \mathbf{Obj}\ \{m_1 \colon \tau_1, \ldots, m_n \colon \tau_n\}} \begin{array}{c} m \leq n \\ \pi \colon \{1, \ldots, m\} \rightarrowtail \\ \{1, \ldots, n\} \end{array}$$

$$\frac{}{\mathbf{CObj}_Z\ X <: \mathbf{Obj}\ X}$$

$$\frac{}{\mathbf{CMeth}\ (\sigma \otimes \tau \to \sigma \otimes \tau') <: \sigma \otimes \tau \to \sigma \otimes \tau'}$$

$$\frac{}{\mathbf{CMeth}_p\ (\sigma \otimes \tau \to \sigma \otimes \tau') <: \sigma \otimes \tau \to \sigma \otimes \tau'}$$

Figure 4.6: Subtyping

stands for the rule

$$\frac{h_0, e_1 \Downarrow h_1, v_1 \ \cdots \ h_{n-1}, e_n \Downarrow h_n, v_n}{h_0, e \Downarrow h_n, v}$$

as only object construction and method invocation need to interact with the heap.

Note that our operational semantics is untyped—types are not required at run time for our language. While the operational semantics would in principle support a larger language including heaps containing circular references, this is not supported by our game model. Instead, our type system prevents cycles from appearing in the heap.[9]

More precisely, so long as every class body in the heap has the restricted **CObj** type, $\Downarrow$ preserves heap acyclicity. Define G_h to be the directed graph with vertices the locations of h, and an edge $l \to l'$ when $l' \in \mathrm{FV}(h(l))$, and $\mathrm{DAG}(G)$ the property that G contains no cycle $l \to \ldots \to l$. Then the following theorem expresses this property:

Theorem 4.1 (Heap acyclicity). *If* $\Delta \vdash e$ *and for* $l_1, \ldots, l_n$ *there are* Δ_i, σ_i, X_i *such that* $\Delta_i \vdash h(l_i) \colon \sigma_i \otimes \mathbf{CObj}\ X$*, and* $h, e \Downarrow h', v$*, then* $\mathrm{DAG}(G_h) \Rightarrow \mathrm{DAG}(G_{h'})$.

In order to prove this theorem one needs to know that the class argument of any **constr** has **CObj** type, which entails a type preservation property, and to

[9]We discuss the removal of this restriction in Section 7.1.

$$\frac{\Gamma \vdash e_1\colon \sigma \quad \Delta \vdash e_2\colon \mathbf{Class}\ \langle \sigma; m\colon \tau_m \to \tau'_m\rangle_{m\in X}}{\Gamma, \Delta \vdash \mathbf{new}\ e_1\ e_2\colon \mathbf{Obj}\ \{m\colon \tau_m \to \tau'_m\}_{m\in X}}$$

$$\frac{\begin{array}{l}\big(\ \Gamma, \varsigma\colon \mathbf{CObj}\ \{m\colon \sigma \otimes \tau_m \to \sigma \otimes \tau'_m\}_{m\in Y} \\ \quad \vdash e_m\colon \sigma \otimes \tau_m \to \sigma \otimes \tau'_m\ \big)_{m\in Y}\end{array}}{\begin{array}{l}\Gamma \vdash \mathbf{class}\ (\varsigma)\ \{m = e_m\}_{m\in Y} \\ \quad\colon \mathbf{Class}\ \{\sigma;\ m\colon \tau_m \to \tau'_m\}_{m\in Y}\end{array}}$$

$$\frac{\begin{array}{l}\Gamma \vdash e\colon \mathbf{Class}\ \langle \sigma;\ m\colon \tau_m \to \tau'_m\rangle_{m\in X} \\ \big(\ \Gamma, \varsigma\colon \mathbf{CObj}\ \{e_m\colon \sigma \otimes \tau_m \to \sigma \otimes \tau'_m\}_{m\in X\cup Y} \\ \quad \vdash e_m\colon \sigma \otimes \tau_m \to \sigma \otimes \tau'_m\ \big)_{m\in Y}\end{array}}{\begin{array}{l}\Gamma \vdash \mathbf{extend}\ e\ \mathbf{with}\ (\varsigma)\ \{m = e_m\}_{m\in Y} \\ \quad\colon \mathbf{Class}\ \{\sigma;\ m\colon \tau_m \to \tau'_m\}_{m\in X\cup Y}\end{array}}$$

$$\frac{\Gamma \vdash e_1\colon \tau_1 \quad \Delta, x\colon \tau_1 \vdash e_2\colon \tau_2}{\Gamma, \Delta \vdash \mathbf{let}\ x\ \mathbf{be}\ e_1\ \mathbf{in}\ e_2\colon \tau_2} \qquad \frac{\Gamma, x\colon \tau_1, y\colon \tau_2 \vdash e\colon \tau}{\Gamma \vdash \lambda\langle x, y\rangle.\ e\colon (\tau_1 \otimes \tau_2) \to \tau}$$

$$\frac{\Gamma, f\colon \tau \to \tau', x\colon \tau \vdash e\colon \tau' \quad \Delta, f\colon \tau \to \tau' \vdash e_2\colon \tau''}{\Gamma, \Delta \vdash \mathbf{letrec}\ f(x) = e_1\ \mathbf{in}\ e_2\colon \tau''}\ re(\Gamma)$$

$$\frac{\Gamma \vdash e\colon \tau' \to \tau'' \quad \Delta \vdash e'\colon \tau \to \tau'}{\Gamma, \Delta \vdash e \circ e'\colon \tau \to \tau''}$$

$$\frac{\Gamma \vdash e\colon \mathbf{Obj}\ \{m\colon \tau'_m \to \tau''_m\}_{m\in X} \quad \Delta \vdash e'\colon \mathbf{Obj}\ \{m\colon \tau_m \to \tau'_m\}_{m\in X}}{\Gamma, \Delta \vdash e \circ e'\colon \mathbf{Obj}\ \{m\colon \tau_m \to \tau''_m\}_{m\in X}}$$

Figure 4.7: Derived Constructs

$$\mathbf{Class}\ \langle \sigma; m\colon \tau_m \to \tau'_m\rangle_{m\in X}$$

$$\leadsto$$

$$\mathbf{CObj}\ \{m\colon \sigma\otimes\tau_m \to \sigma\otimes\tau'_m\}_{m\in X} \quad\to$$
$$\mathbf{CObj}\ \{m\colon \sigma\otimes\tau_m \to \sigma\otimes\tau'_m\}_{m\in X}$$

$$\mathbf{extend}\ c\ \mathbf{with}\ (\varsigma)\ \{m = e_m\}_{m\in Y}$$

$$\leadsto$$

$$\lambda\varsigma\colon \mathbf{CObj}\ \{e_m\colon \sigma\otimes\tau_m \to \sigma\otimes\tau'_m\}_{m\in X\cup Y}.\ \mathbf{obj}\ \left\{\begin{array}{ll} m = (c\ \varsigma).m, & m \in X\backslash Y \\ m = e_m & m\in Y \end{array}\right\}$$

$$\begin{array}{rcl}
\mathbf{class}\ (\varsigma)\ X & \leadsto & \mathbf{extend}\ \lambda x.\ \{\}\ \mathbf{with}\ (\varsigma)\ X \\
\mathbf{new}\ e\ c & \leadsto & \mathbf{constr}\ e\ (\mathbf{Y}c) \\
\lambda\langle x,y\rangle.\ e & \leadsto & \lambda z.\ \mathbf{let}\ \langle x,y\rangle\ \mathbf{be}\ z\ \mathbf{in}\ e \\
\mathbf{let}\ x\ \mathbf{be}\ e_1\ \mathbf{in}\ e_2 & \leadsto & (\lambda x.\ e_2)\ e_1 \\
\mathbf{letrec}\ f(x) = e\ \mathbf{in}\ e' & \leadsto & \mathbf{let}\ f\ \mathbf{be}\ Y(\lambda f.\ \lambda x.\ e)\ \mathbf{in}\ e' \\
g\circ f & \leadsto & \lambda x.\ g(f(x)) \\
o_2\circ o_1 & \leadsto & \mathbf{obj}\ \{\ m = o_2\cdot m\circ o_1\cdot m\ \}_{m\in X}
\end{array}$$

Figure 4.8: Translation of Derived Forms

$$\frac{}{v \Downarrow v} \qquad \frac{e_1 \Downarrow n_1 \ \cdots \ e_n \Downarrow n_k}{c_\varphi \ e_1 \ \cdots \ e_k \Downarrow m} \ \varphi(n_1, \ldots, n_k) = m$$

$$\frac{e \Downarrow 0 \quad e_1 \Downarrow v}{\textbf{ifz}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \Downarrow v} \qquad \frac{e \Downarrow n \quad e_2 \Downarrow v}{\textbf{ifz}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \Downarrow v} \ n \neq 0$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{\langle e_1, e_2 \rangle \Downarrow \langle v_1, v_2 \rangle} \qquad \frac{e \Downarrow \langle v_1, v_2 \rangle \quad e'[v_1/x, v_2/y] \Downarrow v}{\textbf{let}\ \langle x, y \rangle\ \textbf{be}\ e\ \textbf{in}\ e' \Downarrow v}$$

$$\frac{e_1 \Downarrow \lambda x.e' \quad e_2 \Downarrow v' \quad e'[v'/x] \Downarrow v}{e_1\ e_2 \Downarrow v}$$

$$\frac{e \Downarrow \textbf{obj}\ \{m_1 = v_1, \ldots, m_n = v_n\}}{e \cdot m_i \Downarrow v_i} \ 1 \leq i \leq n$$

$$\frac{e \Downarrow v}{\mathbf{Y}(e) \Downarrow \mathbf{Y}(v)} \qquad \frac{e \Downarrow \mathbf{Y}(v)}{e \cdot m \Downarrow \mathbf{Y}(v) \cdot m}$$

$$\frac{e_1 \Downarrow \mathbf{Y}(v') \quad (v'\ \mathbf{Y}(v'))\ e_2 \Downarrow v}{e_1\ e_2 \Downarrow v} \qquad \frac{e_1 \Downarrow \mathbf{Y}(v') \cdot m \quad (v'\ \mathbf{Y}(v')) \cdot m\ e_2 \Downarrow v}{e_1\ e_2 \Downarrow v}$$

$$\frac{h, e_s \Downarrow h', v_s \quad h', e_c \Downarrow h'', v_c}{h, \textbf{constr}\ e_s\ e_c \Downarrow h''[l \mapsto \langle v_s, v_c \rangle], l} \ l \textit{ fresh} \qquad \frac{e \Downarrow l}{e \cdot m \Downarrow l \cdot m}$$

$$\frac{h, e_1 \Downarrow h', l \cdot m \quad h', v_c \cdot m\ \langle v_s, e_2 \rangle \Downarrow h'', \langle v_s', v \rangle}{h, e_1\ e_2 \Downarrow h''[l \mapsto \langle v_s', v_c \rangle], v} \ h(l) = \langle v_s, v_c \rangle$$

Figure 4.9: Operational Semantics

prove type preservation we shall need to extend our type system to expressions in heaps. It shall be most convenient to give typing rules inductively for *acyclic* heaps, for simplicity of presentation and for the purposes of our denotational semantics, so in fact a type preservation argument will also give the heap acyclicity result.

The typing we require for heaps is as follows. The heap cell $(l_i \mapsto \langle s, c\rangle)$ can be created by an expression **constr** $s\ c$—even if it arose as some other use of **constr** with some subsequent method invocations, it may as well have been created directly with its current state. Given a heap h with locations $l_1, \ldots, l_n$ and an expression e, the pair h, e is assigned the type τ in some context if the expression

$$\textbf{let}\ l_1\ \textbf{be constr}\ h(l_1)\ \textbf{in}\ \ldots \textbf{let}\ l_n\ \textbf{be constr}\ h(l_n)\ \textbf{in}\ e$$

would be assigned type τ in the empty context.[10] We define a function Φ on types which relates the types of $h(l_i)$ and **constr** $h(l_i)$, so that if the pair of state and object implementation has type τ, the object created from these would have type $\Phi(\tau)$.

$$\begin{gathered}\Phi(\ \sigma \otimes \textbf{CObj}\ \{m\colon \sigma\otimes\tau_m \to \sigma\otimes\tau'_m\}_{m\in X}\ )\\ =\\ \textbf{Obj}\ \{m\colon \tau \multimap \tau'\}_{m\in X}\end{gathered}$$

$$\Phi(l_1\colon \tau_1, \ldots, l_n\colon \tau_n) = l_1\colon \Phi(\tau_1), \ldots, l_n\colon \Phi(\tau_n)$$

We can now introduce a typing judgement for heaps $\Delta \vdash h$, meaning that h is a well-typed heap in context Δ. Our typing rules for heaps are as follows:

$$\frac{}{\emptyset \vdash \emptyset} \qquad \frac{\Delta \vdash h \quad \Phi(\Delta) \vdash v\colon \tau}{\Delta, l\colon \tau \vdash h, l \mapsto v}\ \tau = \sigma \otimes \textbf{CObj}\ \{m\colon \sigma\otimes\tau_m \to \sigma\otimes\tau'_m\}_{m\in X}$$

For expressions in heaps, we then define

$$\begin{aligned}\Delta \vdash h, e\colon \tau &\iff \Delta \vdash h \wedge \Phi(\Delta) \vdash e\colon \tau\\ \Delta \vdash h, e &\iff \exists\tau.\Delta \vdash h, e\colon \tau\end{aligned}$$

We are now in a position to give our type preservation result.

Theorem 4.2 (Type preservation)**.** *If $\Delta \vdash h, e\colon \tau$ and $h, e \Downarrow h', v$ then there exists $\Delta' \sqsupseteq \Delta$ with $\Delta' \vdash h', v\colon \tau$.*

[10]Strictly speaking $h(l_i)$ is a pair while **constr** is a curried function taking two arguments.

The heap typing defined above actually imposes the condition that all pointers in h (and in h') point left with respect to the ordering given by Δ (respectively by Δ'). Note that as well as adding new locations to the end of Δ which may appear in v, Δ' can add new locations in the middle, since the evaluation of a method invocation on the object at l may construct an object at the new location l' and store this in the state.

We now sketch the proof of type preservation. We first give an alternate formulation of the method invocation rule which indicates the intended ordering on the resulting heap h'; clearly the set of derivations is essentially unchanged, as none of the above rules is affected by the order of the heap. Define FV^* as the least relation such that

$$l \in \mathrm{FV}(e) \Rightarrow \big(l \in \mathrm{FV}^*(e) \wedge \mathrm{FV}(h(l)) \subseteq \mathrm{FV}^*(e)\big)$$

Let $h\upharpoonright_X$ be $h' \sqsubseteq h$ such that $(l \mapsto v) \in h' \Leftrightarrow l \in X$, and h_R be the portion of h'' between l and the last location in h', in the following:

$$\frac{h, e_1 \Downarrow h', l \cdot m \quad h', v_c \cdot m\ \langle v_s, e_2\rangle \Downarrow h'', \langle v_s', v\rangle}{h, e_1\ e_2 \Downarrow h_L, h_s, (l \mapsto \langle v_s', v_c\rangle), h_R, h_v,\ v} \quad \begin{array}{c} h'' = h_L, h''(l), h_R, h_e \\ h_s = h_e\upharpoonright_{\mathrm{FV}^*(v_s')}, h_v = h_e\upharpoonright_{\overline{\mathrm{FV}^*(v_s')}} \\ h(l) = \langle v_s, v_c\rangle \end{array}$$

The proof is then by induction on operational derivations. The majority of cases are uninteresting, and we concentrate on the two involving heap manipulation. Firstly, if

$$h, \mathbf{constr}\ e_s\ e_c \Downarrow h''[l \mapsto (v_s, v_c)], l$$

then by the **constr** typing rule, $e_c\colon \mathbf{CObj}\ \{m\colon \sigma \otimes \tau_m \to \sigma \otimes \tau_m'\}_{m \in X}$ and $e_s\colon \sigma$, so $\langle v_s, v_c\rangle$ has the correct type for a location at the rightmost end of the heap (agreeing with l).

For the method invocation rule, the concern is that v_s' may contain a reference which is not left-pointing. With reference to the alternate method invocation rule above, this situation would mean that v_s' contained a reference l' in h_R, or one of the new locations pointed to by v_s' does so. The type of $v_c \cdot m$ is a **CMeth**, so by some tedious syntactic analysis of the possible forms of derivation trees for the second premise, we see that there must be some e_m (a method body) such that $e_m\ \langle v_s, e_n\rangle \Downarrow \langle v_s', v\rangle$ and the typing derivation for e_m ends in either the $\mathcal{L}_{\mathrm{arg}}$ rule or the identity rule. Then by inspection of these two rules, the result state cannot contain a location from the argument (since a value of a ground type γ

contains no free variables), but only from v_c and v_s (including a newly created location only referring to such locations).

Lemma 4.3 (Pair-like typing)**.** *In $\mathcal{L}_{\text{pair}}$, if $\Delta \vdash h, e \colon \tau$, and $h, e \Downarrow h', v$ with $\Delta' \vdash h', v$, and*

$$\begin{array}{rcl} \Delta & = & l_1 \colon \lambda_1, \ldots, l_n \colon \lambda_n \\ \Delta' & = & \Xi_1, l_1 \colon \lambda_1, \ldots, \Xi_n, l_n \colon \lambda_n, \Xi_{n+1} \end{array}$$

then for each $l'_i \in \Xi_i$ there is no $l \in \{l_j\} \cup \mathrm{Dom}(\Xi_j)$ for $j > i$ such that $l'_i \in \mathrm{FV}(h'(l))$.

In other words, this lemma states that any location appearing after l_i can be typed in the context without Ξ_i, i.e. Ξ_i is a portion of heap in a sense owned by l_i.

The truth of this lemma can again be seen from the modified method invocation rule above. The lemma holds exactly when for any $l \in h_s, l' \in h_v$, it is the case that $l \notin \mathrm{FV}(h_s)$ and $l \notin \mathrm{FV}(v)$. This holds because of the **CMeth** rule of $\mathcal{L}_{\text{pair}}$. In both $\mathcal{L}_{\text{arg}}$ and $\mathcal{L}_{\text{pair}}$ it is the case that newly created objects cannot be directly shared between e_1 and e_2 because they are given as a pair; but in $\mathcal{L}_{\text{arg}}$ or $\mathcal{L}_{\text{ret}}$ a location created in e_1 can later become available to e_2 via s, either after having been stored in the state, or immediately via the application $e_m \langle e_1, e_2 \rangle$. In $\mathcal{L}_{\text{pair}}$ this is prohibited by the type of e_2, meaning that the above property holds.

4.4 Denotational Semantics

We give a semantics in $\mathbf{BG}^V$ of the form

$$[\![\Gamma \vdash e \colon \tau]\!] \colon [\![\Gamma]\!] \to [\![\tau]\!]_\bot$$

according to typing derivation. We interpret contexts $\Gamma = x_1 \colon \tau_1, \ldots, x_n \colon \tau_n$ as products $[\![\Gamma]\!] = [\![\tau_1]\!] \otimes \ldots \otimes [\![\tau_n]\!]$. Definitions of $[\![-]\!]$ for types and terms are given in Figures 4.10–4.12; we explain and reproduce these in the text below. We shall use the abbreviation $[\![e]\!]_\Gamma$ for $[\![\Gamma \vdash e \colon \tau]\!]$ when e is typeable in context Γ and it is clear (or unimportant) what τ is assigned.

Note that unlike our operational semantics, we have no notion of a heap here, just giving the denotation of a term in context. Stateful behaviour of objects is instead modelled by the behaviour of strategies of ! type. The fact that no explicit modelling of heaps is needed here is a crucial aspect of our approach, and is one measure of the degree to which our denotational semantics is more abstract than the operational semantics.

We interpret types which we have designated *reusable* as objects $!A_1 \otimes \ldots \otimes !A_n$. The linear exponential provides this reusability for objects $!A$ via the contraction $!A \rightarrow !A \otimes !A$, but we extend this to products of such objects.

Definition 4.4 (Reusable objects)**.** An object of $\mathbf{BG}^V$ is *reusable* if it is of the form $!A$, or $B \otimes C$ where B and C are themselves reusable. We extend contraction to reusable objects as follows. Where B and C are reusable, define

$$\begin{array}{rcllcl} d_{!A} & = & d_A & : & !A \rightarrow !A \otimes !A \\ d_{B \otimes C} & = & (d_B \otimes d_C); (id \otimes \gamma \otimes id) & : & (B \otimes C) \rightarrow (B \otimes C) \otimes (B \otimes C) \end{array}$$

There is some notational confusion arising from the interpretation of $d_{!A}$ now potentially referring to contraction for objects of $!A$ or $!!A$ type, but the meaning will be clear from context. The intuition of the above definition is clear, but further semantic justification comes from the fact that $d_{!A \otimes !B}$ coincides with

$$!A \otimes !B \cong !(A \& B) \xrightarrow{d_{A\&B}} !(A\&B) \otimes !(A\&B) \cong (!A \otimes !B) \otimes (!A \otimes !B)$$

Hence we give the following definition.

Definition 4.5. If A is a reusable object, for any morphism $f\colon A \rightarrow \bigotimes_{j \in J} B_j$, if $A = \bigotimes_{i \in I} A_i$ there is a morphism $f^{\ddagger}\colon A \rightarrow \bigotimes_{i \in I} !B_i$. Where

$$g = !\&_{i \in I} A_i \cong \bigotimes_{i \in I} !A_i \xrightarrow{f} \bigotimes_{j \in J} B_j \cong \&_{j \in J} B_j$$

define

$$f^{\ddagger} = A \cong !\&_{i \in I} A_i \xrightarrow{g^{\ddagger}} !\&_{j \in J} B_j \cong \bigotimes_{j \in J} !B_j$$

For natural numbers we take $[\![\iota]\!] = !N = N$ where N is the object $(\mathbb{N}, 1_{\mathbf{BG}})$, justifying the earlier nomination of ι as a reusable type. The other basic type constructors we take as standard using the computational monad $\bot$, namely

$$\begin{array}{rcl} [\![\tau \rightarrow \tau']\!] & = & [\![\tau]\!] \multimap [\![\tau']\!]_{\bot} \\ [\![\sigma \otimes \tau]\!] & = & [\![\sigma]\!] \otimes [\![\tau]\!] \end{array}$$

We then complete the interpretation of types by taking

$$[\![\mathbf{Obj}\ \{m_1 = \tau_1, \ldots, m_n = \tau_n\}]\!] = !\&_{m \in \{1 \ldots n\}} [\![\tau_m]\!] \ \cong \ \otimes_{m \in \{1 \ldots n\}} ![\![\tau_m]\!]$$

which correctly interprets an object type as a reusable record, and setting

$$[\![\mathbf{CObj}\ X]\!] = [\![\mathbf{Obj}\ X]\!]$$

for all X such that the types in question are well formed. Similarly, we set

$$[\![\mathbf{CMeth}\ \tau]\!] = [\![\tau]\!]$$

The subtyping relation is interpreted as a projection from the subtype to the supertype, discarding any unnecessary components.

$$\begin{array}{rcl}
[\![\tau <: \tau]\!] & = & id_{[\![\tau]\!]} \\
[\![\tau_1 \otimes \tau_2 <: \tau_1' \otimes \tau_2']\!] & = & [\![\tau_1 <: \tau_1']\!] \otimes [\![\tau_2 <: \tau_2']\!] \\
[\![\mathbf{Obj}\ \{m_{\pi 1} \colon \tau_{\pi 1}', \ldots, m_{\pi m} \colon \tau_{\pi m}'\} <: & = & !\&_{i \in 1,\ldots,m}(\Pi_{\pi i}; [\![\tau_{\pi i} <: \tau_{\pi i}']\!]) \\
\mathbf{Obj}\ \{m_1 \colon \tau_1, \ldots, m_n \colon \tau_n\}]\!] & & \\
[\![\mathbf{CObj}\ X <: \mathbf{Obj}\ X]\!] & = & id_{[\![\mathbf{Obj}\ X]\!]} \\
[\![\mathbf{CMeth}_\sigma\ \ (\tau \otimes \tau' \to \tau \otimes <) : (\sigma \otimes \tau \to \sigma \otimes \tau')]\!] & = & id_{[\![\sigma \otimes \tau \to \sigma \otimes \tau']\!]}
\end{array}$$

The exception to this general pattern is the subtyping rule for function types. The contravariance in the argument type of the subtyping rule for function types matches that of the $\multimap$ functor:

$$[\![\tau_1 \to \tau_2 <: \tau_1' \to \tau_2']\!] = [\![\tau_1' <: \tau_1]\!] \multimap [\bot([\![\tau_2 <: \tau_2']\!])]$$

The core of our language, an affine λ-calculus, is again interpreted in the usual way (noting that we have chosen to give explicit definitions in $\mathbf{BG}^V$ rather than make use of the Kleisli category $\mathbf{BG}^V_\bot$).

Interpret structural rules, and constructs for pairs and functions as follows. Recall that $\psi \colon X_\bot \otimes Y_\bot \to (X \otimes Y)_\bot$ is the double-strength morphism, the unit of $\bot$ is $\eta \colon X \to X_\bot$, and $-^\dagger$ is the promotion taking $f \colon X \to Y_\bot$ to $f^\dagger \colon X_\bot \to Y_\bot$.

$$\begin{array}{rcl}
[\![\Gamma, x \colon \tau, \Delta \vdash x \colon \tau]\!] & = & \eta_{[\![\tau]\!]} \circ \Pi_{[\![\tau]\!]} \\
[\![\Gamma, \Delta \vdash \langle e, e' \rangle \colon \tau \otimes \tau']\!] & = & \psi \circ ([\![\Gamma \vdash e \colon \tau]\!] \otimes [\![\Delta \vdash e' \colon \tau']\!]) \\
[\![\Gamma, \Delta \vdash \mathbf{let}\ \langle x, y \rangle\ \mathbf{be}\ e_1\ \mathbf{in}\ e_2 \colon \tau]\!] & = & [\![\Gamma, x \colon \tau_1, y \colon \tau_2 \vdash e_2 \colon \tau]\!]^\dagger \circ \psi \circ \\
& & \quad (\eta_{[\![\Gamma]\!]} \otimes [\![\Delta \vdash e_1 \colon \tau_1 \otimes \tau_2]\!]) \\
[\![\Gamma \vdash \lambda x.e \colon \tau \to \tau']\!] & = & \eta \circ \lambda_\tau([\![\Gamma, x \colon \tau \vdash e \colon \tau']\!]) \\
[\![\Gamma, \Delta \vdash e_1\ e_2 \colon \tau]\!] & = & eval^\dagger \circ \psi \circ \\
& & \qquad [\![\Gamma \vdash e_1 \colon \sigma \to \tau]\!] \otimes [\![\Delta \vdash e_2 \colon \sigma]\!]
\end{array}$$

Constants are trivially interpreted as discussed earlier:

$$[\![\Gamma \vdash c_\varphi \colon \iota \otimes \ldots \otimes \iota \to \iota]\!] = \eta \circ \lambda(\bar{\varphi}) \circ 1_\Gamma$$

The conditional uses the *ifz* morphism we introduced earlier. Notice that there is no use of the double strength ψ here; *ifz* evaluates the condition argument, and then only one of e_1 or e_2 as required by the & connective.

$$[\![\Gamma, \Delta \vdash \textbf{ifz}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \colon \tau]\!] =$$

$$[\![\Gamma]\!] \otimes [\![\Delta]\!] \xrightarrow{[\![e]\!]_\Gamma \otimes ([\![e_1]\!]_\Delta \& [\![e_2]\!]_\Delta)} [\![\iota]\!]_\perp \otimes ([\![\tau]\!]_\perp \& [\![\tau]\!]_\perp) \xrightarrow{ifz_\tau} [\![\tau]\!]_\perp$$

We now come to the expressions specific to reusable types. We write d_τ for the derived contraction map $d_{[\![\tau]\!]}$ on the reusable object $[\![\tau]\!]$ as described above.

$$[\![\Gamma, z \colon \tau, \Delta \vdash e[z/x, z/y] \colon \tau']\!] = [\![\Gamma, x \colon \tau, y, \Delta \colon \tau \vdash e \colon \tau']\!] \circ (id_{[\![\Gamma]\!]} \otimes d_\tau \otimes id_{[\![\Delta]\!]})$$

It should be emphasised that this correctly manages interfering (stateful) behaviour in its two components, or in other words this is the place in our semantics where interesting stateful behaviour is propagated.

The **obj** constructor involves promoting all the constituent terms. Recall that for $f \colon !X \to Y_\perp$, the morphism $f^\sharp \colon !X \to (!Y)_\perp$ is defined via the !-promotion and distributivity $!X \xrightarrow{f^\ddagger} !(Y_\perp) \xrightarrow{dist^{!\perp}} (!Y)_\perp$, and $pp^n \colon \bigotimes_{1\le i\le n} !X_i \cong !\&_{1\le i\le n} X_i$.

$$\begin{array}{lcl} [\![\Gamma \vdash \textbf{obj}\ \{m_1 = e_1, \ldots, m_n = e_n\} & = & \perp(pp^n) \circ \psi^n \circ \\ \quad \colon \textbf{Obj}\ \{m_1 \colon \tau_1, \ldots, m_n \colon \tau_n\}]\!] & & (\bigotimes_{1\le i\le n} [\![\Gamma \vdash e_i \colon \tau_i]\!]^\sharp \circ d^n) \end{array}$$

while field selection is simply projection combined with dereliction:

$$[\![\Gamma \vdash e \cdot m \colon \tau_m]\!] = [\![\Gamma]\!] \xrightarrow{[\![e]\!]_\Gamma} (!\&_{m\in X}[\![\tau_m]\!])_\perp \xrightarrow{\perp(\varepsilon)} (\&_{m\in X}[\![\tau_m]\!])_\perp \xrightarrow{\perp(\Pi_m)} [\![\tau_m]\!]_\perp$$

The **obj** syntax is also used to construct terms of **CObj** type. The denotation of such a term shall be the same regardless of which of these types it is assigned, the difference being that a term of the latter type shall obey an additional semantic property not apparent in the type. We thus use the following definition, with the understanding that a derivation of the typing judgement on the left yields a derivation of the judgement on the right.

$$[\![\Gamma \vdash \textbf{obj}\ X \colon \textbf{CObj}_T\ ]\!] = [\![\Gamma \vdash \textbf{obj}\ X \colon \textbf{Obj}\ T]\!]$$

Now consider the fixpoint operator **Y**. For each of the possible cases for ρ, there is a morphism $\mu_\rho \colon [\![\rho]\!]_\perp \to [\![\rho]\!]$. In the case of $\rho = \tau \to \tau'$, this is

$$\mu_{\multimap} \colon ([\![\tau]\!] \multimap [\![\tau']\!]_\perp)_\perp \to [\![\tau]\!] \multimap [\![\tau']\!]_\perp$$

While in the case of $\rho = \mathbf{Obj}\ \{\sigma_m \to \tau_m\}_{m\in X}$, this is

$$\mathit{dist}^{!\perp}; !\&_{m\in X}(\Pi_m; \mu_{\multimap})\colon\ (!\&_{m\in X}(\llbracket\tau\rrbracket \multimap \llbracket\tau'\rrbracket_\perp))_\perp \to !\&_{m\in X}(\llbracket\tau\rrbracket \multimap \llbracket\tau'\rrbracket_\perp)$$

where $\mathit{dist}^{\perp !}\colon (!X)_\perp \to !(X_\perp)$ is the other part of the distributivity used above. Then define:

$$\llbracket\Gamma \vdash \mathbf{Y}(e)\colon \rho\rrbracket =\ \llbracket\Gamma\rrbracket \xrightarrow{\llbracket e\rrbracket^\sharp_\Gamma} (!\llbracket\rho \to \rho\rrbracket)_\perp \xrightarrow{\perp(!(id_{\llbracket\rho\rrbracket} \multimap \mu_\rho); Y_{\llbracket\rho\rrbracket})} \llbracket\rho\rrbracket_\perp$$

Finally, and perhaps most importantly, to implement **constr** we use the *thread* operation as follows:

$$\llbracket\Gamma, \Delta \vdash \mathbf{constr}\ e_s\ e_c\colon \tau\rrbracket =$$

$$\llbracket\Gamma\rrbracket \otimes \llbracket\Delta\rrbracket \xrightarrow{\llbracket e_s\rrbracket_\Gamma \otimes \llbracket e_c\rrbracket_\Delta} \llbracket\tau_1\rrbracket_\perp \otimes \llbracket\tau_2\rrbracket_\perp \xrightarrow{\psi} (\llbracket\tau_1\rrbracket \otimes \llbracket\tau_2\rrbracket)_\perp \xrightarrow{\perp(\mathit{thread})} \llbracket\tau\rrbracket_\perp$$

On a point of notation, we will write $[e]_\Delta$ for the morphism such that $[e]_\Delta\,;\eta = \llbracket e\rrbracket_\Delta$ when it exists (i.e. when e is a value or of the form **constr** $v_1\ v_2$ for values v_1 and v_2).

4.4.1 Coherence

Given that some of our typing rules have no corresponding syntax, there is a potential worry about coherence. For any two derivations of the judgement $\Gamma \vdash e\colon \tau$, the associated denotations $\llbracket\Gamma \vdash e\colon \tau\rrbracket$ must agree if we are to think of *the* denotation of a typing judgement. The potentially problematic areas are subtyping (when to apply the subsumption rule) and the structural rules (when to apply weakening and contraction). Since the proof of these various coherence properties proceeds along standard lines, we shall simply sketch the arguments here.

Subtyping is generally unproblematic due to the language's explicit typing discipline. As in [34], it is possible to define a type- and semantics-preserving rewriting system transforming typing derivations (or explicitly annotated terms) into a normal form—we will briefly sketch this here, but omit the formal details.

The general idea is to push the use of subsumption as far down the derivation as possible, at which point the normal form will have been reached (this will possibly involve the insertion of the identity $\tau <: \tau$). Call an instance of a typing judgement within a derivation *coerced* if it appears as the conclusion of the subsumption rule. Then our normal forms are characterised as follows. In the instance of the **ifz** rule, the premises corresponding to e_1 and e_2 are coerced;[11] in the case of **let** $\langle x, y\rangle$ **be** e **in** e', the premise e is coerced; in the case of an application $e\ e'$, the premise e' is coerced; in the case of $e \cdot m$, e is coerced (by width subtyping only, and not depth subtyping); in an instance of $\mathbf{Y}_\rho(e)$, e is coerced; and in an instance of **constr** $e\ c$, the state e is coerced. Finally, the conclusion of the whole derivation is coerced, and no other judgement appearing in the derivation is coerced.

