Eriskay: a programming language based on game semantics

John Longley Nicholas Wolverson

February 12, 2008

Abstract

We report on an ongoing project to design a strongly typed, class-based object-oriented language based around ideas from game semantics. Part of our goal is to create a powerful modern programming language whose clean semantic basis renders it amenable to work in program verification; however, we argue that our semantically inspired approach also yields benefits of more immediate relevance to programmers, such as expressive new language constructs and novel type systems for enforcing security properties of the language. We describe a simple-minded game model with a rich mathematical structure, and explain how this model may be used to guide the design of our language. We then focus on three specific areas where our approach appears to offer something new: linear types and continuations; observational equivalence for class types; and static control of the use of higher-order store.

In a substantial appendix, we present the formal definition of a fragment of our language which embodies many of the innovative features of the full language.

1 Introduction and motivations

This paper reports on an ongoing experiment in applied semantics. Our goal, broadly, is to take some simple and appealing mathematical model of computation, and to allow the structure of this model to guide the design of a full-scale programming language. The hope is that this will result in a clean and elegant language with a coherent design and pleasant logical properties.

Our project in some ways seeks to emulate the development of Standard ML [24]. Part of the explicit intention of the designers of ML was to provide a language with a sound mathematical basis, not just in the sense of having a rigorous formal definition, but in the sense of being based around a mathematically natural conception of computable functions, as embodied by Scott's LCF and its domain-theoretic model. As a result, the functional fragment of ML, in particular, is widely considered to be a clean and beautiful system and turns out to be particularly amenable from a logical point of view. (see e.g. [18]). Of course, the demands of programming practice also motivated the inclusion of "non-functional" features which could not be straightforwardly interpreted in the mathematical model, and with hindsight, one can identify aspects of the design of these parts of ML which are troublesome from a logical point of view (for instance, the possibility of *anonymous exceptions*). Nowadays, however, we have mathematically appealing semantic models for much more than purely functional computation, and even some rather simple *game models* (see e.g. [2, 10]) are sufficient to underpin many of the features typical of modern programming practice, such as state and control features, subtyping, second order polymorphism, and a class and object system with inheritance and dynamic binding. We therefore believe that the time has come for another attempt to design a mathematically based language, but making use of some computationally much richer models that have emerged more recently from work in game semantics.

Part of our aim is to create a language whose clean semantic basis makes it a suitable framework for future research in program verification. Besides the traditional problem of proving a program correct relative to a specification, we have in mind problems such as proving that two different implementations of a certain (Java-style) class are observationally equivalent – an issue of clear relevance to modular program construction. (We shall explain how game semantics sheds light on this particular problem in Section 5). The methodology we have in mind is that the underlying and relatively simple model should guide the design of a clean and expressive program logic (as in the case of LCF). Statements in the logic will thus refer in principle to elements of the model; but if one has a strong correlation between language and model known as *logical full abstraction*, they can equally well be interpreted directly as statements about programs couched in terms of more familiar operational concepts [17]. We will therefore have a particular interest in questions of full abstraction and definability for our language and model. Another possible strength of this approach is that it allows one to reason directly with expressions of the programming language itself, rather than indirectly via a translation into some logical language as in e.g. [11].

Of course, only a small minority of programmers are likely to be excited by the prospect of formal program verification. However, we believe that the 'logical hygiene' that makes a language amenable to program verification also yields other benefits that are more immediately relevant to the concerns of programmers. On one level, the search for a 'tight fit' between the language and the model can sometimes lead to the discovery of expressive new language constructs. (For example, in Section 4 we present a semantically inspired control operator with a 'coroutining' flavour, while in Section 5 we show how the semantic perspective leads naturally to a calculus in which class implementations are first-class values.) On another level, our model suggests certain kinds of type systems which enforce certain safety properties of programs, such as the security of the exception, continuation and name generation mechanisms, in a manner quite different from that offered by effect type systems as in [23]. We discuss the static control of exceptions as an example in Section 6.

Broadly speaking, we have in mind a strongly typed, class-based, object-oriented language, with powerful polymorphism somewhat as in Generic Java, and with a functional, higher-order flavour somewhat as in OCaml. Our main goals are:

- To develop a programming language, comparable in scale with ML or Haskell, which owing to its clean semantic basis, will provide a suitable medium for future research in program verification.
- To provide a showcase for certain language features inspired by game semantics, thus enabling other language designers to assess the potential benefits of these features and of our semantically inspired approach in general.
- To provide a platform which allows experimentation with the kinds of programming

styles supported by these features.

A formal definition of our language (which we call Eriskay) is underway, and a working draft is available online [20]. We also intend to develop a prototype implementation to a level suitable for research and teaching purposes. For the purpose of pedagogical presentation and theoretical study, we have also isolated a subset of the language, called Lingay, which embodies many of the innovative features of Eriskay but is not suitable as a practical programming language. A complete definition of (the core of) Lingay is given in Appendix A.[1]

In this paper, our purpose is not to catalogue all the features of the proposed language, but rather to explain in general terms how ideas from game semantics may be used to shape the design of our language, and then to focus on some particular areas where our approach seems to have something new to contribute. In Section 2 we informally introduce the game-theoretic conception of computation and give a broad outline of the project as a whole. In Section 3 we give the precise definition of our game model, and explain in particular the paradigmatic interpretation of *object types* within it. We then consider three specific areas of our language in more detail: the use of linear types in connection with continuation operators (Section 4); the treatment of class implementations and observational equivalence thereof (Section 5); and static control of the use of higher order store, with an application to the analysis of exceptions (Section 6).

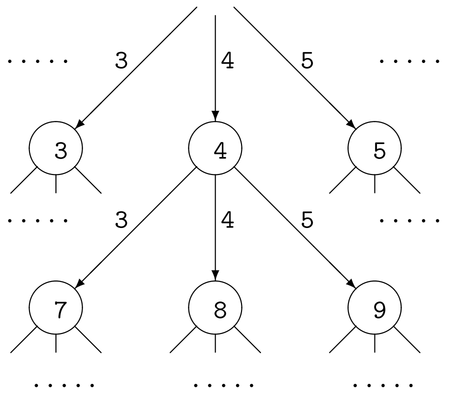
We are grateful to Bob Atkey, Sam Lindley, Don Sannella, Alex Simpson and Ian Star for valuable feedback, and also to the referees of an earlier version of this paper for their helpful comments. This work was supported by EPSRC Grant GR/T08791: “A programming language based on game semantics”.

2 Programs as strategies

We now give a broad overview of the main ideas behind our project. Here, a comparison with typed functional languages such as Standard ML and Haskell is instructive. In broad terms, the design of these languages is based on a conception of program behaviour expressed by the slogan: *Programs are functions.* (More precisely, the behaviour of a program may be adequately captured by a mathematical function, say between suitable domains.) For both ML and Haskell, this conception leads to a very clean and beautiful ‘core language’ which admits pleasant reasoning principles. However, the demands of programming practice also necessitate the inclusion of other features that do not immediately fit into a purely functional framework (e.g. exceptions, state, I/O).

By analogy, our basic conception of program behaviour may be expressed by the slogan: *Programs are strategies.* This says, for example, that the behaviour of an object on the heap is adequately captured by a strategy or ‘decision tree’ determining how the object responds under all possible interactions with its environment (typically the rest of the program). For instance, in Fig. 1 we illustrate how a decision tree may represent the behaviour of a stateful object with an `int->int` method under all possible sequences of method calls; and one may readily imagine how more complex objects give rise to interaction strategies of a more

[1]Our languages, like Java, bear the names of islands. Eriskay is a small island in the Outer Hebrides, and Lingay is a tiny uninhabited island nearby. The latter name also seems appropriate in view of the associations with *linearity* and *games*.



```
Object adder = new Object() {
   private int total = 0;
   public int add (int i) {
    total += i; return total
    }
   }
```

Figure 1: Part of the strategy associated with a Java-style object

elaborate kind. Here the labelled edges, representing method invocations with specified arguments, correspond to possible moves played by an Opponent (the environment to the object), whilst the labelled vertices, giving the return values, correspond to the response of a Player who represents the object itself.

Clearly, objects with a more complicated interface type will be modelled by more elaborate kinds of strategies: for instance, if the argument to some method m is itself an object o, the Player's response might be first to interrogate o before returning from the call to m; or if the return value of a call to m is itself an object o', a possible Opponent response might then be to interrogate o'. The basic paradigm, then, is that each programming language type will be modelled by a *game*, which specifies the allowable forms of potential interactions, while a term of some type will be modelled by a strategy of the corresponding game. A more precise formulation of the game-theoretic model we have in mind will be given in the next section.

On an intuitive level, this idea seems to fit very naturally with an object-oriented way of thinking, in that it captures the essence of data abstraction — the concrete implementation of an object is hidden while only the externally visible interactions are modelled. It is therefore not too surprising that interpretations based on strategies lead to *fully abstract* models of suitable object-oriented languages. Moreover, games naturally embody a *reactive* conception of computation as an ongoing process of interaction rather than a simple terminating procedure that computes some final result.

Our approach, then, is to construct a simple mathematical model – the category $\mathcal{G}_V$ to be defined in the next section – which formalizes the ideas of games and strategies in an intuitive way, and then to ask precisely which aspects of object-oriented languages can be naturally and adequately captured in this model. This will lead to the definition of a *core language* which can be smoothly interpreted in the model. Furthermore, we may ensure that we are getting 'value for money' from our model, by requiring that our language can express *every* (computable) strategy of appropriate type that the model supports.

The core language thus obtained turns out to be much richer than, say, the functional core of ML or Haskell, but it still lacks some features that one would like for programming purposes. For example, it is intuitively clear that a simple-minded semantics of objects in terms of behaviour under message passing will not be able to account for reference equality tests (`==` in Java). A more subtle example of the limitations of $\mathcal{G}_V$ will be given in Section 6. Following the example of ML, we therefore augment our core language to a *full language* which includes some of these missing features. However, in contrast to ML, we restrict ourselves (even in the full language) to features which we are able to model within our game-semantical framework, albeit at the cost of some additional complication. This will mean that we will in principle have a relatively clean semantic basis for the whole language, although it is for the core language that everything will work most smoothly.

In principle, then, the boundary between the core and the rest of the language is determined by what can and cannot be naturally modelled in $\mathcal{G}_V$. From a syntactic point of view, however, the drawing of this line turns out to be surprisingly delicate, and some rather subtle syntactic criteria are involved (see Section 6). In our view, it is one of the main virtues of a semantically inspired approach that it helps us to identify the 'right' place to draw the boundary around certain classes of well-behaved programs.

To anticipate the outcome of this investigation, the scope of our core and full languages respectively may be broadly summarized as follows. In the *core language*, we have (or can naturally simulate):

- The kernel of a Java-style class and object system, including fields, methods and constructors; public, protected and private access levels; and single inheritance with method overriding and dynamic method invocation. However, all this is subject to certain restrictions on the manipulation of higher-order store (see Section 6), which imply (among other things) that only *acyclic* heaps may be constructed.
- Class types, allowing a treatment of class implementations as first-class expressions (see Section 5).
- Recursion and mutual recursion for terms and types (including recursive object and class types).
- Structural (sub)typing for objects, along with (unbounded and bounded) second-order polymorphism at the level of both types and first-order type operators (this gives us much of the essential power of *F-bounded polymorphism*).
- A system of linear types in the spirit of [27] (see Section 4).
- Control constructs sufficient to support coroutining (see Section 4).

In the smaller Lingay language the main omissions are recursive types and polymorphism (except for a limited form of *row variable polymorphism* required for the class inheritance machinery — see Section5). Furthermore, declarations are omitted, and the language is explicitly typed so that type inference is not an issue.

In the *full* Eriskay language, we also have the following non-core features:

- Unrestricted use of higher order store, including cyclic heap structures.
- A system of reference types, including reference equality.
- Exceptions and input/output.

We should also comment on features we are unable to include because our semantic framework cannot accommodate them. Firstly, and most obviously, our chosen game models do not support threads or concurrency, although the possibility of coroutining does offer some compensation for this. Secondly, as is typical for systems that treat object types structurally, we cannot deal with strong binary methods (those whose implementations can access the private members of both the objects in question). Thirdly, we cannot cope with typical 'object-based' features such as method update. An interesting question for further investigation is whether our framework can accommodate type systems with `MyType` as described in [8].

Another point to note is that our semantic approach tells us little about what *concrete syntax* to use for the various language features. For practical purposes, the syntax presented in Appendix A will need to be supplemented with a variety of sugared forms, which we intend to finalize once an experimental prototype implementation has become available.

3 A simple game model

In this section we make precise the definition of our game model and summarize its key properties. We use a category of games introduced by Lamarche [15], along with a '!' operator described by Hyland in [10]. We concentrate here on details required for specifying our interpretation of object types and class types. Readers not conversant with all the mathematical concepts here should still be able to follow the rest of the paper on an informal level.

Our games will involve *opponent* and *player moves*. It is technically convenient to define a fixed universe of moves from which the moves involved in all games are drawn. In general, moves in a complex game will need to be 'tagged' to show which constituent game they belong to; we therefore define moves to be formal expressions generated by a simple grammar which allows such tagging. For the purpose of the present paper, it suffices to define sets O and P by the following grammar, in which i ranges over $\mathbb{N}$ and z ranges over $\mathbb{Z}$:

$$\begin{array}{lcl} o \in O & ::= & \mathsf{q}(z) \mid \mathsf{andL}(o) \mid \mathsf{andR}(o) \mid \mathsf{impL}(p) \mid \mathsf{impR}(o) \mid \mathsf{tag}(i,o) \\ p \in P & ::= & \mathsf{a}(z) \mid \mathsf{andL}(p) \mid \mathsf{andR}(p) \mid \mathsf{impL}(o) \mid \mathsf{impR}(p) \mid \mathsf{tag}(i,p) \end{array}$$

Fundamentally, every move is either a *question* $\mathsf{q}(z)$ or an *answer* $\mathsf{a}(z)$ labelled by a value z of type `int` (which we regard as representative of all basic types). The remaining constructs in the above grammar are used to specify in which constituent of a complex game the move takes place. We use m to range over $M = O \cup P$. We write And for the set of all moves $\mathsf{andL}(m)$ or $\mathsf{andR}(m)$, and similarly for Imp; likewise, we write Tag for the set of all moves $\mathsf{tag}(i,m)$. We also write Alt for the set of all finite sequences of the form $o_1 p_1 \ldots o_n p_n$ ($n \geq 0$) or $o_1 p_1 \ldots o_n$ ($n \geq 1$). If $S \subseteq$ Alt, we write S^{odd}, S^{even} for the sets of odd- and even-length sequences in S. If $s, s' \in$ Alt, $s' \sqsubseteq s$ means s' is a prefix of s. If $s \in$ Alt and μ is an injective mapping $M \to M$, we write $s \upharpoonright \mu$ for the inverse image under μ of the subsequence of s consisting of moves in the range of μ. We abbreviate $s \upharpoonright \mathsf{tag}(i, -)$ to $s \upharpoonright_i$.

Definition 1 *(i) A* game *G is a non-empty prefix-closed subset of* Alt*. The sequences $s \in G$ are called the* legal plays *or* positions *of G.*

(ii) Given games G, H, we may define games $G \otimes H$, $G \multimap H$, $!G$ as follows:

$$\begin{array}{rcl} G \otimes H & = & \{s \in \mathrm{Alt} \cap \mathrm{And}^* \mid s \upharpoonright \mathsf{andL} \in G,\ s \upharpoonright \mathsf{andR} \in H\} \\ G \multimap H & = & \{s \in \mathrm{Alt} \cap \mathrm{Imp}^* \mid s \upharpoonright \mathsf{impL} \in G,\ s \upharpoonright \mathsf{impR} \in H\} \\ !G & = & \{s \in \mathrm{Alt} \cap \mathrm{Tag}^* \mid \forall i \in \mathbb{N}.\ s \upharpoonright_i \in G\ \wedge \\ & & \qquad \forall s' \sqsubseteq s.\ s' \upharpoonright_i = \epsilon \Rightarrow s' \upharpoonright_{i+1} = \epsilon\} \end{array}$$

Note that there is no notion of winning or losing in our games.

Intuitively, $G \otimes H$ allows two games G, H to be played concurrently, where Opponent is free to choose which component to play in at each stage (Player must follow suit). Likewise, $!G$ allows arbitrarily many copies of G to be played concurrently; at each stage, Opponent may choose to play in any of the existing copies (indexed by $0, \ldots, j-1$, say) or to start a play in the next available fresh copy (indexed by j). The game $G \multimap H$ is best understood in the light of:

Definition 2 *(i) A* strategy *for G is a partial function $f : G^{odd} \rightharpoonup P$ such that* dom f *is closed under odd-length prefixes, and if $f(s) = p$ then $sp \in G$.*

(ii) The category $\mathcal{G}$ is defined as follows: objects are games, morphisms $G \to H$ are strategies for $G \multimap H$, and identities and composition are defined in the natural way (see e.g. [2, 10]).

A strategy f for $G \multimap H$ is something whose back end may be hooked up to a strategy for G, and whose front end will then behave as a strategy for H, with f itself mediating the back-and-forth interactions between G and H.

A crucial point to note is that a strategy f for $G \otimes H$ need not be simply a strategy for G combined with a strategy for H, since (for example) the response of f to an opponent move in G may depend on what moves have occurred in H. This corresponds to the idea that interacting with an H-program can influence the behaviour of a G-program if the two programs share mutable state. Likewise, a strategy f for $!G$ need not be just an $\mathbb{N}$-indexed family of strategies for G; this corresponds to the idea that different (and possibly concurrent) invocations of a G-program may interfere in complex ways if G has mutable state.

The definition of our category of games is rather simple compared with many of those in the literature. This reflects the fact that many of the other models are specifically tailored to match some given language, whereas for us, the idea is to let the language be determined by a simple and natural model. Our model is also comparatively rich in terms of its expressive power, i.e. the class of computations it supports. Some other game models surpass ours in this respect (see e.g. [4]), but the combination of simplicity, intuitiveness and expressivity offered by our model make it, in our view, an attractive choice as a basis for our project. One should also note that expressive richness is a two-edged sword, in that the possibility of modelling additional language features is typically achieved at the cost of more complex denotations even for programs not involving these features, which increases the difficulty of reasoning about such programs.

Our category $\mathcal{G}$ turns out to be well-endowed with computational and mathematical structure. For example:

- The operator ! can be given the structure of a *linear exponential comonad*, so that $\mathcal{G}$ is a model for the $(\otimes, \multimap, !)$ fragment of linear logic (see [10, 29]).

- $\mathcal{G}$ is CPO-enriched, allowing us to interpret recursive programs. Moreover, the set of all games ordered by inclusion is itself (literally) a CPO, so we may immediately interpret recursive types as fixpoints of type operators.