Given a derivation with premises in normal form, it is then straightforward to place the whole derivation in normal form, and verify that the denotations agree. For example, in the case of application, this involves taking some use of subsumption on the first premise $\tau_2 \to \tau_1' <: \tau_1 \to \tau_2'$, and replacing this with subsumption in the second premise $\tau_1 <: \tau_2$ and the conclusion $\tau_1' <: \tau_2'$, and furthermore collapsing the two successive uses of subsumption on the second premise. Some calculation from the denotation of subtyping and application then shows that these two derivations agree.

The coherence of the structural rules can be established similarly. Consider the possible derivations for an application $\Gamma \vdash e_1\ e_2$. There may be multiple ways to split Γ into Δ_1, Δ_2 such that $\Delta_1 \vdash e_1$ and $\Delta_2 \vdash e_2$. However, any such choice can only differ in which of Δ_1 or Δ_2 an unused (i.e. weakened) variable is placed. It is then easy to show by induction on typing derivations that given a derivation of $\Gamma, x\colon \tau, \Delta \vdash e$, if $x \neq \mathrm{FV}(e)$ then $[\![\Gamma, x\colon \tau, \Delta \vdash e]\!] = [\![\Gamma, \Delta \vdash e]\!] \circ id_\Gamma \otimes 1_{[\![\tau]\!]} \otimes id_\Delta$, where one can obtain the second derivation by erasing x from every context of the first. Therefore for every rule with multiple premises, unused variables can be rearranged as desired (e.g. we could say they are canonically discarded by the leftmost premise).

Finally, the coherence of contraction is a little more interesting. Here we again transform to a normal form where contraction happens towards the conclusion of the derivation. A derivation is in normal form if after every instance of a rule with multiple premises (except contraction itself and **obj**), the contraction rule occurs once for every reusable variable in the context of the conclusion, and the contraction rule does not occur anywhere else. The rules split into a number of groups: of course there is nothing to be done for the axioms, while the denotations of the two simple one-premise rules $e \cdot m$ and subsumption take the form of a postcomposition with the denotation of their premises, while contraction takes

[11]Without the τ annotation on **ifz** , we would have to consider an intersection here.

the form of a precomposition, so the composition rule can be pushed down just by associativity of composition. Three two-premise rules **constr** e_1 e_2, $\langle e_1, e_2 \rangle$, and e_1 e_2, all have the form $f \circ (\llbracket e_1 \rrbracket \otimes \llbracket e_2 \rrbracket)$: an instance of contraction in one of their premises can moved to the conclusion because contraction commutes with weakening (and the previous paragraph allows us to introduce weakening). The **ifz** rule works out similarly. In the case of $\mathbf{Y}(e)$ and **obj**, we use the fact that since $f^\ddagger \circ d = f \circ d^\ddagger$, $f^\sharp \circ d = f \circ d^\sharp$. Finally, we can ignore the order of contractions because $(!X, d)$ form a comonoid, and the interpretation of the contraction rule is the identity on unaffected parts of the context.

4.4.2 Properties of method implementations

We now show how our **CMeth** and **CObj** types are related to the semantic properties of strategies introduced in Section 3.4.

Lemma 4.6 (Argument-safe methods)**.** *In $\mathcal{L}_{\mathrm{arg}}$, if*

$$\Delta \vdash e \colon \mathbf{CMeth}\ (\sigma \otimes \tau \to \sigma \otimes \tau')$$

then $\llbracket e \rrbracket_\Delta$ is a disciplined strategy as per Definition 3.1. If

$$\Delta \vdash e \colon \mathbf{CObj}\ \{\sigma \otimes \tau_m \to \sigma \otimes \tau'_m\}_{m \in X}$$

then for each $m \in X$, $\llbracket e \rrbracket_\Delta ; \Pi_m$ is a disciplined strategy.

The two clauses are proved by a simultaneous induction on the size of typing derivations. The only non-trivial rules are the $\mathcal{L}_{\mathrm{arg}}$ **CMeth** rule and the rule for $\mathbf{Y}$.

Recall that a disciplined strategy $\llbracket e \rrbracket \colon \llbracket \Delta \rrbracket \to \llbracket \sigma \rrbracket \otimes \llbracket \tau \rrbracket \multimap (\llbracket \sigma \rrbracket \otimes \llbracket \tau' \rrbracket)_\perp$ is one in which after the initial question has been answered (i.e. e has returned a value) no move in the right-hand $\llbracket \sigma \rrbracket$ triggers a move in $\llbracket \tau \rrbracket$. This property is satisfied by the "ground type funnelling" in our **CMeth** rule. Since γ is a ground type, $\llbracket \Gamma, s \colon \sigma, x \colon \tau \vdash e \colon \gamma \rrbracket$ contains no move after the initial question has been answered, so cannot cause a move in $\llbracket \tau \rrbracket$.

Consider the post-return interaction with $\llbracket e \rrbracket$. A move in $\llbracket \sigma \rrbracket$ in $\llbracket e_m\ \langle e_1, e_2 \rangle \rrbracket$ causes a move in $\llbracket \sigma \rrbracket$ in $\llbracket e_m \rrbracket$ via the evaluation morphism. Now e_m is also of **CMeth** type, so we can assume $\llbracket e_m \rrbracket$ is disciplined, and so the only possible resulting move in its argument is in $\llbracket \sigma \rrbracket$ of its argument, and not $\llbracket \tau_1 \rrbracket$. Thus there is no interaction with $[\![e_2]\!]$, only with $[\![e_1]\!]$. By the type of e_1, $[\![\Gamma, y\colon \gamma, s\colon \sigma \vdash e_1]\!]$ cannot make a move in $[\![\tau]\!]$.

In the case of the $\mathbf{Y}$ rule, we note that $\bot$ is of course disciplined, and semantically the disciplined property is closed under limits, so $[\![\mathbf{Y}(e)]\!]; \Pi_m$ is disciplined.

Lemma 4.7 (Pair-like methods)**.** *In $\mathcal{L}_{\text{pair}}$ and $\mathcal{L}_{\text{ret}}$, if*

$$\Delta \vdash e\colon \mathbf{CMeth}\ (\sigma \otimes \tau \to \sigma \otimes \tau')$$

then $[\![e]\!]_\Delta$ is a pair-like strategy as per Definition 3.7.

Similar reasoning holds for this stronger property.

4.4.3 Heaps

As we have explained, the denotational semantics given above makes no reference to heaps. However, for the purposes of proving soundness with respect to the operational semantics, we need to extend the denotational semantics to the augmented language where an expression is interpreted in the context of some heap. No modification is required to our existing semantics of expressions—instead, we consider locations to be variables in an appropriate context. We then complete the interpretation by defining the semantics of a heap.

As noted previously, a single heap cell $(l \mapsto \langle s, c\rangle)$ could be constructed with the expression $(\mathbf{constr}\ s\ c)$, and thus the denotation of this heap cell may just be taken to be $[\![\mathbf{constr}\ s\ c]\!]$. This can then be supplied as the denotation of l in any expression e involving l:

$$\begin{aligned}
[\![l\colon \sigma \vdash l \mapsto \langle s, c\rangle,\ e\colon \tau]\!] &= [\![\emptyset \vdash \mathbf{constr}\ s\ c\colon \Phi(\sigma)]\!]; [\![l\colon \Phi(\sigma) \vdash e\colon \tau]\!] \\
&= [\![\emptyset \vdash \mathbf{let}\ l\colon \Phi(\sigma)\ \mathbf{be}\ (\mathbf{constr}\ s\ c)\ \mathbf{in}\ e\colon \tau]\!]
\end{aligned}$$

We define the denotation of a heap inductively according to this scheme, where $[\![\Delta \vdash h]\!]\colon 1 \to [\![\Delta]\!]$, following the typing rules for heaps given in Section 4.3:

$$\begin{aligned}
[\![\emptyset \vdash \emptyset]\!] &= id_1 \\
[\![\Delta, l\colon \tau \vdash h, l \mapsto \langle s, c\rangle]\!] &= [\![\Delta \vdash h]\!]; d; \left(id_{[\![\Phi(\Delta)]\!]} \otimes [\mathbf{constr}\ s\ c]_{\Phi(\Delta)}\right)
\end{aligned}$$

Note that we use $[\mathbf{constr}\ s\ c]_{\Phi(\Delta)}$ in place of $[\![\mathbf{constr}\ s\ c]\!]_{\Phi(\Delta)}$ so that it is clear from the types that a heap cell is never undefined. The denotation of an expression in heap is then as follows:

$$[\![\Delta \vdash h, e\colon \tau]\!] = [\![\Delta \vdash h]\!]; [\![\Phi(\Delta) \vdash e\colon \tau]\!]$$

Just as we write $[\![e]\!]_\Delta$ for $[\![\Delta \vdash e\colon \tau]\!]$, we shall write $[\![h]\!]_\Delta$ for $[\![\Delta \vdash h]\!]$.

In the case of *flat* heaps, where there are no references to the heap contained in any heap cell, we could define

$$[\![\Delta \vdash l_0 \mapsto \langle s_0, c_0\rangle, \dots, l_n \mapsto \langle s_n, c_n\rangle = [\mathbf{constr}\ s_0\ c_0]_\emptyset \otimes \cdots \otimes [\mathbf{constr}\ s_0\ c_0]_\emptyset$$

The general definition agrees with this where it is defined. When a heap is flat, it can easily be split into two parts in any way we desire, using simple projections. We shall need to split the heap in two in the next chapter, as after evaluating an expression the updated pre-existing heap and the newly created heap are treated differently in our proof. Unfortunately, this is not possible for general heaps with the above definition, since the new heap may depend upon the old. We therefore introduce the following *relativised* denotation of heaps:

$$\begin{array}{rcl} [\![h]\!]^\Delta_{\Delta'} & : & [\![\Delta]\!] \to [\![\Delta, \Delta']\!] \\ [\![h]\!]^\Delta_\emptyset & = & id_{[\![\Phi(\Delta)]\!]} \\ [\![h, l \mapsto \langle s, c\rangle]\!]^\Delta_{\Delta', l\colon \tau} & = & [\![h]\!]^\Delta_{\Delta'}; d_{[\![\Phi(\Delta,\Delta')]\!]}; \\ & & \quad (id_{[\![\Phi(\Delta,\Delta')]\!]} \otimes [\mathbf{constr}\ s\ c]_{\Phi(\Delta,\Delta')}) \end{array}$$

Here we think of $[\![h]\!]^\Delta_{\Delta'}$ as being the denotation of in the context of some Δ-type heap. If heap we then supply is $h{\upharpoonright}_\Delta$, the expected property holds:

$$[\![h]\!]^\emptyset_{\Delta,\Delta'} = [\![h]\!]^\Delta_{\Delta'} \circ [\![h{\upharpoonright}_\Delta]\!]^\emptyset_\Delta$$

We shall in fact make use of the following more general property:

Lemma 4.8 (Relativised heaps). *For any heap h such that $\Delta_1, \Delta_2, \Delta_3 \vdash h$*

$$[\![h]\!]^{\Delta_1}_{\Delta_2,\Delta_3} = [\![h{\upharpoonright}_{\Delta_1,\Delta_2}]\!]^{\Delta_1}_{\Delta_2}; [\![h]\!]^{\Delta_1,\Delta_2}_{\Delta_3}$$

Proof. Trivial by induction on the structure of Δ. □

Note also that $[\![h]\!]_\Delta = [\![h]\!]^\emptyset_\Delta$.

We now have a result specific to $\mathcal{L}_{\text{pair}}$, showing that semantically "new locations don't escape", corresponding to the syntactic property of Lemma 4.3.

Lemma 4.9 ($\mathcal{L}_{\text{pair}}$ heap semantics). *Suppose for $\Delta \vdash h, e$ and $\Delta' \vdash h', v$ we have in $\mathcal{L}_{\text{pair}}$*

$$h, e \Downarrow h', v$$

where (ordering by the left-pointing typing of Theorem 4.2)

$$\Delta = l_1 \colon \lambda_1, \ldots, l_n \colon \lambda_n$$
$$\Delta' = \Xi_1, l_1 \colon \lambda_1, \ldots, \Xi_n, l_n \colon \lambda_n, \Xi_{n+1}$$

Then for each l_i, l_j *if* $\Xi = \Xi_{i+1}, l_{i+1} \colon \lambda_{i+1} \ldots \Xi_j, \lambda_j \colon l_j$ *there exists a morphism* $[\![h'_i]\!]^{\Delta_j}_{\Xi}$ *such that there is a factorisation*

$$\begin{tikzcd}
[\![\Delta'_j]\!] \arrow[r, "{[\![h'_i]\!]^{\Delta'_j}_{\Xi}}"] \arrow[d, "\Pi"'] & [\![\Delta'_i]\!] \\
[\![\Delta_j]\!] \arrow[ur, dashed, "{[\![h'_i]\!]^{\Delta_j}_{\Xi}}"'] &
\end{tikzcd}$$

We note that the new notation $[\![h'_i]\!]^{\Delta_j}_{\Xi}$ does not conflict with the existing notation due to the specification of Δ_j—the only ambiguity might be if $\Delta_j = \Delta'_j$, and then $\Pi = id_{[\![\Delta_j]\!]}$ anyway.

Proof. By induction on the length of Ξ. If $\Xi = l \colon \lambda$ then by Lemma 4.3, for each $l' \in \Delta'_j \backslash \Delta_j$, $l' \notin \mathrm{FV}(h'(l))$. $[\![h'_i]\!]^{\Delta_j}_{l \colon \lambda} = [\![h'(l)]\!]_{\Delta_j} ; \bot(\mathit{thread})$, and by weakening $[\![h'(l)]\!]_{\Delta'_j} = \Pi_{\Delta_j} ; [\![h'(l)]\!]_{\Delta_j}$. $\square$

$$
\begin{array}{rcl}
[\![\iota]\!] & = & !N = N \\
[\![\tau \to \tau']\!] & = & [\![\tau]\!] \multimap [\![\tau']\!]_\perp \\
[\![\sigma \otimes \tau]\!] & = & [\![\sigma]\!] \otimes [\![\tau]\!] \\
[\![\mathbf{Obj}\ \{m_1 = \tau_1, \ldots, m_n = \tau_n\}]\!] & = & !\&_{m \in \{1 \ldots n\}} [\![\tau_m]\!] \\
[\![\mathbf{CObj}_X\ ]\!] & = & [\![\mathbf{Obj}\ X]\!]
\end{array}
$$

Figure 4.10: Denotation of Types

$$
\begin{array}{rcl}
[\![\tau <: \tau]\!] & = & id_{[\![\tau]\!]} \\
[\![\tau_1 \otimes \tau_2 <: \tau_1' \otimes \tau_2']\!] & = & [\![\tau_1 <: \tau_1']\!] \otimes [\![\tau_2 <: \tau_2']\!] \\
[\![\mathbf{Obj}\ \{m_{\pi 1} \colon \tau'_{\pi 1}, \ldots, m_{\pi m} \colon \tau'_{\pi m}\} <: & = & !\&_{i \in 1,\ldots,m} (\Pi_{\pi i} ; [\![\tau_{\pi i} <: \tau'_{\pi i}]\!]) \\
\mathbf{Obj}\ \{m_1 \colon \tau_1, \ldots, m_n \colon \tau_n\}]\!] & & \\
[\![\mathbf{CObj}_X\ <: \mathbf{Obj}\ X]\!] & = & id_{[\![\mathbf{Obj}\ X]\!]} \\
[\![\mathbf{CMeth}_\sigma\ (\tau \otimes \tau' \to \tau \otimes <) : (\sigma \otimes \tau \to \sigma \otimes \tau')]\!] & = & id_{[\![\sigma \otimes \tau \to \sigma \otimes \tau']\!]}
\end{array}
$$

Figure 4.11: Denotation of Subtyping

$$
\begin{array}{rcl}
\llbracket \Gamma, x\colon \tau, \Delta \vdash x\colon \tau \rrbracket & = & \eta_{\llbracket \tau \rrbracket} \circ \Pi_{\llbracket \tau \rrbracket} \\
\llbracket \Gamma \vdash c_{\varphi}\colon \iota \otimes \ldots \otimes \iota \to \iota \rrbracket & = & \eta \circ \lambda(\bar{\varphi}) \circ 1_{\llbracket \Gamma \rrbracket} \\
\llbracket \Gamma, \Delta \vdash \mathbf{ifz}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\colon \tau \rrbracket & = & \mathit{ifz}_{\tau} \circ (\llbracket \Gamma \vdash e\colon \iota \rrbracket \otimes \\
 & & \quad \llbracket \Delta \vdash e_1\colon \tau \rrbracket \& \llbracket \Delta \vdash e_2\colon \tau \rrbracket) \\
\llbracket \Gamma, \Delta \vdash \langle e, e' \rangle\colon \tau \otimes \tau' \rrbracket & = & \psi \circ (\llbracket \Gamma \vdash e\colon \tau \rrbracket \otimes \llbracket \Delta \vdash e'\colon \tau' \rrbracket) \\
\llbracket \Gamma, \Delta \vdash \mathbf{let}\ \langle x, y \rangle\ \mathbf{be}\ e_1\ \mathbf{in}\ e_2\colon \tau \rrbracket & = & \llbracket \Gamma, x\colon \tau_1, y\colon \tau_2 \vdash e_2\colon \tau \rrbracket^{\dagger} \circ \psi \circ \\
 & & \quad (\eta_{\llbracket \Gamma \rrbracket} \otimes \llbracket \Delta \vdash e_1\colon \tau_1 \otimes \tau_2 \rrbracket) \\
\llbracket \Gamma \vdash \mathbf{obj}\ \{m_1 = e_1, \ldots, m_n = e_n\} & = & \bot(pp^n) \circ \psi^n \circ \\
\colon \mathbf{Obj}\ \{m_1\colon \tau_1, \ldots, m_n\colon \tau_n\} \rrbracket & & \quad (\bigotimes_{1 \le i \le n} \llbracket \Gamma \vdash e_i\colon \tau_i \rrbracket^{\sharp} \circ d^n) \\
\llbracket \Gamma \vdash \mathbf{obj}\ X\colon \mathbf{CObj}_T \rrbracket & = & \llbracket \Gamma \vdash \mathbf{obj}\ X\colon \mathbf{Obj}\ T \rrbracket \\
\llbracket \Gamma, z\colon \tau, \Delta \vdash e[z/x, z/y]\colon \tau' \rrbracket & = & \llbracket \Gamma, x\colon \tau, y, \colon \tau, \Delta \vdash e\colon \tau' \rrbracket \circ \\
 & & \quad (id_{\llbracket \Gamma \rrbracket} \otimes d_{\tau} \otimes id_{\llbracket \Delta \rrbracket}) \\
\llbracket \Gamma \vdash e \cdot m\colon \tau \rrbracket & = & \bot(\Pi_m \circ \varepsilon) \circ \llbracket \Gamma \vdash e\colon \mathbf{Obj}\ \{m\colon \tau\} \rrbracket \\
\llbracket \Gamma \vdash \lambda x.e\colon \tau \to \tau' \rrbracket & = & \eta \circ \lambda_{\tau}(\llbracket \Gamma, x\colon \tau \vdash e\colon \tau' \rrbracket) \\
\llbracket \Gamma, \Delta \vdash e_1\ e_2\colon \tau \rrbracket & = & \mathit{eval}^{\dagger} \circ \psi \circ \\
 & & \quad \llbracket \Gamma \vdash e_1\colon \sigma \to \tau \rrbracket \otimes \llbracket \Delta \vdash e_2\colon \sigma \rrbracket \\
\llbracket \Gamma \vdash \mathbf{Y}(e)\colon \rho \rrbracket & = & \bot(Y_{\llbracket \rho \rrbracket} \circ !(id_{\llbracket \rho \rrbracket} \multimap \mu_{\rho})) \circ \llbracket e \rrbracket^{\sharp}_{\Gamma} \\
\llbracket \Gamma, \Delta \vdash \mathbf{constr}\ e_s\ e_c\colon \tau \rrbracket & = & \bot(\mathit{thread}) \circ \psi \circ \\
 & & \quad (\llbracket \Gamma \vdash e_s\colon \tau_1 \rrbracket \otimes \llbracket \Delta \vdash e_c\colon \tau_2 \rrbracket) \\
\llbracket \Gamma \vdash e\colon \tau' \rrbracket & = & \llbracket \tau <: \tau' \rrbracket \circ \llbracket \Gamma \vdash e\colon \tau \rrbracket
\end{array}
$$

Figure 4.12: Denotation of Terms

Chapter 5

Proof of soundness

Having given a language with a denotational semantics, and an operational semantics matching our intuitive understanding of the language, we naturally wish to show that these agree, or in the usual terminology that our denotational semantics is *adequate* with respect to our operational semantics. We break adequacy into two directions, that any result predicted by the operational semantics is also given by the denotational semantics (soundness), and the converse.

In this chapter we shall give a soundness proof. Typically a soundness proof is a straightforward induction on derivations, however in our setting it turns out to be highly non-trivial, requiring a surprisingly strong induction claim and the introduction of some seemingly new ideas. The necessary techniques are one of the main contributions of the thesis. There is considerable distance between our denotational and operational semantics, and the proof requires an analysis of the relationship between the abstract and concrete views of the behaviour of objects in particular. Even so, we have so far succeeded in completing the proof only for $\mathcal{L}_{\text{pair}}$, whereas we believe the result to hold for the whole of $\mathcal{L}_{\text{arg}}$.

As it is not at all clear from the statement of our soundness claim why our formulation is the right one, we shall guide the reader through a series of successive refinements to a proposed soundness property. We then spend some time giving necessary lemmas, before giving a proof by induction on operational semantics derivations. We conclude the chapter by discussing the prospects for the extension of our proof to $\mathcal{L}_{\text{arg}}$, and also proof of the remaining direction of adequacy.

5.1 Choice of induction claim

The essence of soundness is that the observations which one can make operationally on the behaviour of an expression correspond to properties of its denotation (in our case a strategy). The most basic property one can observe is termination:

$$e \Downarrow \quad \Rightarrow \quad [\![e]\!] \neq \bot$$

This property[1] simply says that if an expression terminates in our operational semantics, the corresponding strategy is non-empty, i.e. makes some response to the initial question. Alternatively, one might thus consider allowing arbitrary observations at ground type:

$$e \Downarrow v \colon \gamma \quad \Rightarrow \quad [\![e]\!] = [\![v]\!]$$

This statement is equivalent to the previous one, in the sense that any ground-type observation can be expressed as an observation on termination through the use of conditional statements, but is perhaps more amenable to proof. Explicitly, if $e \Downarrow v \colon \gamma$ then we are requiring that $[\![e]\!]$ is in fact the same strategy as $[\![v]\!]$, which responds to the initial question with the *move* corresponding to the ground-type value v.

To prove the above property for ground types inductively on operational semantics derivations one naturally needs to consider expressions of higher type, as these may appear in the evaluation of ground-type terms:

$$e \Downarrow v \colon \tau \quad \Rightarrow \quad [\![e]\!] = [\![v]\!]$$

Here even at higher types we are simply requiring the equality of $[\![e]\!]$ and $[\![v]\!]$, although now this means asserting that two strategies potentially consisting of infinitely deep trees are identical. However, the equation $[\![e]\!] = [\![v]\!]$ is not even well-typed in general, since v may include heap locations even when e does not. We should therefore consider the following:

$$\emptyset, e \Downarrow h', v \colon \tau \quad \Rightarrow \quad [\![e]\!] = [\![v]\!] \circ [\![h']\!]$$

Here the expression v may contain locations from h'; if $l \in FV(v)$ the corresponding component l of $[\![h']\!]$ provides the behaviour of the object $h(l)$, whereas on the

[1] In this section we implicitly assume all terms are well-typed, as our untyped operational semantics can only be asked to coincide with the semantics given to typed terms on terms which can actually be assigned a type.

left hand side the corresponding behaviour is incorporated into the behaviour of $[\![e]\!]$ itself. Most simply, consider the following evaluation for values c, s:

$$\emptyset, \mathbf{constr}\ s\ c \Downarrow (l \mapsto \langle s, c\rangle), l$$

Here $[\![\emptyset \vdash \mathbf{constr}\ s\ c]\!] = [\![l \vdash l]\!] \circ [\![\mathbf{constr}\ s\ c]\!]$, and in fact the value l contributes no interesting behaviour. Again, to prove the above one must take a stronger induction claim since evaluation of an expression in an empty heap may involve evaluating some other expression in a non-empty heap. This property is particularly natural, matching the form of our operational semantics, so we choose this formulation as our main soundness property:

Theorem 5.1 (Soundness)**.**

$$h, e \Downarrow h', v \quad \Rightarrow \quad [\![e]\!] \circ [\![h]\!] = [\![v]\!] \circ [\![h']\!]$$

So far we have discussed a series of induction claims which are clearly far too weak, but we have now entered the realm of apparent possibility. However, we will in fact need a much stronger claim. The problem here is that there is not enough information exposed regarding the interaction between heap and expression in $[\![e]\!] \circ [\![h]\!]$ and $[\![v]\!] \circ [\![h']\!]$—the expression and heap are too tightly coupled. Consider the evaluation of $h, e_1; e_2$ (or of $h, \langle e_1, e_2\rangle$), where e_1 is of unit type. In this case if $h, e_1 \Downarrow h', v_1$, clearly $[\![v]\!] \circ [\![h']\!]$ does not say anything about $[\![h']\!]$, while $[\![e_2]\!] \circ [\![h']\!]$ requires information about $[\![h']\!]$.

In essence, we need to know that the denotation of the resulting heap $[\![h']\!]$ behaves correctly not only under interaction with $[\![v]\!]$, but more generally under all possible interactions that might arise from other parts of a larger program. Assuming for now the existence of some operation $\rightsquigarrow_s$ taking a heap to that heap after the interaction s, the following might appear to express this idea:

$$h, e \Downarrow h', v \quad \Rightarrow \quad \exists s. [\![h]\!] \rightsquigarrow_s [\![h']\!] \ \wedge\ \forall t.\ qsa^v t \in [\![h]\!] \| [\![e]\!] \Leftrightarrow qa^v t \in [\![h']\!] \| [\![v]\!]$$

Here we are saying that for every play qsa^vt in $[\![e]\!]$ in the original heap, where q and a^v are the initial question and matching answer (with ground-type data v), there is a play qa^vt in $[\![v]\!]$ in the new heap which omits the initial heap interaction (since that is how the new heap arose) but gives the same result and is thereafter the same. Note that this implies the statement of Theorem 5.1, but says more since the relation of the denotations of the heap before and after evaluation are related by $[\![h]\!] \rightsquigarrow_s [\![h']\!]$. This is the idea used in [7] for Idealized Algol.

To see why this formulation is not sufficient for our purposes, consider the evaluation of $h, \langle e_1, e_2\rangle$—in effect equivalent to command sequencing in Idealized Algol if e_1 has unit type—which proceeds as follows

$$\frac{h, e_1 \Downarrow h_1, v_1 \quad h_1, e_2 \Downarrow h_2, v_2}{h, \langle e_1, e_2\rangle \Downarrow h_2, \langle v_1, v_2\rangle}$$

Roughly speaking, we also need to know that the denotation of the resulting value $[\![v]\!]$ behaves correctly not only under interaction with the corresponding heap $[\![h']\!]$, but more generally under all possible interactions with heaps that might arise in the evaluation of a larger program. Assume that $[\![h]\!] \rightsquigarrow_{s_1} [\![h_1]\!]$ with $qs_1a^{v_1}t_1 \in [\![h]\!]||[\![e_1]\!] \Leftrightarrow qa^{v_1}t_1 \in [\![h_1]\!]||[\![v_1]\!]$ and $[\![h_1]\!] \rightsquigarrow_{s_2} [\![h_2]\!]$ with $qs_2a^{v_2}t_2 \in [\![h_1]\!]||[\![v_1]\!] \Leftrightarrow qa^{v_2}t_2 \in [\![h_2]\!]||[\![v_2]\!]$. One can show that $[\![h]\!] \rightsquigarrow_{s_1s_2} [\![h_2]\!]$ and

$$qs_1s_2a^{v_1,v_2}t_2 \in [\![h]\!]||[\![\langle e_1, e_2\rangle]\!] \Leftrightarrow qa^{v_1,v_2}t_2 \in [\![h_2]\!]||[\![\langle v_1, v_2\rangle]\!]$$

where t_2 is some play in the right component (so matches the t_2 in our assumption). However one does not know that

$$qs_1s_2a^{v_1,v_2}t_1 \in [\![h]\!]||[\![\langle e_1, e_2\rangle]\!] \Leftrightarrow qa^{v_1,v_2}t_1 \in [\![h_2]\!]||[\![\langle v_1, v_2\rangle]\!]$$

where t_1 is some play in the left component, since the assumption only gives information on the behaviour of e_1 and v_1 under the assumption that after the initial evaluation the heap behaves like h_1. If the heap is updated before further interaction with v_1, we need to know the behaviour under that new heap. We note that this issue does not apply in the case of the Idealized Algol proof, since in the corresponding situation e_1 is of ground type, and so permits no further interaction.

We must thus enrich our induction claim to give information on the behaviour of the expression in all possible *future* heaps. The behaviour of $[\![e]\!]$ in all possible heaps after some initial interaction qsa is just the subforest of the strategy $[\![e]\!]$ rooted below the prefix qsa. The substrategy σ of $[\![e]\!]$ consisting of all plays prefixed by qsa is of interest; the strategy (not in fact a substrategy) $[\![e]\!]_s$—defined by memoization in Section 2.5—which omits s but otherwise behaves as σ is just $[\![e]\!]$ after the initial heap interaction s, and so should match $[\![v]\!]$:

$$h, e \Downarrow h', v \quad \Rightarrow \quad \exists qsa^v \in [\![h]\!]||[\![e]\!].\ [\![e]\!]^s = [\![v]\!] \wedge [\![h]\!]_s = [\![h']\!]$$

Here $[\![h]\!]_s$ is the matching notion for heaps, and indeed with relation to the previous statement one could say that $[\![h]\!] \rightsquigarrow_s [\![h']\!]$ where $[\![h]\!]_s = [\![h']\!]$. It should be noted that when the above claim gives $\llbracket h' \rrbracket ; \llbracket v \rrbracket = \llbracket h \rrbracket_s ; \llbracket e \rrbracket^s$, it is a general fact that $\sigma_s ; \tau^s = \sigma ; \tau$, so $\llbracket h' \rrbracket ; \llbracket v \rrbracket = \llbracket h \rrbracket ; \llbracket e \rrbracket$ as desired.

Before moving onto more complex issues, we resolve one problem with the above formulation: it does not take account of new heap locations in h'. It is not reasonable to ask that $\llbracket h \rrbracket_s = \llbracket h' \rrbracket$ when h' is a larger heap than h; instead, we merely wish to assert that they agree on their common portion. As in the earlier formulation, the additional portion of the heap should be thought of in conjunction with $\llbracket v \rrbracket$ as follows:

$$\Delta \vdash h, e \Downarrow \Delta, \Delta' \vdash h', v \quad \Rightarrow \quad \exists qsa^v \in \llbracket h \rrbracket \| \llbracket e \rrbracket .\ \llbracket e \rrbracket^s = \llbracket v \rrbracket \circ \llbracket h' \rrbracket^{\Delta}_{\Delta'} \wedge \llbracket h \rrbracket_s = \llbracket h' \rrbracket_{\Delta}$$

Here $\llbracket h' \rrbracket^{\Delta}_{\Delta'}$ is the portion of h' in Δ', leaving the Δ portion to be filled in later.

The above relates the unevaluated and evaluated expressions and corresponding heaps correctly where it is defined, but unfortunately still does not make sense in general, which is to say the statement is not well-typed. The problem is that the types of $\llbracket h \rrbracket_s$ and $\llbracket h' \rrbracket$, and of $\llbracket e \rrbracket^s$ and $\llbracket v \rrbracket$, still do not quite match. If the heap type is Δ, we have $1 \xrightarrow{\llbracket h \rrbracket_s} \llbracket \Delta \rrbracket_s \xrightarrow{\llbracket e \rrbracket^s} X_\perp$. From Lemma 2.13, $\llbracket \Delta \rrbracket_s$ can be decomposed as $Z_s \otimes \llbracket \Delta \rrbracket$—a copy of $\llbracket \Delta \rrbracket$ for future interaction, and a *residue* Z_s. This residue corresponds to the fact that $\llbracket \Delta \rrbracket_s$ may allow for further play in the components already opened in s of any given heap object, or in other words allows for further interaction with the argument or result of method invocations in s. If we could exclude this interaction with Z_s (e.g. by restricting method types), the following formulation might suffice, if the isomorphism is the canonical one given by Lemma 2.13:

$$\begin{tikzcd}
 & 1 \arrow[r, "\llbracket \Delta \vdash h \rrbracket^s"] \arrow[ddr, "\llbracket \Delta \vdash h' \restriction \Delta \rrbracket"'] & \llbracket \Delta \rrbracket_s \arrow[rr, "\llbracket \Delta \vdash e \rrbracket_s"] \arrow[d, phantom, "\cong"] & & X \\
h, e \Downarrow h', v \quad \Rightarrow & & Z_s \otimes \llbracket \Delta \rrbracket \arrow[d, "\Pi"] & & \\
 & & \llbracket \Delta \rrbracket \arrow[rr, "\llbracket \Delta, \Delta' \vdash h' \rrbracket^{\Delta}_{\Delta'}"'] & & \llbracket \Delta \rrbracket \otimes \llbracket \Delta' \rrbracket \arrow[uu, "\llbracket \Delta, \Delta' \vdash v \rrbracket"']
\end{tikzcd}$$

Here we simply throw away any possible continued interaction in opened components with a projection, leaving an updated heap of the original type.

In general however, we do wish to allow for methods which return interesting results, and expressions which do interesting things with them, so we must find a way to add back the "missing" information. We first introduce a little notation.

Definition 5.2. Given a morphism $f\colon X \to Y$ where X is a reusable object (as per Definition 4.4), define $f^{\lhd}\colon X \to Y \otimes X$ as

$$f^{\lhd} = d_X; (f \otimes id_X)$$

and $f^{\rhd}\colon X \to X \otimes Y$ as

$$f^{\rhd} = d_X; (id_X \otimes f)$$

Then we can consider the following revised formulation:

$$\begin{tikzcd}
 & & 1 \arrow[r, "{[\![\Delta \vdash h]\!]^s}"] \arrow[ddr, "{[\![\Delta \vdash h'\upharpoonright_{\Delta}]\!]}" sloped] & {[\![\Delta]\!]_s} \arrow[r, "{[\![\Delta \vdash e]\!]_s}"] \arrow[d, phantom, "\cong"] & X \\
h, e \Downarrow h', v & \Rightarrow & & Z_s \otimes [\![\Delta]\!] & \\
 & & & {[\![\Delta]\!]} \arrow[u, "\theta^{\lhd}"'] \arrow[r, "{[\![\Delta, \Delta' \vdash h']\!]^{\Delta}_{\Delta'}}"'] & {[\![\Delta]\!] \otimes [\![\Delta']\!]} \arrow[uu, "{[\![\Delta, \Delta' \vdash v]\!]}"']
\end{tikzcd}$$

Here the morphism $\theta\colon [\![\Delta]\!] \to Z_s$ supplies the continued behaviour in question. In particular, for each method invocation m that occurs in s when evaluating e to a value, θ gives the results of future interaction with m—namely the content of any higher-type return value, which may depend on objects in the heap.