- There is a natural *subtyping* relation on games (cf. [2]): we set $G <: H$ if

$$s \in G^{even} \wedge so \in H \;\Rightarrow\; so \in G \qquad\qquad t \in H^{odd} \wedge tp \in G \;\Rightarrow\; tp \in H$$

- For any set $\mathcal{F}$ of games, we may define its 'generic intersection' $\bigwedge \mathcal{F}$ to be the unique game F satisfying the following (cf. [2]):

$$\begin{array}{rl} \forall s \in F^{even}, o \in O. & (so \in F \;\Leftrightarrow\; \exists G \in \mathcal{F}.\, s \in G \wedge so \in G) \\ \forall t \in F^{odd}, p \in P. & (tp \in F \;\Leftrightarrow\; \forall G \in \mathcal{F}.\, t \in G \Rightarrow tp \in G) \end{array}$$

  This, together with the subtyping relation, allows us to interpret System F style polymorphism along with its bounded and F-bounded variants.

- $\mathcal{G}$ has a *universal object* U, in the sense that every game G is a *retract* of U (i.e. there are morphisms $\iota : G \to U$, $\pi : U \to G$ with $\pi\iota = \mathrm{id}_G$). Specifically, we may take $U = \mathrm{Alt} \cap V^*$, where V is the set of moves $\mathsf{q}(z)$ or $\mathsf{a}(z)$.

- $\mathcal{G}$ has a *universal projection*, corresponding informally to a "type of types". However, our language design does not currently exploit this feature.

Broadly speaking, we wish to model types of our programming language by games, and programs by strategies. For example, the type `int` of natural numbers can be represented by the game $Z = \{\epsilon, ?, \ldots, ?\mathsf{a}(-1), ?\mathsf{a}(0), ?\mathsf{a}(1), \ldots\}$, where $? = \mathsf{q}(0)$ say. Intuitively, the Opponent move ? corresponds to a request from the environment for an expression to be evaluated: the strategy $\{? \mapsto \mathsf{a}(n)\}$ will respond with the Player move n, whilst the *empty* strategy for Z will go undefined at this point. Thus, the game Z really models *computations* of type `int`, playing the role of the object $\mathbb{Z}_\perp$ in classical domain theory.

Actually, since we wish to model a call-by-value language, we need to extend $\mathcal{G}$ to a category $\mathcal{G}_V$ which allows us also to represent already computed *values* of ground type. We do this by means of a general construction given in [3]. In $\mathcal{G}_V$, objects are indexed families of games $(G_i \mid i \in I)$; the idea is that a choice of $i \in I$ allows us to specify the ground type information present in some value, while G_i is used to model any higher type ingredients. A morphism $(G_i \mid i \in I) \to (H_j \mid j \in J)$ is a (dependently typed) function mapping each $i \in I$ to a pair (j, f) where $j \in J$ and $f \in \mathrm{Hom}_{\mathcal{G}}(G_i, H_j)$. Identities and composition in $\mathcal{G}_V$ are defined in the natural way, and we regard $\mathcal{G}$ as a full subcategory of $\mathcal{G}_V$ by identifying a game G with the corresponding singleton family. The definitions of $\otimes$ and ! lift easily from $\mathcal{G}$ to $\mathcal{G}_V$, and it is also straightforward to define a linear function space constructor $\multimap : \mathcal{G}_V^{op} \times \mathcal{G} \to \mathcal{G}$, a lift functor $-_\perp : \mathcal{G}_V \to \mathcal{G}$, and a disjoint sum operation $\oplus : \mathcal{G}_V \times \mathcal{G}_V \to \mathcal{G}_V$. The definitions of subtyping and generic intersection lift to $\mathcal{G}_V$ as follows:

$$(G_i \mid i \in I) <: (H_j \mid j \in J) \;\text{ iff }\; I \subseteq J \text{ and } \forall i \in I.\, G_i <: H_i$$

$$\bigwedge\{(G_i^k \mid i \in I_k) \mid k \in K\} \;=\; \left(\bigwedge\{G_i^k \mid k \in K\} \,\middle|\, i \in \bigcap\{I_k \mid k \in K\}\right)$$

We are now able to give the paradigmatic interpretation of *object types* in our setting. We adopt a structural view of types, according to which the type of an object is essentially the interface that it implements. Suppose an object t has public methods with names $l_1, \ldots, l_n$, where the method l_i has argument type ζ_i and return type ξ_i. Assuming that these types have already been assigned suitable denotations $[\![\,\zeta_i\,]\!], [\![\,\xi_i\,]\!]$ in $\mathcal{G}_V$, the type of x may be interpreted in $\mathcal{G}_V$ by the object

$$\bigotimes_{1 \leq i \leq n} !([\![\,\zeta_i\,]\!] \multimap [\![\,\xi_i\,]\!]_\perp)$$

and the (externally visible) behaviour of t itself will be modelled by a strategy of this type. Here the '$\bigotimes$' embodies the idea that the object's environment may choose, at any time, which method to invoke, while the '!' captures the idea that each method may be called more than once. Note that even in a sequential programming language, multiple concurrent invocations of (the same or different) methods are possible in connection with method arguments of higher type, for instance in the evaluation of re-entrant expressions such as $t.l_1(\texttt{fn}\ x \texttt{=>} t.l_2())$. (By contrast, if the body of the method l_1 itself directly calls l_2, only the outer call will show up in the external view of the behaviour of t.)

An interpretation of objects of this kind was briefly outlined in [4], and is further developed in [29], which analyses the way in which a strategy for the external behaviour of an object o may be obtained from a strategy for its internal behaviour (i.e. its concrete method implementations), and works out the details of a game semantics for an object-oriented language somewhat smaller than Lingay (e.g. without control features), along with a soundness proof relative to a heap-based operational semantics. A corresponding account of the game semantics for Lingay is in preparation.

Some comparison with other (syntactic or semantic) object encodings considered in the literature may be helpful at this point (cf. [9]). Firstly, our interpretation is quite different in flavour from those based on existential types, since an object's internal representation does not appear at all in the denotation of the object, even in a hidden way. In this respect, our approach is closer to a classical 'recursive record' encoding insofar as it offers a purely external view of objects. In contrast to this, however, our encoding is not a *functional* encoding, in that our objects 'manage their own state', so we do not require each method call to return an updated version of the target object. In our view, the key to our approach resides in the inherently stateful nature of $\otimes$ and !, in conjunction with the use of linearity to distinguish between different invocations of a method.

On another axis, our approach relates closely to recent work on *trace semantics* for fragments of Java [1, 12], in which one models objects solely in terms of their interactions with the environment, leading to full abstraction results. The languages treated in these papers are richer than ours in some respects (e.g. concurrency is addressed). However, the set of traces is defined on whole programs in a rather operational, non-compositional way. We believe our more structural approach will facilitate the formulation of natural reasoning principles.

4 Linear types and continuations

We now turn our attention to some particular features of our core language.

Firstly, since our model supports linear (more precisely, *affine*) types, we may incorporate a linear type system up-front in our language. For the time being, we consider the types generated by the grammar:

$$\sigma \ ::= \ \texttt{int} \mid \sigma_1\texttt{*}\sigma_2 \mid \sigma_1\texttt{+}\sigma_2 \mid \sigma_1\texttt{->}\sigma_2 \mid \texttt{!}\sigma_1 \mid \{l_1\texttt{:}\sigma_1\texttt{,}\ \ldots\texttt{,}l_n\texttt{:}\sigma_n\}$$

Semantically, we define $[\![\,\texttt{int}\,]\!] = (\{\epsilon\} \mid i \in \mathbb{Z})$ and $[\![\,\sigma_1\texttt{->}\sigma_2\,]\!] = [\![\,\sigma_1\,]\!] \multimap [\![\,\sigma_2\,]\!]_\perp$; we interpret `!` using !, `+` using $\oplus$, and `*` and record types using $\otimes$. The essential point here is that if we are passed a function of type $\sigma\texttt{->}\tau$, we may invoke this function at most once, whilst a function of type $\texttt{!}(\sigma\texttt{->}\tau)$ may be invoked as many times as we please. The type system of our language statically enforces the 'once-only' discipline for linear values by keeping track of which variables are and are not *reusable* (cf. [5]).

A linear type system offers several potential advantages. Firstly, there are possible implementation benefits, as discussed in [27]. Secondly, for the purpose of reasoning about programs, it is a great help if we know that some expression can be interpreted in the game $[\![\,\sigma\,]\!] \multimap [\![\,\tau\,]\!]_\perp$ rather than the more complicated game $![\![\,\sigma\,]\!] \multimap [\![\,\tau\,]\!]_\perp$, since in the former case we are dealing with a much smaller decision tree. We here focus on a third kind of advantage: the ability to enforce (one kind of) *linear use of continuations* as discussed in [6].

We introduce (various versions of) a control operator `catchcont`, which may be regarded as a typed version of the *resumable exception* construct of Common Lisp, or else as a resumable variant of the Cartwright/Felleisen `catch` operator. Consider first the following typing rules (where ρ, τ are ground types).

$$\frac{x : \rho\texttt{->}\sigma \ \vdash \ e : \tau}{\vdash \texttt{catchcont}_1\ x \texttt{=>} e \ :\ \{\texttt{value:}\,\tau\}\ \texttt{+}\ \{\texttt{arg:}\,\rho\texttt{,}\ \texttt{resume} : \sigma\texttt{->}\tau\}} \qquad \frac{x : \texttt{!}(\rho\texttt{->}\sigma) \ \vdash \ e : \tau}{\vdash \texttt{catchcont}_2\ x \texttt{=>} e \ :\ \{\texttt{value:}\,\tau\}\ \texttt{+}\ \{\texttt{arg:}\,\rho\texttt{,}\ \texttt{resume} : \sigma\texttt{->!}(\rho\texttt{->}\sigma)\texttt{->}\tau\}}$$

To evaluate $\texttt{catchcont}_1\ x \texttt{=>} e$, we try to evaluate e, treating x as a dummy variable of function type. If the evaluation completes without ever attempting to call x, we obtain a result of type τ. Otherwise, if we attempt to call x on some argument $y : \rho$, the value of y is returned, along with a function allowing us to resume the evaluation of e at some later date (this latter feature is what distinguishes our operator from `catch`).

Linear typing is used in two distinct ways to enforce the safety of this operator. Firstly, the `resume` function may only be invoked once; this in essence means that the continuation at the point of calling x is only used 'linearly'. Secondly, the form of the premise means that the evaluation of e can attempt to call x at most once; thus, when we invoke the `resume` function, there is no danger that we will encounter a second application of x. A more general behaviour is admitted by the operator $\texttt{catchcont}_2$, where e may attempt to call x many times; here we must supply the `resume` function with an operator to provide the value of x for the purpose of subsequent invocations. (We can, if we like, trap the second use of x by a further use of $\texttt{catchcont}_2$.)

The restriction that τ be of ground type is important. If the evaluation of $\texttt{catchcont}_2\ x \texttt{=>}\ e$ were to return a result of some non-ground type τ, further interaction with this result might run into lingering occurrences of x which are now out of scope. However, we can overcome this restriction with a yet more powerful version of `catchcont`. Here we split the type of e into a ground type component τ and a component of arbitrary type τ'. In order to make use of the latter, one must again supply an argument giving the 'meaning' of x.

$$\frac{x : \texttt{!}(\rho\texttt{->}\sigma) \ \vdash\ e : \tau\texttt{*}\tau'}{\begin{array}{l}\vdash \texttt{catchcont}_3\ x \texttt{=>}\ e \\ \quad : \ \{\texttt{value:}\tau\texttt{, more:!}(\rho\texttt{->}\sigma)\texttt{->}\tau'\}\ \texttt{+} \\ \quad\ \ \{\texttt{arg:}\rho\texttt{, resume} : \sigma\texttt{->!}(\rho\texttt{->}\sigma)\texttt{->}\tau\texttt{*}\tau'\}\end{array}}\ \rho, \tau \text{ ground}$$

The details of an operational semantics for this operator are included in Appendix A.

This last version of `catchcont` was suggested to us by semantic considerations, and in particular by the idea that all computable strategies (at the types considered here) ought to be definable in the language. As explained in [16], we can establish both definability and full abstraction at such types with the help of the universal object U from Section 3, which is essentially the denotation of the *universal type* `univ` = {`play:!(int->int)`}. The properties of definability and full abstraction are trivial for `univ` itself, and they extend to all types if we can show that for every type τ we have a language-definable retraction $[\![\tau]\!] \lhd [\![\texttt{univ}]\!]$ (see [16]). For this, it suffices to check each of the types `int`, `univ*univ`, `univ+univ`, `univ->univ`, `!univ` is a definable retract of `univ`.

It turns out that all the relevant operations are programmable using familiar language constructs, except for the abstraction operation (`univ->univ`) $\rightarrow$ `univ`, which calls for something with the power of $\texttt{catchcont}_3$.[2] In Figure 2 we give the code for the abstraction operation, which constitutes the most interesting part of the definability proof. (An ML version of this program is given in [22].) We write `pre_univ` for the type `int->int*univ`, and we presuppose here the existence of a program `fold:pre_univ->univ` which collects a family of `univ` objects into a single object which chooses one of them according to the argument passed on the first invocation of `play`, and a program `unfold:univ->pre_univ` for the corresponding inverse operation. The programs `tagL`, `tagR` serve as the left and right components of some coding function $\mathbb{Z} + \mathbb{Z} \rightarrow \mathbb{Z}$.

Our semantics has thus guided us to an interesting new language construct, which naturally provides support for 'coroutining' styles of programming within our type system. The fact that such a construct (in the context of 'linear PCF', for example) yields full abstraction and definability for our model $\mathcal{G}$ appears to be a new theoretical result in game semantics, although Laird [14] shows that a closely related coroutining operation (which is definable from ours) yields full abstraction and definability for the corresponding intuitionistic model.

All these versions of `catchcont` can in essence be implemented in New Jersey ML using `callcc` (see [22]). However, there are several possible advantages to packaging up the power of `callcc` in the above constructs. Firstly, the types involved are more familiar, and for some programmers more intuitive. Secondly, we get better run-time security, since our operators can never give rise to a `TopLevelCallCC` error. (Indeed, a theorem asserting the run-time security of our dummy variable mechanism follows readily from the adequacy of

[2]In fact, the first author originally conjectured that $\texttt{catchcont}_2$ would suffice, and it was the attempt to code the abstraction operation that led us to the definition of $\texttt{catchcont}_3$.

```
val abstract : (univ->univ)->univ =
   split rec (abstr',abstr'') =>
          (fn P:pre_univ->unit =>
             fold (fn h:int => abstr''
             (fn g:pre_univ => unfold (P g) h)),
           fn R:pre_univ->int*univ =>
                  case catchcont x:!pre_univ => R(der x) of
                     inl {result=r, more=P'} => (tagL r, abstr' P')
                   | inr {arg=q, resume=Q} =>
                     (tagR q,
                     fold (fn h:int => abstr''
                              (fn branches:pre_unit =>
                                  Q (a, fold branches) dummy))))
   as (abstr',abstr'') in
     fn F:univ->univ =>
          abstr' (fn branches:int->int*univ => F (fold branches))
   end
```

Figure 2: Code for abstraction operation

our denotational semantics.) Thirdly, our operators enforce linear use of the underlying continuations, which significantly simplifies the task of implementation of the language as it avoids the need for reification of the stack.

Finally, we mention that our category $\mathcal{G}$ also supports a more powerful linear exponential '!' operator which would allow one to model 'non-linear' uses of continuations (e.g. backtracking). The '!' in question is described (implicitly) in Section 6.6 of [16], and provides the basis of the Stratagem system [21], which allows dynamic explorations (possibly including backtracking) of the strategies underlying a given ML program. However, we have chosen not to follow this route, partly because of the known implementation difficulties, and also in the interests of simplying reasoning about programs: working with the more subtle linear exponential would result in much larger decision trees, and a correspondingly finer notion of observational equivalence, imposing an added burden on reasoning even for programs not involving control features.

5 Class implementations

Next, we explain our treatment of class implementations and their semantics. We adopt a system of *class types* close to that of [7]. We concentrate here mostly on a 'default' scenario in which all methods are public and all fields are protected.

Omitting some bells and whistles, a class implementation c will be an expression of some *class type* of the form `classimpl` τ_f`,`τ_m`,`τ_k, where τ_f is a labelled record type giving details of any fields; $\tau_m = \{l_1\texttt{:!(}\zeta_1\texttt{->}\xi_1\texttt{)}, \ldots, l_n\texttt{:!(}\zeta_n\texttt{->}\xi_n\texttt{)}\}$ gives the names and externally visible types of the methods (so that objects obtained as instances of c will have type τ_m); and τ_k is the argument type of the (single) constructor for the class.

In any given object of such a class, the concrete implementation of the methods may be considered as having type

$$\tau_m \natural \tau_f \;=\; \{l_1 \texttt{:!(}\zeta_1\texttt{*}\tau_f\texttt{->}\xi_1\texttt{*}\tau_f\texttt{)}\texttt{,}\ \ldots\texttt{,}\ l_n\texttt{:!(}\zeta_n\texttt{*}\tau_f\texttt{->}\xi_n\texttt{*}\tau_f\texttt{)}\}$$

(Note the 'functional' treatment of the internal state at this point, in contrast to the imperative treatment in [7]. This has some rather subtle implications for the precise semantics of method calling, since (in effect) an object's state can only be updated in-place at the *end* of a method call; however, these subtleties only manifest themselves for re-entrant method calls involving higher-order arguments.)

However, in the class implementation itself, we wish to leave these method implementations 'open' in order to allow for method overriding in subclasses. We achieve this using the idea of *early self binding* as in [25, 28]. Specifically, in a class implementation, the method bodies (of type $\tau_m \natural \tau_f$) are parameterized by a variable `self` (also of type $\tau_m \natural \tau_f$). This operator $\tau_m \natural \tau_f$ `->` $\tau_m \natural \tau_f$ may be modified and extended in subclasses, and its fixpoint is taken at object creation time to yield the concrete implementation of the object in question. As is well-known, this correctly captures the effect of dynamic method invocation.

Two further ingredients should be mentioned. Firstly, our method implementations should also be parameterized by a variable `super` in order to admit superclass method calls. Secondly, as explained e.g. in [8], in order to allow for the possibility of additional fields present in future subclasses, we need to replace τ_f above by an arbitrary type τ_f`**`ρ, where ρ is a *row variable* ranging over sets of labelled record type components, and `**` denotes record extension. (The more usual formulation in terms of an arbitrary type ζ`<:`τ_f does not work out correctly in our setting.)