Given θ, there would now seem to be enough information; indeed, unlike the previous few formulations this version is both well-typed and *true* in general. Indeed, the claim we have arrived at seems more or less equivalent in strength to the Soundness claim used in [10]. The claim there is that "if for some term M we have $(L, s)\ M \Downarrow (L', s')\ V$ then $[\![\texttt{new}\ L, s\ \texttt{in}\ (\lambda x.N)\ M]\!] = [\![\texttt{new}\ L', s'\ \texttt{in}\ (\lambda x.N)\ V]\!]$ for any suitably typed term N". Here the `new` expressions correspond to our $[\![h]\!]$ and $[\![h']\!]$ and M and V to e and v, while the observing term $\lambda x.\ N$ seems to play the rôle of ensuring that both heap and location agree.

This formulation is very elegant (although we feel there is also a virtue to the directness of our more explicit formulation). However, in our case there is still insufficient information for the purpose of proof. In the context of the general references of [10], the entirety of the heap is created in one step, and each location initialised in a second, while in our context it is not possible to divorce object creation and initialisation. This means that we are not able to consider heap locations as reorderable and must instead take seriously the dependence of one heap location on earlier locations, giving a proof which respects this structure.

Consider a method invocation on some location l in the middle of the heap. This will result in the update of the heap location h_l to some new value h'_l, and some particular reasoning about the semantics of method invocation will relate the memoized strategy $h_l^{s_l}$ with the syntactically derived updated one h'_l. However, the triangle of the above diagram only relates the two versions of the later heap cells in the context of the *old* value of h_l, and says nothing relevant to the composition with the updated h'_l.

We must therefore specify the relation of the two versions of *each* given heap cell, so that if a cell references earlier heap cells which are updated, there is enough information to know that the relationship still holds.

Recall from the previous chapter that if $\Delta_i = l_1 \colon \lambda_1, \ldots, l_n \colon \lambda_i$ (and $\Delta = \Delta_n$), by the property of the "relativised" denotation of heaps

$$\llbracket h \rrbracket_\Delta \;=\; 1 \xrightarrow{\llbracket h_1 \rrbracket} \llbracket \lambda_1 \rrbracket \longrightarrow \ldots \longrightarrow \llbracket \Delta_{i-1} \rrbracket \xrightarrow{\llbracket h_i \rrbracket^{\Delta_{i-1}}_{l_i \colon \lambda_i}} \llbracket \Delta_i \rrbracket \longrightarrow \ldots \xrightarrow{\llbracket h \rrbracket^{\Delta_{n-1}}_{l_n \colon \lambda_n}} \llbracket \Delta \rrbracket$$

where we write h_i for $h \upharpoonright_{\Delta_i}$ (the initial portion of the syntactic heap h up to location l_i). Given a terminating play qta^v in $\llbracket h \rrbracket_\Delta \| \llbracket e \rrbracket_\Delta$, the process of resplitting (as given by Lemma 2.6) yields when applied repeatedly play t_i at each $\llbracket \Delta_i \rrbracket$ such that

$$\begin{aligned} (\llbracket h \rrbracket_\Delta)_{t_i} \;&=\; 1 \xrightarrow{\llbracket h_1 \rrbracket_{t_1}} \llbracket \lambda_1 \rrbracket_{t_1} \longrightarrow \ldots \\ &\ldots \longrightarrow \llbracket \Delta_{i-1} \rrbracket_{t_{i-1}} \xrightarrow{(\llbracket h_i \rrbracket^{\Delta_{i-1}}_{l_i \colon \lambda_i})^{t_{i-1}}_{t_i}} \llbracket \Delta_i \rrbracket_{t_i} \longrightarrow \ldots \xrightarrow{(\llbracket h \rrbracket^{\Delta_{n-1}}_{l_n \colon \lambda_n})^{t_{n-1}}_{t_n}} \llbracket \Delta \rrbracket_{t_n} \end{aligned} \tag{5.1}$$

where $t_n = t$. From here on, we shall abbreviate $\llbracket h_i \rrbracket^{\Delta_{i-1}}_{l_i \colon \lambda_i}$ as H_i. We can introduce a morphism $\zeta_i \colon Z_{i-1} \otimes \llbracket \Delta_{i-1} \rrbracket \to Z_i$ and demand that

$$\forall i \in \mathrm{dom}(h). \qquad \begin{tikzcd} \llbracket \Delta_{i-1} \rrbracket_{t_{i-1}} \arrow[r, "(H_i)^{t_{i-1}}_{t_i}"] \arrow[d, phantom, "\cong"] & \llbracket \Delta_i \rrbracket_{t_i} \arrow[d, phantom, "\cong"] \\ Z_{i-1} \otimes \llbracket \Delta_{i-1} \rrbracket \arrow[r, "(id_{Z_{i-1}} \otimes d_{\Delta_{i-1}});(\zeta_i \otimes \llbracket h'_i \rrbracket^{\Delta_{i-1}}_{l_i \colon \lambda_i})"] & Z_i \otimes \llbracket \Delta_i \rrbracket \end{tikzcd}$$

The collection of morphisms $\zeta_1, \ldots, \zeta_n$ serve to specify θ as in the previous diagram: we define $\theta = \mathrm{zip}(\zeta_1, \ldots, \zeta_n) \colon \llbracket \Delta \rrbracket \to Z_s$, the *zipping* of $\zeta_1, \ldots, \zeta_n$, i.e. their composition with the appropriate copying of (the appropriate parts of) $\llbracket \Delta \rrbracket$. Then the fact that for each i the above square commutes means that the triangle in the previous diagram commutes; the square in that diagram can remain unchanged. We introduce the formal definition of zipping. Here and in the next few definitions, we write Δ for the object playing the rôle of $\llbracket \Delta \rrbracket$.

Definition 5.3 (Zipping)**.** For reusable objects Δ, Ξ, λ, given morphisms $\theta\colon \Delta \to Z$ and $\zeta\colon Z \otimes \Delta \otimes \Xi \to Z'$ where $\Delta' = \Delta \otimes \Xi \otimes \lambda$, define the zipping of these as

$$\mathrm{zip}(\theta, \zeta) = \Pi_{\Delta,\Xi}; (\Pi_{\Delta}; \theta)^{\lhd}; \zeta\colon \Delta' \to Z'$$

Given a collection of morphisms $\zeta_1, \ldots, \zeta_n$ which have compatible types

$$\zeta_{i+1}\colon Z_i \otimes \Delta_i \otimes \Xi_i \to Z_{i+1}$$

(where $Z_0 = 1, \Delta_{i+1} = \Delta_i \otimes \Xi_i \otimes \lambda_i$) extend the notation as follows:

$$\mathrm{zip}(\zeta_1, \ldots, \zeta_n) = \mathrm{zip}(\mathrm{zip}(\zeta_1, \ldots, \zeta_{n-1}), \zeta_n)$$

This formulation is very close to being sufficient. In fact the property we have specified is slightly too strong: the Z_{i-1} at the bottom left of the above diagram allows for *any* continued interaction with the residue Z_{i-1}, while the square may only hold for some particular interaction in Z_{i-1}. The issue here is that part of Z_{i-1} may represent interaction between h_i and the results of some method invocation on an object h_j $(j < i)$. Then the memoized h_i will make use of ζ_j (while $[\![h'_i]\!]$ does not), and so the square will only commute when this particular ζ_j is supplied.

The notion of zipping comes to the rescue here. Where $\theta_i = \mathrm{zip}(\zeta_1, \ldots, \zeta_i)$, the following diagram fills in the missing information correctly:

$$\forall i \in \mathrm{dom}(h). \quad \begin{tikzcd}
{[\![\Delta_{i-1}]\!]_{t_{i-1}}} \arrow[r, "(H_i)^{t_{i-1}}_{t_i}"] \arrow[d, phantom, "\cong"] & {[\![\Delta_i]\!]_{t_i}} \arrow[d, phantom, "\cong"] \\
Z_{i-1} \otimes [\![\Delta_{i-1}]\!] & Z_i \otimes [\![\Delta_i]\!] \\
{[\![\Delta_{i-1}]\!]} \arrow[u, "\theta_{i-1}^{\lhd}"] \arrow[r, "{[\![h'_i]\!]^{\Delta_{i-1}}_{l_i\,:\,\lambda_i}}"] & {[\![\Delta_i]\!]} \arrow[u, "\theta_i^{\lhd}"']
\end{tikzcd}$$

The sequence of such diagrams then composes to the triangle given before.

It turns out that we need to know a little more about ζ_i, namely that on the parts of the residue on which H_i is simply a copycat (and so any play appears both in t_i and t_{i-1}), ζ_i also passes through unaltered. We introduce an auxiliary notion ζ_i^*, where ζ_i will specify only the parts which are required, and elsewhere ζ_i^* will behave as the identity.

Definition 5.4. For a morphism $\zeta\colon Z^R \otimes \Delta \to Z'^B$ and suitable objects Z, Z_1 we can define a morphism $\zeta^*\colon Z \otimes \Delta \to Z'$ which extends ζ with some copycat behaviour. With respect to a given pair of isomorphisms $\imath\colon Z \cong Z^L \otimes Z^R$, $\jmath\colon Z^L \otimes Z'^B \cong Z'$ (typically determined by the context) we define

$$\zeta^* = (\imath \otimes id_\Delta); (id_{Z^L} \otimes \zeta); \jmath$$

Thus we instead require that $\theta_i = \mathrm{zip}(\zeta_1^*, \ldots, \zeta_i^*)$ for *suitable* $\imath$, $\jmath$ in the diagram above. From the definition of H_i, if $h(l_i) = \langle v_s, v_c\rangle$ then

$$[\![h_i]\!]^{\Delta_{i-1}}_{l_i\colon \lambda_i} = d; (id \otimes [\mathbf{constr}\ v_s\ v_c]_{\Delta_{i-1}})$$

Therefore there is some $\hat{t}_{i-1}$ with

$$(H_i)^{t_{i-1}}_{t_i} \;=\; (\Delta_{i-1})_{t_{i-1}} \xrightarrow{d^{t_{i-1}}_{\hat{t}_{i-1}}} (\Delta_{i-1} \otimes \Delta_{i-1})_{\hat{t}_{i-1}} \xrightarrow{(id \otimes [\mathbf{constr}\ v_s\ v_c]_{\Delta_{i-1}})^{\hat{t}_{i-1}}_{\hat{t}_i}} (\Delta_i)_{t_i}$$

In addition to $(\Delta_i)_{t_i} \cong Z_i \otimes [\![\Delta_i]\!]$, we can read off from Lemma 2.13 that since $[\![\Delta_i]\!] = [\![\Delta_{i-1}]\!] \otimes [\![\lambda_i]\!]$, there is a matching decomposition $Z_i \cong (Z_i^A \otimes Z_i^B)$. Similarly, where $([\![\Delta_{i-1}]\!] \otimes [\![\Delta_{i-1}]\!])_{\hat{t}_{i-1}} \cong ([\![\Delta_{i-1}]\!] \otimes [\![\Delta_{i-1}]\!]) \otimes Z'_{i-1}$, there is a decomposition $Z'_{i-1} \cong Z^L_{i-1} \otimes Z^R_{i-1}$. Since the morphism $(id \otimes [\mathbf{constr}\ v_s\ v_c]_{\Delta_{i-1}})^{\hat{t}_{i-1}}_{\hat{t}_i}$ is the memoization of a pair of morphisms, the left of which is the identity, we can see from Lemmas 2.8, 2.10 that we must have $Z^L_{i-1} = Z^A_i$. Furthermore, since the contraction d allocates components between the two copies of Δ_{i-1} and thereafter is the copycat on each of these components, the memoized contraction $d^{\hat{t}_{i-1}}_{\hat{t}_{i-1}}$ induces an isomorphism $Z_{i-1} \cong Z^L_{i-1} \otimes Z^R_{i-1}$ corresponding to these two sets of allocated components. We then take the isomorphisms:

$$\imath_i\colon Z_{i-1} \cong Z^L_{i-1} \otimes Z^R_{i-1} \qquad Z^L_{i-1} \otimes Z^B_i \cong Z_i\colon \jmath_i \tag{5.2}$$

We note that we have so far failed to account for the possibility of new heap cells in h' between l_{i-1} and l_i. Instead of $[\![h'_i]\!]^{\Delta_{i-1}}_{l_i\colon \lambda_i}$ we should include $[\![h'_i]\!]^{\Delta'_{i-1}}_{\Xi_i, l_i\colon \lambda_i}$ in the bottom line of the above diagram. While in $\mathcal{L}_{\text{pair}}$ the new portion of heap Ξ_i may still appear before l_i, but is not involved h' after l_i, in $\mathcal{L}_{\text{ret}}$ or $\mathcal{L}_{\text{arg}}$ this restriction is lifted.

We thus now have the enlarged type Δ'_i on the bottom line of the diagram (and an enlarged type for ζ_i and θ_i). We shall write H'_i to denote

$$[\![h'_i]\!]^{\Delta'_{i-1}}_{\Xi_i, l_i\colon \lambda_i}\colon\ [\![\Delta'_{i-1}]\!] \to [\![\Delta'_{i-1}, \Xi_i, l_i\colon \lambda_i]\!]$$

We shall also define $\theta^\pi_i = \theta^\lhd_i; (id_{Z_i} \otimes \Pi_{\Delta_i});$ where $\Pi_{\Delta_i}\colon \Delta'_i \to \Delta_i$. Where

$\theta_i = \mathrm{zip}(\zeta_1^*, \ldots, \zeta_i^*)$:

$$\forall i \in \mathrm{dom}(h). \qquad \begin{tikzcd} \llbracket \Delta_{i-1} \rrbracket_{t_{i-1}} \arrow[r, "(H_i)_{t_i}^{t_{i-1}}"] \arrow[d, phantom, "\cong"] & \llbracket \Delta_i \rrbracket_{t_i} \arrow[d, phantom, "\cong"] \\ Z_{i-1} \otimes \llbracket \Delta_{i-1} \rrbracket & Z_i \otimes \llbracket \Delta_i \rrbracket \\ \llbracket \Delta'_{i-1} \rrbracket \arrow[u, "\theta_{i-1}^{\pi}"] \arrow[r, "H'_i"] & \llbracket \Delta'_i \rrbracket \arrow[u, "\theta_i^{\pi}"] \end{tikzcd}$$

At this point we make our first simplification for the purposes of the proof, and restrict to the language $\mathcal{L}_{\mathrm{ret}}$ (we discuss the prospects for removal of this restriction after the proof). This gives some additional structure which we shall make use of in the proof. In $\mathcal{L}_{\mathrm{ret}}$, the result of a method invocation m_1 (or a locally constructed object) is never directly "split" between the state and return value in a calling method m_2. As a consequence, we can think of θ_i as being a product of a number of morphisms, each giving the behaviour of some component of the residue Z_i corresponding to the result of one method call (we shall call these *fibres*). Similarly, ζ_i is a product of a number of morphisms, each giving the behaviour of the part of the residue corresponding to one method call. Each such morphism might depend on a number of different fibres of θ_{i-1}, but since each such fibre is only used once, θ_i also consists of a (possibly smaller) number of fibres.

Definition 5.5 (Fibred morphisms)**.** A morphism $f\colon A \otimes \Delta \to B$ for a reusable object Δ is *fibred* with respect to decompositions $\imath\colon A \cong \bigotimes_{i\in X} A_i$, $\jmath\colon B \cong \bigotimes_{j\in Y} B_j$ for finite sets X and Y if there are disjoint sets X_j with $X = \bigsqcup_{j\in Y} X_j$ and for $j \in Y$ morphisms $f_j\colon \bigotimes_{i\in X_j} A_i \otimes \Delta \to B_j$ such that (omitting some evident reorderings of products)

$$f = \left(\imath \otimes d^{|Y|}\right) ; \left(\bigotimes_{j\in Y} f_j\right) ; \jmath^{-1}$$

We will insist that $\zeta_i^*\colon Z_{i-1} \otimes \llbracket \Delta_{i-1} \rrbracket \to Z_i$ is fibred with respect to the following decomposition of Z_{i-1} and Z_i, given for each i by Lemma 2.13:

$$Z_i \cong \bigotimes_{\substack{j\in J \\ 0\le k<n_j}} (\lambda_j)_{t_{j,k}} \tag{5.3}$$

where $t_{j,k} = t_i \upharpoonleft_{(X_j)_k}$ and $\llbracket \lambda_i \rrbracket = !\lambda_i$. If ζ_i is fibred with respect to the corresponding decomposition of Z_{i-1}^R and Z_i^B, then ζ_i^* will obviously be fibred with respect to Z_{i-1} and Z_i).

We can now present our main induction claim. As the preceding pages have shown, this property is surprisingly strong, and its formulation is not at all obvious. We shall postpone proof of this lemma until after we establish some other technical lemmas which are required for the proof.

Lemma 5.6 (Soundness)**.** *Suppose for $\Delta \vdash h, e$ and $\Delta' \vdash h', v$ we have in $\mathcal{L}_{\text{ret}}$*

$$h, e \Downarrow h', v$$

where (ordering by the left-pointing typing of Theorem 4.2)

$$\begin{aligned}\Delta &= l_1 \colon \lambda_1, \dots, l_n \colon \lambda_n \\ \Delta' &= \Xi_1, l_1 \colon \lambda_1, \dots, \Xi_n, l_n \colon \lambda_n, \Xi_{n+1}\end{aligned}$$

Let $\Delta_i = l_1 \colon \lambda_1, \dots, l_i \colon \lambda_i$ and $\Delta'_i = \Xi_1, l_1 \colon \lambda_1, \dots, \Xi_i, l_i \colon \lambda_i$, and let $h_i = h \restriction_{\Delta_i}$ and $h'_i = h' \restriction_{\Delta'_i}$. Then

1. (Termination property) $[\![h]\!]; [\![e]\!]$ *terminates.*

2. (Heap property) *Suppose for each $1 \le i \le n$, t_i denotes the resulting play in $[\![\Delta_i]\!]$ as in (5.1), and $\imath_i \colon Z_{i-1} \cong Z^L_{i-1} \otimes Z^R_{i-1}$ and $\jmath_i \colon Z^L_{i-1} \otimes Z^B_i \cong Z_i$ are as in (5.2). Then for each i there exists $\zeta_i \colon Z^R_{i-1} \otimes [\![\Delta'_i]\!] \to Z^B_i$ which is fibred as described in (5.3), such that if $\theta_i = \mathrm{zip}(\zeta^*_1, \dots, \zeta^*_i)$ with ζ^*_i defined relative to $\imath_i, \jmath_i$, then*

$$\begin{tikzcd}
{[\![\Delta_{i-1}]\!]_{t_{i-1}}} \arrow[r, "(H_i)^{t_{i-1}}_{t_i}"] & {[\![\Delta_i]\!]_{t_i}} \\
\cong & \cong \\
Z_{i-1} \otimes [\![\Delta_{i-1}]\!] & Z_i \otimes [\![\Delta_i]\!] \\
{[\![\Delta'_{i-1}]\!]} \arrow[u, "\theta^\pi_{i-1}"] \arrow[r, "H'_i"] & {[\![\Delta'_i]\!]} \arrow[u, "\theta^\pi_i"']
\end{tikzcd}$$

where

$$\begin{aligned}H_i &= [\![h_i]\!]^{\Delta_{i-1}}_{l_i \colon \lambda_i} &&\colon [\![\Delta_{i-1}]\!] \to [\![\Delta, \lambda_i]\!] \\ H'_i &= [\![h'_i]\!]^{\Delta'_{i-1}}_{\Xi_i, l_i \colon \lambda_i} &&\colon [\![\Delta'_{i-1}]\!] \to [\![\Delta, \Xi_i, l_i \colon \lambda_i]\!]\end{aligned}$$

3. (Expression property) *If $\theta_n = \mathrm{zip}(\zeta^*_1, \dots, \zeta^*_n)$ then*

$$\begin{tikzcd}
{[\![\Delta]\!]_{t_n}} \arrow[r, "{[\![e]\!]^{t_n}_{\Delta}}"] & {[\![\tau]\!]_\perp} \\
\cong & \\
Z_n \otimes [\![\Delta]\!] & \\
{[\![\Delta']\!]} \arrow[u, "\theta^\pi_n"'] \arrow[r, "{[\![h']\!]^{\Delta'}_{\Xi_{n+1}}}"'] & {[\![\Delta', \Xi_{n+1}]\!]} \arrow[uuu, "{[\![v]\!]_{\Delta', \Xi_{n+1}}}"']
\end{tikzcd}$$

Proposition 5.7. *Lemma 5.6 implies Theorem 5.1.*

Proof. The content of this is that the squares in (2) (for $1 \le i \le n$) and the square in (3) all compose, where the the left/top composite is $[\![h]\!]_\Delta ; [\![e]\!]_\Delta$, and the bottom/right composite is $[\![h']\!]_{\Delta'} ; [\![v]\!]_{\Delta'}$. Clearly the verticals agree. The leftmost vertical $[\![\Delta_0]\!] \xrightarrow{\theta_0^+} Z_0 \otimes [\![\Delta_0]\!] \cong [\![\Delta_0]\!]_{t_0}$ is just $1 \cong 1$, while the top composite was obtained earlier from resplitting and the definition of heaps from $[\![\Delta \vdash h]\!]_t ; [\![\Delta \vdash e]\!]^t = [\![h]\!]_\Delta ; [\![e]\!]_\Delta$ (where qta is the terminating play of $[\![h]\!] ; [\![e]\!]$). The bottom line is

$$[\![h_1']\!]_{\Xi_1, l_1 \colon \lambda_1} ; \ldots ; [\![h_n']\!]^{\Delta'_{n-1}}_{\Xi_n, l_n \colon \lambda_n} ; [\![v]\!]_{\Delta'}$$

By definition this gives the composite $[\![h']\!]_{\Delta'} ; [\![v]\!]_{\Delta'}$ as required. □

5.2 Various lemmas

Before proving the main soundness lemma which will give Theorem 5.1, we shall first give some useful subsidiary lemmas. The most substantial relates composition and syntactic substitution. Other lemmas show that values of reusable type are promoted morphisms and hence commute with contraction, and also show some facts about zippings (and fibred zippings).

It is not true in general that the obvious substitution property holds (cf. [72]), and here we identify the situation under which it does—only for values, and only because any reusable value being substituted into some expression must be interpreted by a promoted morphism.

Firstly, all values of reusable type are promoted morphisms.

Lemma 5.8 (Promoted values)**.** *When $\Delta \vdash v \colon \tau$ with $re(\Delta), re(\tau)$ there exists f such that*

$$[\![v]\!]_\Delta = \eta \circ f^\ddagger$$

Proof. By induction on the structure of v. Consider possible cases for v. If $v = l \colon \lambda \in \Delta$, then

$$\begin{aligned} [\![v]\!]_\Delta &= [\![\Delta_1, l, \Delta_2 \vdash l]\!] \\ &= [\![l \vdash l]\!] \circ (1_{\Delta_1} \otimes id_{[\![\lambda]\!]} \otimes 1_{\Delta_2}) \\ &= \eta \circ (1_{\Delta_1} \otimes \varepsilon_\lambda \otimes 1_{\Delta_2})^\ddagger \end{aligned}$$

If $v = \textbf{obj}\ \{m = v_m\}_{m \in X}$ then

$$\begin{aligned}
[\![v]\!]_\Delta &= \bot(pp^n) \circ \psi^n \circ \textstyle\bigotimes_{0<i<n}([\![v_i]\!]_\Delta^\ddagger) \circ d^n \\
&= \bot(pp^n) \circ \eta \circ \textstyle\bigotimes_{0<i<n} [v_i]^\ddagger \circ d^n \\
&= \eta \circ (\&_{0<i<n} [v_i]_\Delta)^\ddagger
\end{aligned}$$

If $v = \mathbf{Y}(v')$, and $\Delta \vdash v \colon \rho \to \rho$ where $\rho = \textbf{Obj}\ \{m \colon \sigma_m \to \tau_m\}_{m \in X}$, then by expansion of the definition of $\mathbf{Y}$ we note that

$$\begin{aligned}
[\![v]\!]_\Delta &= [\![\mathbf{Y}(v')]\!]_\Delta \\
&= [\![\textbf{obj}\ \{m = \lambda x.v\ (\mathbf{Y}v) \cdot m\ x\}_{m \in X}]\!]_\Delta
\end{aligned}$$

which has the required form by the reasoning above. Note that the case where $v = \mathbf{Y}(v') \cdot m$ does not arise, because this must be of some non-reusable type $\tau \multimap \tau'$.

If $v = \langle v_1, v_2 \rangle$ then since by the inductive hypothesis $[\![v_1]\!] = f^\ddagger$ and $[\![v_2]\!] = g^\ddagger$,

$$\begin{aligned}
[\![v]\!]_\Delta &= \psi \circ ([\![v_1]\!]_\Delta \otimes [\![v_2]\!]_\Delta) \circ d_\Delta \\
&= \eta \circ ([v_1]_\Delta \otimes [v_2]_\Delta) \circ d_\Delta \\
&= \eta \circ (f^\ddagger \otimes g^\ddagger) \circ d_\Delta \\
&= \eta \circ (f \& g)^\ddagger
\end{aligned}$$

□

Next, contraction can be performed before or after promotion.

Lemma 5.9 (Promotion-contraction)**.** *For $f \colon !X \to Y$,*

$$d_Y \circ f^\ddagger = (f^\ddagger \otimes f^\ddagger) \circ d_X$$

Proof. The following diagram commutes:

$$\begin{tikzcd}
!X \arrow[r, "\delta_X"] \arrow[d, "d_X"] & !!X \arrow[r, "!f"] \arrow[d, "d_{!X}"] & !Y \arrow[d, "d_Y"] \\
!X \otimes !X \arrow[r, "\delta_X \otimes \delta_X"'] & !!X \otimes !!X \arrow[r, "!f \otimes !f"'] & !Y \otimes !Y
\end{tikzcd}$$

the left square because δ is a morphism of comonoids, and the right square by naturality of d. □

We now combine these two lemmas to show that contraction can be performed before or after the denotation of a value of reusable type.

Corollary 5.10. *If $\Delta \vdash v \colon \tau$ with $re(\Delta), re(\tau)$ then*

$$d_\tau \circ [v]_\Delta = ([v]_\Delta \otimes [v]_\Delta) \circ d_\Delta$$

and hence

$$\bot(d_\tau) \circ [\![v]\!]_\Delta = \psi \circ ([\![v]\!]_\Delta \otimes [\![v]\!]_\Delta) \circ d_\Delta$$

Proof. By Lemma 5.8, there exists a morphism f with $[\![v]\!]_\Delta = \eta \circ f^\ddagger$, i.e. $[v]_\Delta = f^\ddagger$ thus by Lemma 5.9, $d_\tau \circ [v]_\Delta = ([v]_\Delta \otimes [v]_\Delta) \circ d_\Delta$, giving the first equality. Note that $\eta \circ d_\tau \circ [v]_\Delta = \bot(d_\tau) \circ \eta \circ [v]_\Delta = \bot(d_\tau) \circ [\![v]\!]_\Delta$, while similarly $\eta \circ ([v]_\Delta \otimes [v]_\Delta) \circ d_\Delta = \psi \circ ([\![v]\!]_\Delta \otimes [\![v]\!]_\Delta) \circ d_\Delta$, giving the second equality. □

The following lemma relates the syntactic substitution of values into expressions (which is just the usual capture-avoiding substitution) to composition. Note that there is a contraction on the right of the below equation, since on the left there may be variables used both in e and v, and hence a contraction in the semantics of the substituted term.

Both the restriction to a reusable context Δ and to a value v are necessary for the property to hold. Lemma 5.9 says that $d \circ f^\ddagger = (f^\ddagger \otimes f^\ddagger) \circ d$, but it is not true in general that $d \circ g = (g \otimes g) \circ d$, and it is certainly not true in general that expressions of reusable type are promoted morphisms (thanks to **constr**).

Lemma 5.11 (Substitution)**.** *If $\Delta \vdash v \colon \tau'$ and $\Delta, x \colon \tau' \vdash e \colon \tau$, and $re(\Delta)$ then*

$$[\![e[v/x]]\!]_\Delta = [\![e]\!]_{\Delta, x \colon \tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta$$

This means that, in briefer notation, $[\![e[v/x]]\!]_\Delta = [\![e]\!]_{\Delta,\tau'} \circ [v]^{\triangleright}_\Delta$, and also $[\![e[v/x]]\!]_\Delta = [\![e]\!]^\dagger_{\Delta,\tau'} \circ \psi \circ (\eta_\Delta \otimes [\![v]\!]_\Delta) \circ d_\Delta$.

Proof. By induction on the structure of e.

Case (constants)

$$\begin{aligned} [\![c_\varphi[v/x]]\!]_\Delta &= [\![c_\varphi]\!]_\Delta \\ &= [\![c_\varphi]\!]_{\Delta, x \colon \tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d \end{aligned}$$

Case (var 1)

$$\begin{aligned} [\![x[v/x]]\!]_\Delta &= [\![v]\!]_\Delta \\ &= (1_\Delta \otimes \eta_{\tau'}) \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta \\ &= [\![x]\!]_{\Delta, x} \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta \end{aligned}$$

Case (var 2)

For $y \neq x$,

$$\begin{aligned}
[\![y[v/x]]\!]_\Delta &= [\![y]\!]_\Delta \\
&= [\![y]\!]_\Delta \circ (id_\Delta \otimes 1_{\tau'}) \circ (id_\Delta \otimes [v]_\Delta) \circ d \\
&= [\![y]\!]_{\Delta, x:\,\tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d
\end{aligned}$$

Case (abs 1)

$$\begin{aligned}
[\![(\lambda x.\ e)[v/x]]\!]_\Delta &= [\![\lambda x.\ e]\!]_\Delta \\
&= [\![\lambda x.\ e]\!]_\Delta \circ (id_\Delta \otimes 1_{\tau'}) \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta \\
&= [\![\lambda x.\ e]\!]_{\Delta, x:\,\tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta
\end{aligned}$$

Case (abs 2)

$$\begin{aligned}
[\![(\lambda y.\ e)[v/x]]\!]_\Delta &= [\![\lambda y.\ (e[v/x])]\!]_\Delta \\
&= \lambda_Y([\![e[v/x]]\!]_\Delta) \\
&= \lambda_Y([\![e]\!]_{\Delta, x:\,\tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta) \\
&= \lambda_Y([\![e]\!]_{\Delta, x:\,\tau'}) \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta \\
&= [\![\lambda y.\ e]\!]_{\Delta, x:\,\tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta
\end{aligned}$$

Case (unpairing)

The case for **let** $\langle x, y \rangle$ **be** e **in** e' proceeds as for the two cases (abs 1), (abs 2).

Case (pair)

$$\begin{aligned}
[\![\langle e_1, e_2 \rangle\, [v/x]]\!]_\Delta &= [\![\langle e_1[v/x], e_2[v/x] \rangle]\!]_\Delta \\
&= \psi \circ ([\![e_1[v/x]]\!]_\Delta \otimes [\![e_2[v/x]]\!]_\Delta) \circ d_\Delta \\
&= \psi \circ \big(([\![e_1]\!]_{\Delta, x:\,\tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta) \otimes \\
&\qquad ([\![e_2]\!]_{\Delta, x:\,\tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta)\big) \\
&= \psi \circ ([\![e_1]\!]_{\Delta, x:\,\tau'} \otimes [\![e_2]\!]_{\Delta, x:\,\tau'}) \circ \\
&\qquad \big((id_\Delta \otimes [v]_\Delta) \otimes (id_\Delta \otimes [v]_\Delta)\big) \circ (d_\Delta \otimes d_\Delta) \circ d_\Delta
\end{aligned}$$

How does this relate to $[\![\Delta, x \vdash \langle e_1, e_2 \rangle]\!]$? We consider two possibilities for the derivation of $\Delta, x \vdash \langle e_1, e_2 \rangle$. Firstly, if x is of reusable type $re(\tau)$, we can assume by coherence that the last steps perform contraction for the whole of (Δ, x), i.e.

$$[\![\langle e_1, e_2 \rangle]\!]_{\Delta, x:\,\tau'} = \psi \circ ([\![e_1]\!]_{\Delta, x:\,\tau'} \otimes [\![e_2]\!]_{\Delta, x:\,\tau'}) \circ d_{\Delta, \tau}$$

Since we have shown in Corollary 5.10 that

$$((id_\Delta \otimes [v]_\Delta) \otimes (id_\Delta \otimes [v]_\Delta)) \circ (d_\Delta \otimes d_\Delta) = d_{\Delta,\tau'} \circ (id_\Delta \otimes [v]_\Delta)$$

we have

$$\begin{aligned} [\![\langle e_1, e_2\rangle\,[v/x]]\!]_\Delta &= \psi \circ ([\![e_1]\!]_{\Delta,x:\,\tau'} \otimes [\![e_2]\!]_{\Delta,x:\,\tau'}) \circ \\ &\qquad\quad \big((id_\Delta \otimes [v]_\Delta) \otimes (id_\Delta \otimes [v]_\Delta)\big) \circ (d_\Delta \otimes d_\Delta) \circ d_\Delta \\ &= [\![\langle e_1, e_2\rangle]\!]_{\Delta,x:\,\tau'} \circ (id_\Delta \circ [v]_\Delta) \circ d_\Delta \end{aligned}$$

Alternatively, if x is of non-reusable type, then either x appears in e_1, or in e_2 (or neither), and we shall assume by coherence that the last steps perform contraction of Δ. If x occurs in neither e_1 nor e_2 the options will agree. Consider the second case, the first being similar:

$$[\![\langle e_1, e_2\rangle]\!]_{\Delta,x:\,\tau'} = \psi \circ ([\![e_1]\!]_\Delta \otimes [\![e_2]\!]_{\Delta,x:\,\tau'}) \circ (d_\Delta \otimes id_{\tau'})$$

In the second case since $\Delta \vdash e_1$

$$[\![e_1]\!]_\Delta = [\![e_1]\!]_{\Delta,x:\,\tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta$$

and so

$$\begin{aligned} [\![\langle e_1, e_2\rangle\,[v/x]]\!]_\Delta &= \psi \circ ([\![e_1]\!]_{\Delta,x:\,\tau'} \otimes [\![e_2]\!]_{\Delta,x:\,\tau'}) \circ \\ &\qquad\quad \big((id_\Delta \otimes [v]_\Delta) \otimes (id_\Delta \otimes [v]_\Delta)\big) \circ (d_\Delta \otimes d_\Delta) \circ d_\Delta \\ &= \psi \circ [\![e_1]\!]_\Delta \otimes \big([\![e_2]\!]_{\Delta,x:\,\tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta\big) \circ d_\Delta \\ &= \psi \circ [\![e_1]\!]_\Delta \otimes \big([\![e_2]\!]_{\Delta,x:\,\tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta\big) \circ d_\Delta \\ &= [\![\langle e_1, e_2\rangle]\!]_{\Delta,x:\,\tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta \end{aligned}$$

Case (app)

$$\begin{aligned} [\![(e_1\ e_2)[v/x]]\!]_\Delta &= [\![e_1[v/x]\ e_2[v/x]]\!]_\Delta \\ &= eval^\dagger \circ \psi \circ ([\![e_1[v/x]]\!]_\Delta \otimes [\![e_2[v/x]]\!]_\Delta \circ d_\Delta \end{aligned}$$

Since this is simply $eval^\dagger$ composed with what we had before, the reasoning in the pairing case gives

$$[\![(e_1\ e_2)[v/x]]\!]_\Delta = [\![e_1\ e_2]\!]_{\Delta,x:\,\tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta$$

Case (constr)

$$\begin{array}{rcl}
[\![(\mathbf{constr}\ e_1\ e_2)[v/x]]\!]_\Delta & = & [\![\mathbf{constr}\ e_1[v/x]\ e_2[v/x]]\!]_\Delta \\
& = & \bot(\mathit{thread}) \circ \psi \circ ([\![e_1[v/x]]\!]_\Delta \otimes [\![e_2[v/x]]\!]_\Delta \circ d_\Delta
\end{array}$$

Since this is simply $\mathit{thread}^\dagger$ composed with what we had before, the reasoning in the pairing case gives

$$[\![(\mathbf{constr}\ e_1\ e_2)[v/x]]\!]_\Delta = [\![\mathbf{constr}\ e_1\ e_2]\!]_{\Delta,x\colon\tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta$$

Case (ifz)

Again, this case proceeds as for pairing.