In the light of this discussion, we may formulate the following typing rules for our basic (unsugared) class constructs. Here, for convenience, we temporarily assume that the language supports explicit System F style quantification over row variables;[3] later we will show how one may dispense with this assumption.

$$\frac{}{\texttt{root} : \texttt{classimpl}\{\}\texttt{,}\{\}\texttt{,}\{\}} \qquad \frac{c : \texttt{classimpl}\ \tau_f\texttt{,}\tau_m\texttt{,}\tau_k}{\texttt{new}\ c : \texttt{!(}\tau_k\texttt{->}\tau_m\texttt{)}}$$

$$\frac{\begin{array}{c} c : \texttt{classimpl}\ \tau_f\texttt{,}\tau_m\texttt{,}\tau_k \\ e_m : \texttt{polytype}\ \rho\ \texttt{=>}\ \texttt{!(}\tau_{super}\texttt{->}\tau_{self}\texttt{->}\tau_{self}\texttt{)} \\ e_k : \texttt{!(}\tau'_k\texttt{->}\tau_k\texttt{*(}\tau_f\texttt{->}\tau'_f\texttt{))} \end{array}}{\texttt{extend}\ c\ \texttt{with}\ e_m\texttt{,}e_k : \texttt{classimpl}\ \tau''_f\texttt{,}\tau''_m\texttt{,}\tau'_k} \qquad \begin{array}{l} \tau_{super} = \tau_m \natural (\tau''_f\texttt{**}\rho) \\ \tau_{self} = \tau'_m \natural (\tau''_f\texttt{**}\rho) \\ \tau''_f = \tau_f \sharp \tau'_f \\ \tau''_m = \tau_m \sharp \tau'_m \\ \tau_f, \tau'_f \text{ have disjoint labels} \end{array}$$

This can be seen as a type-theoretic presentation of the kernel of a Java-style class system with inheritance. Classes are constructed by (repeatedly) extending a single vacuous root class. The constructor implementation e_k returns a value to be passed up to the superclass constructor, together with an operation for filling out the new fields once the superclass fields have been initialized. We will also (temporarily) assume that the language contains a simpler version of the extend rule for *final* classes, i.e. expressions of the form

[3]This feature will probably not be included in Eriskay, owing to the complications involved in avoiding component name clashes.

`final extend` c `with` e_m`,`e_k, in which the variable ρ is omitted everywhere. (This feature is present in Eriskay, but not in Lingay.)

What is an appropriate denotational semantics for (non-final) class types? First, we note that the behaviour of a class implementation c : `classimpl` τ_f`,`τ_m`,`τ_k may intuitively be captured by a strategy of the following 'representing type':

$$\mathsf{classrep}(\tau_f, \tau_m, \tau_k) \;=\; (\texttt{polytype}\ \rho\ \texttt{=>}\ \texttt{!}(\tau_m\natural(\tau_f\texttt{**}\rho)\texttt{->}\tau_m\natural(\tau_f\texttt{**}\rho)))\ \ \texttt{*}\ \ \texttt{!}(\tau_k\texttt{->}\tau_f)$$

Here the first component gives the behaviour of the method implementations relative to `self`, and the second component gives the behaviour of the constructor implementation. (There is no need to incorporate a dependence on `super` here, since any class expression may be normalized to one that directly extends the root class.) This suggests that we could simply *define* $[\![$ `classimpl` τ_f`,`τ_m`,`τ_k $]\!]$ to be $[\![\,\mathsf{classrep}(\tau_f, \tau_m, \tau_k)\,]\!]$ (the `polytype` being interpreted by generic intersection over all possible ground instantiations of ρ). In fact, this works out well, and allows us to give adequate interpretations of `extend` and `new` expressions (cf. [19, 29]).

Actually, this interpretation of class types turns out to be 'best possible', in that the types `classimpl` τ_f`,`τ_m`,`τ_k and $\mathsf{classrep}(\tau_f, \tau_m, \tau_k)$ are *definably isomorphic*. That is, there are language-programmable mappings in both directions whose denotations in $\mathcal{G}_V$ are mutually inverse. Specifically, given a variable `rep` of type $\mathsf{classrep}(\tau_f, \tau_m, \tau_k)$, we may define a corresponding class implementation by the expression is straightforwardly defined by the expression

```
split rep as (meth-rep, constr-rep) in
    extend triv with meth-rep, constr-rep
end
```

Conversely, given a variable `impl` of type `classimpl`(τ_f, τ_m, τ_k), we may extract the underlying representation by means of a programming trick involving overriding and `super`. We illustrate this in Figure 3 for the case τ_f = `tf`, τ_k = `tk`, τ_m = {`m:!(ta->tb)`}.

Thus, questions of observational equivalence for class implementations can be reduced to the corresponding questions for their representations. Furthermore, if τ_f, τ_m, τ_k do not themselves involve class types or polymorphism, then our interpretation of $\mathsf{classrep}(\tau_f, \tau_m, \tau_k)$ is itself fully abstract (cf. Section 4), so we have a fully abstract model for `classimpl` τ_f`,`τ_m`,`τ_k.

We have been over-simplifying slightly here, since (as we will explain in Section 6) only *argument-safe* method bodies may legitimately appear in a class implementation. This does not pose a serious problem, since we may cut down from arbitrary strategies to argument-safe ones by means of an appropriate retraction (see [22]). Adapting the definition of $[\![$ `classimpl` τ_f`,`τ_m`,`τ_k $]\!]$ to take account of this, we do indeed obtain full abstraction for class implementations as claimed. Moreover, exactly the same picture remains valid if *private* fields of ground type are added to the mix (the situation for non-ground private fields requires further investigation).

In fact, we can make our interpretation of class implementations even more concrete than this, and (at the same time) dispense with any dependence on row variable quantification in the language. A slightly troubling feature of the above interpretation is that the game $[\![$ `classimpl` τ_f`,`τ_m`,`τ_k $]\!]$, being defined as a generic intersection, contains rather a lot of junk for our purposes. (More specifically, we only really wish to consider those 'uniform'

```
(polytype '@r => prom
   (fn source: {m: !(ta*(tf**'@r) -> tb*(tf**'@r))} =>
      let val impl' = final extend impl with
           prom (fn super => fn self =>
                      {m = source.m,
                       m' = super.m,
               read = prom (fn (x:unit,s:tf**'@r)=>(s,s))} ) ,
                   prom (fn s:tf**'@r => (dummy, fn t:tf => s))
      in {m = prom (fn (a:ta, s:tf**'@r) =>
                   let val o = new impl' s in
                      o.m'(a) ; o.read void
                   end)}
      end) ,
 let val impl'' = final extend impl with
                   prom (fn super => fn self =>
                      {read = prom (fn (x:unit,s:tf) => (s,s))}) ,
                   prom (fn k:tk => (k, fn s:tf => s))
 in prom (fn k:tk => der (new impl'' k).read void)
 end)
```

Figure 3: Code for class representation extraction

strategies which behave as copycat on the portions corresponding to the indeterminate game for ρ.) However, we have the following useful fact: a polymorphic strategy of such a type is uniquely determined by *any* of its instances (e.g. the instantiation given by $\rho := \{\,\}$). Intuitively, this is because in the course of any play of this strategy, there is, at any time, at most one value of type τ_f**ρ available, since ρ is treated by the type system as non-reusable. We may think here in terms of a single 'token' which we give away whenever we invoke a `self` or `super` method implementation, and which is returned to us when the method call returns. It is now fairly clear that the way in which the token is threaded through the computation can be recovered from just the single instance $\rho := \{\,\}$.

More precisely, the strategies of interest in $[\![\,\texttt{polytype}\ \rho\ \texttt{=>}\ \texttt{!}(\tau_m\natural(\tau_f\texttt{**}\rho) \to \tau_m\natural(\tau_f\texttt{**}\rho))\,]\!]$ may equally be represented as the strategies for a certain retract of $[\![\,\texttt{!}(\tau_m\natural\tau_f \to \tau_m\natural\tau_f)\,]\!]$. This suggests that we may work with the alternative representation

$$\mathsf{classrep}'\ \tau_f\,,\tau_m\,,\tau_k \;=\; \texttt{!}(\tau_m\natural\tau_f \to \tau_m\natural\tau_f)\ \texttt{*}\ \texttt{!}(\tau_k \to \tau_f)$$

Moreover, in the presence of `catchcont`, both halves of the corresponding retraction turn out to be syntactically definable, so that we may define $[\![\,\texttt{classimpl}\ \tau_f\,,\tau_m\,,\tau_k\,]\!]$ to be the relevant retract of $[\![\,\mathsf{classrep}'\ \tau_f\,,\tau_m\,,\tau_k\,]\!]$. This gives us a remarkably concrete semantics for class implementations for which we have both full abstraction and definability; moreover, the limited row polymorphism available in Lingay suffices to define the relevant programs.

These ideas offer a seemingly novel approach to the question of observational equivalence for non-final class implementations (where 'observations' may involve subclassing the given class), at least for our chosen language. Whereas in e.g. [13, 26] this question is reduced to the existence of an appropriate logical relation or bisimulation, we reduce it to the question of equality for particular strategies which may be described very concretely. We expect this to make the problem much easier, although we have yet to work through any particular cases in detail.

We note in passing that the question of semantics for *final* classes is very much easier: if c is some class implementation which we are not allowed to extend, we may take its interpretation $[\![c]\!]$ to be simply $[\![\texttt{new}\ c]\!]$, a strategy of type $[\![\tau_k \to \tau_m]\!]$ where we already have full abstraction if τ_k, τ_m are simple types.

One further aspect of our class system deserves a brief mention. If the type τ_f of an object's internal state is not reusable, it will be impossible for the object's environment to engage in two method invocations concurrently, since the state is 'used up' by the first method call and we must wait for the updated state to be returned before initiating a second call. Since methods with higher-order arguments naturally give rise to the possibility of concurrent invocations via re-entrant method calls, we had better restrict arguments to be of *ground type* for classes with non-reusable state. This motivates a system of *linear classes* in which this restriction is imposed, and it turns out that this gives just what is needed to achieve definability and full abstraction at all (linear) types.

As an aside, we also believe that it will be of interest to explore in more detail the 'functorized' styles of programming in the large which our first-class treatment of class implementations appears to support.

6 Restrictions on higher-order store

We now come to consider the most severe and perhaps the most embarrassing limitation of our core language: the use of higher-order store is significantly restricted, so that it is not even possible to implement a simple higher-order store cell. Of course, this is not fatal from the programmer's point of view, since the restrictions may be bypassed in the full language; however, we shall argue that there are still reasons why programmers should care about these restrictions.

We start by explaining what is permitted in the core language, what is excluded, and why. As a running example, we consider a class `c` containing a field `f` of type `int->int`, which we regard as representative of arbitrary non-ground types (including object types). Firstly, we do allow objects to have fields of non-ground type, and initial values of such fields may be supplied via constructor arguments. For instance, the evaluation of `new c (g:int->int)` might perform `f:=g`. So in this sense, we may store non-ground values on the heap.

However, we do not permit a *method* to store a non-ground argument in a field of the target object. (For instance, a method call `m1(g)` cannot perform `f:=g`.) More generally, a method may not store any non-ground value obtained from such an argument (thus, a call `m2(h:int->int->int)` cannot perform `f:=h(5)`). The precise meaning of 'obtained from' here will be clarified below.

On the positive side, a method call *can* do any or all of the following:

- Interact with non-ground fields (e.g. a call `m()` can return `f(5)`).
- Update non-ground fields (`m()` can perform `f:=fn i=>f(i+1)`).

- Make unrestricted use of *ground type* information extracted from its argument (`m1(g)` can perform `p:=g(5); f:=fn i=>i+p`).
- Export non-ground field contents as return values (`m()` can return `f`).
- Use arguments without restriction in return values (`m1(g)` can return `g`).

To summarize, information of non-ground type can flow freely from the object's state to the return value, or from the argument to the return value, but any dependency of the updated state on the argument must be funnelled through a value of ground type. A useful intuition may be gained by thinking about objects and pointers: in the course of a method call, the target object may make use of pointers acquired from the method argument, but it is not allowed to retain hold of such pointers after we return from the method call.

Programs which obey the above discipline are called *argument-safe*. This discipline is statically enforced by the typing rules of our core language, which keep track of which variables are safe (i.e. may be used unrestrictedly), and which terms depend on unsafe variables only in safe ways. Here are some sample typing rules in a simplified form; the idea is worked out in detail for Lingay in Appendix A.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{}{\Gamma, x : \tau \ \mathsf{safe} \vdash x : \tau \ \mathsf{safe}} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau \ \mathsf{safe}}\ \tau \text{ ground}$$

$$\frac{\Gamma,\ x_{arg} : \rho,\ y_{state} : \sigma \vdash e_{return} : \xi \qquad \Gamma,\ x_{arg} : \rho,\ y_{state} : \sigma\,\mathsf{safe} \vdash e'_{state} : \sigma\ \mathsf{safe}}{\Gamma \vdash \texttt{fn}\ \langle x_{arg}, y_{state}\rangle \texttt{=>}\ \langle e_{return}, e'_{state}\rangle\ :\ \rho\texttt{*}\sigma\texttt{->}\xi\texttt{*}\sigma\ \mathsf{argsafe}}$$

To explain the reason for these rather odd syntactic restrictions, let us look at what goes wrong in our semantics if a program is not argument safe. For example, suppose `s` is a store cell for the type `int->int` with `get` and `put` methods, and consider the interactions with `s` when we try to execute `s.put(fn x=>x); s.get()(5)`:

	`put : (int->int)`	`-> unit`	`get : unit ->`	`(int->int)`
O		?–		
P		!()		
O			?()	
P			!–	
O				?5
P	?5			

In the last move here, Player (presumably) will respond to Opponent's query by interrogating the argument passed to `put`. However, at this point in the dialogue, such a move will be out of turn from the point of view of the `put` component of the game, where it is Opponent's turn to move.

Naturally, these difficulties can be overcome by moving to a more elaborate game model (as in [4]), but at the cost of complicating the whole setup. More specifically, the decision tree that represents the behaviour of a program up to observational equivalence will now be larger inasmuch as it may involve non-alternating subplays in certain components, resulting in more fine-grained observational distinctions even for purely functional programs. One response would be to accept the added difficulty in reasoning about programs as a price worth paying for the expressive power we gain; alternatively, we may ask whether there are any positive benefits to be reaped from the above restrictions.

In fact, argument-safety does give rise to various pleasant consequences – for instance, it can be shown that argument-safe programs never give rise to *cycles* in the heap – and this reinforces the impression that argument-safety is somehow a natural condition which facilitates reasoning about programs. However, we will here focus on an application to a problem of more immediate relevance to programmers: the static control of *exceptions* in the presence of higher-order store.

In general, the combination of exceptions and higher-order store can give rise to strange phenomena. In ML, for instance, we may create a store cell, then use it to store a function with raises a *locally declared* exception. Later, outside the scope of this exception, we may apply this function, causing an anonymous (out of scope) exception to be raised. In Java, by contrast, 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 interface `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):

```
interface Function {Element f (Element x);}
interface List {void add (Element x);
                void map (Function F);
                Element nth (int n);}
```

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.

In our full language, we plan to introduce a more programmer-friendly exception mechanism than that offered by `catchcont`, in which the type of an expression M is in general annotated by a single set X of exceptions, documenting the exceptions which M might be responsible for raising in some context $C[M]$. Some sample typing rules are:

$$\frac{M : (\sigma\texttt{->}\tau)^X \qquad M' : \sigma^{X'}}{M\,M' : \tau^{X \cup X'}} \qquad\qquad \frac{M : \tau^{X\cup\{e\}} \qquad M' : \tau^{X'}}{M\ \texttt{handle}\ e\ \texttt{=>}\ M' \ : \tau^{X\cup X'}}\ \ \tau \text{ ground}$$

Furthermore, for any specified type τ^X we may declare an associated type of *references* to values of τ^X. For example:

```
reftype EltRef for Element{e} with (ref,deref) ;
```

has the effect of introducing an abstract type `EltRef` for `Element{e}`, which comes equipped with referencing and dereferencing operations which are bound to `ref`, `deref`. From the point of view of the type system, `EltRef` is treated as a ground type; there is therefore no violation of argument-safety in writing a `List` class for storing *references* to elements. (This shows how the higher-order store restrictions may be circumvented in the full language when necessary.) The `add` method of such a class will have type `EltRef->unit`, and this enforces the fact that we may not add elements that raise exceptions other than `e`. (Note that the scope of `e` must enclose that of `EltRef` itself.) By contrast, the `map` method, being argument-safe anyway, can be given the type `!(Element->Element)->unit`, and may be invoked with an argument `F` involving any exceptions whatever, by virtue of the above typing rule for application.

Although in this article we have not given details of our proposal for interpreting the full language within our game model, we expect to be able to do so, and the soundness of this interpretation will then readily imply the following *exception safety* property:

if the evaluation of $M : \tau^X$ raises e, then $e \in X$.

Furthermore, we anticipate that what we are able to achieve for exceptions will equally be feasible for other impure effects: for example, our type system will likewise enforce a non-extrusion property for references themselves.

It is pleasing to find that the notion of argument-safety, which arose from a quest for mathematical simplicity, seems also to have applications to more practical problems. We regard this, along with the other results described in this paper, as supporting our view that a simple mathematical model can lead to a harmonious language design with some unexpected good properties. We expect our game model to remain a fruitful source of inspiration as our language design continues to take shape, and thereafter in the design of suitable program logics for the language.

References

[1] Àbrahàm, E., Bonsangue, M., de Boer, F., Steffen, M.: Object connectivity and full abstraction for a concurrent calculus of classes. Proc. 1st ICTAC, Z. Liu and K. Araki, eds., Springer LNCS 3407 (2004) 37–51

[2] Abramsky, S.: Semantics of Interaction: an introduction to game semantics. Proc. CLiCS Summer School, P. Dybjer and A. Pitts, eds., CUP (1997) 1–31

[3] Abramsky, S., McCusker, G.: Call-by-value games. Proc. 11th CSL, M. Nielsen and W. Thomas, eds., Springer LNCS 1414 (1998) 1–17

[4] Abramsky, S., Honda, K., McCusker, G.: A fully abstract game semantics for general references. Proc. 13th LICS, IEEE Press (1998) 334–344

[5] Barber, A., Plotkin, G.: Dual intuitionistic linear logic. University of Edinburgh Technical Report ECS-LFCS-96-347 (1997)

[6] Berdine, J., O'Hearn, P., Reddy, U., Thielecke, H.: Linear continuation-passing. Higher Order and Symbolic Computation vol. 15 (2002) 181–208

[7] Bono, V., Patel, A., Shmatikov, V., Mitchell, J.: A core calculus of classes and objects. Proc. MFPS'99, Elsevier ENTCS 20 (1999) 1–22

[8] Bruce, K.: Foundations of Object-Oriented Languages. MIT Press (2002)

[9] Bruce, K., Cardelli, L., Pierce, B.: Comparing object encodings. Information and Computation 155 (1999) 108–133

[10] Hyland, M.: Game semantics. Proc. CLiCS Summer School, P. Dybjer and A. Pitts, eds., CUP (1997) 131–184

[11] Jacobs, B. *et al*: Reasoning about Java Classes (Preliminary Report). Object-Oriented Programming Systems, Languages and Applications, ACM Press (1998) 329–340

[12] Jeffrey, A., Rathke, J.: A fully abstract may testing semantics for concurrent objects. Theor. Comp. Sci. 228 (2005) 17–63

[13] Koutavas, V., Wand, M.: Reasoning about class behaviour. Presented at FOOL'07.