Case (object)

In this case we use the value restriction on **Obj** , to ensure that the components of the object are promoted morphisms.

$$\begin{array}{rl}
& [\![\mathbf{obj}\ \{m_i = v_i\}_{i\in X}[v/x]]\!]_\Delta \\
= & [\![\mathbf{obj}\ \{m_i = v_i[v/x]\}_{i\in X}]\!]_\Delta \\
= & \bot(pp^n) \circ \psi^n \circ [\![v_i[v/x]]\!]^\dagger_\Delta d \\
= & \bot(pp^n) \circ \psi^n \circ ([\![v_i]\!]_{\Delta,x\colon\tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta)^\dagger \\
= & \bot(pp^n) \circ \psi^n \circ ([\![v_i]\!]_{\Delta,x\colon\tau'})^\dagger \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta \\
= & [\![\mathbf{obj}\ \{m_i = v_i\}_{i\in X}]\!]_{\Delta,x\colon\tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d_\Delta
\end{array}$$

since by application of the comonad laws (which apply because $[v_i]$ is a promoted morphism):

$$(f \circ (id_{!X} \otimes g^\ddagger) \circ d_X)^\ddagger = f^\ddagger \circ (id_{!X} \otimes g^\ddagger) \circ d_X$$

Case (select)

$$\begin{array}{rcl}
[\![e \cdot m[v/x]]\!]_\Delta & = & [\![e[v/x] \cdot m]\!]_\Delta \\
& = & \Pi_m \circ \varepsilon \circ [\![e[v/x]]\!]_\Delta \\
& = & \Pi_m \circ \varepsilon \circ [\![e]\!]_{\Delta,x\colon\tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d \\
& = & [e \cdot m]_{\Delta,x\colon\tau'} \circ (id_\Delta \otimes [v]_\Delta) \circ d
\end{array}$$

□

Lemma 5.12 (Strictness)**.** *Suppose $h\colon 1 \to \Delta$, $f\colon \Delta \to X_\bot$, $g\colon X_\bot \to Y_\bot$. If g is* strict *(as in Section 2.1.7) and*

$$\begin{array}{rcl}
q_1 s_1 a_1 & \in & h \| f \\
q_2 s_2 a_2 & \in & h_{s_1} \| (f^{s_1}; g)
\end{array}$$

then

$$\begin{array}{rcl} q_2s_1s_2a_2 & \in & h\|(f;g) \\ (f;g)^{s_1s_2} & = & (f^{s_1};g)^{s_2} \end{array}$$

Proof. Clearly if $s_2 \in h_{s_1}$ then $s_1s_2 \in h$. By definition if g is strict it must respond to q_2 with q_1, which is relayed to f. So we know by the first requirement that $q_2q_1s_1a_1 \in h\|(f\|g)$, and hence $q_2s_1 \in h\|(f;g) = (h\|f);g$. Since $q_2s_2a_2 \in (h_{s_1}\|f^{s_1});g$, $q_2s_1s_2a_2 \in (h_{s_1}\|f^{s_1});g = h_{s_1}\|(f^{s_1});g$. □

Lemma 5.13 (Fibred zippings).

(i) If ζ is fibred with respect to $\imath, \jmath$, and ζ^* is defined relative to $\imath', \jmath'$ then ζ^* is fibred with respect to $\imath';(id \otimes \imath)$, $\jmath'^{-1};(id \otimes \jmath)$.

(ii) If for $0 < i \leq n$ there is a decomposition $\imath_i \colon Z_i \cong \bigotimes_{j \in X_i} Z_{i,j}$, and each ζ_i is fibred with respect to $\imath_i$, $\imath_{i+1}$ then the zipping $\theta_n \colon !\Delta_n \to Z_n$ of $\zeta_1, \ldots \zeta_n$ is a fibred morphism with respect to the trivial decomposition $\jmath \colon 1 \cong 1$ and $\imath_n$

In essence, the above lemma simply states that fibred morphisms compose appropriately, where we may bundle together a collection of fibres if we wish, but not split fibres.

Proof. (i). Easy to verify by expansion of the definition. (ii). By induction on n. The trivial zipping $\theta_0 \colon 1 \to 1$ is trivially fibred. If θ_i is fibred with respect to $\jmath$ and $\imath_i$, there must be for $j \in X_i$ morphisms $\vartheta_{i,j} \colon 1 \otimes !\Delta_i \to Z_{i,j}$, such that

$$\theta_i = (\jmath \otimes d^{|X_i|}); \left(\bigotimes_{j \in X_i} \vartheta_{i,j} \right); \imath_i$$

If ζ_{i+1} is fibred with respect to $\imath_i$ and $\imath_{i+1}$ then there must be disjoint sets $X_{i,k}$ with $X_i = \bigsqcup_{k \in X_{i+1}} X_{i,k}$ and for $k \in X_{i+1}$ morphisms

$$\zeta_{i+1,k} \colon \left(\bigotimes_{w \in X_{i+1,k}} Z_{i,w} \right) \otimes !\Delta \to Z_{i+1,k}$$

such that

$$\zeta_{i+1} = (\imath_i \otimes d^{|X_{i+1}|}); \left(\bigotimes_{k \in X_{i+1}} \zeta_{i+1,k} \right); (\imath_{i+1})^{-1}$$

By definition $\mathrm{zip}(\theta_i, \zeta_{i+1}) = \Pi_\Delta; \theta_i^+; \zeta_{i+1} \colon 1 \otimes !\Delta_{i+1} \to Z_{i+1}$. Construct for $k \in X_{i+1}$ morphisms $\vartheta_{i+1,k} \colon 1 \otimes 1_{\Delta_{i+1}} \to Z_{i+1,k}$ such that

$$\theta_{i+1} = (\jmath \otimes d^{|X_{i+1}|}); \left(\bigotimes_{k \in X_{i+1}} \vartheta_{i+1,k} \right); \imath_{i+1}$$

as

$$\vartheta_{i+1,k} = \Pi_\Delta; \left(\bigotimes_{w \in X_{i+1,k}} \vartheta_{i,w} \right)^+ ; \ \zeta_{i+1,k}$$

□

Lemma 5.14 (Zip-pair). *For reusable objects Δ, Ξ_1, and Ξ_2, and morphisms $\zeta_1 \colon Z_1 \otimes \Delta \otimes \Xi_1 \to Z_1'$, $\zeta_2 \colon Z_2 \otimes \Delta \otimes \Xi_2 \to Z_2'$ equipped with isomorphisms*

$$\begin{aligned} \imath_1 \colon \ & Z_A \cong (W \otimes Z_2) \otimes Z_1 \\ & Z_B \cong (W \otimes Z_2) \otimes Z_1' \colon \jmath_1 \\ \imath_2 \colon \ & Z_B \cong (W \otimes Z_1') \otimes Z_2 \\ & Z_C \cong (W \otimes Z_1') \otimes Z_2' \colon \jmath_2 \end{aligned}$$

giving

$$\zeta_1^* \colon Z_A \otimes \Delta \otimes \Xi_1 \to Z_B \quad \zeta_2^* \colon Z_B \otimes \Delta \otimes \Xi_2 \to Z_C$$

there exists a morphism

$$(\!|\zeta_1, \zeta_2|\!) \colon (Z_1 \otimes Z_2) \otimes \Delta \otimes \Xi_1 \otimes \Xi_2 \to (Z_1' \otimes Z_2')$$

equipped with isomorphisms

$$\begin{aligned} \imath \colon \ & Z_A \cong W \otimes (Z_1 \otimes Z_2) \\ & Z_B \cong W \otimes (Z_1' \otimes Z_2') \colon \jmath \end{aligned}$$

giving

$$(\!|\zeta_1, \zeta_2|\!)^* \colon Z_A \otimes \Delta \otimes \Xi_1 \otimes \Xi_2 \to Z_C$$

such that:

$$\mathrm{zip}(\vec{\zeta}, \zeta_1^*, (\zeta_2 \circ \Pi_{\Delta,\Xi_2})^*) = \mathrm{zip}(\vec{\zeta}, (\!|\zeta_1, \zeta_2|\!)^*)$$

where $\Pi_{\Delta,\Xi_2} \colon Z_2 \otimes \Delta \otimes \Xi_1 \otimes \lambda_1 \otimes \Xi_2 \to Z_2 \otimes \Delta \otimes \Xi_2$ is the evident projection. Furthermore, if ζ_1, ζ_2 are fibred with respect to decompositions $\imath_1' \colon Z_1 \cong \bigotimes_{j \in X_1} Z_j$, $\imath_2' \colon Z_1' \cong \bigotimes_{i \in Y_1} Z_i$, $\imath_3' \colon Z_2 \cong \bigotimes_{j \in X_2} Z_j'$, $\imath_4' \colon Z_2' \cong \bigotimes_{i \in Y_2} Z_i'$ then $(\!|\zeta_1, \zeta_2|\!)$ is fibred with respect to the decompositions $\imath_1' \otimes \imath_3'$ and $\imath_2' \otimes \imath_4'$.

Proof. Take

$$(\!|\zeta_1, \zeta_2|\!) = (id_{Z_1,Z_2} \otimes d_\Delta); (id_{Z_1} \otimes \gamma_{Z_2,\Delta} \otimes id_\Delta); (\zeta_1 \otimes \zeta_2)$$

Looking at the types above, the isomorphisms $\imath, \jmath$ are self-evident. Calculation from the definition of zip$(-)$ gives the desired equality, since the types of the isomorphisms $\imath_1, \jmath_1, \imath_2, \jmath_2$ ensure that ζ_2^* does not depend on the result of ζ_1, only the identity component of ζ_1^*. The definition is also easily seen to be fibred. □

Corollary 5.15 (Zip-swap). *For reusable objects Δ, Ξ_1, Ξ_2, and morphisms $\zeta_1 \colon Z_1 \otimes \Delta \otimes \Xi_1 \to Z_2$, $\zeta_2 \colon Z_3 \otimes \Delta \otimes \Xi_2 \to Z_4$ equipped with isomorphisms $\imath_1, \jmath_1, \imath_2, \jmath_2$ as in the previous Lemma, there exist suitable $\imath_1', \jmath_1', \imath_2', \jmath_2'$ such that*

$$\text{zip}(\vec{\zeta}, \zeta_1^*, (\zeta_2 \circ \Pi_{\Delta,\Xi_2})^*) = \text{zip}(\vec{\zeta}, \zeta_2^*, (\zeta_1 \circ \Pi_{\Delta,\Xi_1})^*)$$

Proof. From the previous lemma, note that

$$\text{zip}(\vec{\zeta}, \zeta_1^*, (\zeta_2 \circ \Pi_{\Delta,\Xi_1})^*) = \text{zip}(\vec{\zeta}, (\!|\zeta_1, \zeta_2|\!)^*) = \text{zip}(\vec{\zeta}, \zeta_2^*, (\zeta_1 \circ \Pi_{\Delta,\Xi_2})^*)$$

since $(\!|\zeta_1, \zeta_2|\!) \cong (\!|\zeta_2, \zeta_1|\!)$ modulo some twist maps. □

Lemma 5.16 (Memoization of (fibred) zippings). *Let $\theta_n = \text{zip}(\zeta_1^*, \dots, \zeta_n^*)$ and $\theta_n' = \text{zip}(\zeta_1'^*, \dots, \zeta_n'^*)$ for appropriately fibred morphisms ζ_i, ζ_i', such that*

$$\Delta_n'' \xrightarrow{\theta_n'^\pi} (\Delta_n')_s \qquad \Delta_n' \xrightarrow{\theta_n^\pi} (\Delta_n)_u$$

Assume furthermore that there exists a play t such that there is a memoization $(\theta_n^\pi)_t^s$. Then there exist fibred morphisms $\zeta_1'', \dots \zeta_n''$ such that if $\theta_n'' = \text{zip}(\zeta_1''^*, \dots, \zeta_n''^*)$ with

$$\Delta_n'' \xrightarrow{\theta_n''^\pi} (\Delta_n)_{ut}$$

then $\theta_n''^\pi = \theta_n'^\pi; (\theta_n^\pi)_t^s$.

Proof (sketch). By induction on n. Trivial base case. For the inductive step, assume the property holds at $n-1$. Note that we can expand

$$\theta_n^\pi = d_{\Delta_n'}; (id_{\Delta_n'} \otimes (\Pi_{\Delta_{n-1}'}; \theta_{n-1}^\pi)); \iota; (\zeta_n^* \otimes id_{\lambda_n})$$

where ι is the isomorphsim $\Delta_n' \otimes Z_{n-1} \otimes \Delta_{n-1} \cong Z_{n-1} \otimes \Delta_{n-1}' \otimes \Xi_n \otimes \Delta_{n-1} \otimes \lambda_n$. We similarly expand θ_n'.

By manipualting $(\theta_n^\pi)_t^s$ we push the memoization into subexpressions, including two of the form $(\theta_{n-1}^\pi)_{s_2}^{s_2}$ and $(\zeta_n^*)_{t_1}^{t_1'}$. We can subsequently move the former subexpression past the $(\zeta_n')^*$ term, since the play t_1' must only occur in the identity-part of the $-^*$ construction, arriving a the composition $\theta'^{\pi}_{n-1}; (\theta^{\pi}_{n-1})^{s'}_{t'}$. By the inductive hypothesis this is equal to some fibred zipping θ''^{π}_{n-1}.

Further structural manipulation moves the term $(\zeta^*_n)^{t'_1}_{t_1}$ beside ζ'^*_n, resulting in the composition $(\zeta'_n); \jmath; (id \otimes (\zeta_n)^{s''}_{t''})$ for a suitable isomorphism $\jmath$. We thus take $\zeta''_n = \zeta'^{\pi}_n; \jmath; (id \otimes (\zeta_n)^{s''}_{t''})$.

Finally, we note that if we take $\theta''_n = \mathrm{zip}(\theta''_{n-1}, \zeta''_n)$, then the expression is in the form of $(\theta''_n)^{\pi}$ expanded as described above. □

5.3 Main induction

We shall prove the Lemma by induction on operational semantics derivations. Before we do so, we perform a little "preprocessing" in order to simplify the proof. Firstly, given an expression e, and a second expression e' such that

$$h, e \Downarrow h', v \iff h, e' \Downarrow h', v \quad \text{and} \quad [\![e]\!] = [\![e']\!]$$

then proof of the lemma for e' also provides proof for e. To this end we make the following simplifications wherever they apply in the operational derivation, for fresh variables x_1, x_2:

$$\begin{array}{rcl} e_1\ e_2 & \rightsquigarrow & \mathbf{let}\ \langle x_1, x_2\rangle\ \mathbf{be}\ \langle e_1, e_2\rangle\ \mathbf{in}\ (x_1\ x_2) \\ \mathbf{constr}\ e_1\ e_2 & \rightsquigarrow & \mathbf{let}\ \langle x_1, x_2\rangle\ \mathbf{be}\ \langle e_1, e_2\rangle\ \mathbf{in}\ (\mathbf{constr}\ x_1\ x_2) \\ \mathbf{ifz}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 & \rightsquigarrow & \mathbf{let}\ \langle x_1, x_2\rangle\ \mathbf{be}\ \langle e, 1\rangle\ \mathbf{in}\ \big(\mathbf{ifz}\ x_1\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\big) \end{array}$$

It is clear that operationally these expansions give the same results, as in all four application rules and in the **constr** rule e_1 and e_2 are evaluated in turn, as in the pairing rule for $\langle e_1, e_2\rangle$. Therefore we get essentially the same derivation.

Now note that

$$[\![x_1, x_2 \vdash x_1\ x_2]\!] = \mathit{eval}^{\dagger} \circ \psi \circ \eta \otimes \eta = \mathit{eval}$$

and hence

$$\begin{array}{rcl} [\![\Delta \vdash \mathbf{let}\ \langle x_1, x_2\rangle\ \mathbf{be}\ \langle e_1, e_2\rangle\ \mathbf{in}\ x_1\ x_2]\!] & = & \mathit{eval}^{\dagger} \circ [\![\langle e_1, e_2\rangle]\!] \\ & = & [\![\Delta \vdash e_1\ e_2]\!] \end{array}$$

Similarly

$$[\![x_1, x_2 \vdash \mathbf{constr}\ x_1\ x_2]\!] = \bot(\mathit{thread}) \circ \psi \circ \eta \otimes \eta = \eta \circ \mathit{thread}$$

and hence

$$\begin{aligned}\llbracket \Delta \vdash \mathbf{let}\ \langle x_1, x_2\rangle\ \mathbf{be}\ \langle e_1, e_2\rangle\ \mathbf{in}\ \mathbf{constr}\ x_1\ x_2 \rrbracket &= \bot(\mathit{thread}) \circ \llbracket \langle e_1, e_2\rangle \rrbracket \\ &= \llbracket \Delta \vdash \mathbf{constr}\ e_1\ e_2 \rrbracket\end{aligned}$$

Again, it is easy to check that

$$\begin{aligned}\llbracket \mathbf{ifz}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \rrbracket_\Delta &= (\llbracket \mathbf{ifz}\ x_1\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \rrbracket_{x_1 :\, \iota, x_2 :\, \iota, \Delta})^\dagger \circ \llbracket \langle e, 1\rangle \rrbracket_\Delta \\ &= \Big\llbracket \mathbf{let}\ \langle x_1, x_2\rangle\ \mathbf{be}\ \langle e, 1\rangle\ \mathbf{in}\ \big(\mathbf{ifz}\ x_1\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\big) \Big\rrbracket\end{aligned}$$

These substitutions therefore allow us to consider the relevant rules as applying (in our derivation) only to values. We make the obvious simplifications to the rules given in the previous chapter under this restriction—we shall present the appropriate version of each rule as we consider each case in turn.

We now prove Lemma 5.6 by induction on the operational derivation of $h, e \Downarrow h', v$. The most interesting cases are those for pairing and unpairing, and of course the method invocation rule. After our preprocessing stage, the cases for the pairing and unpairing rules handle all the reasoning about the heap which would otherwise be required for most rules. The remaining rules then generally have zero or one premises, and so at least the heap property is trivial or follows immediately from the inductive hypothesis. Of course the method invocation rule genuinely involves detailed reasoning about the heap.

Case (Values)

Assume $\Delta \vdash h, v$. The value rule is:

$$\frac{}{h, v \Downarrow h, v}$$

Since $\llbracket v \rrbracket_\Delta = \eta \circ f$, the composition $\llbracket h \rrbracket_\Delta ; \llbracket v \rrbracket_\Delta$ terminates with some play qa, and the play at each $\llbracket \Delta_i \rrbracket$ is ε. Then since $\llbracket \Delta_i \rrbracket_\varepsilon \cong 1 \otimes \llbracket \Delta_i \rrbracket$ we take

$$\zeta_i = 1_{1 \otimes \Delta_i} : 1 \otimes \llbracket \Delta_{i-1} \rrbracket \to 1$$

and note that

$$\theta_i = \mathrm{zip}(\zeta_1, \ldots, \zeta_i) = 1_{\Delta_i}$$

and since the heap is unchanged,

$$H_i' = \llbracket h_i \rrbracket^{\Delta_{i-1}}_{\Xi_i, l_i :\, \lambda_i} = \llbracket h_i \rrbracket^{\Delta_{i-1}}_{l_i :\, \lambda_i} = H_i$$

Therefore we have the heap property, since

$$\theta^{\pi}_{i-1}; (H_i)^{\varepsilon}_{\varepsilon} = H_i = H'_i; \theta^{\pi}_i$$

For the expression property,

$$[\![v]\!]_{\Delta} \circ \theta^{\pi}_n = [\![v]\!]_{\Delta} = [\![v]\!]_{\Delta} \circ [\![h]\!]^{\Delta}_{\Xi_{n+1}}$$

since $[\![h]\!]^{\Delta}_{\Xi_{n+1}} = id_{\Delta}$.

Case (Record selection)

Assume $\Delta \vdash h, e \cdot m_i$ and $\Delta' \vdash h', v_i$. The record selection rule is:

$$\frac{h, e \Downarrow h', \mathbf{obj}\ \{m_1 = v_1, \ldots, m_n = v_n\}}{h, e \cdot m_i \Downarrow h', v_i}\ 1 \leq i \leq n$$

By definition $[\![e \cdot m_i]\!]_{\Delta} = [\![e]\!]_{\Delta}; \bot(\varepsilon; \Pi_{m_i})$. Since $[\![h]\!]; [\![e]\!]_{\Delta}$ terminates and $\bot(-)$ is strict, $[\![h]\!]_{\Delta}; [\![e \cdot m_i]\!]_{\Delta}$ terminates. Since play in the heap is identical in both cases, the heap property follows trivially with the same choice of ζ_i. The expression property from the IH is

$$\theta^{\lhd}_n; [\![e]\!]^{t_n}_{\Delta} = [\![h']\!]^{\Delta_n}_{\Xi_{n+1}}; [\![\mathbf{obj}\ \{m_j = v_j\}_{j \in X}]\!]_{\Delta}$$

Expanding the definition of $[\![\mathbf{obj}\ -]\!]$, and in particular some calculation with the promotion operator shows that

$$[\![v_i]\!]_{\Delta} = [\![\mathbf{obj}\ \{m_j = v_j\}_{j \in X}]\!]_{\Delta}; \bot(\varepsilon; \Pi_i)$$

which together with the above gives the expression property:

$$\theta^{\lhd}_n; ([\![e \cdot m_i]\!]_{\Delta})^{t_n} = [\![h']\!]^{\Delta_n}_{\Xi_{n+1}}; [\![v_i]\!]_{\Delta}$$

Case (Method selection)

The method selection rule (for heap objects) is:

$$\frac{h, e \Downarrow h', l}{h, e \cdot m \Downarrow h', l \cdot m}$$

Since $[\![h]\!]; [\![e]\!]$ terminates and $\bot(\Pi_m)$ is strict, $[\![h]\!]; [\![e \cdot m]\!] = [\![h]\!]; [\![e]\!]; \bot(\Pi_m)$ terminates. Since play in the heap is identical in both cases, the heap property again carries over, and since $\theta^{\lhd}_n; [\![e]\!]^{t_n} = [\![h']\!]^{\Delta}_{\Xi}; [\![l]\!]$ then $\theta^{\lhd}; [\![e \cdot m]\!]^{t_n} = [\![h']\!]^{\Delta}_{\Xi}; [\![l \cdot m]\!]$.

The case for the similar method rule for $\mathbf{Y}$, which is:

$$\frac{h, e \Downarrow h', \mathbf{Y}(v)}{h, e \cdot m \Downarrow h', \mathbf{Y}(v) \cdot m}$$

holds by the same reasoning.

Case (Ifz)

The reduced **ifz** rules are:

$$\frac{h, e_1 \Downarrow h', v}{h, \textbf{ifz}\ 0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \Downarrow h', v} \qquad \frac{h, e_2 \Downarrow h', v}{h, \textbf{ifz}\ n\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \Downarrow h', v}\ n \neq 0$$

In the first case, just from the definition of $[\![-]\!]$ and **ifz**

$$\begin{aligned} [\![\textbf{ifz}\ 0\ \textbf{then}\ e_1\ \textbf{else}\ e_2]\!] &= \mathit{ifz} \circ ([\![0]\!] \otimes ([\![e_1]\!] \& [\![e_2]\!])) \\ &= [\![e_1]\!] \end{aligned}$$

while in the second case

$$\begin{aligned} [\![\textbf{ifz}\ n\ \textbf{then}\ e_1\ \textbf{else}\ e_2]\!] &= \mathit{ifz} \circ ([\![n]\!] \otimes ([\![e_1]\!] \& [\![e_2]\!])) \\ &= [\![e_2]\!] \end{aligned}$$

Therefore in both cases, the termination, heap and expression properties hold unchanged from the inductive hypothesis.

Case (Application)

Assume $\Delta \vdash (\lambda x.e)\ v'$. The reduced application rule is:

$$\frac{h, e[v/x] \Downarrow h', v}{h, (\lambda x.e)\ v' \Downarrow h', v}$$

By Lemma 5.11,

$$\begin{aligned} [\![e[v'/x]]\!]_\Delta &= d_\Delta; (id_\Delta \otimes [\![v']\!]_\Delta); [\![e]\!]_{\Delta,\tau'} \\ &= d_\Delta; [\lambda([\![e]\!]_{\Delta,\tau'}) \otimes [\![v']\!]_\Delta)]; \mathit{eval} \\ &= [\![(\lambda x.e)\ v']\!]_\Delta \end{aligned}$$

By the inductive hypothesis, $[\![h]\!]; [\![e[v/x]]\!]$ terminates, so $[\![h]\!]; [\![(\lambda x.e)\ v']\!]$ terminates. Similarly, the expression and heap properties hold unchanged from the I.H. by the above equality.

Case (Y)

The reduced **Y** application rule is:

$$\frac{(v_1\ (\mathbf{Y} v_1))\ v_2 \Downarrow v}{(\mathbf{Y} v_1)\ v_2 \Downarrow v}$$

Calculate from the definition of $[\![\mathbf{Y} v_1]\!]$:

$$\begin{array}{rl} & \bot(Y \circ !(id \multimap \mu_{\multimap})) \circ [\![v_1]\!]^{\sharp}_{\Delta} \\ = & \bot(Y \circ !(id \multimap \mu_{\multimap})) \circ \eta \circ [v_1]^{\ddagger}_{\Delta} \\ = & \eta \circ Y \circ ((id \multimap \mu_{\multimap}) \circ [v_1]_{\Delta})^{\ddagger} \\ = & \eta \circ \mathit{eval} \circ ((id \multimap \mu_{\multimap}) \circ [v_1]_{\Delta}) \otimes Y \circ ((id \multimap \mu_{\multimap}) \circ [v_1]_{\Delta})^{\ddagger} \\ = & \eta \circ \mu_{\multimap} \circ \mathit{eval} \circ [v_1]_{\Delta} \otimes Y \circ ((id \multimap \mu_{\multimap}) \circ [v_1]_{\Delta})^{\ddagger} \end{array}$$

Then

$$\begin{array}{rl} & [\![(\mathbf{Y} v_1)\ v_2]\!] = \\ = & \mathit{eval}^{\dagger} \circ \psi \circ ([\![\mathbf{Y} v_1]\!] \otimes [\![v_2]\!]) \\ = & \mathit{eval}^{\dagger} \circ \psi \circ \big(\eta \circ \mu_{\multimap} \circ \mathit{eval} \circ [v_1]_{\Delta} \otimes Y \circ ((id \multimap \mu_{\multimap}) \circ [v_1]_{\Delta})^{\ddagger}\big) \otimes [\![v_2]\!] \\ & \qquad \text{(as } \mathit{eval} \circ (\mu_{\multimap} \circ f \otimes g) = \mathit{eval}^{\dagger} \circ \psi \circ (f \otimes \eta \circ g)) \\ = & \mathit{eval}^{\dagger} \circ \psi \circ \big(\mathit{eval} \circ [v_1]_{\Delta} \otimes Y \circ ((id \multimap \mu_{\multimap}) \circ [v_1]_{\Delta})^{\ddagger}\big) \otimes [\![v_2]\!] \\ = & \mathit{eval}^{\dagger} \circ \psi \circ \big(\mathit{eval}^{\dagger} \circ \psi \circ ([\![v_1]\!]_{\Delta} \otimes \bot(Y \circ !(id \multimap \mu_{\multimap})) \circ [\![v_1]\!]^{\sharp}_{\Delta}\big) \otimes [\![v_2]\!] \\ = & [\![(v_1\ (\mathbf{Y} v_1))\ v_2]\!] \end{array}$$

Thus the termination, heap and expression properties all hold unchanged from the premise.

Case (Constr)

The reduced **constr** rule is:

$$\frac{}{h, \mathbf{constr}\ v_s, v_c \Downarrow h[l \mapsto \langle v_s, v_c\rangle], l}$$

The composition $[\![h]\!]; [\![\mathbf{constr}\ v_s\ v_c]\!]$ terminates with $qa \in [\![h]\!] \| [\![\mathbf{constr}\ s_1\ v_c]\!]$, since:

$$\begin{array}{rcl} [\![\mathbf{constr}\ v_s\ v_c]\!]_{\Gamma,\Delta} & = & ([\![v_s]\!] \otimes [\![v_c]\!]); \psi; \bot(\mathit{thread}) \\ & = & ([v_s] \otimes [v_c]); \mathit{thread}; \eta \end{array}$$

As l is a new location, we again take each $\zeta_i = 1_{\Delta_i}$, noting that this makes $\theta_i = 1_{\Delta_i}$. The heap property is thus trivial, and for the expression property we have:

$$\begin{array}{rcl} \theta_n^+; [\![\mathbf{constr}\ v_c\ s_1]\!] & = & [\![\mathbf{constr}\ v_c\ s_1]\!] \\ & = & [\![\mathbf{constr}\ v_c\ s_1]\!]^{\triangleright}; [\![\Delta, l \vdash l]\!] \\ & = & [\![h[l \mapsto \langle s_1, v_c\rangle]]\!]^{\Delta}_{l:\,\lambda}; [\![\Delta, l \vdash l]\!] \end{array}$$

Case (Pair)

Assume $\Delta \vdash h, e_1$, $\Delta' \vdash h', v_1$, $\Delta' \vdash h', e_2$, $\Delta'' \vdash h'', v_2$. The pair rule is:

$$\frac{h, e_1 \Downarrow h', v_1 \quad h', e_2 \Downarrow h'', v_2}{h, \langle e_1, e_2 \rangle \Downarrow h'', \langle v_1, v_2 \rangle}$$

As $[\![h]\!]; [\![e_1]\!]_\Delta$ and $[\![h']\!]; [\![e_2]\!]_\Delta$ terminate, we have $qta^{v_1} \in [\![h]\!] \| [\![e_1]\!]_\Delta$ and $qua^{v_2} \in [\![h']\!] \| [\![e_2]\!]_{\Delta'}$, and for e_1 we also have $\zeta_1 \ldots \zeta_n$ with zipping θ_n (for e_2 see below). From the heap property for e_1 we have $[\![h' \upharpoonleft_{\Delta'}]\!]_{\Delta'}; \theta_n^\pi = [\![h]\!]^t$. So by definition of ψ and d we have $qt\overline{u}a^{(v_1,v_2)} \in [\![h]\!] \| d; ([\![e_1]\!] \otimes [\![e_2]\!]); \psi$, that is $[\![h]\!]; [\![\langle e_1, e_2 \rangle]\!]$ terminates. Here u is as $\overline{u}$ but with component renaming via the contraction (since e_2 must now start wherever e_1 finishes). This establishes the termination property.

Since we have two "levels" of newly created heap locations, we fix some notation. We decompose Δ, Δ' as follows:

$$\begin{aligned}
\Delta' &= \Xi_1, \lambda_1, \ldots \Xi_n, \lambda_n, \Xi_{n+1} \\
\Xi_i &= \lambda_{i,1}, \ldots \lambda_{i,m_i} \\
\Delta'' &= \ldots \Xi'_{i,1}, \lambda_{i,1}, \ldots \Xi'_{i,m_i}, \lambda_{i,m_i}, \Xi'_i, \lambda_i \ldots \\
\Xi''_i &= \Xi'_{i,1}, \lambda_{i,1}, \ldots, \Xi'_{i,m_i}, \lambda_{i,m_i}
\end{aligned}$$

We will then for e_2 refer to the corresponding $\zeta'_{1,1}, \ldots, \zeta'_{1,m_1}, \zeta'_1, \ldots, \zeta'_{i,m_i}, \zeta'_i$ with zipping θ'_i, so that $\theta'_i \colon \Delta''_i \to Z_{u_i}$.

For each λ_i, there is one instance of the heap property for e_1, while e_2 gives a corresponding collection of instances at each of $\lambda_{i,1}, \ldots, \lambda_{i,m_i}, \lambda_i$. We therefore create one instance from this latter collection before pasting the two together.

The diagrams for e_2 compose, giving

$$\begin{aligned}
& \theta'^{\pi}_{i-1}; ([\![h']\!]^{\Delta'_{i-1}}_{\Xi_i, \lambda_{i,1}})^{u_{i-1}}_{u_i} \\
= \; & [\![h'']\!]^{\Delta''_{i-1}}_{\Xi'_{i,1}, \lambda_{i,1}}; \ldots; [\![h'']\!]^{\Delta''_{i-1}, \Xi_i}_{\Xi'_i, \lambda_i}; \theta'^{\pi}_i \\
= \; & [\![h'']\!]^{\Delta''_{i-1}}_{\Xi''_i, \Xi'_i, \lambda_i}; \theta'^{\pi}_i
\end{aligned}$$

To complete the construction of the "glued together" heap property, we construct ζ''_i to make $\text{zip}(\theta'_{i-1}, \zeta''_i) = \theta'_i$. First define for $\zeta \colon \Delta \otimes \Xi \otimes Z \to Z'$, $\zeta^\pi = (\Pi_{\Delta,\Xi} \otimes id); \zeta \colon (\Delta \otimes \Xi \otimes \lambda) \otimes Z \to Z'$. Then define the following auxiliary zipping construction:

$$\text{rzip}(\zeta_1, \zeta_2) = (id \otimes d); (\zeta_1^\pi \otimes id); \zeta_2$$

Define as before $\text{rzip}(\zeta_1, \ldots, \zeta_i, \zeta_{i+1}) = \text{rzip}(\text{rzip}(\zeta_1, \ldots, \zeta_i), \zeta_{i+1})$. Now take $\zeta''_i =$

$\mathrm{rzip}(\zeta'_{i,1},\ldots,\zeta'_{i,m_i},\zeta'_i)$, and note that

$$\begin{aligned}\mathrm{zip}(\theta'_{i-1},\zeta''_i) &= \mathrm{zip}(\theta'_{i-1},\mathrm{rzip}(\zeta'_{i,1},\ldots,\zeta'_{i,m_i},\zeta'_i))\\ &= \mathrm{zip}(\theta'_{i-1},\zeta'_{i,1},\ldots,\zeta'_{i,m_i},\zeta'_i)\\ &= \theta'_i\end{aligned}$$

This completes the "glued" e_2 heap property.

By resplitting the memoization $\llbracket h\rrbracket^{t\bar{u}}$, the play at each Δ_i must then be $t_i\overline{u_i}$ where the play was t_i for e_1 and u_i for e_2 respectively. Similarly, we can resplit the heap diagram at various other points, yielding the following all-round memoization (at e_1):

$$(\theta^\pi_{i-1})^{u_{i-1}}_{\bar{u}_{i-1}};(H_i)^{t_{i-1}\bar{u}_{i-1}}_{t_i\bar{u}_i} = (\llbracket h'\rrbracket^{\Delta'_{i-1}}_{\Xi_i,\lambda_i})^{u_{i-1}}_{u_i};(\theta^\pi_i)^{u_{i-1}}_{\bar{u}_{i-1}}$$

Note that $(\theta^\pi_{i-1})^{u_{i-1}}_{\bar{u}_{i-1}}$ does not memoize θ_{i-1}, merely the promoted part. Composing these two constructed properties we get

$$\theta'^\pi_{i-1};(\theta^\pi_{i-1})^{u_{i-1}}_{\bar{u}_{i-1}};(H_i)^{t_{i-1}\bar{u}_{i-1}}_{t_i\bar{u}_i} = \llbracket h''\rrbracket^{\Delta'_{i-1}}_{\Xi''_i,\Xi'_i};\theta'^\pi_i;(\theta^\pi_i)^{u_{i-1}}_{\bar{u}_{i-1}}$$

We then note that by Lemma 5.16, there exist ζ'''_i such that $\theta''_i = \mathrm{zip}(\zeta'''_1,\ldots,\zeta'''_i) = \theta'^\pi_i;(\theta^\pi_i)^{u_{i-1}}_{\bar{u}_{i-1}}$. In fact in this case $\zeta'''_i = (\!|\zeta_i,\zeta''_i|\!)$. This gives the desired heap property that

$$\theta''^\pi_{i-1};(H_i)^{t_{i-1}\bar{u}_{i-1}}_{t_i\bar{u}_i} = \llbracket h''\rrbracket^{\Delta;_{i-1}}_{\Xi''_i,\Xi'_i};\theta''^\pi_i$$

We now move on to the expression property. This is established by the following chain of reasoning:

$$\begin{aligned}&\theta''^\pi_n;(\llbracket\langle e_1,e_2\rangle\rrbracket_\Delta)^{t\overline{u}}\\ =\;&\theta''^\pi_n;d^{t\overline{u}}_{\hat{t}\hat{u}};\imath;(\llbracket e_1\rrbracket^t\otimes\llbracket e_2\rrbracket^u);\psi\\ &\qquad\text{(where } \hat{t} \text{ and } \hat{u} \text{ are the evident relabellings of } t \text{ and } \overline{u}\text{, and}\\ &\qquad\imath\colon(\Delta'\otimes\Delta')_{\hat{t}\hat{u}}\cong\Delta'_t\otimes\Delta'_u)\\ =\;&d;\theta^\pi_n;\llbracket e_1\rrbracket^t\otimes\theta'^\pi_n;\llbracket e_2\rrbracket^u\\ &\qquad(\text{since } \mathrm{zip}(\ldots,(\!|\zeta_i,\zeta''_i|\!),\ldots) = d;[\mathrm{zip}(\ldots,\zeta_i,\ldots)\otimes\mathrm{zip}(\ldots,\zeta''_i,\ldots)])\\ =\;&d;(\llbracket h'\rrbracket^{\Delta'}_{\Xi_{n+1}};\llbracket v_1\rrbracket_{\Delta',\Xi_{n+1}}\otimes\llbracket h''\rrbracket^{\Delta'}_{\Xi'_{n+1}};\llbracket v\rrbracket_{\Delta',\Xi'_{n+1}};\psi\\ =\;&\llbracket h''\rrbracket^{\Delta'}_{\Xi'_{n+1}};\llbracket\langle v_1,v_2\rangle\rrbracket_{\Delta',\Xi_{n+1},\Xi'_{n+1}}\end{aligned}$$

Case (Let-pair)

Assume $\Delta\vdash e$ and $\Delta'\vdash e'[v_1/x,v_2/y]$. The let-pair rule is:

$$\frac{h,e\Downarrow h',\langle v_1,v_2\rangle\quad h',e'[v_1/x,v_2/y]\Downarrow h'',v}{h,\mathbf{let}\ \langle x,y\rangle\ \mathbf{be}\ e\ \mathbf{in}\ e'\Downarrow h'',v}$$

We first show the termination property—this requires not only the termination property for each premise, but also the expression property for e, since v_1 and v_2 are used in the second premise. By definition

$$\begin{aligned}[\![\textbf{let } \langle x, y\rangle \textbf{ be } e \textbf{ in } e']\!] &= [\![e']\!]^{\dagger}_{\Delta,x,y} \circ \psi \circ (\eta_\Delta \otimes id) \circ [\![e]\!]^{\triangleright}_{\Delta} \\ &= [\![e']\!]^{\dagger}_{\Delta,x,y} \circ \psi \circ (\eta_\Delta \otimes [\![e]\!]_\Delta) \circ d_\Delta\end{aligned}$$

From the inductive hypothesis, $[\![h]\!]; [\![e]\!]$ terminates, and there are given $\zeta_1, \ldots, \zeta_n$ with zipping θ_n. If the terminating play is $qta^{v_1,v_2} \in [\![h]\!] \| [\![e]\!]$, then the I.H. also gives

$$\theta_n^\pi; [\![e]\!]^t_\Delta = [\![h']\!]^{\Delta'}_{\Xi_{n+1}}; [\![\langle v_1, v_2\rangle]\!]_{\Delta',\Xi_{n+1}}$$

By Lemma 5.11,

$$[\![e'[v_1/x, v_2/y]]\!] = [\![e']\!]^{\dagger}_{\Delta,x,y} \circ \psi \circ (((\eta_\Delta \circ \Pi_\Delta) \otimes id) \circ [\![\langle v_1, v_2\rangle]\!]^{\triangleright}_{\Delta',\Xi_{n+1}}$$

Therefore

$$\begin{aligned}&[\![e'[v_1/x, v_2/y]]\!] \circ [\![h']\!]^{\Delta'}_{\Xi_{n+1}} \\ =\ & [\![e']\!]^{\dagger}_{\Delta,x,y} \circ \psi \circ (((\eta \circ \Pi) \otimes id) \circ [\![\langle v_1, v_2\rangle]\!]^{\triangleright}_{\Delta',\Xi_{n+1}} \circ [\![h']\!]^{\Delta'}_{\Xi_{n+1}} \\ =\ & [\![e']\!]^{\dagger}_{\Delta,x,y} \circ \psi \circ ((\eta \otimes id) \circ [\![e]\!]^t_\Delta \circ \theta_n^\pi\end{aligned}$$

So by Lemma 5.12, since $[\![e'[v_1/x, v_2/y]]\!] \circ [\![h']\!]^{\Delta}_{\Xi_{n+1}}$ terminates and $[\![e]\!]^\dagger$ is strict,

$$[\![e']\!]^{\dagger}_{\Delta,x,y} \circ \psi \circ (\eta \otimes id) \circ [\![e]\!]_\Delta$$

terminates, i.e. $[\![\textbf{let } \langle x, y\rangle \textbf{ be } e \textbf{ in } e']\!]$ terminates, completing proof of the termination property.

The heap property proceeds as for the pairing case, excepting that u and $\bar{u}$ differ in a more interesting way, since there is play in the residue. Specifically, the construction of ζ_i''' via Lemma 5.16 takes care of our issues by ensuring $\theta_i'' = \text{zip}(\zeta_1''', \ldots, \zeta_i''') = \theta_i'^\pi; (\theta_i^\pi)^{u_{i-1}}_{\bar{u}_{i-1}}$. Notice that each ζ_i''' as given by that Lemma contains both ζ_i' and a *memoization* of ζ_i'', rather than that morphism itself as in the pairing case, because there can be interaction in e' with the result of e as supplied by ζ_i. This gives the desired heap property that

$$\theta''^\pi_{i-1}; (H_i)^{t_{i-1}\bar{u}_{i-1}}_{t_i\bar{u}_i} = [\![h'']\!]^{\Delta;_{i-1}}_{\Xi_i'',\Xi_i'}; \theta_i''^\pi$$

We now prove the expression property. Given that $\theta_n = \text{zip}(\zeta_1^*, \ldots, \zeta_n^*)$, by the I.H. for the first premise

$$\theta_n^\pi; [\![e]\!]^{t_n}_\Delta = [\![h']\!]^{\Delta'}_{\Xi_{n+1}}; [\![\langle v_1, v_2\rangle]\!]_{\Delta',\Xi_{n+1}}$$

and by the I.H. for the second premise there are morphisms $\zeta'_1, \dots, \zeta'_m$ such that where $\theta'_m = \mathrm{zip}({\zeta'_1}^*, \dots, {\zeta'_m}^*)$,

$$\theta'^{\pi}_m; [\![e'[v_1/x, v_2/y]]\!]^{u_m}_{\Delta'} = [\![h']\!]^{\Delta''}_{\Xi_{m+1}}; [\![v]\!]_{\Delta'',\Xi_{m+1}}$$

Therefore,

$$\begin{array}{rl} & \theta''^{\pi}_n; [([\![e]\!]^t)^{\triangleright}]^{u'_n}_u; ([\![e']\!]^{\dagger}_{\Delta,x,y} \circ \psi \circ (\eta_{!\Delta} \otimes id))^u \\ = & \theta'^{\pi}_n; [(\theta^{\pi}_n; [\![e]\!]^t)^{\triangleright}]^{u_n}_u; ([\![e']\!]^{\dagger}_{\Delta,x,y} \circ \psi \circ (\eta_{!\Delta} \otimes id))^u \\ = & \theta'^{\pi}_n; [([\![h']\!]^{\Delta}_{\Xi_{n+1}}; [\![\langle v_1, v_2\rangle]\!]_{\Delta',\Xi_{n+1}})^{\triangleright}]^{u_n}_u; ([\![e']\!]^{\dagger}_{\Delta,x,y} \circ \psi \circ (\eta_{!\Delta} \otimes id))^u \\ = & \theta'^{\pi}_n; ([\![h']\!]^{\Delta}_{\Xi_{n+1}})^{u_n}_{u_m}; [[\![\langle v_1, v_2\rangle]\!]^{\triangleright}_{\Delta',\Xi_{n+1}}]^{u_m}_u; ([\![e']\!]^{\dagger}_{\Delta,x,y} \circ \psi \circ (\eta_{!\Delta} \otimes id))^u \\ = & h''; \theta'^{\pi}_m; [[\![\langle v_1, v_2\rangle]\!]^{\triangleright}_{\Delta',\Xi_{n+1}}; [\![e']\!]^{\dagger}_{\Delta,x,y} \circ \psi \circ (\eta_{!\Delta} \otimes id)]^u_m \\ = & h''; \theta'^{\pi}_m; [\![e'[v_1/x, v_2/y]]\!]^{u_m} \\ = & [\![h']\!]^{\Delta''}_{\Xi_{m+1}}; [\![v]\!]_{\Delta'',\Xi_{m+1}} \end{array}$$

This completes proof of the expression property.

Case (Method Invocation)

Assume $\Delta \vdash l \cdot m\ v_1$, where $\Delta = \Delta_L, l\colon \lambda, \Delta_R$, so that $\Delta_L \vdash v_c, s_1$, and suppose $\Delta \vdash l \cdot m\colon \tau \to \tau'$ and $\Delta \vdash s_1\colon \sigma$. The reduced method invocation rule:

$$\frac{h, v_c \cdot m\ \langle s_1, v_1\rangle \Downarrow h', \langle s_2, v_2\rangle}{h, l \cdot m\ v_1 \Downarrow h'[l \mapsto \langle s_2, v_c\rangle], v_2}\ h(l) = \langle s_1, v_c\rangle$$

This is by far the most demanding induction case, and where most of the interesting context of the proof resides (details of the proof of this case were worked out with John Longley).

Argument outline

Before proving the method invocation case, we attempt to give an intuition and describe the structure of the proof.

From the inductive hypothesis we get a set of equations (the heap property) expressing the heap update during evaluation of $v_c \cdot m\ \langle s_1, v_1\rangle$, and an equation giving the result $\langle s_2, v_2\rangle$ (the expression property). Since the method invocation causes an update of the heap at l, the expression property for $v_c \cdot m\ \langle s_1, v_1\rangle$ will feed into the construction of both expression and heap properties for $l \cdot m\ v_1$.

As an intermediate step, we shall construct a heap cell l' as a copy of l, so that $l' \cdot m$ will correspond to $v_c \cdot m$ resulting in heap h', and $l \cdot m$ will be as $l' \cdot m$

but resulting in heap $h'[l \mapsto \langle s_2, v_c \rangle]$. Take a fresh location l' containing $\langle v_c, s_1 \rangle$, located at the end of the heap. By the properties of *thread* from Chapter 3, one finds that the result of $l' \cdot m\ v_1$ agrees with $v_c \cdot m\ \langle s_1, v_1 \rangle$, and the updated object l' after evaluation agrees with a new object constructed from v_c and the state part of $v_c \cdot m\ \langle s_1, v_1 \rangle$. From the inductive hypothesis, the result and state are v_2 and s_2 respectively.

By observing that v_c can not refer to the portion of the heap from l onwards, and the result of the method invocation cannot because $v_c \cdot m$ cannot capture pointers from v_1, we can consider l' to reside beside l in the heap. Given this newly constructed heap, since they are constructed with the same implementation and state, $l' \cdot m$ and $l \cdot m$ agree up to the point of termination, and yet may subsequently differ. There are two reasons for this. Firstly, heap locations subsequent to l may of course observe the updated state here. Secondly, if there are any nested invocations of l, directly or indirectly caused by interaction with v_1, in the case of $l' \cdot m$ this will be observable as a state-change of l, while in the case of $l \cdot m$ it will not.

We finally observe that we can merge any nested invocations in l into l', so that l' after $l' \cdot m$ is the same in both cases, the play in l' is the desired interaction and the updated state (s_2) agrees with the result of the operational rule, where we rename l' to l (throwing away the original l).

Termination Property

By the induction hypothesis,

$$[\![h]\!]; [\![v_c \cdot m\langle s_1, v_1 \rangle]\!]_\Delta \colon 1 \to [\![\sigma \otimes \tau']\!]_\perp$$

terminates. We begin by throwing away the "state" component of the result. Let

$$\Pi' = \perp(\Pi_{[\![\tau']\!]}) \colon [\![\sigma \otimes \tau']\!]_\perp \to [\![\tau']\!]_\perp$$

Then clearly $[\![h]\!]; [\![v_c \cdot m\ \langle s_1, v_1 \rangle]\!]_\Delta; \Pi'$ terminates: that is, there is some play

$$qta^{v_2} \in [\![h]\!] || [\![v_c \cdot m\ \langle s_1, v_1 \rangle]\!]_\Delta; \Pi'$$

But

$$\begin{array}{ll} & [\![v_c \cdot m\,\langle s_1, v_1\rangle]\!]_\Delta ; \Pi' \\ = & d^3; (([\![v_c]\!]_\Delta ; \bot(\varepsilon; \Pi_m)) \otimes [\![s_1]\!]_\Delta \otimes [\![v_1]\!]_\Delta); \mathit{eval}^\dagger; \Pi' \\ & \qquad \text{(by definition of } [\![-]\!]\text{)} \\ = & d^3; (([v_c]_\Delta\,; \varepsilon; \Pi_m) \otimes [s_1]_\Delta \otimes [v_1]_\Delta); \mathit{eval}; \Pi' \\ & \qquad \text{(since } s_1, v_1, v_c \text{ are values)} \\ = & d^2; \big(((([\langle s_1, v_c\rangle]_\Delta\,; \mathit{thread}) \otimes \mathit{id}); (\varepsilon; \Pi_m) \otimes [v_1]_\Delta\big); \mathit{eval} \\ & \qquad \text{(by Thread Property 1, since both } [v_c]_\Delta \text{ and } [s_1]_\Delta \text{ are promoted} \\ & \qquad \text{morphisms by Lemma 5.8)} \end{array}$$

$$\begin{array}{ll} = & [\mathbf{constr}\ s_1\ v_c]^{\lhd}_\Delta\,; \big((\varepsilon; \Pi_m) \otimes [v_1]_\Delta\big); \mathit{eval} \\ & \qquad \text{(by definition of } [-]\text{)} \\ = & [\mathbf{constr}\ s_1\ v_c]^{\rhd}_\Delta\,; \gamma_{\Delta,\lambda}; \big((\varepsilon; \Pi_m) \otimes [v_1]_\Delta\big); \mathit{eval} \\ & \qquad \text{(swapping for later convenience)} \end{array}$$

Thus,

$$qta^{v_2} \in [\![h]\!] \,\|\, [\mathbf{constr}\ s_1\ v_c]^{\rhd}_\Delta\,; -@v_1$$

where $-@v_1$ abbreviates $\gamma_{\Delta,\lambda}; (\varepsilon; \Pi_m \otimes [v_1]_\Delta); \mathit{eval}$. Shifting the interaction boundary to the right, there is some play

$$q\bar{t}a^{v_2} \in [\![h]\!]; [\mathbf{constr}\ s_1\ v_c]^{\rhd}_\Delta \,\| {-@v_1}$$

where the internal interaction $\bar{t}$ takes place in the game $[\![\Delta]\!] \otimes [\![\lambda]\!]$. Moreover, it is easy to see that the strategy responds to the initial question q with a question $q_0^{v_1}$ in the m-component of $[\![\lambda]\!]$, and generates the answer a^{v_2} immediately from the corresponding answer $a_0^{v_2}$. Thus, $\bar{t}$ has the form $q_0^{v_1} t' a_0^{v_2}$, and moreover all moves in t' are in the left component of $[\![\Delta]\!] \otimes [\![\lambda]\!]$ (corresponding to interactions with the heap arising from locations appearing in v_1).

We now concentrate on the left half of the above interaction, that is, the play

$$\bar{t} = q_0^{v_1} t' a_0^{v_2} \in [\![h]\!]; [\mathbf{constr}\ s_1\ v_c]^{\rhd}_\Delta : 1 \to [\![\Delta]\!] \otimes [\![\lambda]\!]$$

We show that essentially the same play is possible if the "real" heap cell l in h is used for the method call rather than the copy $[\mathbf{constr}\ s_1\ v_c]_\Delta$ just created (i.e. l' in the earlier discussion).

First, let h_L, h_R be the heap portions corresponding to Δ_L, Δ_R respectively, and define

$$\begin{array}{rcll} H_L & = & [\![h_L]\!] & : \ 1 \to [\![\Delta_L]\!] \\ H_\ell & = & [\![h_L, l \mapsto \langle s_1, v_c\rangle]\!]^{\Delta_L}_{l:\,\lambda} & : \ [\![\Delta_L]\!] \to [\![\lambda]\!] \\ H_R & = & [\![h]\!]^{\Delta_L, l:\,\lambda}_{\Delta} & : \ [\![\Delta_L]\!] \otimes [\![\lambda]\!] \to [\![\Delta_R]\!] \end{array}$$

Now note that by the left-pointing property of heaps,

$$[\mathbf{constr}\ s_1\ v_c]_\Delta = [\mathbf{constr}\ s_1\ v_c]_{\Delta_L} \otimes 1_{\lambda,\Delta_R}$$

so we can effectively "commute" $[\mathbf{constr}\ s_1\ v_c]$ past H_R, so as to be adjacent to the real heap cell l:

$$\begin{array}{rl} & [\![h]\!]; [\mathbf{constr}\ s_1\ v_c]^{\rhd}_\Delta \\ = & H_L; H_\ell^{\rhd}; H_R^{\rhd}; ([\mathbf{constr}\ s_1\ v_c]_{\Delta_L} \otimes 1_{\lambda,\Delta_R})^{\rhd} \\ = & H_L; H_\ell^{\rhd}; ([\mathbf{constr}\ s_1\ v_c]_{\Delta_L} \otimes 1_\lambda)^{\rhd}; (H_R^{\rhd} \otimes id_\lambda) \\ = & H_L; H_\ell^{\rhd}; (H_\ell \otimes 1_\lambda)^{\rhd}; (H_R^{\rhd} \otimes id_\lambda) \\ & \qquad \text{(by definition of } H_\ell) \\ = & H_L; (d_{\Delta_L}; H_\ell \otimes H_\ell)^{\rhd}; (H_R^{\rhd} \otimes id_\lambda) \end{array}$$

So the play $q_0^{v_1} t' a_0^{v_2}$ is present in this latter strategy. Now consider the "exposed" interaction in

$$H_L; (d_{\Delta_L}; H_\ell \otimes H_\ell)^{\rhd} \| (H_R^{\rhd} \otimes id_\lambda)$$

Since the moves $q_0^{v_1}$ and $a_0^{v_2}$ are simply copied back and forth by id_λ (as well as play in the method argument), we have some play

$$q_0^{v_1} t'' a_0^{v_2} \in H_L; (d_{\Delta_L}; H_\ell \otimes H_\ell)^{\rhd} \colon 1 \to [\![\Delta_L]\!] \otimes [\![\lambda]\!] \otimes [\![\lambda]\!]$$

The strategy H_ℓ appears twice here: the first copy (corresponding to the left hand $[\![\lambda]\!]$) is used for any nested invocations of the object at l triggered (directly or indirectly) by interaction with $[v_1]$, while the second copy is used for the method call under consideration, and only for that. As we noted earlier, $[v_c]_\Delta$ and $[s_1]_\Delta$ are promoted morphisms, and since $\Delta_L \vdash v_c \colon \mathbf{Obj}\ X$, by Lemma 4.6 $[v_c]$ is a disciplined strategy. Therefore we are in the situation addressed by Thread Property 3(a), which shows that all these interactions may as well use only a single copy of H_ℓ via the contraction map $[\![\lambda]\!] \to [\![\lambda]\!] \otimes [\![\lambda]\!]$:

$$q_0^{v_1} t'' a_0^{v_2} \in H_L; (H_\ell; d_\lambda)^{\rhd} \colon 1 \to [\![\Delta_L]\!] \otimes [\![\lambda]\!] \otimes [\![\lambda]\!]$$

We may now put back the parts of the composition we stripped away. Since $H_L; (H_\ell; d)^\triangleright$ behaves identically to $H_L; H_\ell^\triangleright; (H_\ell \otimes 1_\lambda)^\triangleright$ under the relevant interactions in $[\![\Delta]\!] \otimes [\![\lambda]\!] \otimes [\![\lambda]\!]$, we have

$$\begin{aligned} q_0^{v_1} t' a_0^{v_2} &\in H_L; (H_\ell; d_\lambda)^\triangleright; (H_R^\triangleright \otimes id_\lambda) \\ &= [\![h]\!]; \Pi_\lambda^\triangleright \colon 1 \to [\![\Delta]\!] \otimes [\![\lambda]\!] \end{aligned}$$

and likewise $q\bar{t}a^{v_2} \in [\![h]\!]; \Pi_\lambda^\triangleright \| {-}@v_1$. But

$$\begin{aligned} \Pi_\lambda^\triangleright; -@v_1 &= d; (\Pi_\lambda \otimes id); (\varepsilon; \Pi_m \otimes [v_1]_\Delta); eval \\ &= d; ((\Pi_\lambda; \varepsilon; \Pi_m) \otimes [v_1]_\Delta); eval \\ &= [\![l \cdot m\ v_1]\!]_\Delta \end{aligned}$$

So $qa^{v_2} \in [\![h]\!]; [\![l \cdot m\ v_1]\!]_\Delta$ as required.

Constructing a new heap cell

As a first step to establishing the heap property for the evaluation of $l \cdot m\ v_1$ in h, we consider the evaluation of $l' \cdot m\ v_1$ in a heap $h[l' \mapsto \langle s_1, v_c\rangle]$, and first establish the heap property for this situation. Throughout this section, let θ_n and t_n mean what they do in the context of the inductive hypothesis: here we will construct a $\zeta_{l'}$ for the additional heap cell.

From Theorem 4.2, s_2 cannot contain (existing) locations from the right of l_n, and in $\mathcal{L}_{\text{ret}}$ the set of new heap locations in s_2 and v_2 must be disjoint, i.e. $\Delta'_L, \Xi_S \vdash s_2$ and $\Delta', \Xi_V \vdash v_2$. Then from part 3 of the I.H.:

$$\theta_n^\pi; [\![v_c \cdot m\langle s_1, v_1\rangle]\!]_\Delta^{t_n} = [\![h']\!]_{\Xi_S, \Xi_V}^{\Delta'}; [\![\langle s_2, v_2\rangle]\!]_{\Delta', \Xi_S, \Xi_V}$$

We now throw away the non-state component of the result. The state component projects as follows:

$$\theta_n^\pi; [\![v_c \cdot m\langle s_1, v_1\rangle]\!]_\Delta^{t_n}; \bot(\Pi_{[\![\sigma]\!]}) = [\![h']\!]_{\Xi_S}^{\Delta'_L}; [\![s_2]\!]_{\Delta'_L, \Xi_S}$$

By Thread Property 2, there is a play u such that

$$[\mathbf{constr}\ s_1\ v_c]_u^{t_n}; \Pi_E = \big(([\![v_c \cdot m\langle s_1, v_1\rangle]\!]_\Delta^{t_n}; \bot(\Pi_{[\![\sigma]\!]})) \otimes [v_c]_\Delta\big); thread^\star$$

where $\Pi_E \colon (!([\![\tau]\!] \to [\![\tau']\!]_\bot))_u \cong (Z_u \otimes !([\![\tau]\!] \to [\![\tau']\!]_\bot)) \xrightarrow{\Pi_R} !([\![\tau]\!] \to [\![\tau']\!]_\bot)$. Therefore

$$\theta_n^\pi; [\mathbf{constr}\ s_1\ v_c]_u^{t_n}; \Pi_E = [\![h']\!]_{\Xi_S}^{\Delta'_L}; [\mathbf{constr}\ s_2\ v_c]$$

Since $[v_c]$ is pair-like, by Thread Property 4 there are morphisms f and g

$$\Delta_{t_n} \cong \Delta_{t_n^L} \otimes \Delta_{t_n^R} \xrightarrow{f \otimes g} Z_u \otimes !(\llbracket\tau\rrbracket \to \llbracket\tau'\rrbracket_\perp)$$

such that $[\mathbf{constr}\ s_1\ v_c]_u^{t_n} = f \otimes g$. Take the decompositions as $\Delta_{t_n^L} \cong \Delta \otimes Z_L$ and $\Delta_{t_n^R} \cong \Delta \otimes Z_R$, so that $Z_n \cong Z_L \otimes Z_R$. The inductive hypothesis states that θ_n is fibred, so splitting θ_n as $\theta_n = \theta_S \otimes \theta_V$ for fibres $\theta_S \colon \Delta' \to Z_L$ and $\theta_V \colon \Delta' \to Z_R$ we have that

$$\begin{aligned}(\theta_S^\pi \otimes \theta_V^\pi); \llbracket\mathbf{constr}\ s_1\ v_c\rrbracket_u^{t_n} \quad = \quad & \llbracket h'\rrbracket_{\Xi_S}^{\Delta'_L}; \llbracket\mathbf{constr}\ s_2\ v_c\rrbracket \otimes \\ & \theta_V^\pi; \llbracket\mathbf{constr}\ s_1\ v_c\rrbracket_u^{t_n}; \Pi_{Z_u}\end{aligned}$$

We therefore define $\zeta_{l'} \colon \Delta' \otimes Z_n \to Z_u$ as

$$\zeta_{l'} = 1_{Z_L} \otimes \llbracket\mathbf{constr}\ s_1\ v_c\rrbracket_u^{t_n}; \Pi_{Z_u}$$

and note that $\zeta_{l'}$ is trivially fibred (consisting of only one fibre). Then we have shown a suitable heap property for l':

$$\theta_n^\pi; ([\mathbf{constr}\ s_1\ v_c]_u^{t_n})^\rhd = (\llbracket h'\rrbracket_{\Xi_S}^{\Delta'_L}; [\mathbf{constr}\ s_2\ v_c]_{\Delta'_L})^\rhd; (\mathrm{zip}(\theta_n, \zeta_{l'}^*))^\pi$$

Moving the new cell left

At this point we must move the new heap cell l' from the end of the heap to the position just after l, before we can merge the two. Here we show that one can switch l' with a single cell on its left (without altering the results of the composition), because l' is actually independent of that cell. This fact is then repeatedly applied until l' lies immediately to the right of l, which it *does* depend on. At each stage, we wish to know that the heap property still holds for the computation arising form evaluating $l' \cdot m\ v_1$. From the data $\zeta_j, \zeta_{l'}$ we construct the new data simply as $\zeta_{l'}, \zeta_j$, since the $-^*$ operation will do the work for us.

Given

$$\begin{aligned}\theta_{j-1}^\pi; (H_j)_{t_j}^{t_{j-1}} &= H'_j; (\mathrm{zip}(\theta_{j-1}, \zeta_j^*))^\pi \\ (\mathrm{zip}(\theta_{j-1}, \zeta_j^*))^\pi; [(\Pi_{\Delta_{j-1}}; H_l)]_{t_l}^{t_j} &= (\Pi_{\Delta_{j-1}}; H'_{l'}); (\mathrm{zip}(\theta_{j-1}, \zeta_j^*, (\Pi_{\Delta_{j-1}}; \zeta_{l'})^*))^\pi\end{aligned}$$

calculation involving (memoizations of) the projections and the contraction and identity in the various instances of $-^\pi$ shows that

$$\begin{aligned}\theta_{j-1}^\pi; (H_\ell)_{t'_j}^{t_{j-1}} &= H'_{l'}; (\mathrm{zip}(\theta_{j-1}, \zeta_{l'}^*))^\pi \\ (\mathrm{zip}(\theta_{j-1}, \zeta_{l'}^*))^\pi; [(\Pi_{\Delta_{j-1}}; H_j)]_{t_l}^{t'_j} &= (\Pi_{\Delta_{j-1}}; H'_j); (\mathrm{zip}(\theta_{j-1}, \zeta_{l'}^*, (\Pi_{\Delta_{j-1}}; \zeta_j)^*))^\pi\end{aligned}$$

where t'_j is as t_j but instead of passing through moves from h_l passes through moves from h_j. Notice that the precise meaning of $-^*$ has changed between these two sets of equations, in particular the type of the identity part. Furthermore, from Lemma 5.15 and reasoning as above we have that the identity of the composite is maintained:

$$\begin{array}{rcl} \mathrm{zip}(\theta_{j-1}, \zeta_j^*, (\Pi_{\Delta_{j-1}}; \zeta_{l'})^*); & = & \mathrm{zip}(\theta_{j-1}, \zeta_{l'}^*, (\Pi_{\Delta_{j-1}}; \zeta_j)^*) \\ (H_j)_{t_j}^{t_{j-1}}; [(\Pi_{\Delta_{j-1}}; H_\ell)]_{t_l}^{t_j} & = & (H_\ell)_{t'_j}^{t_{j-1}}; [(\Pi_{\Delta_{j-1}}; H_j)]_{t_l}^{t'_j}; \gamma \\ H'_j; (\Pi_{\Delta_{j-1}}; H'_{l'}) & = & H'_{l'}; (\Pi_{\Delta_{j-1}}; H'_j); \gamma \end{array}$$

Merging heap cells—coincidence of play / diagram construction

We now have a heap with adjacent cells l and l', which we wish to merge for our resulting l cell. The key here is noticing that the play in l and l' translates to a play in l alone, and the resulting object at l' after the former play is the object at l after the latter. The object at l after the former play is *not* related to anything in the latter, as it represents the discarded state update in a nested method call.

We shall now construct Diagram 5.1, showing the merged heap cell, and show that it commutes. Take $\zeta'_l = (\!|\zeta_l, \zeta_{l'}|\!)$, so that where $\theta_l = \mathrm{zip}(\theta_{l-1}, \zeta_l)$ and $\theta_{l'} = \mathrm{zip}(\theta_{l-1}, \zeta_l, \zeta_{l'})$, we have the new $\theta'_l = \mathrm{zip}(\theta_{l-1}, \zeta'_l)$.