[14] Laird, J.: Functional programs as coroutines. Draft paper (2007).

[15] Lamarche, F.: Sequentiality, games and linear logic. Manuscript (1992)

[16] Longley, J.: Universal types and what they are good for. Proc. 2nd ISDT, GQ Zhang, J. Lawson, Y.-M. Liu and M.-K. Luo, eds., Kluwer (2003) 25–63

[17] Longley, J., Plotkin, G.: Logical full abstraction and PCF. Tbilisi Symposium on Logic, Language and Computation, SiLLI/CSLI (1997) 333–352

[18] Longley, J., Pollack, R.: Reasoning about CBV functional programs in Isabelle/HOL. In Theorem Proving in Higher Order Logics, 17th International Conference, 2004, Utah, proceedings, Springer LNCS 3223 (2004), 201-216.

[19] Longley, J., Wolverson, N.: Game semantics for object-oriented languages: a progress report. Presented at GaLoP II, Seattle (2006). Available online from `homepages.inf.ed.ac.uk/jrl/Research`

[20] Longley, J.: Definition of the Eriskay programming language. Working draft (63 pages) available from the above URL.

[21] Longley, J.: Stratagem. NJ-SML software available from the above URL (2002)

[22] Longley, J.: Catchcont and friends. NJ-SML source file from the above URL (2007)

[23] Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. Proc. 15th POPL Symposium, ACM Press (1988) 47–57

[24] Milner, R., Tofte, M., Harper, R., MacQueen, D.: The definition of Standard ML (revised). MIT Press (1997)

[25] Reddy, U.: Objects as closures: Abstract semantics of object-oriented languages. Proc. ACM Symp. Lisp and Functional Programming Languages (1998) 289–297

[26] Sumii, E., Pierce, B.: A bisimulation for type abstraction and recursion. Proc. 32nd POPL Symposium, ACM Press (2005) 63–74

[27] Wadler, P.: Linear types can change the world! Programming Concepts and Methods, Israel, North-Holland (1990)

[28] Wand, M.: Type inference for objects with instance variables and inheritance. Theoretical Aspects of Object-Oriented Programming, C. Gunter and J. Mitchell, eds., MIT Press (1994) 97–120

[29] Wolverson, N.: Game semantics of object oriented languages. PhD thesis, University of Edinburgh, submitted December 2007.

Appendix A: Provisional definition of Lingay

The Lingay programming language is a medium-scale fragment of the Eriskay language, embodying many of the innovative features of Eriskay and intended as a suitable target language for theoretical study, but not possessing the full range of features needed for a practical programming language. It is intended that Lingay is literally a sublanguage of Eriskay in the following strong sense:

- Every valid Lingay program is a valid Eriskay program with the same observable evaluation behaviour.
- If two Lingay programs are observationally equivalent in Lingay, they are also observationally equivalent in Eriskay.
- Every Eriskay program whose type exists in Lingay is observationally equivalent (in Eriskay) to some Lingay program.

This appendix contains a provisional formal definition of Lingay, to a level of rigour which provides a suitable basis for proofs of metatheoretical results. Note that the definition of Lingay may be subject to further change in the near future: in particular, a backtracking version of `catchcont` is under consideration for inclusion, as is a more imperative treatment of method implementations based on `read` and `write` operations.

Where convenient we adopt a similar style of definition for the two languages, although we are occasionally somewhat less formal and complete here than in the full Eriskay definition.

Lexical matters

A Lingay program may be regarded as a sequence of lexical tokens. Each token is a non-empty sequence of non-whitespace ASCII characters, and tokens are of two kinds: *reserved* and *non-reserved*. The reserved tokens featuring in Lingay are as follows:[4]

```
{ } [ ] ( ) : ; , ! -> => :: % | = $ ** +>
as bool case classimpl der else end extend false
fn if in int linear new of prom rec split then triv true with
```

Each non-reserved token belongs to one of six lexical categories, which are pairwise disjoint. In addition, certain reserved tokens are classified as belonging to one of these categories. The lexical categories are as follows; we refer to the Eriskay definition for full details.

- *Identifiers* such as `x` and `foo5`, ranged over by the metavariables x, y, k, f, m, t according to the context.
- *Special constants*, which in Lingay are just the tokens `_intPlus`, `_intMinus`, `_intTimes` and `_intLeq`.
- *Boolean literals*, which are just the reserved tokens `true` and `false`.
- *Integer literals* such as `00` and `+23`.
- *Row variables* such as `'@a`, ranged over by ρ.
- *Type constants*, which in Lingay are just the reserved tokens `bool` and `int`, ranged over by κ.

We also define the class of *expression constants* (ranged over by *ec*) to consist of the special constants, boolean literals, and integer literals.

Types

The syntax of *type expressions* is given in Figure 4. We let τ, σ, ζ, ξ range over type expressions and π over labelled product type expressions.

Strictly speaking, type expressions are parse trees generated by the above grammar rather than sequences of lexical tokens, as the grammar is formally ambiguous. However, we shall of course denote type expressions by sequences of tokens, freely adding brackets to disambiguate where necessary.[5] As usual, we treat `->` as right-associative.

The intention behind row variables is that they should range over sets of components for labelled product types; the type expression `{a:int}**`ρ refers to the labelled product type obtained by extending `{a:int}` with whatever components are included in ρ; thus,

[4]In addition, for the sake of compatibility, all other reserved tokens of Eriskay should be considered as reserved in Lingay. We refer to the Eriskay definition for the complete list.

[5]The brackets in binary product types are made obligatory in Lingay to avoid confusion over type expressions such as τ_0`*`τ_1`*`τ_2 which in Eriskay denote ternary products.

type-expr	::=	κ	(nullary type constant)
	$\vert$	`(` *type-expr* `*` *type-expr* `)`	(binary product)
	$\vert$	*lab-prod-type*	(labelled product)
	$\vert$	`[` *Clist*(*comp-type*) `]`	(labelled sum)
	$\vert$	`!` *type-expr*	(reusable type)
	$\vert$	*type-expr* `->` *type-expr*	(linear function type)
	$\vert$	*lin-opt* `classimpl` *type-expr*`,` *type-expr*`,` *type-expr*	(class implementation)
lab-prod-type	::=	`{` *Clist*(*comp-type*) `}`	(simple labelled product)
	$\vert$	*lab-prod-type* `**` ρ	
comp-type	::=	k`:`*type-expr*	(component typing)
lin-opt	::=	ϵ $\vert$ `linear`	

Figure 4: Context-free grammar for types

instantiating ρ to {`b:bool`} yields the type expression {`a:int, b:bool`}. The following "shallow" definition of row variable instantiation $\pi[\rho := \pi']$ suffices for our purposes. If

$$\begin{aligned}\pi &= \{k_0 : \tau_0\texttt{,} \cdots \texttt{,}k_{n-1} : \tau_{n-1}\}\texttt{**}\rho_0 \cdots \texttt{**}\rho_{m-1}\\ \pi' &= \{k'_0 : \tau'_0\texttt{,} \cdots \texttt{,}k'_{n'-1} : \tau'_{n'-1}\}\texttt{**}\rho'_0 \cdots \texttt{**}\rho'_{m-1}\end{aligned}$$

where $k_0, \ldots, k_{n-1}, k'_0, \ldots, k'_{n'-1}$ are all distinct, $\rho_0, \ldots, \rho_{m-1}, \rho'_0, \ldots, \rho'_{m'-1}$ are all distinct, and $\rho = \rho_i$, we define $\pi[\rho := \pi']$ to be the labelled product type expression

$$\begin{gathered}\{k_0 : \tau_0\texttt{,} \cdots \texttt{,}k_{n-1} : \tau_{n-1}\texttt{,}k'_0 : \tau'_0\texttt{,} \cdots \texttt{,}k'_{n'-1} : \tau'_{n'-1}\}\\ \texttt{**}\rho_0 \cdots \texttt{**}\rho_{i-1}\texttt{**}\rho_{i+1} \cdots \texttt{**}\rho_{m-1}\texttt{**}\rho'_0 \cdots \texttt{**}\rho'_{m-1}\end{gathered}$$

Note that $\pi[\rho := \pi']$ is undefined if π' contains component names clashing with those in π, or if the row variables themselves clash.

The rules below generate judgements of the form $\vdash \tau \propto sd$, meaning "τ is a well-formed type expression of status *sd*". Here *sd* ranges over the three *status descriptors* ordinary, reusable, ground.

The rules for `classimpl` types require the following definitions. A pair (π_f, π_m) is said to be *of class form* relative to $\sigma_0, \ldots, \sigma_{q-1}$ if:

- π_f is a simple labelled product expression;
- π_m is of the form

  $$\{\ m_0\texttt{:!(}\sigma_0\texttt{->}\sigma'_0\texttt{)}\texttt{,} \ldots \texttt{,}m_{q-1}\texttt{:!(}\sigma_{q-1}\texttt{->}\sigma'_{q-1}\texttt{)}\ \}$$

  for some $\sigma'_0, \ldots, \sigma'_{q-1}$ and distinct $m_0, \ldots, m_{q-1}$.