Recall from the termination argument that from Thread Property 3, the play $t_{l'}$ in $\Delta_{l-1} \otimes \lambda$ (which is a terminating play in λ) is admitted both by $(H_\ell; d_{\Delta,\lambda})^\triangleright$ and $H_\ell^\triangleright; (H_\ell \otimes 1_\lambda)^\triangleright$. Furthermore, Thread Property 3 says that if

$$\tilde{\Pi} = (\llbracket\lambda\rrbracket \otimes \llbracket\lambda\rrbracket)_{t_{l'}} \cong Z_{t_{l'}} \otimes \llbracket\lambda\rrbracket \otimes \llbracket\lambda\rrbracket \xrightarrow{id_{Z_{t_{l'}}} \otimes 1_{\llbracket\lambda\rrbracket} \otimes id_{\llbracket\lambda\rrbracket}} Z_{t_{l'}} \otimes \llbracket\lambda\rrbracket \cong \llbracket\lambda\rrbracket_{t_{l'}}$$

then

$$[H_\ell^\triangleright; (H_\ell \otimes 1_\lambda)^\triangleright]_{t_{l'}}^{t_{l-1}}; \tilde{\Pi} = ((H_\ell; d)^\triangleright)_{t_{l'}}^{t_{l-1}}; \tilde{\Pi} = (H_\ell^\triangleright)_{t'_l}^{t_{l-1}}$$

For the corresponding lower triangle, take

$$\Pi = (id \otimes 1_\lambda \otimes id)$$

and

$$\begin{array}{rcl} H''_l & = & \llbracket h''_l \rrbracket^{\Delta_L}_{\Xi_l, \Xi_{l'}, l_i \colon \lambda_i} \\ & : & \llbracket\Delta_L\rrbracket \to \llbracket\Delta_L, \Xi_l, \Xi'_l, l_i \colon \lambda_i\rrbracket \end{array}$$

then by examination of the definition of relativised heaps and $-^\triangleright$,

$$(H''_l)^\triangleright = H_l'^\triangleright; (\Pi_{\Delta_L}; H'_{l'})^\triangleright; \Pi_{\Delta_L, \Xi_l, \Xi_{l'}, \lambda}$$

The right-hand trapezoid $\tilde{\Pi} \circ \theta^{\pi}_{l'} = \theta'^{\pi} \circ \Pi$ holds from the definition of Π and $\tilde{\Pi}$, plus the fact that by Lemma 5.14

$$\theta_{l'} = \text{zip}(\theta_{l-1}, \zeta^*_l, (\Pi_\Delta \circ \zeta_{l'})^*) = \text{zip}(\theta_{l-1}, (\!|\zeta_l, \zeta_{l'}|\!)^*) = \theta'_l$$

The two inner diagrams are just the existing heap properties for l and l', therefore we have shown that Diagram 5.1 commutes.

Heap property

Since we have collapsed l and l', we can now remove the contraction to produce the desired final heap. Since each heap cell now has the form $\Pi_{\Delta_{j-1}}; H_j$ we note that

$$(id \otimes d_\lambda); (\Pi_{\Delta_{j-1}}; H_j); (id \otimes \gamma) = H_j; (id \otimes d_\lambda \otimes id); (id \otimes \gamma)$$

The memoization is then for the play t_j at Δ_j, where all play in λ is in the same copy as desired. Similarly, $d; \text{zip}(\theta, (\Pi; \zeta_j)^*) = \text{zip}(\theta, \zeta^*_j)$.

We have therefore proved the desired heap property.

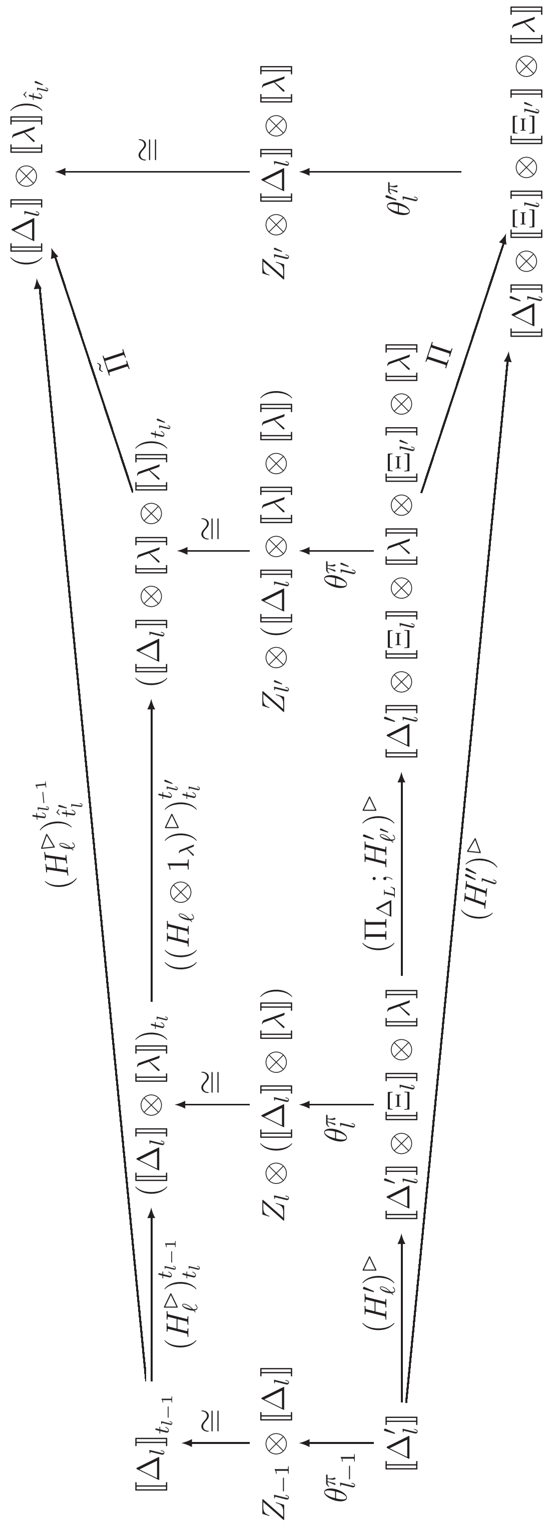
Expression property

The verification of the expression property proceeds somewhat similarly to the section "Constructing a new heap cell". As in that section, from the inductive hypothesis we have

$$\theta^{\pi}_n; [\![v_c \cdot m\langle s_1, v_1\rangle]\!]^{t_n}; \bot(\Pi_V) = [\![h']\!]^{\Delta'}_{\Xi_V}; [\![v_2]\!]_{\Delta', \Xi_V}$$

Then if we define

$$\begin{aligned}\theta'' &= \text{zip}(\theta_n, (\Pi_\Delta \circ \zeta_{l'})^*) \\ &= \text{zip}(\zeta^*_1, \ldots, (\!|\zeta_l, \zeta_{l'}|\!)^*, \ldots, \zeta^*_n)\end{aligned}$$



$$\begin{tikzcd}
 & & & (\llbracket \Delta_l \rrbracket \otimes \llbracket \lambda \rrbracket)_{\hat{t}_{l'}} \\
\llbracket \Delta_l \rrbracket_{t_{l-1}} \arrow[r, "(H_\ell^{\rhd})_{t_l}^{t_{l-1}}"'] \arrow[urrr, "(H_\ell^{\rhd})_{\hat{t}'_l}^{t_{l-1}}"] & (\llbracket \Delta_l \rrbracket \otimes \llbracket \lambda \rrbracket)_{t_l} \arrow[r, "((H_\ell \otimes 1_\lambda)^{\rhd})_{t_l}^{t_{l'}}"'] & (\llbracket \Delta_l \rrbracket \otimes \llbracket \lambda \rrbracket \otimes \llbracket \lambda \rrbracket)_{t_{l'}} \arrow[ur, "\tilde{\Pi}"'] & \\
Z_{l-1} \otimes \llbracket \Delta_l \rrbracket \arrow[u, "\cong"'] & Z_l \otimes (\llbracket \Delta_l \rrbracket \otimes \llbracket \lambda \rrbracket) \arrow[u, "\cong"'] & Z_{l'} \otimes (\llbracket \Delta_l \rrbracket \otimes \llbracket \lambda \rrbracket \otimes \llbracket \lambda \rrbracket) \arrow[u, "\cong"'] & Z_{l'} \otimes \llbracket \Delta_l \rrbracket \otimes \llbracket \lambda \rrbracket \arrow[uu, "\cong"'] \\
\llbracket \Delta'_l \rrbracket \arrow[u, "\theta_{l-1}^{\pi}"] \arrow[r, "(H'_\ell)^{\rhd}"] \arrow[drrr, "(H''_l)^{\rhd}"'] & \llbracket \Delta'_l \rrbracket \otimes \llbracket \Xi_l \rrbracket \otimes \llbracket \lambda \rrbracket \arrow[u, "\theta_l^{\pi}"] \arrow[r, "(\Pi_{\Delta_L}; H'_{\ell'})^{\rhd}"] & \llbracket \Delta'_l \rrbracket \otimes \llbracket \Xi_l \rrbracket \otimes \llbracket \lambda \rrbracket \otimes \llbracket \Xi_{l'} \rrbracket \otimes \llbracket \lambda \rrbracket \arrow[u, "\theta_{l'}^{\pi}"] \arrow[dr, "\Pi"] & \\
 & & & \llbracket \Delta'_l \rrbracket \otimes \llbracket \Xi_l \rrbracket \otimes \llbracket \Xi_{l'} \rrbracket \otimes \llbracket \lambda \rrbracket \arrow[uu, "\theta_l'^{\pi}"]
\end{tikzcd}$$

Figure 5.1: Merging heap cells

the following reasoning establishes the expression square:

$$
\begin{aligned}
& \theta^{\pi}_{n}; [\![v_c \cdot m\langle s_1, v_1\rangle]\!]^{t_n}; \bot(\Pi_V) \\
= \quad & \theta^{\pi}_{n}; ([\textbf{constr}\ s_1\ v_c]^{\lhd}_{\Delta}; (\Pi_m; \varepsilon \otimes [v_1]_{\Delta}); \mathit{eval})^{t_n} \\
& \qquad \text{(By Thread Property 1.)} \\
= \quad & \theta^{\pi}_{n}; ([\textbf{constr}\ s_1\ v_c]^{\lhd}_{\Delta})^{t_n}_{u}; ((\Pi_m; \varepsilon \otimes [v_1]_{\Delta}); \mathit{eval})^{u} \\
& \qquad \text{(Resplitting.)} \\
= \quad & \theta^{\pi}_{n}; ([\textbf{constr}\ s_1\ v_c]^{\lhd}_{\Delta})^{t_n}_{u}; \Pi_Z; [(\Pi_m; \varepsilon \otimes [v_1]_{\Delta}); \mathit{eval}]^{u} \\
& \qquad \text{(By examination of } [-]^{u}\text{—as } \varepsilon \text{ means no more uses of } l'.) \\
= \quad & \theta^{\pi}_{X}; ([\textbf{constr}\ s_1\ v_c]^{\lhd}_{\Delta})^{t_n}_{u}; \Pi_Z; \theta^{\pi}_{Y}; ((\Pi_m; \varepsilon \otimes [v_1]_{\Delta}); \mathit{eval})^{u} \\
& \qquad (\theta \text{ is fibred.}) \\
= \quad & (\theta' \otimes \theta_Y)^{\pi}; ((\Pi_m; \varepsilon \otimes [v_1]_{\Delta}); \mathit{eval})^{u} \\
& \qquad \text{(By construction of } \zeta'_l \text{ and hence } \theta' \text{ from the earlier section.)} \\
= \quad & \theta''^{\pi}; [\![l \cdot m\ v_1]\!]^{u}_{\Delta} \\
& \qquad \text{(Definition of } \theta'', [\![-]\!].)
\end{aligned}
$$

5.4 Further issues

We have given a soundness proof for the restricted language $\mathcal{L}_{\text{ret}}$ rather than the more general case of $\mathcal{L}_{\text{arg}}$. This is a genuine restriction, which prohibits methods which create new objects and both store them in their state and return them. However, unlike in $\mathcal{L}_{\text{pair}}$, methods may return higher-type results from their state. In both cases, arbitrary results may be returned from other objects in the heap (but again, the results of such a method call may be stored in the state, or returned, but not both). Despite the restriction of $\mathcal{L}_{\text{ret}}$, it may be possible to accomplish the same results as a method in $\mathcal{L}_{\text{arg}}$ via multiple method calls, first storing some locally created object in the state then subsequently retrieving that object and returning it. Here we discuss the implications to the proof of removing this restriction.

The restriction manifests istelf in the statement of the soundness property. The restriction of $\vec{\zeta}$ (and hence θ) to fibred morphisms builds in a separation between the results of any two method invocations. This is important in the method invocation case, where we both make use of the assumption of fibredness, and construct a new morphism which is fibred because of the pair-like property of the method implementation.

Recall that the *shape* of our heap is always a DAG, i.e. there are no circular references. While the shape of dependencies on method return results is generally a DAG, in the restricted situation this structure is a tree. This can be most easily seen in the definition of fibred morphisms: several fibres can be "tied together" when a method calls several methods and returns a result constructed from their return values, but a fibre can never be split by using it both to update the state and return a value (this would be non-pair-like behaviour). Our proof follows this tree-structure, while a more general proof would seem to have to follow the DAG-structure.

5.5 Adequacy

The property of *computational adequacy* states that the operational and denotational semantics agree:

$$h, e \Downarrow h', v \iff [\![h]\!]; [\![e]\!] = [\![h']\!]; [\![v]\!]$$

The soundness property we have proved gives the $\Rightarrow$ direction. The remainder amounts to proof that $[\![-]\!]$ is not over-defined, i.e.

$$[\![h]\!]; [\![e]\!] \neq \bot \implies \exists h', v.\ h, e \Downarrow h', v$$

For reasons of time we do not prove this property here. It seems highly implausible that it is false—it is hard to imagine where our denotational semantics might "invent" a value where we do not intend to produce one. Moreover, we have had to impose considerable restrictions to ensure that our operational semantics is not *over*-defined, not just by the pair-like restriction but by the restricted typing for method implementations to produce disciplined strategies and avoid cycles in the heap.

The probable truth of this property aside, the proof will still be non-trivial. However, we expect that the standard proof technique using logical relations will suffice. In this case, the essence of the problem will be to formulate the correct relation. We believe that the detailed analysis of our semantics which has been required for the soundness proof will also go much of the way towards proving the remainder of the adequacy property.

Chapter 6

Definability and full abstraction

In this chapter we shall show that the interpretation of our object-oriented language in $\mathbf{BG}^V$ satisfies two important properties. We show that *definability* and *full abstraction* hold at a certain (somewhat unusual) subset of denotable types, including types of any given rank (and in particular all pure types).

Define the sets of types T and $ArgT$ by the following grammar:

$$\begin{array}{rcl} T & ::= & \iota \mid (ArgT \Rightarrow T) \\ ArgT & ::= & \iota \mid (T \Rightarrow \iota) \end{array}$$

We shall now state the the properties proven in this chapter. Firstly, call a strategy *effective* if the underlying function from odd-length plays to moves is effectively computable. Then at types in T all effective strategies are definable:

Theorem 6.1 (Definability)**.** *For any type $\tau \in T$, and any effective strategy $a\colon 1 \to [\![\tau]\!]$, there exist suitable sets of computable constant functions $\Phi = \{\Phi_k \mid k \in \mathbb{N}\}$, and a closed term $\emptyset \vdash e : \tau$ in $\mathcal{L}_{\text{arg}}$ with constants from Φ, such that $[\![\emptyset \vdash e : \tau]\!] = a$.*

Secondly, at types in $ArgT$ the interpretation is fully abstract:[1]

Theorem 6.2 (Full abstraction)**.** *For any type $\tau \in ArgT$ and terms $\vdash e\colon \tau$, $\vdash e'\colon \tau$:*

$$\big(\forall C[-], v.\, C[e] \Downarrow v \Leftrightarrow C[e'] \Downarrow v\big) \;\Rightarrow\; [\![\vdash e : \tau]\!] = [\![\vdash e' : \tau]\!]$$

where $C[-]$ ranges over ground-type contexts and $e \Downarrow v$ abbreviates $\exists h'.\ \emptyset, e \Downarrow h', v$.

[1]Technically our full abstraction result is not quite for the interpretation in $\mathbf{BG}^V$, but for a "truncated" version—see Section 6.4.

We shall show both of these properties by constructing a suitable family of programs. There is a type ρ (namely the type of objects with a single $\iota \to \iota$ method) such that for any strategy $\sigma : [\![\tau]\!]$ we can give a program $\hat{\sigma}$ of type ρ encoding the sequence of moves in σ (we apologise for the unfortunate clash of notation with regards σ and ρ, τ). We then define a program $\mathit{interpret}_\tau \colon \rho \to \tau$ with the property that

$$[\![\mathit{interpret}_\tau\ \hat{\sigma}]\!] = \sigma$$

Both properties follow easily from the existence of these programs.

For the purposes of this chapter, we shall restrict our attention to the fragment of our language without product types and where objects have only single methods—these features do not seem to add any new issues with respect to definability, but would add a great deal of complication in the proof. We also restrict ourselves to the *intuitionistic* fragment; additional language features are required for definability at certain types involving linear functions, and we discuss this issue in Section 7.3.4. As noted above, our proof will fail to cover the entire range of product-free intuitionistic types; we shall describe the problematic behaviour, and suggest a possible language extension which might remove this restriction.

6.1 Notation

In this chapter we are interested in product-free types, and so we will often be discussing objects with a single method. We will abbreviate the "intuitionistic" type **Obj** $\{m \colon \tau \to \tau'\}$ as $\tau \Rightarrow \tau'$, and for an object o of that type abbreviate

$$o \bullet e \rightsquigarrow o \cdot m\ e$$

Note that we are *not* defining $\tau \Rightarrow \tau'$ as **Obj** $\{m \colon \tau\} \to \tau'$ as one might expect from the usual linear-logic decomposition of $A \Rightarrow B$ as $!A \multimap B$, since we want to use **constr** to construct stateful objects.

When defining such objects, we further abbreviate

$$\textbf{letrec } f \bullet x = e \textbf{ in } e' \rightsquigarrow \textbf{let } f \textbf{ be } [\mathbf{Y}(\lambda f.\ \textbf{obj } \{m = \lambda x.e\})] \textbf{ in } e'$$

We shall also make use of the derived forms given in Chapter 4. In particular we recall:

$$\textbf{letrec } f(x) = e \textbf{ in } e' \rightsquigarrow \textbf{let } f \textbf{ be } Y(\lambda f.\ \lambda x.\ e) \textbf{ in } e'$$

$$g \circ f \rightsquigarrow \lambda x.g(f(x))$$

$$o_2 \circ o_1 \rightsquigarrow \textbf{obj } \{\ m = o_2.m \circ o_1.m\ \}_{m \in X}$$

We will also find it convenient to use ι in the rôle of a unit type, using $*$ in place of a binding variable and $()$ as the unit value, standing for some unimportant ι value.

6.2 Coding

For any type τ, a strategy of that type is simply a function from sequences of moves to moves. Given an encoding of moves of $[\![\tau]\!]$ as natural numbers

$$\Psi_\tau \colon M_{[\![\tau]\!]} \to \mathbb{N}$$

one can thus code each effective strategy as a program of a suitable function type, by adding a constant representing the strategy's underlying function to our set of function names. We choose to use the type

$$\rho \quad = \quad \iota \Rightarrow \iota \quad = \quad \mathbf{Obj}\,\{m \colon \iota \to \iota\}$$

The program e representing a strategy σ is thus a *stateful* one which given an opponent move responds with the appropriate player move for σ given the sequence of moves which e has seen so far. It is easy to write such a program by creating an object with an integer state, given a bijection $\mathbb{N}^* \cong \mathbb{N}$.

Since

$$[\![\iota \Rightarrow \iota]\!] = !(\&_{\mathbb{N}}\Sigma_{\mathbb{N}}1)$$

we refer to the moves of this game in component i with value n as q_i^n or a_i^n.

Now consider the relation between the game $[\![\tau]\!]$ and its encoding at the above type. One can imagine a strategy in each direction translating the move m to $\Psi(m)$ and vice versa.

However, only one direction results in well-bracketed play. We define a morphism $\mathit{codes}_\tau \colon [\![\iota \Rightarrow \iota]\!] \to [\![\tau]\!]$ as follows:

$$\begin{aligned} \mathit{codes}_\tau(sa_i^{\Psi(m_1)}m_2) &= q_{i+1}^{\Psi(m_2)} \\ \mathit{codes}_\tau(sa_i^{\Psi(m_1)}) &= m_1 \qquad s\!\upharpoonright_{[\![\tau]\!]}\ m_1 \in [\![\tau]\!] \end{aligned}$$

Note that the restriction $s\!\upharpoonright_{[\![\tau]\!]}\ m_1 \in [\![\tau]\!]$ avoids copying moves from the generic game that are not permissible in the game $[\![\tau]\!]$; removal of this restriction results in plays which violate bracketing or simply the basic rules of the game.

The property of the desired programs $\mathit{interpret}_\tau$ above is then that

$$\forall(\emptyset \vdash e \colon \iota \Rightarrow \iota).\ \mathit{codes}_\tau \circ [\![e]\!] = [\![\mathit{interpret}_\tau\ e]\!]$$

We now construct a term $\hat{\sigma} \colon \iota \Rightarrow \iota$ such that $codes \circ [\![\hat{\sigma}]\!] = \sigma$. If σ is an effective strategy, we can assume a function φ_σ such that $\varphi_\sigma(\Psi^*(t)) = \Psi(a) \Leftrightarrow \sigma(t) = a$, where Ψ^* extends Ψ to a coding on sequences. Then

$$\hat{\sigma} = \textbf{constr}\ \Psi^*(\varepsilon)\ \textbf{obj}\ \{m = \lambda\langle t, a\rangle.\ \textbf{let}\ b\ \textbf{be}\ c_{\varphi_\sigma}(t \cdot a)\ \textbf{in}\ \langle t \cdot a \cdot b, b\rangle\}$$

where $\cdot$ is a function such that $\Psi^*(t) \cdot \Psi(a) = \Psi^*(ta)$.

The existence of these programs means that the above property of $interpret_\tau$ gives definability at τ. It should be noticed that the use of state in $\hat{\sigma}$ explains why we will be able to define *interpret* programs for some types without using **new** which nevertheless cater correctly for strategies involving some stateful behaviour.

6.3 The "interpret" programs

Here we define a family of programs $interpret_\tau$ parametrised over types τ, where

$$interpret_\tau \colon (\iota \Rightarrow \iota) \to \tau$$

The definition of $interpret_\tau$ at higher types quickly becomes rather complex, so we shall give definitions at successively higher types, incrementally adding features to the programs to deal with the increasing range of possible behaviour. Figure 6.1 shows the language constructs and behaviours required at each of these types.

In each case, $interpret_\tau$ must construct an object of type τ from an argument which is an encoding of a strategy for $[\![\tau]\!]$, so that the denotation of the constructed object is the strategy being simulated. At higher types we give programs for types involving variables $X, Y, Z, \ldots$—these range over types for which the appropriate *interpret* program has already been defined, and will use that program recursively to construct the result. The definitions below should be thought of as an informal meta-program which, given some concrete type τ, constructs a program of our language by repeatedly expanding the appropriate definition of *interpret* according to the structure of τ.[2]

It is notable that the *interpret* program can be defined by such an expansion. In [58] Longley and Plotkin similarly construct programs to interpret a coding of some term as the corresponding term itself. In that case the coding is a Gödel-numbering, and the interpreting program has to be defined as a mutual recursion over all the types involved. In a sense our programs perform a similar

[2]To do this in the language itself would require some form of polytypic programming.

1	ι	method invocation, arithmetic.
2	$\iota \to X$	1 + **obj** $\{\ldots\}$, λ, **ifz** .
3	$\iota \Rightarrow X$	2 + **constr** , $\langle -, - \rangle$, **let** $\langle x, y \rangle$ **be** $-$.
4	$(X \Rightarrow \iota) \to Y$	3 + **Y** (with nested method calls).
5	$(X \Rightarrow (Y \Rightarrow \iota)) \to Z$	4 + object state, . . . —not handled here.

Figure 6.1: Language features required for $\mathit{interpret}_\tau$ at various types τ

recursion "lazily": each $\mathit{interpret}_\tau$ program is recursively defined, and any use of $\mathit{interpret}_X$ for a smaller type X is with an argument object constructed from both the strategy $\hat{\sigma}$ being interpreted and a recursive use of the $\mathit{interpret}_\tau$ program itself.[3] We are able to be lazy here because we interpret strategies of type $\iota \Rightarrow \iota$ rather than an encoding of type ι.

Interpreting ι

Most simply, $\mathit{interpret}_\iota$ must return a number (type ι) from the encoding of a strategy representing that number. Given coding functions

$$q_\iota : \iota \qquad a_\iota : \iota \to \iota$$

satisfying

$$q_\iota = \Psi_\iota(q) \qquad a_\iota(\Psi_\iota(a^n)) = n$$

we define

$$\mathit{interpret}_\iota(\hat{\sigma}) = a_\iota(\hat{\sigma} \bullet q_\iota)$$

This program satisfies the required property, since if $\sigma(q) = \perp$, $[\![\hat{\sigma} \bullet q_\iota]\!] = \perp$ and hence $[\![a_\iota(\hat{\sigma} \bullet q_\iota)]\!](q) = \perp$, and if $\sigma(q) = a^n$, $[\![\hat{\sigma} \bullet q_\iota]\!] = [\![\Psi_\iota(a^n)]\!]$ and hence $[\![a_\iota(\hat{\sigma} \bullet q_\iota)]\!](q) = n$.

Interpreting $\iota \to X$

We now consider types of the form $\iota \to X$, where X is a type for which $\mathit{interpret}_X$ is defined. While this is a linear and not an intuitionistic type, we present this definition as a suitable stepping stone towards the types we are interested in, and

[3]This is a simplification, as it is not $\mathit{interpret}_\tau$ which is recursively defined but some function within its definition.

an aid to understanding those. Given coding functions

$$\begin{array}{rclrclrcl} q_* & : & \iota & \mathit{isq}_X & : & \iota \to \iota & \mathit{out}_X & : & \iota \to \iota \\ q_X & : & \iota \to \iota & & & & \mathit{in}_X & : & \iota \to \iota \end{array}$$

satisfying

$$\begin{array}{rclrcl} q_* & = & \Psi_{\iota\to X}(q) & \mathit{out}_X(\Psi_{\iota\to X}(x)) & = & \Psi_X(x) \\ q_X(n) & = & \Psi_{\iota\to X}(q^n) & \mathit{in}_X(\Psi_X(x)) & = & \Psi_{\iota\to X}(x) \\ \mathit{isq}_X(\Psi_X(q)) & = & 0 & & & \\ \mathit{isq}_X(n) & = & 1 \quad (n \neq \Psi_X(q)) & & & \end{array}$$

define the program

$$\begin{array}{l} \mathit{interpret}_{\iota\to X}(\hat\sigma) = \\ \hat\sigma \bullet q_*;\ \lambda n. \\ \quad \mathit{interpret}_X(\mathbf{obj}\ \{m = \lambda v.\ \mathbf{ifz}\ \mathit{isq}_X(v)\ \ \mathbf{then}\ \mathit{out}_X(\hat\sigma \bullet q_X(n)) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \mathbf{else}\ \mathit{out}_X(\hat\sigma \bullet \mathit{in}_X(v))\}) \end{array}$$

Given the argument n, we extract an encoded strategy of type $[\![X]\!]$ from s, from which $\mathit{interpret}_X$ constructs a genuine expression of type X. The encoded strategy is easy to extract, being simply a recoding of moves with the exception of the initial question, which carries a value in $[\![\iota \to X]\!]$ but not $[\![X]\!]_\perp$.

Interpreting $\iota \Rightarrow X$

We now move on to object types. For $\mathit{interpret}_{\iota\Rightarrow X}$ we must construct an object with a method which at each invocation extracts an appropriate strategy from $\hat\sigma$. Each method call is associated with a new component in $\hat\sigma$, so ι-type state must be used to keep track of which component is to be associated with the next invocation. Other than this, the program is the same as that for $\iota \to X$.

Given coding functions

$$\begin{array}{rclrclrcl} q_* & : & \iota & \mathit{isq}_X & : & \iota \to \iota & \mathit{out}_X & : & \iota \to \iota \\ q_X & : & \iota \otimes \iota \to \iota & & & & \mathit{in}_X & : & \iota \otimes \iota \to \iota \end{array}$$

satisfying

$$\begin{array}{rclrcl} q_* & = & \Psi_{\iota\Rightarrow X}(q) & \mathit{out}_X(\Psi_{\iota\Rightarrow X}(x_i)) & = & \Psi_X(x) \\ q_X(i,n) & = & \Psi_{\iota\Rightarrow X}(q_i^n) & \mathit{in}_X(i, \Psi_X(x)) & = & \Psi_{\iota\Rightarrow X}(x_i) \\ \mathit{isq}_X(\Psi_X(q)) & = & 0 & & & \\ \mathit{isq}_X(n) & = & 1 \quad (n \neq \Psi_X(q)) & & & \end{array}$$

define the program

$$\begin{aligned}&\mathit{interpret}_{\iota\Rightarrow X}(\hat{\sigma}) =\\ &\quad \hat{\sigma}\bullet q_*;\ \textbf{constr}\ 0\ \textbf{obj}\ \{\lambda\langle i,n\rangle.\ \langle i+1, \mathit{interpret}_X(f(\hat{\sigma},i,n))\rangle\}\end{aligned}$$

where

$$\begin{aligned}f(\hat{\sigma},i,n) = \textbf{obj}\ \{m = \lambda v.\ \textbf{ifz}\ \mathit{isq}_X(v)\ \textbf{then}\ \mathit{out}_X(\hat{\sigma}\bullet q_X\langle i,n\rangle)\\ \textbf{else}\ \mathit{out}_X(\hat{\sigma}\bullet \mathit{in}_X\langle i,v\rangle)\end{aligned}$$

Here $f(\hat{\sigma},i,n)$ is simply factored out for space and convenience. It should also be noted that while the argument to out_X could be some number not representing a move the correct X component, this will not occur with any $\hat{\sigma}$ coding some strategy σ simply by the switching condition of the game $![\![\iota \to X]\!]$.

Now if $\sigma(tq_j^m) = a_j^x$ we show $[\![\mathit{interpret}(\hat{\sigma})]\!](tq_j^m) = a_j^x$. Note that by the interpretation of **constr**, it is the case that

$$\begin{aligned}[\![\hat{\sigma}\vdash \textbf{constr}\ 0\ \textbf{Obj}\ \{\lambda\langle i,n\rangle.\langle i+1, \mathit{interpret}_X(f\langle\hat{\sigma},i,n\rangle)\rangle\}| =\\ \lambda(\delta; \textstyle\bigotimes_i [\![\hat{\sigma}\vdash \lambda n.\mathit{interpret}_X(f\langle s,i,n\rangle)]\!])\end{aligned}$$

So

$$[\![\mathit{interpret}(\hat{\sigma})]\!](tq_j^m) = \big([\![\hat{\sigma}]\!];\delta;\bigotimes_i \&_n[\![\hat{\sigma}\vdash \mathit{interpret}_X f\langle\hat{\sigma},i,n\rangle]\!]\big)(tq_j^m)$$

Since $\mathit{interpret}_X$ is given a view of component j of $\hat{\sigma}$, we are interested in the restriction of the play in question to component j. We can always construct a strategy containing this play—but this is dependent on the particular play in question, and σ may contain plays leading to various different strategies for component j (i.e. there may be interference). Define

$$\begin{aligned}\sigma_i^t(q_i) = a_i^v &\iff \sigma(tq_i) = a_i^v\\ \sigma_i^t(t{\restriction_i}\ \ x_i) = x_i' &\iff \sigma(tx_i) = x_i'\end{aligned}$$

and note that

$$([\![\hat{\sigma}]\!];\delta;\bigotimes_i f_i)(tx) = \bigotimes_i [\![\hat{\sigma_i^t}]\!]; f_i$$

Then

$$\begin{aligned}\big([\![\hat{\sigma}]\!];\delta;\textstyle\bigotimes_i \&_n[\![\hat{\sigma}\vdash \mathit{interpret}_X f\langle\hat{\sigma},i,n\rangle]\!]\big)(tq_j^m) =\\ \big(\textstyle\bigotimes_i \&_n[\![\hat{\sigma_i^t}]\!];[\![\hat{\sigma}\vdash \mathit{interpret}_X f\langle\hat{\sigma},i,n\rangle]\!]\big)(tq_j^m)\end{aligned}$$

Now define

$$\begin{aligned}\sigma_{i,n}^t(q) = a^v &\iff \sigma(tq_i^n) = a_i^v\\ \sigma_{i,n}^t(t{\restriction_i}\ \ x) = x' &\iff \sigma(tx_i) = x_i'\end{aligned}$$

and note that $[\![f\langle\hat\sigma, i, n\rangle]\!] = [\![\sigma^t_{i,n}]\!]$, so

$$\begin{array}{rl} & \big(\bigotimes_i \&_n [\![\hat{\sigma^t_i}]\!]; [\![\hat\sigma \vdash \mathit{interpret}_X f\langle\hat\sigma, i, n\rangle]\!]\big)(tq^m_j) \\ = & \big(\bigotimes_i \&_n [\![\hat\sigma \vdash \mathit{interpret}_X \hat{\sigma^t_i}]\!]\big)(tq^m_j) \\ = & a^x_j \end{array}$$

Thus $\mathit{interpret}_{\iota\Rightarrow X}$ satsifies the required property.

Interpreting $(X \Rightarrow \iota) \to Y$

We can now give a definition for $(X \Rightarrow \iota) \to Y$. The ideas required to handle strategies of this type combine in a straightforward fashion with that from $\iota \Rightarrow X$ above to give a program for types of the form $(X \Rightarrow \iota) \Rightarrow Y$, but we refrain from doing so here in order to reduce clutter and simplify the presentation.

The new feature at this type is the interesting argument type $X \Rightarrow \iota$. We shall define a program which, given an argument g of that type, and where $\hat\sigma$ has requested the value of some component of $[\![g]\!]$, constructs an expression of type X with which to supply g, again by using $\mathit{interpret}_X$ with an argument derived from interaction with $\hat\sigma$. When g finishes its interaction with this argument, and returns some number n we have the result to supply to $\hat\sigma$ as the answer of $[\![g]\!]$.

The response of $\hat\sigma$ at this point may be a move in Y (which we may simply pass on to $\mathit{interpret}_Y$) or another question of $[\![g]\!]$. In this case we must go through the same process again, and repeatedly so until $\hat\sigma$ responds in Y, resulting in the need for the recursively defined function f in the below definition.

There is also a possibility for nested method calls during interaction in X, as would occur when $\hat\sigma$ is the encoding of the strategy for a program which supplies g an argument which on evaluation makes use of a further call to g. The following is such a program, with $X = (\iota \Rightarrow \iota)$ and $Y = \iota$, the simplest (intuitionistic) type at which this occurs:

$$\sigma = \left[\!\!\left[\begin{array}{l} \lambda g\colon ((\iota \Rightarrow \iota) \Rightarrow \iota). \\ \quad g\ (\mathbf{obj}\ \{\ m = \lambda x.\ 2 * g\ (\mathbf{obj}\ \{m = \lambda y.\ y + 1\})\ \}) \end{array} \right]\!\!\right]$$

An example play from σ, is as follows:

$$\begin{array}{llll}
((\iota \Rightarrow & \iota) \Rightarrow & \iota) \rightarrow & \iota \\
 & & & q \\
 & & q_0 & \\
 & q^3_{0,0} & & \\
 & & q_1 & \\
 & q^4_{1,0} & & \\
 & a^5_{1,0} & & \\
 & & a^{24}_1 & \\
 & a^{42}_{0,0} & & \\
 & & a^{24}_0 & \\
 & & & a^{24}
\end{array}$$

This play might arise from g playing the following stateful function:

$$\left[\!\!\left[\begin{array}{l} \textbf{let } \mathit{flag} \textbf{ be constr } 0 \textbf{ obj } \{m = \lambda\langle n, *\rangle.\ \langle 1, n\rangle\} \\ \quad \textbf{in obj } \{m = \lambda f.\ \textbf{ifz } \mathit{flag} \bullet () \textbf{ then } f \bullet 3 \textbf{ else } f \bullet 4\} \end{array} \right]\!\!\right]$$

Notice that an auxiliary integer state cell is used to give the behaviour of g, as the state must be updated before the nested call rather than after the outer call.

The recursive definition of f also accounts for this, as we make use of $\mathit{interpret}_X$ with an argument constructed from f as well as $\hat{\sigma}$. This argument behaves much as the one constructed for $\mathit{interpret}_Y$, with moves in X simply being recoded (in this case involving the use of the index j) while moves corresponding to nested invocations of g are handled by a recursive invocation of f.

The function f thus behaves as follows. When given some move by $\hat{\sigma}$, f gives the next move in the *same game* (i.e. Y or some particular copy of X) after any nested interaction with g—just what is needed for $\mathit{interpret}_Y$ or $\mathit{interpret}_X$.

Given coding functions

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
q_* & : & \iota & \mathit{isq}_X & : & \iota \rightarrow \iota & \mathit{out}_X & : & \iota \rightarrow \iota \\
\mathit{ans}_\iota & : & \iota \otimes \iota \rightarrow \iota & \mathit{is}_X & : & \iota \rightarrow \iota & \mathit{in}_X & : & \iota \rightarrow \iota \\
\mathit{qval} & : & \iota \rightarrow \iota & \mathit{is}_Y & : & \iota \rightarrow \iota & \mathit{out}_Y & : & \iota \rightarrow \iota \\
\mathit{index} & : & \iota \rightarrow \iota & & & & \mathit{in}_Y & : & \iota \rightarrow \iota
\end{array}$$

satisfying

$$\begin{array}{rcll rcll}
q_* & = & \Psi_{(X\Rightarrow\iota)\to Y}(q) & & out_X(\Psi_{(X\Rightarrow\iota)\to Y}(x)) & = & \Psi_X(x) & \\
ans_\iota(i,n) & = & \Psi_{(X\Rightarrow\iota)\to Y}(a_i^n) & & in_X(\Psi_X(x)) & = & \Psi_{(X\Rightarrow\iota)\to Y}(x) & \\
isq_X(\Psi_X(q)) & = & 0 & & out_Y(\Psi_{(X\Rightarrow\iota)\to Y}(y)) & = & \Psi_Y(y) & \\
isq_X(n) & = & 1 & (n \neq \Psi_X(q)) & in_Y(\Psi_Y(y)) & = & \Psi_{(X\Rightarrow\iota)\to Y}(y) & \\
isq_Y(\Psi_Y(q)) & = & 0 & & qval(\Psi_{(X\Rightarrow\iota)\to Y}(q^x)) & = & \Psi_X(a^x) & \\
isq_Y(n) & = & 1 & (n \neq \Psi_Y(q)) & index(\Psi_{(X\Rightarrow\iota)\to Y}(q_i^x)) & = & i & \\
is_X(\Psi_{(X\Rightarrow\iota)\to Y}(x)) & = & 0 & & is_Y(\Psi_{(X\Rightarrow\iota)\to Y}(y) & = & 0 & \\
is_X(n) & = & 1 & (n \neq \Psi_{\dots}(x)) & is_Y(n) & = & 1 & (n \neq \Psi_{\dots}(y))
\end{array}$$

define the program

```
$interpret_{(X\Rightarrow\iota)\to Y}(\hat\sigma) =$
  $\hat\sigma \bullet q_*;\ \lambda g\colon (X \Rightarrow \iota).$
    **let** $o = \lambda\langle\hat\sigma, j, v\rangle.$ **obj** $\left\{ m = \lambda z.\ \begin{array}{l}\textbf{ifz}\ isq_X(z)\ \textbf{then}\ qval(v) \\ \textbf{else}\ out_X(\hat\sigma \bullet (in_X(j,z)))\end{array} \right\}$
    **in letrec** $f \bullet v =$
      **ifz** $is_Y(v)$ **then** $out_Y(v)$
      **else ifz** $is_X(v)$ **then** $out_X(v)$
      **else let** $j$ **be** $index(v)$ **in**
        $f \bullet (\hat\sigma \bullet (ans_\iota\langle j, g \bullet (interpret_X(o\langle f \circ \hat\sigma, j, v\rangle))\rangle))$
    **in** $interpret_Y(f \circ \hat\sigma \circ in_Y)$
```

We prove correctness by induction on length of plays via the following property:

$$\begin{aligned}
P(n) &= \forall t(\text{with } length(t) = n).\ \forall\sigma. \\
&\quad \sigma(m_1 t) = m_2 \Rightarrow [\![g \vdash f \circ \hat\sigma]\!](q^{\Psi(m_1)}\Psi^\bullet(m_1,t)) = \begin{cases} a^{\Psi(m_2)} & (sc(m_1,m_2)) \\ m_2 & (\text{otherwise}) \end{cases} \\
&\quad \sigma(m_1 t) = \bot \Rightarrow [\![g \vdash f \circ \hat\sigma]\!](q^{\Psi(m_1)}\Psi^\bullet(m_1,t)) = \bot
\end{aligned}$$

where $sc(m_1, m_2)$ if m_1 and m_2 are in the "same component" in $(X \Rightarrow \iota) \to Y$, i.e. are both in Y or in the same copy of X. This means m_2 is in a sense a response to m_1, but they need not be a question/answer pair. Also define

$$\Psi^\bullet(m, m_1 t) = \begin{cases} q^{\Psi(m_1)}\Psi^\bullet(m,t) & sc(m,m_1) \text{ and } \lambda(m_1) = O \\ a^{\Psi(m_1)}\Psi^\bullet(m,t) & sc(m,m_1) \text{ and } \lambda(m_1) = P \\ m_1\Psi^\bullet(m,t) & \text{otherwise} \end{cases}$$

and let $\Psi^\bullet(mt) = \Psi^\bullet(m, t)$—here the m identifies the location of the component which should be encoded, and any moves elsewhere are left unchanged.

Note that the above property gives

$$[\![g \vdash \mathit{interpret}_Y(f \circ \hat{\sigma})]\!](t) = \sigma(t)$$

when combined with the property of $\mathit{interpret}_Y$, and thus we have

$$\left[\!\left[\mathit{interpret}_{(X \Rightarrow \iota) \to Y}(\hat{\sigma})\right]\!\right](t) = \sigma(t)$$

The proof splits into three parts, each of which use the inductive hypothesis in a different way. Firstly, a move in Y or X with an immediate response in that same component, or an answer in ι with a response in X or Y, corresponds to a single invocation of f, and any further play is simply a shorter play and hence handled correctly by subsequent invocations of f.

Secondly, an unclosed call to g comprises some shorter play in $\mathit{interpret}_Y(...)$ (or $\mathit{interpret}_X(...)$ if nested), which is handled by a nested use of f. The above property holds of the nested f, and by the property of the nested *interpret* the "decoded" version holds of $\mathit{interpret}(\ldots f \ldots)$, giving the property for the outer f.

Thirdly, a completed call to g results in a tail-call of f, and the shorter play starting from this point is thus correctly handled by f.

We start with the case of a response in X or Y. Assume $\sigma(m_1) = m_2$ with $m_2 \colon X$ or $m_2 \colon Y$. Then as f is the identity here, $[\![g \vdash f \circ \hat{\sigma}]\!](q^{\Psi(m_1)}) = a^{\Psi(m_2)}$. Otherwise if $\sigma(m_1) = \bot$ then $[\![f \circ \hat{\sigma}]\!](q^{\Psi(m_1)}) = \bot$. We shall omit consideration of any further cases where $\sigma(tm_1) = \bot$, as it is always the case that we evaluate $[\![\hat{\sigma}]\!](q^{\Psi(m_1)})$ for each move m_1, and as this is $\bot$ the expression in question will be too.

Now assume $\sigma(m_1 m_2 t) = m'$. There is another strategy $\sigma_{m_1 m_2}$ with $\sigma_{m_1 m_2}(t) = \sigma(m_1 m_2 t)$. Since $[\![f]\!]_{q^{\Psi(m_1)} a^{\Psi(m_2)}} = [\![f]\!]$ and $[\![\hat{\sigma}]\!]_{q^{\Psi(m_1)} a^{\Psi(m_2)} t} = \left[\!\left[\widehat{\sigma_{m_1 m_2}}\right]\!\right]$

$$[\![f \circ \hat{\sigma}]\!]_{q^{\Psi(m_1)} a^{\Psi(m_2)}}(t) = \left[\!\left[f \circ \widehat{\sigma_{m_1 m_2}}\right]\!\right](t)$$

and thus the property holds.

We now move on to a call to g. If $\sigma(m_1) = q$ then examination of $f(\Psi(q))$ reveals that by evaluation order the initial question is asked to determine the value of g. So $[\![g \vdash f \circ \hat{\sigma}]\!](m_1) = q$. For continued play in the argument to g, assume $\sigma(m_1 q t) = m_2$ (where t does not answer q). There is a strategy σ' with

$\sigma'(t) = \sigma(m_1qt)$; by the inductive hypothesis $[\![g \vdash f \circ \hat{\sigma}']\!](\Psi^\bullet(t)) = m_2$ (or $a^{\Psi(m_2)}$), so $[\![g \vdash \mathit{interpret}(f \circ \hat{\sigma})]\!](t) = m_2$ (in either case). Thus $[\![g \vdash f \circ \hat{\sigma}]\!](m_1qt) = m_2$.

Now we reach an answer to g. Note that where g returns a value a, f constructs the coded answer $\Phi(a)$ and returns $(f \circ \hat{\sigma})(\Phi(a))$. So

$$[\![f \circ \hat{\sigma}]\!](m_1qtm_2a) = \left[\!\left[f \circ \widehat{\sigma_{m_1qtm_2}}\right]\!\right](q^{\Psi(a)})$$

6.3.1 Definable types

The programs presented so far handle definability only at a subset of all possible intuitionistic product-free types. However, these types include those of arbitrarily high *rank*. Define rank as follows:

$$\mathit{rank}(\iota) = 0 \quad \mathit{rank}(\sigma \Rightarrow \tau) = \max(\mathit{rank}(\sigma) + 1, \mathit{rank}(\tau))$$

For any rank k there is a pure type $\overline{k}$ of that rank defined as follows:

$$\overline{0} = \iota \qquad \overline{k+1} = \overline{k} \Rightarrow \iota$$

The above interpret programs give definability for these types. However, we can go much further. The following grammar captures precisely the subset of the intuitionistic product-free types for which we have definability:

$$\begin{array}{rcl} T & ::= & \iota \mid (\mathit{ArgT} \Rightarrow T) \\ \mathit{ArgT} & ::= & \iota \mid (T \Rightarrow \iota) \end{array}$$

This grammar shows that *parity* is important—arrows nested at even levels on the left can have general return types, whereas arrows nested at odd levels may only return ι. These types could also be characterised as those (intuitionistic, product free) types which contain no type of the form $T \Rightarrow (T \Rightarrow T)$ at an odd-rank position.

It is rather unusual that we have definability at types of arbitrary depth (i.e. rank) but bounded width, as the more normal situation is that every type is a definable retract of some pure type $\overline{k}$. The terms which would normally define these retractions do not compose to give the identity in our setting, because our model is so intensionally fine-grained.

6.3.2 Issues at more complex types

We shall now discuss a proposed *interpret* program for the type

$$(X \Rightarrow (Y \Rightarrow \iota)) \rightarrow Z$$

Inspection of the shortcomings of this program will reveal underlying issues with regards to the present language definition. It is notable that the program comes rather close to performing its intended purpose; it is also the case that (if correct) the program would seem to contain all the ideas needed to handle *all* product-free intuitionistic types, i.e. those of the form $[X \Rightarrow (Y \Rightarrow (\ldots \Rightarrow \iota))] \Rightarrow Z$.

The main addition in the program below is that on receiving a result of type $(Y \Rightarrow \iota)$ from the argument g this program must store that result for use at some arbitrary later point (or points), where the previous program could just return a ground-type answer once and for all.

In order to store a number of such objects, one can maintain an accumulator, using the following implementation of a functional *vector*:

$$\begin{aligned}
\mathit{vec}[\tau] &= (\iota \Rightarrow \tau)\\
\mathit{empty} &= \mathbf{obj}\ \{\lambda n.\bot\}\\
\mathit{update} &: \mathit{vec}[\tau] \otimes \tau \otimes \iota \to \mathit{vec}[t]\\
&= \lambda\langle f, x, n\rangle.\ \mathbf{obj}\ \{\mathit{get} = \lambda m.\mathbf{ifz}\ m = n\ \mathbf{then}\ x\ \mathbf{else}\ f \cdot m\}
\end{aligned}$$

For convenience we use the following syntactic sugar below:

$$\begin{aligned}
[] &= \mathit{empty}\\
e[n] &= e \cdot \mathit{get}(n)\\
e[n \mapsto e_1] &= \mathit{update}\langle e, e_1, n\rangle
\end{aligned}$$

We can then replace the recursive definition of f with a definition using the **constr** operator, maintaining an accumulator of a vector type and saving this as state between calls to f. There is some subtlety in this definition. It is not possible to correctly bind f recursively within its definition,[4] and instead we create a *clone* of the implementation f_{impl} as f' for the recursive use within $\mathit{interpret}_X$ and $\mathit{interpret}_Y$; the consequences of this are discussed below.

Given coding functions for $W = ((X \Rightarrow (Y \Rightarrow \iota)) \to Z)$, and for each of $T \in \{X, Y, Z\}$:

[4]This would result in the creation of a fresh object each recursion, losing the accumulated state.

```
$\mathit{interpret}_{(X \Rightarrow (Y \Rightarrow \iota)) \to Z}(\hat{\sigma}) =$
  $\hat{\sigma} \bullet q;\ \lambda g \colon X \Rightarrow (Y \Rightarrow \iota).$

—Helper functions $o$ and $o'$ generate object for uses of $\mathit{interpret}$.
—Note $s$ will be $f' \circ \hat{\sigma}$
  $\textbf{let } o = \lambda\langle s, j, v\rangle.\ \textbf{obj} \left\{ m = \lambda z. \begin{array}{l} \textbf{ifz } \mathit{isq}_X(z) \textbf{ then } \mathit{qval}_X(v) \\ \textbf{else } \mathit{out}_X(s \bullet (\mathit{in}_X(j, z))) \end{array} \right\}$
  $\textbf{in let } o' = \lambda\langle s, j, k, v\rangle.\ \textbf{obj} \left\{ m = \lambda z. \begin{array}{l} \textbf{ifz } \mathit{isq}_Y(z) \textbf{ then } \mathit{qval}_Y(v) \\ \textbf{else } \mathit{out}_Y(s \bullet (\mathit{in}_Y(j, k, z))) \end{array} \right\}$
—The main definition. Construct outer $f$ from recursively defined $f_{\mathit{impl}}$:
  $\textbf{in let } f = \textbf{constr } [\,]\ (\textbf{letrec } f_{\mathit{impl}} = \textbf{obj } \{m = \lambda\langle \mathit{acc} \colon \mathit{vec}[(Y \Rightarrow \iota)], v \colon \iota\rangle.$
    $\textbf{ifz } \mathit{is}_Z(v) \textbf{ then } \langle \mathit{acc}, \mathit{out}_Z(v)\rangle$
    $\textbf{else ifz } \mathit{is}_Y(v) \textbf{ then } \langle \mathit{acc}, \mathit{out}_Y(v)\rangle$
    $\textbf{else ifz } \mathit{is}_X(v) \textbf{ then } \langle \mathit{acc}, \mathit{out}_X(v)\rangle$

—Case for outer $\Rightarrow$. Clone $f$ as $f'$, call argument $g$ using $\mathit{interpret}_X$ with $f'$
—and store result $o \colon Y \Rightarrow \iota$ under key $j$ (the outer $\Rightarrow$ index) in the most
—recent store from $f'$, then recurse:
    $\textbf{else ifz } \mathit{isq}_1(v) \textbf{ then let } j \textbf{ be } \mathit{index}_1(v) \textbf{ in}$
      $f_{\mathit{impl}} \bullet \langle\ \textbf{let } f' \textbf{ be constr } \mathit{acc}\ f_{\mathit{impl}} \textbf{ in}$
                $\textbf{let } o \textbf{ be } g \bullet \mathit{interpret}_X(o\langle f' \circ \hat{\sigma}, j, v\rangle)$
                $\textbf{in } (f' \cdot \mathit{acc}())[j \mapsto o],\quad \hat{\sigma} \bullet \mathit{ans}_{(Y \Rightarrow \iota)}(j)\ \rangle$

—Case for inner $\Rightarrow$. Clone $f$ as $f'$, pick up $o \colon Y \Rightarrow \iota$ stored in previous case
—as $\mathit{acc}[j]$ and call $o$, with no need to store the returned value $n$, then recurse:
    $\textbf{else ifz } \mathit{isq}_2(v) \textbf{ then let } \langle j, k\rangle \textbf{ be } \mathit{index}_2(v) \textbf{ in}$
      $f_{\mathit{impl}} \bullet \langle \mathit{acc},\quad \textbf{let } f' \textbf{ be constr } \mathit{acc}\ f_{\mathit{impl}}$
                  $\textbf{in let } n \textbf{ be } \mathit{acc}[j] \bullet (\mathit{interpret}_Y(o'\langle f' \circ \hat{\sigma}, j, k, v\rangle))$
                  $\textbf{in } \hat{\sigma} \bullet (\mathit{ans}_\iota\langle j, k, n\rangle)\ \rangle$
    $\textbf{else } \langle \mathit{acc}, 0\rangle,$—Default case, unused.

—The read method $\mathit{acc}$ extracts the state from $f'$ in $f$:
    $\mathit{acc} = \lambda\langle \mathit{acc}, *\rangle.\ \langle \mathit{acc}, \mathit{acc}\rangle\})$
  $\textbf{in } \mathit{interpret}_Z(f \circ \hat{\sigma} \circ \mathit{in}_Z)$
```

Figure 6.2: Extended *interpret* program.

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
q_* & : & \iota & isq_T & : & \iota \to \iota & out_T & : & \iota \to \iota \\
ans_\iota & : & \iota \otimes \iota \otimes \iota \to \iota & isq_1 & : & \iota \to \iota & in_T & : & \iota \to \iota \\
ans_{(Y \Rightarrow \iota)} & : & \iota \to \iota & isq_2 & : & \iota \to \iota & index_1 & : & \iota \to \iota \\
qval_T & : & \iota \to \iota & & & & index_2 & : & \iota \to \iota \otimes \iota
\end{array}$$

satisfying

$$\begin{array}{rcl@{\qquad}rcl}
q_* & = & \Psi_W(q) & in_T(\Psi_T(m)) & = & \Psi_W(m) \\
ans_\iota(i,j,n) & = & \Psi_W(a^n_{i,j}) & out_T(\Psi_W(m)) & = & \Psi_T(m) \\
ans_{(Y\Rightarrow\iota)}(i) & = & \Psi_W(a^*_i) & qval_T(\Psi_W(q^v)) & = & \Psi_T(a^v) \\
index_1(\Psi_W(q^x_i)) & = & i & isq_T(\Psi_T(q)) & = & 0 \\
index_2(\Psi_W(q^y_{i,j})) & = & \langle i,j\rangle & isq_T(n) & = & 1 \quad (n \neq \Psi_T(q)) \\
isq_1(\Psi_W(q^x_i)) & = & 0 & isq_2(\Psi_W(q^y_{i,j})) & = & 0 \\
isq_1(n) & = & 1 \quad (n \neq \Psi_W(q^x_i)) & isq_2(\Psi_W(n)) & = & 1 \quad (n \neq \Psi_W(q^y_{i,j}))
\end{array}$$

define the program in Figure 6.2.

Before discussing the limitations of this program, we shall illustrate the wide range of strategies for which it suffices. We shall give some plays at the type

$$((\iota \to \iota) \Rightarrow (\iota \Rightarrow \iota)) \to \iota$$

We use the type $(\iota \to \iota)$ rather than $(\iota \Rightarrow \iota)$ for X for simplicity of notation, but one should bear in mind that it is the latter type which is under consideration. A move q^3_0 in the below is a question carrying value 3 in component 0 of the ! corresponding to the outer $\Rightarrow$; similarly moves with two subscripts indicate the components of the !'s corresponding to the outer and inner $\Rightarrow$ (in that order).

Consider the following strategy:

$$\sigma = [\![\lambda g.\ \mathbf{let}\ o\ \mathbf{be}\ g \bullet \mathbf{obj}\ \{m = \lambda x.x+1\}\ \mathbf{in}\ o \bullet 1 + o \bullet 2]\!]$$

An interaction with an argument such as

$$g = [\![\lambda z.\ \mathbf{let}\ n\ \mathbf{be}\ z \bullet 3\ \mathbf{in}\ \mathbf{obj}\ \{m = \lambda]\!] n.n + z\}$$

gives the following play:

$$\begin{array}{cccc}
\multicolumn{4}{c}{(\iota \to \iota) \Rightarrow (\iota \Rightarrow \iota) \to \iota} \\
 & & & q \\
 & q_0 & & \\
q_0^3 & & & \\
a_0^4 & & & \\
 & a_0 & & \\
 & & q_{0,0}^1 & \\
 & & a_{0,0}^5 & \\
 & & q_{0,1}^0 & \\
 & & a_{0,1}^4 & \\
 & & & a^9
\end{array}$$

It can be seen by stepping through the definition of interpret that this play is indeed in $[\![\mathit{interpret}(\hat{\sigma})]\!]$. In particular, f stores the object returned by g, and twice extracts that object from the state and calls a method on it, and as directed by σ adds the results to obtain the result value.

The following play illustrates more complex behaviour which is still correctly handled by *interpret*:

$$\begin{array}{cccc}
\multicolumn{4}{c}{(\iota \to \iota) \Rightarrow (\iota \Rightarrow \iota) \to \iota} \\
 & & & q \\
 & q_0 & & \\
q_0^{37} & & & \\
 & q_1 & & \\
 & a_1 & & \\
a_0^{37} & & & \\
 & a_0 & & \\
 & & q_{1,0}^{42} & \\
 & & a_{1,0} & \\
 & & & a
\end{array}$$

This play is included in the strategy

$$\begin{aligned}
\sigma = [\![\lambda g.\ \ & \mathbf{let}\ f\ \mathbf{be\ constr}\ \bot \\
& \quad \mathbf{obj} \left\{ \begin{array}{rcl} m & = & \lambda\langle s, x\rangle.\ \langle g \bullet (\mathbf{obj}\ \{m = \lambda x.\ x\}), x\rangle, \\ \mathit{read} & = & \lambda\langle s, \_\rangle.\ \langle s, s\rangle \end{array} \right\} \\
& \mathbf{in}\ g \bullet (f);\ f \cdot \mathit{read}() \bullet 42 \quad]\!]
\end{aligned}$$

Note that the above program for σ calls g with an argument which itself calls g on a call of its method m. The typing derivation makes crucial use of subtyping, to assign f the type $(\iota \Rightarrow \iota)$ in order to pass it to g.

The interest in the above play is twofold. Firstly, *interpret* is able to handle strategies containing nested calls. Secondly, the object returned by one of those nested calls is available *after* that call has terminated (due to the expression $f' \bullet acc$ extracting the resulting state of the nested call). Indeed, the object is available after other objects have been stored as a result of the containing call. This could be thought of as a violation of some weak bracketing property—in a sense $q^{42}_{1,0}$ refers to the response a_1—but of course the play above is a valid fully bracketed play in our setting.

We now come to the end of the line. There are further extensions to *interpret* which will admit further plays similar to the above. However, the complexity of these alterations still does not seem to buy us full generality. The following play does *not* elicit a correct response by *interpret*:

$$\begin{array}{ccccc}
(\iota \rightarrow \iota) & \Rightarrow & (\iota \Rightarrow \iota) & \rightarrow & \iota \\
 & & & q & \\
 & q_0 & & & \\
q^n_0 & & & & \\
 & q_1 & & & \\
 & a_1 & & & \\
a^m_0 & & & & \\
 & a_0 & & & \\
 & & q^n_{1,0} & & \\
q^n_1 & & & & \\
 & & q^n_{0,0} & & \\
 & & a^m_{0,0} & & \\
a^m_1 & & & & \\
 & & a^m_{1,0} & & \\
 & & & a^m &
\end{array}$$

Not only does *interpret* not cater for the above play, but we are at present not able to directly give a program for any strategy containing this play. We conjecture that there is in fact no such program in our language, and dealing with this behaviour will require some language extension.

It is not obvious what goes wrong here; the above play appears to be the shortest violating example, and here the problem occurs at the tenth move $q^n_{0,0}$. The play up until just before this point is as the previous example, the program for which required a little trickery to construct, so the reader could be forgiven for thinking this is rather obscure behaviour. In terms of that program, the behaviour here corresponds to $g \bullet (f)$ updating the state of f and the method $f \cdot m$ calling g with an argument which inspects the state of f and makes use of the object there, i.e. that returned by the outer call of g. Neither of these things is possible, the former being more problematic, but it is not obvious that there is no (possibly unrelated) program which exhibits this behaviour.

Consideration of *interpret* may be more illuminating. In the move from the previous iteration of the program, the recursive use of f inside the nested *interpret* has been switched to a new object f' which starts as a duplicate of f. In a sense, if we were actually able to use f here the problem would be solved—the failure of *interpret* is the divergence of state between these two objects. It is possible for f to keep up to date with changes to f' by calling a method on f' to extract its state, since f has a reference to f'. However, f cannot update f' with new objects—f' would then be acting as a general reference cell.

The initial reason that we cannot simply use a recursive call to f is that there is no such thing at this point—f_{impl} is the only recursive binding here. One cannot simply recursively bind f within f_{impl}, for this would just mean a new object is constructed for each use of f. The only way to get a reference to the relevant object is to give the method an additional argument, so that f is passed to its own method. The problem is that we need to store the result of $g \bullet \mathit{interpret}_X(o\langle f' \circ \hat{\sigma}, j, v\rangle)$, and this would entail storing a reference to f. This is not only a captured pointer, but a circular reference.

Indeed, any method of giving f' a reference to the result object is again the same problem. Consider the execution of the above “problem play”. The result heap is structured roughly as follows, where o_i is the result of the ith call of g, and *interpret* stands in for the $\mathit{interpret}_X$ expression involving $\hat{\sigma}$ and f':

$$[f \mapsto (o_0, o_1); o_0 \mapsto \mathit{interpret}; \mathit{interpret} \mapsto f'; f' \mapsto o_1]$$

This example illustrates why we cannot collapse f and f'. If we could somehow allow f' to obtain a reference to o_0 then we would be happy in this case, since there would be no need to store that reference, but in the case of the next type

$$(X \Rrightarrow (Y \Rrightarrow (Z \Rrightarrow \iota))) \to W$$

this is no longer true.

It is natural at this point to ask for a language extension in order to achieve definability. In Chapter 3, when discussing the behaviour of nested method calls we observed that the choice to give method implementations at the type

$$\sigma \otimes \tau \Rrightarrow \sigma \otimes \tau$$

has the consequence that the state updates from nested calls are discarded. In our definition in Figure 6.2 we have managed to sidestep this in one case, where the clone f' has made an update during the initial interaction with g, but this is not a general solution. It seems possible to give method implementations at the following more generous type:

$$(1 \Rrightarrow \sigma) \otimes \tau \Rrightarrow \sigma \otimes \tau$$

Here instead of a state, the method implementation receives a *read* function (of type $1 \Rrightarrow \sigma$) allowing the state to be inspected at any point. This ability to read the state at any time would seem to give strictly more expressive method implementation, and may be a large part of what is required to prove definability. However, it appears that not all uses of such an operation are semantically sound,[5] and one would need to identify a syntactic restriction (perhaps along the lines of argument-safety) under which a read operation is permissible. It seems likely that we could also define a corresponding write operation (for values obeying the argument-safety restriction).

It certainly is the case that there is some language extension which gives the required definability result—since there is a strategy for any $interpret_\tau$, namely $\lambda(codes_\tau)$, we can add a constant in the language for each one. The real question is whether there is some *useful* construct we can add—one which is natural from a program-writing point of view, or at least illuminating, aiding understanding of the semantic behaviour and the corresponding expressive power. It may be that the constructs in question are suitably restricted *read* and *write* operations, but we must leave the investigation of this to future work.

[5]There are potential issues such as the possibility of circular references.

6.4 Full abstraction

Two expressions e and e' are *observationally equivalent* ($e \equiv e'$) if when placed in *any* context $C[-]$ they evaluate to the same value:

$$e \equiv e' \quad\Longleftrightarrow\quad \forall C[-], v.\ C[e] \Downarrow v \Leftrightarrow C[e'] \Downarrow v$$

Adequacy as discussed in the previous chapter would show that if two terms are equivalent in our games model then they are observationally equivalent:

$$[\![e]\!]_\Gamma = [\![e']\!]_\Gamma \ \Rightarrow\ e \equiv e'$$

Full abstraction is the converse, i.e. that any two observationally equivalent terms have exactly the same denotation. We shall first show how definability (for *all* types) gives the full abstraction result, before discussing the limitations described above.

We prove the result in the following form:

$$[\![e]\!]_\Gamma \neq [\![e']\!]_\Gamma \ \Rightarrow\ e \not\equiv e'$$

Firstly note that without loss of generality we can restrict attention to closed terms. Where $\Gamma = x_1 \colon \tau_1, \ldots, x_n \colon \tau_n$ one can equivalently consider instead the terms $\lambda x_1. \ldots \lambda x_n.\ e$ and $\lambda x_1. \ldots \lambda x_n.\ e'$, so below we assume $\emptyset \vdash e \colon \tau$ and $\emptyset \vdash e' \colon \tau$. Strictly speaking, to remain within the intuitionistic fragment we must consider $\mathbf{obj}\ \{\lambda x_1. ...\mathbf{obj}\ \{\lambda x_n.\ e\}...\}$ (and similarly for e'), but it is still the case that the denotation of these closed expressions differ when $[\![e]\!]_\Gamma$ and $[\![e']\!]_\Gamma$ differ.

6.4.1 Identifying indistinguishable strategies

There are some strategies in **BG** which are distinct yet arise as the denotation of two programs not distinguishable in our language. In fact these are exactly the strategies $f, g \colon 1 \to [\![\tau]\!]$ not distinguishable by any $h \colon [\![\tau]\!] \to [\![\iota]\!]_\perp$. If we try to construct a suitable h, we find that it is not well-bracketed, even though f and g are. The problem is that strategies may "chatter" before going undefined. Consider the following two programs:

$$\lambda f \colon \iota \to \iota.\ f\ 1; \perp \qquad \lambda f \colon \iota \to \iota.\ \perp$$

Both of these programs go undefined on any argument, but could be distinguished by a control operator (e.g. the simple *catch* operator). In the existing language, they are indistinguishable by a program context, so in order to achieve full abstraction, we must interpret them the same strategy. This is easily achieved by "trimming" the denotation of the former to match the latter, removing all play that cannot possibly lead to termination.

We define an operation $F(-)$ on strategies which trims them to their maximal fully bracketed plays (and prefixes thereof):

$$F(\sigma) = \{s \in \sigma \mid s \sqsubseteq^{\text{even}} t,\ t \in \sigma \text{ is fully bracketed}\}$$

We then consider full abstraction with respect to such strategies rather than **BG** (in the next section by $[\![-]\!]$ we really mean $F([\![-]\!])$).

This trimming operation seems to be that induced by the quotient construction of [7].

6.4.2 Definability to full abstraction

If $[\![e]\!] \neq [\![e']\!]$, there must be some minimal play s on which the two strategies disagree. Either there are distinct moves a and b such that $[\![e]\!](s) = a$ but $[\![e']\!](s) = b$, or there is an a such that $[\![e]\!](s) = a$ but $[\![e']\!](s) = \bot$; first consider the former situation. A strategy ($F([\![-]\!])$) only contains a play if it also contains some fully bracketed extension, so there must be sequences t_1 and t_2 with $sat_1 \in [\![e]\!]$ and $sbt_2 \in [\![e']\!]$. We construct a morphism

$$\mathit{test} \colon 1 \to [\![\tau]\!] \multimap [\![\iota]\!]_\bot$$

comprising solely the plays $\{qsat_1a^0, qsbt_2a^1\}$ (and prefixes thereof). Since sat_1 and sbt_2 are fully bracketed, test is a valid strategy in **BG**. In the situation where $[\![e']\!]$ does not respond to s, the strategy following the play $qsat_1a^0$ suffices.

By definability, there is an expression

$$\emptyset \vdash e_{\mathit{test}} \colon (\tau \Rightarrow \iota)$$

with $[\![e_{\mathit{test}}]\!] = \eta \circ !(\mathit{test})$—here we bump up the type of test to a reusable type just to remain in the intuitionistic fragment, for simplicity. By the construction of test, $[\![\emptyset \vdash e_{\mathit{test}} \bullet e]\!] = [\![0]\!]$ while $[\![\emptyset \vdash e_{\mathit{test}} \bullet e']\!] = [\![1]\!]$ (or $\bot_{[\![\iota]\!]}$)—from adequacy $\emptyset, e_{\mathit{test}}\ e \Downarrow h, 0$ and $\emptyset, e_{\mathit{test}}\ e' \Downarrow h', 1$ (or $\emptyset, e_{\mathit{test}}\ e' \not\Downarrow$). We have thus found a suitable context to distinguish e and e', so $e \not\equiv e'$.

6.4.3 Restricted full abstraction

In light of our restricted definability result, we do not actually have a fully abstract semantics *at all types*. For closed terms, we have full abstraction at τ, that is

$$\forall e_1, e_2 \colon \tau.\ e_1 \equiv e_2 \Rightarrow [\![e]\!] = [\![e']\!]$$

whenever the test expression at $\tau \to \iota$ is definable. This is the case when $\tau \to \iota$ is in T, or equivalently when τ is in $ArgT$. Thus we have full abstraction at all types in $ArgT$.

The implications for open terms rather illustrate the restrictions implicit in the above statement. The closed term constructed above for an open term e only has a type in $ArgT$ for a single-variable context and $e \colon \iota$. The former restriction could be alleviated by showing definability at product types, but to remove the latter it would seem we need a language extension such as that discussed earlier.

Even so, our present results are enough to yield full abstraction for non-trivial object types admitting complex re-entrant behaviour, such as $(\iota \Rightarrow \iota) \Rightarrow \iota$. Since two observably equivalent objects of this type may have wildly different concrete implementations, our results provide support for the idea that our semantics succeeds in capturing the essence of data abstraction.

Chapter 7

Possible extensions and further work

In this chapter we discuss various areas for future work. These broadly fall into two camps: extensions to our language and the results presented here motivated by semantic concerns, and suggested avenues for further research which support the idea that the language presented here is interesting and worthy of our consideration.

We address the latter point first, beginning with a discussion of our argument safety restriction. We consider a more generous formulation of the restriction, and discuss some desirable properties of programs which obey it.

We then present an interesting program which shows that in our language it is possible to extract the approximation operator underlying a class implementation.

Next, we discuss some natural extensions of the results presented in this thesis, and subsequently suggest some useful language extensions which enable the use of behaviour naturally present in our game model.

Lastly, we discuss the extension of our language to one which is a good match for **SG**, the category of games without bracketing constraints. We introduce a control operator, and discuss modifications to our semantics to take account of the new behaviour.

(The ideas presented in Sections 7.1, 7.2 and 7.5 are due mainly to John Longley.)

7.1 Applications of argument safety

Semantic concerns regarding our game model led us to the imposition of the restriction on method implementations that they be argument-safe. Before we discuss this further, we note that it is possible to relax the type system considerably while maintaining the semantic restriction. We have given the simple restriction presented here—where method bodies must have a particular syntactic form—for ease of presentation and reasoning. However, the idea of "ground type funnelling", where all interaction with the method argument must pass through a value of ground type before being stored in the state, can be the basis for a more intelligent type system. Briefly, instead of forcing a method body to have a particular form, the type system can keep track of what variables have been potentially "tainted" by contact with the method argument, and marking as `safe` those other variables which have only had contact with the argument via expressions of ground type (if at all). There is a little subtlety regarding nested class implementations, but this is the general idea used in [57].

This said, it could be alleged that if our semantics must impose such a restriction it is not up to the demands of "the real world". There are two closely related answers to this. Firstly, we do not attempt to present a full-scale language, but merely a object-oriented calculus which may form the core of a more powerful language, and which is amenable to proof of properties such as soundness and definability. Secondly, our language can represent a *well-behaved* core, so that even in the context of a larger language, reasoning about programs which fall within the purview of our system can exploit their restricted behaviour.

One property which has been highly relevant in this thesis has been that argument-safe programs do not give rise to cycles in the heap. A less obvious property has to do with the static control of exceptions. To quote [59]:

> In Java, the use of exceptions is tightly regulated by requiring all method signatures to declare explicitly any (checked) exceptions the method might throw. However, this system is perhaps overly conservative, and one might hope to allow more whilst still retaining static control over exceptions. Consider for instance a Java class `List`, with a method `add (Element x)` (for adding elements to the list) and a method `map (Function F)` (for applying a given function to all the elements of the list). Then we cannot invoke `L.map` with a function `F` that may raise exceptions not anticipated in the declaration of `Function`. However, there is a sense in which such method invocations are 'safe', since `F` is discarded by `L` after the method call, so that any exceptions present in `F` will not unexpectedly surface later. By contrast, a method invocation `L.add(x)` is 'unsafe' if `x` may raise an unanticipated exception, since this exception may resurface at an arbitrary later point (e.g. outside its static scope). Intuitively, this is related to the fact that `map` is *argument-safe* while `add` is not.

> We therefore suggest that the notion of argument-safety offers a natural answer to the question: "Which uses of higher-order store can safely coexist with exceptions (in the sense of allowing us to retain static control over the latter)?" Furthermore, one may use this idea as the basis for a static type system guaranteeing security of exceptions whilst allowing more flexibility than Java.

We now turn to the question of how one might extend our language in order to interpret non-argument-safe behaviour. On the one hand, one might turn to a more liberal "non-alternating" game model like that of [10, 51], where the particular semantic concern that necessitates the argument-safety restriction in our game model is not an issue. On the other hand, a purely "behavioural" semantics of objects, even if it enables more expressive power than that presented in this thesis, can still not account for the interpretation of a language with features such as reference equality tests. We must regard two objects o_1 and o_2 as equal because they respond in the same way under all possible interactions, but they may be distinguishable if their identity is taken into account (e.g. with the Java `==` operator). We might therefore instead move to a semantics where objects are associated with *names* (cf. [9], [54])—this would allow us to distinguish two independently created objects with identical behaviour, and would also allow the interpretation of cyclic heap structures by "breaking" the dependency of one object on the behaviour of another with a level of indirection.

Either of these options would allow us to remove a restriction in our interpretation of classes, whereby we are at present only able to give a good account of `private` fields of ground type (by making use of an auxiliary reference cell). Of course privacy can easily be imposed in our current system by a purely syntactic constraint, but such a solution is undesirable from a semantic perspective (i.e. with respect to full abstraction).

7.2 Classes and approximation operators

In this thesis we have given a simple class system as a set of derived types and terms, which are translated into the language proper. This shows that we are able to model such behaviour, but one might object that dispensing with an opaque class system ignores the fact that classes give a form of *encapsulation*. In fact, in our language it is possible to write a program which extracts the approximation operator by which we interpret the class, using only the operations we have defined for classes. Therefore while a practical extension of our language might add an opaque class system, we have not lost anything by not doing so here.[1]

Constructing a class from its approximation operator is easy:

$$\lambda F.\ \mathbf{class}\ (\varsigma)\ \{\ m = F(\varsigma)\ \}$$

In the other direction, for each method invocation with a given state argument s, we can make use of the **extend** operation to create a class c' from the original c which is initialised with state s. The new class c' replaces each method m with the version from the argument ς of the approximation operator under construction, and for each m adds a new method m' which uses the version from c.[2] An additional method *get* is added to obtain the resulting state. This means c' extracts a single application of the approximation operator from c.[3]

$$\begin{aligned}
&\lambda c\colon \mathbf{Class}\ \langle \sigma; m\colon \tau_m \to \tau'_m\rangle_{m\in X}.\ \lambda\varsigma\colon \mathbf{CObj}\ \{m\colon \sigma\otimes\tau_m \to \sigma\otimes\tau'_m\}_{m\in X}.\\
&\quad \mathbf{let}\ c'\ \mathbf{be\ extend}\ c\ \mathbf{with}\\
&\qquad \left\{\begin{array}{rcl} m & = & \varsigma\cdot m,\\ m' & = & \lambda\langle s,x\rangle.\ \mathit{super}\cdot m\langle s,x\rangle,\\ \mathit{get} & = & \lambda\langle s,z\rangle.\ \langle s,s\rangle \end{array}\right\}_{m\in X, m'\in X'}\\
&\quad \mathbf{in\ obj}\ \{m = \lambda\langle s,x\rangle.\ \mathbf{let}\ o\ \mathbf{be}\ (\mathbf{new}\ s\ c')\\
&\qquad\qquad\qquad\qquad\quad \mathbf{in\ let}\ r\ \mathbf{be}\ o\cdot m'(x)\\
&\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \langle o\cdot \mathit{get}(), r\rangle\}_{m\in X}
\end{aligned}$$

The class c' is constructed in the method m of the object comprising the result of our approximation operator. This method can be seen to obey a version of the argument-safety restriction, which is a mild relaxation to allow an alternating sequence of argument-safe and non argument-safe **let** bindings. Thus the variable o is safe, while r is not, but since we only store the result of $o\cdot \mathit{get}()$ in the state, nothing goes wrong (the key here is that $o\cdot m'(x)$ cannot capture a reference to x).

We conjecture these programs implement a definable retraction of the follow-

[1]Similar issues are considered in [73].

[2]We use *super* here—this can easily be incorporated into the translation of the derived construct **extend**.

[3]We can only use c' for a single step, because to reuse c' we would need to be able to set the initial state before the next invocation of m', and this falls foul of the argument safety restriction.

ing type:

$$\mathbf{Class}\ \langle \sigma, \tau_m \to \tau'_m \rangle_{m \in X} \ \lhd \ \begin{array}{c} \mathbf{CObj}\ \{\sigma \otimes \tau_m \to \sigma \otimes \tau'_m\}_{m \in X} \to \\ \mathbf{CObj}\ \{\sigma \otimes \tau_m \to \sigma \otimes \tau'_m\}_{m \in X} \end{array}$$

This could be used to give a full abstraction and definability result for classes via the corresponding results for the **CObj** type. Furthermore, an extended version of this idea works even when additional fields in subclasses are added (see Section 7.4).

7.3 Unfinished business

Here we shall briefly review a few natural extensions of the results in this thesis which we previously discussed in the relevant chapters.

7.3.1 Soundness of $\mathcal{L}_{\text{arg}}$

In Chapter 5, we proved soundness for the restricted language $\mathcal{L}_{\text{pair}}$ rather than the full generality of $\mathcal{L}_{\text{arg}}$. As discussed in Section 5.4, the restriction to "pair-like" method implementations ensures the heaps arising in the evaluation of an expression have a restricted structure. We have no reason to believe that the soundness property should not hold for the whole of $\mathcal{L}_{\text{arg}}$. Indeed, we conjecture that a broadly similar heap property would be suitable, but which caters for interaction with heap cells newly created during a method call (and stored in the object's state) via the result of that call. The subtlety seems to be that these possible interactions have a somewhat restricted form, so while the property we have presented is too restrictive, the version suited to $\mathcal{L}_{\text{arg}}$ would have to strike a balance between this and being too permissive.

Several details of our proof are particular to $\mathcal{L}_{\text{pair}}$, but these seem to be simplifications rather than crucial restrictions.

7.3.2 Adequacy

In concluding Chapter 5 we admitted to the lack of a proof of the remaining part of the adequacy property. As we said there, it seems highly implausible that it is false, and given the detailed semantic analysis inherent in our soundness proof, it is likely that the problem boils down to finding the correct property to prove.

7.3.3 Intermediate state update

As noted in Chapter 4, viewing a method as a state-transforming function can be thought of as allowing the state to be read at the start of a method's execution (taking a private copy) and written at the end. We might wish to extend this to allow the state to be read from or written to at arbitrary points, as in most OO languages.

In Chapter 6, we exhibited a strategy which we are at present unable to define in our language, and suggested that a definability result at all types would require a more expressive means of state interaction. In particular, we discussed the introduction of *read* and *write* operations, allowing the state to be read from and written to at intermediate points of a method's execution. We might for example give method implementations a type

$$(1 \Rightarrow \sigma) \otimes (\sigma \Rightarrow 1) \otimes \tau \Rightarrow \tau$$

instead of $(\sigma \otimes \tau) \Rightarrow (\sigma \otimes \tau)$. It is certainly the case that some such operations can be defined (both syntactically and semantically), but not in an unrestricted form. A write operation would at least have to obey the usual restrictions on argument-safety, while it seems there may also be some subtle restriction on the use of a *read* operation.

7.3.4 Linear classes

In Chapter 6 we set out to give a definability result for intuitionistic, product-free types. It does not seem that products would be more than an annoyance, but the extension of our *interpret* programs to linear types requires a language extension. Unlike the issues discussed above, these features are absent more by way of simplification than for any technical reason.

We mentioned in Chapter 4 that it is possible to give a **constr** operation of the following type:

$$\frac{\Gamma \vdash c\colon \mathbf{Obj}\ \{m\colon \sigma \otimes \gamma_m \to \sigma \otimes \tau\}_{m \in X} \quad \Delta \vdash e\colon \sigma}{\Gamma, \Delta \vdash \mathbf{constr}\ e\ c\colon \mathbf{Obj}\ \{m\colon \gamma_m \to \tau_m\}_{m \in X}}\ \mathit{basic}(\gamma_m)$$

This corresponds to the "linear thread" operation we presented in Chapter 3 as a stepping stone on the route to our full definition. This version of **constr** allows us to store values of linear type rather than only reusable type, because the restriction to ground-type arguments excludes the possibility of nested method invocations. Given this,[4] it should be straightforward to extend the adequacy and definability results to include linear types.

7.4 Useful extensions

7.4.1 Polymorphism

With relatively minor alterations, our game model supports parametric polymorphism, as commonly found in functional languages (often called *genericity* in the context of Object-Oriented languages). A suitable interpretation is described in [8].

7.4.1.1 Adding new fields in subclasses

There is an important deficiency in the translation of classes we described earlier. Normally when one creates a subclass, one would expect to add new fields as well as methods, however we have not allowed for this. The problem is that the state appears as both argument to and result of the step function being extended.

Various extensions to our calculus can be used to tackle this problem. When defining a class, we must give the step function a type which includes not only the state of that class, but the *potential subclass state*. When the class is extended, that potential subclass state will be partially instantiated as the newly added state, together with the *new* potential additional state; when the class is instantiated, the additional state is taken to be something trivial.

By treating the potential future state polymorphically, we thus cater for any particular choice of additional state when a class is extended. To interpret a class creating an object of type $\mathbf{Obj}\ \{\tau \to \tau'\}$ with state type σ, instead of an approximation operator of type

$$\mathbf{Obj}\ \{\sigma \otimes \tau \to \sigma \otimes \tau'\} \to \mathbf{Obj}\ \{\sigma \otimes \tau \to \sigma \otimes \tau'\}$$

we would take one of type

$$\forall \upsilon.\ \mathbf{Obj}\ \{(\sigma \otimes \upsilon) \otimes \tau \to (\sigma \otimes \upsilon) \otimes \tau'\} \to \mathbf{Obj}\ \{(\sigma \otimes \upsilon) \otimes \tau \to (\sigma \otimes \upsilon) \otimes \tau'\}$$

The above is a well-known idea in the theory of object-oriented languages, as in [23].

[4]Possibly an operation allowing us to get around a situation in which the type system is over-conservative with respect to linearity is also required.

7.4.2 Recursive types

In Section 2.1.8 we noted that **SG** supports recursive types, however the only use of these we have made is to explain our exponential as $!A = \mu X.A \oslash X$. An obvious extension would be to add the corresponding notion of recursive types to our language. Such an extension is somewhat orthogonal to the other issues we have been discussing. In fact, such an addition gives us access to lots of behaviour already present in our model, but not accessible for typing reasons. Here we shall not concentrate on the addition of recursive types per se, but on this behaviour they enable.

7.4.2.1 Recursive classes

Most fundamentally, recursive types allow us to define "recursive classes". To be a little more precise, recursive types enable the definition of (classes which generate) objects with recursive types, and also (mutually) recursive classes.

Note that since the type of an object does not appear in its state (and we give structural types), we can already create objects with pointers to others of the same type. For example, in *the present language* we can define the following linked list of integers:

$$\begin{array}{l}
\textbf{letrec}\ \mathit{list}_{\mathit{imp}} = \textbf{obj}\ \{ \\
\quad \mathit{add} = \lambda\langle s, x\rangle.\langle\ \langle x, \textbf{constr}\ \mathit{list}_{\mathit{imp}}\ \langle h, t\rangle\ \rangle,\ *\rangle, \\
\quad \mathit{nth} = \lambda\langle s, n\rangle.\langle s,\ \textbf{let}\ \langle h, t\rangle\ \textbf{be}\ s\ \textbf{in}\ \textbf{ifz}\ n\ \textbf{then}\ h\ \textbf{else}\ t\cdot \mathit{nth}(n-1)\rangle \\
\}\ :\ \textbf{Obj}\ \{\mathit{add}\colon S\otimes\iota \to S\otimes\iota,\ \mathit{nth}\colon S\otimes\iota\to S\otimes\iota\}
\end{array}$$

where $S = \textbf{Obj}\ \{\mathit{add}\colon \iota \to \iota,\ \mathit{nth}\colon \iota \to \iota\}$. An object created from $\mathit{list}_{\mathit{imp}}$ via **constr** would also have type S.

On the other hand, we could not add a method *tail* to return the tail of the list, since the type of the entire object should then appear in the type of the *tail* method—but one could easily do this in the presence of recursive types.

7.4.2.2 Clone

One particular example of an operation which we need recursive types to define is a clone method. We can equip an object with a shallow clone method with the following type:

$$\mu X.\ \textbf{Obj}\ \{m_1\colon \sigma_1 \to \tau_1, \ldots, m_n\colon \sigma_n \to \tau_n, \mathit{clone}\colon 1 \to X\}$$

We could do this by defining a "clone **constr**" operation **cconstr** as follows:

$$\begin{aligned}&\mathbf{cconstr} = \lambda s.\lambda e.\ \mathbf{Y}(\lambda \mathit{mknew}.\lambda s.\\&\quad \mathbf{constr}\ s\ (\mathbf{extend}\ e\ \mathbf{with}\ \{\mathit{clone} = \lambda\langle s, *\rangle.\ \langle s, \mathit{mknew}\ s\rangle\}))\ s\end{aligned}$$

Notice that while we can define the above in the current language, the best type we could give *clone* would be some finite expansion of the above recursive type, i.e. we could happily create uncloneable "clones".

7.5 Control

In Chapter 2 we presented **SG** as a simple and elegant game model, but after introducing the well-bracketed model **BG** we proceeded to define a language intended as a "match" for the latter. We could instead ask, what is the appropriate language for **SG**? Given the well-known correspondence between bracketing and control, we could expect to be able to introduce a control operator for **SG**.

7.5.1 A language extension

We mentioned in Section 2.3 that the *catch* operator exhibits the simplest violation of the **BG** bracketing condition. If we allow ourselves sum types $\tau_1 + \tau_2$, which are a straightforward addition to the language, then we could give a catch operator

$$\frac{\Gamma \vdash e\colon (\tau_1 \to \tau_2) \to \tau_3}{\Gamma \vdash \mathbf{catch}\ e\colon \tau_1 + \tau_3}$$

where τ_1, τ_2, τ_3 are all ground types. The catch operator of [27] returns 0 if the evaluation of $e\ z$ (where z is a dummy argument) attempts to evaluate z, or otherwise $n+1$ if e evaluates to n without touching z. In the same way, the above operation would return $\mathbf{inl}(v)$ if the evaluation of $e\ z$ with a dummy argument z attempts to evaluate some $z\ v$, and otherwise if e evaluates as normal to v', then the result would be $\mathbf{inl}(v')$.

This operator captures one way of violating the bracketing condition, namely answering a question prematurely (i.e. when there is some more recent pending question). In **SG** it is also possible to define strategies which answer a question "late" (i.e. after the question has been preempted by such a premature answer). This corresponds to a kind of "resumable exception", or restricted continuation operator

$$\frac{\Gamma \vdash e\colon \mathbf{Obj}\ \{m\colon \tau_1 \to \tau_2\} \to \tau_3}{\Gamma \vdash \mathbf{catchcont}\ e\colon \tau_3 + (\tau_1 \otimes (\tau_2 \to \mathbf{Obj}\ \{m\colon \tau_1 \to \tau_2\} \to \tau_3))}\ basic(\tau_1, \tau_2, \tau_3)$$

Here the linearity of the function type $\tau_2 \to \mathbf{Obj}\ \{m\colon \tau_1 \to \tau_2\} \to \tau_3$ is rather important, meaning that the continuation may only be used once. This is exactly right with reference to **SG**: we can lift the restriction from **BG** on when a question may be answered, but the games we have defined do not support answering a question repeatedly. From the point of view of a more practical implementation, linearly used continuations are important, as they mean that there is no need to copy the relevant portion of the stack.

It is relatively straightforward to give an operational semantics for the language extended with this operator. As usual for the operational semantics of exceptions, one can make use of evaluation contexts $E[-]$. Such an evaluation context is an expression with a "hole" at the current point of evaluation. To give a hint of such a semantics, we could give a small-step rule for **catchcont**:[5]

$$\mathbf{catchcont}\ \lambda x.E[x \cdot m\ v] \to \mathbf{inr}(\langle v, \lambda z.\lambda x.E[z]\rangle)$$

We now come to the interaction of stateful objects and continuations. Our existing method invocation rule has very much a "well-bracketed" character, and does not seem compatible with the addition of continuations. In order to correctly handle the case where the argument to **catchcont** is called from within the evaluation of a method implementation, we can decompose the method invocation into "read" and "update" stages along the following lines, by considering an augmented language with an $\mathbf{update}_l$ operation for each location l:

$$\begin{array}{ll} h, l \cdot m\ v \to h, \mathbf{update}_l\ v_c \cdot m\langle v_s, v\rangle & h(l) = \langle v_s, v_c\rangle \\ h, \mathbf{update}_l\ \langle v_s', v'\rangle \to h[l \mapsto \langle v_s', v_c\rangle], v' & h(l) = \langle v_s, v_c\rangle \end{array}$$

7.5.2 Interpretation in SG

Firstly, we note that it still makes sense to interpret our existing language in **SG**. The semantics would still be adequate in **SG** (at least for the formulation of the property at ground types)—the denotation of a program just happens to be well-bracketed, rather than being constrained to be so. However, it is more interesting to add "all" non-well-bracketed behaviour, in order to make every strategy in **SG** at denotable type definable in some extension of our language.

[5] A big-step evaluation relation more in line with the operational semantics presented earlier is possible with a little work.

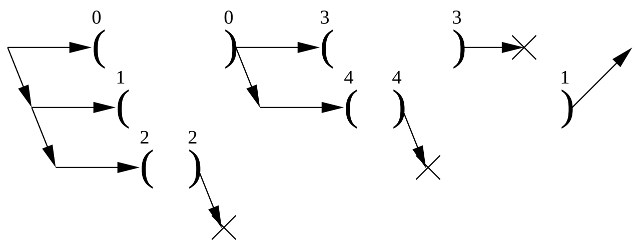


Figure 7.1: Non-well-bracketed method calls

We can interpret **catchcont** by a fairly straightforward extension of the **catch** strategy. However, just as the operational semantics of method invocation has a "well-bracketed" character, our morphism *thread* is defined in a well-bracketed way. Recall that we split method calls into two groups: those which occur during the call in question, and those which occur after. Now consider Figure 7.1, in contrast to Figure 3.2 presented earlier—it is clear that this decomposition is no longer valid, at least if one expects the obvious semantics corresponding to the operational rules sketched above, where the last state update is the one which "sticks".

Fortunately, it seems one can define a version of *thread* which behaves correctly in the presence of such ill-bracketed behaviour, by not relying on the above decomposition of method calls. However, the more structured definition in Chapter 3 allowed us to state and prove our thread properties. Not only would these not seem to hold in the new setting, but one might expect that the proof of the appropriate properties would be more complex.

Indeed this will be more generally true of the proof of adequacy as a whole. As more behaviour is added to the language, it is harder to show that the operational and denotational interpretations agree. In contrast, the move to **SG** should make the proof of definability simpler. One can show definability for the universal object of **SG** (i.e. $[\![\mathbf{Obj}\ \{m\colon \iota \to \iota\}]\!]$), and then with the help of **catchcont**[6] write programs constituting a definable retraction to this object for each type, following the scheme of Longley [60]. The key ingredients of such programs have recently been implemented in [56].

[6]One actually requires a slightly more powerful **catchcont** operator, as discussed in [59].

Chapter 8

Conclusion

In this thesis we have put forward the view that object-oriented programming and game models of computation are well matched, with particular emphasis on the notion of data abstraction. In Chapter 3 we introduced the *thread* operator taking an explicit implementation of an object in terms of an internal state and producing the behaviour of the specified object. In Chapter 4 we then introduced a corresponding language feature **constr**, and described how this allows us to model classes and objects, before giving an operational semantics in terms of heaps. In the soundness proof of Chapter 5 we had to go to some lengths to reconcile the behavioural view of objects given by *thread* with the explicit heap and state manipulation by the operational semantics, suggesting that the interpretation in our game model really is more abstract. In Chapter 6 we gave a limited full abstraction and definability result, which nonetheless demonstrates the validity of our data abstraction operation at some interesting types—no matter what the concrete implementation, if the behaviour of two objects are the same, the denotation they both receive is the same strategy.

The design of our language was heavily guided by our simple game model. This model contains much interesting stateful behaviour, but led us to impose the *disciplined* requirement on strategies implementing objects. Correspondingly, the language we defined in Chapter 4 contains a level of expressivity intermediate in power between that of simple ground-type state and the full generality of higher-type references. This level of expressive power leads to a nontrivial soundness proof, and has proved to be sufficiently subtle that our full abstraction result does not hold at all types in the present language (and yet does so at an interesting range of types).

In Chapter 7 we presented a number of natural and useful extensions to our language and model. These support the notion that the ideas presented in this thesis are of wider interest in the context of a larger language.

Bibliography

[1] Martin Abadi and Luca Cardelli. A theory of primitive objects — untyped and first-order systems. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789, pages 296–320. Springer-Verlag, 1994.

[2] Martín Abadi and Luca Cardelli. *A theory of objects*. Springer, 1996.

[3] Erika Ábrahám, Marcello M. Bonsangue, Frank S. de Boer, and Martin Steffen. Object connectivity and full abstraction for a concurrent calculus of classes. pages 37–51, July 2004.

[4] Erika Ábrahám, Frank S. de Boer, Marcello M. Bonsangue, Andreas Grüner, and Martin Steffen. Observability, connectivity, and replay in a sequential calculus of classes. pages 296–316.

[5] Erika Ábrahám, Andreas Grüner, and Martin Steffen. Dynamic heap-abstraction for open, object-oriented systems with thread classes. In *Proceedings of Computability in Europe 2006: Logical Approaches to Computational Barriers, CiE'06*, January 2006.

[6] S. Abramsky and R. Jagadeesan. Games and full completeness for multiplicative linear logic. In R. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science (FST-TCS'92)*, pages 291–301, New Delhi, India, 1992.

[7] S. Abramsky and G. McCusker. Linearity, sharing and state: a fully abstract game semantics for idealized algol with active expressions, 1997.

[8] Samson Abramsky. Semantics of interaction: an introduction to game semantics. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*. Cambridge University Press, 1997.

[9] Samson Abramsky, Dan Ghica, Andrzej Murawski, Luke Ong, and Ian Stark. Nominal games and full abstraction for the nu-calculus. In *Proceedings of the Nineteenth Annual IEEE Symposium on Logic in Computer Science*, pages 150–159. IEEE Computer Society Press, 2004.

[10] Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references, 1998.

[11] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. In *Theoretical Aspects of Computer Software*, pages 1–15, 1994.

[12] Samson Abramsky and Guy McCusker. Call-by-value games. In *CSL '97: Selected Papers from the 11th International Workshop on Computer Science Logic*, pages 1–17, London, UK, 1998. Springer-Verlag.

[13] Samson Abramsky and Guy McCusker. Game semantics. 1998.

[14] Samson Abramsky and C.-H. Luke Ong. Full abstraction in the lazy lambda calculus. *Inf. Comput.*, 105(2):159–267, 1993.

[15] A. Barber. *Dual Intuitionistic Linear Logic.* University of Edinburgh, Dept. of Computer Science, Laboratory for Foundations of Computer Science, 1996.

[16] P. N. Benton. A mixed linear and non-linear logic: Proofs, terms and models (extended abstract). In *CSL*, pages 121–135, 1994.

[17] P. N. Benton, Gavin M. Bierman, Valeria de Paiva, and Martin Hyland. Linear lambda-calculus and categorial models revisited. In *CSL*, pages 61–84, 1992.

[18] G. Berry and Pierre-Louis Curien. Sequential algorithms on concrete data structures. *Theor. Comput. Sci.*, 20:265–321, 1982.

[19] G. Bierman, M. Parkinson, and A. Pitts. MJ:An imperative core calculus for Java and Java with Effects. 2003.

[20] Gavin M. Bierman. What is a categorical model of intuitionistic linear logic? In *TLCA*, pages 78–93, 1995.

[21] Andreas Blass. A game semantics for linear logic. *Ann. Pure Appl. Logic*, 56(1-3):183–220, 1992.

[22] Viviana Bono, Amit J. Patel, Vitaly Shmatikov, and John C. Mitchell. A core calculus of classes and objects. In *Fifteenth Conference on the Mathematical Foundations of Programming Semantics*, 1999.

[23] Kim B. Bruce. *Foundations of Object-Oriented Languages.* MIT press, 2002.

[24] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[25] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Theoretical Aspects of Computer Software*, pages 415–438, 1997.

[26] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, 1988. A revised version of the paper that appeared in the 1984 Semantics of Data Types Symposium, LNCS 173, pages 51–66.

[27] R. Cartwright and M. Felleisen. *Observable sequentiality and full abstraction.* ACM Press New York, NY, USA, 1992.

[28] Robert Cartwright, Pierre-Louis Curien, and Matthias Felleisen. Fully abstract semantics for observably sequential languages. *Inf. Comput.*, 111(2):297–401, 1994.

[29] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *ACM SIGPLAN Notices*, 24(10):433–443, 1989.

[30] William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135, New York, NY, USA, 1990. ACM Press.

[31] Pierre-Louis Curien. Notes on game semantics. http://www.pps.jussieu.fr/ curien/.

[32] Pierre-Louis Curien. On the symmetry of sequentiality. In *Proceedings of the 9th International Conference on Mathematical Foundations of Programming Semantics*, pages 29–71, London, UK, 1994. Springer-Verlag.

[33] Pierre-Louis Curien. Definability and full abstraction. *Electr. Notes Theor. Comput. Sci.*, 172:301–310, 2007.

[34] P.L. Curien and G. Ghelli. Coherence of Subsumption, Minimum Typing and Type-Checking in. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, 1994.

[35] O-J Dahl, B Myhrhaug, and K Nygaard. Simula 67 common base language. Technical report, 1968.

[36] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–183, 1998.

[37] J.Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[38] Adele Goldberg and David Robson. *SmallTalk-80 The Language and its Implementation.* 1983.

[39] Andrew Gordon and Gareth Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida*, pages 386–395, 1996.

[40] Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In *Proceedings HLCL'98.* Elsevier ENTCS, 1998.

[41] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[42] K. Honda and N. Yoshida. Game-theoretic analysis of call-by-value computation. *Theoretical Computer Science*, 221(1-2):393–456, 1999.

[43] J. M. E. Hyland and C.-H. Ong. On full abstraction for PCF: I. models, observables and the full abstraction problem ii. dialogue games and innocent strategies iii. a fully abstract and universal game model.

[44] J. M. E. Hyland and C.-H. L. Ong. Fair games and full completeness for multiplicative linear logic without the mix-rule. 1993.

[45] Martin Hyland. Game semantics. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*. Cambridge University Press, 1997.

[46] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[47] Alan Jeffrey and Julian Rathke. A fully abstract may testing semantics for concurrent objects. *Theor. Comput. Sci.*, 338(1-3):17–63, 2005.

[48] Alan Jeffrey and Julian Rathke. Java jr: Fully abstract trace semantics for a core java language. In *ESOP*, pages 423–438, 2005.

[49] S.N. Kamin and U.S. Reddy. Two Semantic Models of Object-Oriented Languages. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, 1994.

[50] Alan C. Kay. The early history of smalltalk. pages 511–598, 1996.

[51] J. Laird. A categorical semantics of higher-order store. In *9th Conference on Category Theory and Computer Science*, Electronic notes in Theoretical Computer Science. Elsevier, 2002.

[52] Jim Laird. Full abstraction for functional languages with control. In *Logic in Computer Science*, pages 58–67, 1997.

[53] Jim Laird. *A semantic analysis of control*. PhD thesis, University of Edinburgh, 1998.

[54] Jim Laird. A game semantics of local names and good variables. In *9th Conference on category theory and computer science*, 2002.

[55] F. Lamarche. Sequentiality, games and linear logic. In *Workshop on Categorical Logic in Computer Science*. Aarhus University, 1992.

[56] J. Longley. Catchcont and friends. NJ-SML source file, available from `http://homepages.inf.ed.ac.uk/jrl`.

[57] J. Longley. Definition of the Eriskay programming language. In preparation. Draft available at `http://homepages.inf.ed.ac.uk/homepages/jrl`.

[58] J. Longley and G. Plotkin. Logical full abstraction and PCF. In J. Ginzburg, Z. Khasidashvili, C. Vogel, J.-J. Levy, and E. Vallduvi, editors, *Tbilisi Symposium on Language, Logic and Computation.*, 1997.

[59] J. Longley and N. Wolverson. Eriskay: an experiment in semantically inspired programming language design. In preparation. Draft available at `http://homepages.inf.ed.ac.uk/homepages/jrl`.

[60] John Longley. Universal types and what they are good for. *Domain theory, logic and computation.*

[61] John Longley. Interpreting localized computational effects using operators of higher type. In A. Beckmann, C. Dimitracopoulos, and B. Loewe, editors, *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008, Athens*, volume 5028 of *Lecture Notes in Computer Science.* Springer, 2008. To appear.

[62] P. Lorenzen. Ein dialogisches Konstruktivitätskriterium. In *Infinitistic Methods (Proc. Sympos. Foundations of Math., Warsaw, 1959)*, pages 193–200. Pergamon, Oxford, 1961.

[63] P.A. Mellies. Comparing hierarchies of types in models of linear logic. *Information and Computation*, 189(2):202–234, 2004.

[64] Paul-André Melliès. Sequential algorithms and strongly stable functions. *Theor. Comput. Sci.*, 343(1-2):237–281, 2005.

[65] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997.

[66] J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 109–124, January 1990.

[67] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symp. on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989.

[68] B.C. Pierce and D.N. Turner. Simple Type-Theoretic Foundations for Object-Oriented Programming. *Journal of Functional Programming*, 4(2):207–247, 1994.

[69] Gordon D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):225–255, 1977.

[70] U. Reddy. *Objects as closures: abstract semantics of object-oriented languages.* ACM Press New York, NY, USA, 1988.

[71] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing, 1986.

[72] P. Wadler. There's no substitute for linear logic. Manuscript, December 1991.

[73] Mitchell Wand. Type inference for objects with instance variables and inheritance. In *Theoretical aspects of object-oriented programming: types, semantics, and language design*, pages 97–120. MIT Press, 1994.