We say simply that (π_f, π_m) is *of class form* if it is of class form relative to some $\sigma_0, \ldots, \sigma_{q-1}$.

$$\frac{}{\vdash \kappa \propto sd} \tag{1}$$

$$\frac{\vdash \tau_0 \propto sd \qquad \vdash \tau_1 \propto sd}{\vdash \texttt{(}\tau_0\texttt{*}\tau_1\texttt{)} \propto sd} \tag{2}$$

$$\frac{\vdash \tau_0 \propto sd \qquad \cdots \qquad \vdash \tau_{n-1} \propto sd}{\vdash \texttt{\{}\ k_0\texttt{:}\tau_0\texttt{,}\ \ldots\ \texttt{,}k_{n-1}\texttt{:}\tau_{n-1}\ \texttt{\}} \propto sd}\ k_0,\ldots,k_{n-1} \text{ distinct} \tag{3}$$

$$\frac{\vdash \pi \propto sd}{\vdash \pi\ \texttt{**}\ \rho \propto \textsf{ordinary}}\ \rho \text{ not in } \pi \tag{4}$$

$$\frac{\vdash \tau_0 \propto sd \qquad \cdots \qquad \vdash \tau_{n-1} \propto sd}{\vdash \texttt{[}\ k_0\texttt{:}\tau_0\texttt{,}\ \ldots\ \texttt{,}k_{n-1}\texttt{:}\tau_{n-1}\ \texttt{]} \propto sd}\ k_0,\ldots,k_{n-1} \text{ distinct} \tag{5}$$

$$\frac{\vdash \tau \propto \textsf{ordinary}}{\vdash \texttt{!}\tau \propto \textsf{reusable}} \tag{6}$$

$$\frac{\vdash \tau \propto \textsf{ground}}{\vdash \texttt{!}\tau \propto \textsf{ground}} \tag{7}$$

$$\frac{\vdash \tau \propto \textsf{ordinary} \qquad \vdash \tau' \propto \textsf{ordinary}}{\vdash \tau\texttt{->}\tau' \propto \textsf{ordinary}} \tag{8}$$

$$\frac{\vdash \pi_f \propto \textsf{reusable} \qquad \vdash \pi_m \propto \textsf{reusable} \qquad \vdash \tau_k \propto \textsf{ordinary}}{\vdash \texttt{classimpl}\ \pi_f\texttt{,}\pi_m\texttt{,}\tau_k \propto \textsf{reusable}}\ (\pi_f,\pi_m) \text{ of class form} \tag{9}$$

$$\frac{\begin{array}{c}\vdash \pi_f \propto \textsf{ordinary} \qquad \vdash \pi_m \propto \textsf{reusable} \qquad \vdash \tau_k \propto \textsf{ordinary} \\ \vdash \sigma_0 \propto \textsf{ground} \qquad \cdots \qquad \vdash \sigma_{q-1} \propto \textsf{ground}\end{array}}{\vdash \texttt{linear classimpl}\ \pi_f\texttt{,}\pi_m\texttt{,}\tau_k \propto \textsf{reusable}}\ \begin{array}{l}(\pi_f,\pi_m) \text{ of class form} \\ \text{relative to } \sigma_0,\ldots,\sigma_{q-1}\end{array} \tag{10}$$

Note that $\vdash \tau \propto \textsf{ground}$ implies $\vdash \tau \propto \textsf{reusable}$ which implies $\vdash \tau \propto \textsf{ordinary}$. We say τ is a *well-formed* type expression if $\vdash \tau \propto \textsf{ordinary}$. We say that well-formed type expressions τ, τ' are *equivalent* if they are the same up to permutations of explicitly named components in labelled product and sum types; we omit the formal definition. (Note that e.g. $\pi\texttt{**}\rho\texttt{**}\rho'$ and $\pi\texttt{**}\rho'\texttt{**}\rho$ are *not* deemed equivalent.) For the purpose of Lingay, we may define *underlying types* to be the well-formed types modulo equivalence. We let $\boldsymbol{\tau}, \boldsymbol{\sigma}$ range over underlying types.

Next we define a *subtyping* relation on well-formed type expressions. The following rules generate assertions of the form $\tau <: \tau'$. In the last rule, *lo* ranges over the syntactic category *lin-opt*.

$$\frac{}{\kappa <: \kappa} \tag{11}$$

$$\frac{\tau_0 \mathrel{<:} \tau_0' \qquad \tau_1 \mathrel{<:} \tau_1'}{(\tau_0\texttt{,}\tau_1) \mathrel{<:} (\tau_0'\texttt{,}\tau_1')} \tag{12}$$

$$\frac{\tau_0 \mathrel{<:} \tau_0' \qquad \cdots \qquad \tau_{n-1} \mathrel{<:} \tau_{n-1}'}{\{k_0 : \tau_0\texttt{,} \cdots \texttt{,}k_{n'-1} : \tau_{n'-1}\} \mathrel{<:} \{k_{p0} : \tau_{p0}'\texttt{,} \cdots \texttt{,}k_{p(n-1)} : \tau_{p(n-1)}'\}} \quad \begin{array}{l} n' \geq n \\ p \in S_n \\ \tau_n, \ldots, \tau_{n'-1} \text{ not ground} \end{array} \tag{13}$$

$$\frac{\pi \mathrel{<:} \pi'}{\pi\texttt{**}\rho \mathrel{<:} \pi'\texttt{**}\rho} \tag{14}$$

$$\frac{\tau_0 \mathrel{<:} \tau_0' \qquad \cdots \qquad \tau_{n-1} \mathrel{<:} \tau_{n-1}'}{\texttt{[}k_{p0} : \tau_{p0}\texttt{,} \cdots \texttt{,}k_{p(n-1)} : \tau_{p(n-1)}\texttt{]} \mathrel{<:} \texttt{[}k_0 : \tau_0'\texttt{,} \cdots \texttt{,}k_{n'-1} : \tau_{n'-1}'\texttt{]}} \quad \begin{array}{l} n' \geq n \\ p \in S_n \end{array} \tag{15}$$

$$\frac{\tau \mathrel{<:} \tau'}{\texttt{!}\tau \mathrel{<:} \texttt{!}\tau'} \tag{16}$$

$$\frac{\tau_0' \mathrel{<:} \tau_0 \qquad \tau_1 \mathrel{<:} \tau_1'}{\tau_0\texttt{->}\tau_1 \mathrel{<:} \tau_0'\texttt{->}\tau_1'} \tag{17}$$

$$\frac{\pi_k' \mathrel{<:} \pi_k}{\mathit{lo}\ \texttt{classimpl}\ \pi_f\texttt{,}\pi_m\texttt{,}\pi_k \mathrel{<:} \mathit{lo}\ \texttt{classimpl}\ \pi_f\texttt{,}\pi_m\texttt{,}\pi_k'} \tag{18}$$

Clearly the relation <: is compatible with equivalence of type expressions, so that we may regard <: as a relation (in fact, a partial order) on underlying types.

Finally, to each expression constant ec we associate a type $ty(ec)$ as follows. Boolean literals have type `bool`; integer literals have type `int`; `_intPlus`, `_intMinus`, `_intTimes` have type `int*int->int`; and `_intLeq` has type `int*int->bool`.

Expressions

The syntax of *expressions* is given in Figure 5. As with type expressions, an expression is officially a parse tree generated by the above grammar, and we freely use brackets for disambiguation as required. We let e range over expressions.

Certain expressions are syntactically designated as *values*. The notion of value is of particular importance in the dynamic semantics, but it also features in the static semantics in rule 28. The grammar for values is given in Figure 6. We let v range over values.

We also allow ourselves the use of the following syntactic sugar:

$\texttt{let val } x : \tau \texttt{ = } e \texttt{ in } e' \texttt{ end}$	for	$\texttt{(fn } x : \tau \texttt{ => } e'\texttt{) \$ } e$
$\texttt{fn (}x : \sigma\texttt{,}y : \tau\texttt{) => } e$	for	$\texttt{fn } z : (\sigma\texttt{*}\tau) \texttt{ => split } z \texttt{ as (}x\texttt{,}y\texttt{) in } e \texttt{ end}$
$\texttt{rec (}x : \sigma\texttt{,}y : \tau\texttt{) => } e$	for	$\texttt{rec } z : (\sigma\texttt{*}\tau) \texttt{ => split } z \texttt{ as (}x\texttt{,}y\texttt{) in } e \texttt{ end}$
$e\ e'$	for	$\texttt{(der } e\texttt{) \$ } e'$
$e\texttt{.}m$	for	$\texttt{split } e \texttt{ as } \{m\texttt{=}x\texttt{,} \cdots\} \texttt{ in } x \texttt{ end}$

where z does not occur free in e.

expr	::=	x	(variable)
	$\mid$	*ec*	(expression constant)
	$\mid$	`(` *expr*`,` *expr* `)`	(pair)
	$\mid$	`split` *expr* `as` `(` x_0`,`x_1 `)`	
		`in` *expr* `end`	(pair separation)
	$\mid$	{ *Clist*(*comp-assign*) }	(labelled record)
	$\mid$	*expr* `+>` *expr*	(record merge/override)
	$\mid$	`split` *expr* `as` { *Clist*(*comp-bind*) }	
		`in` *expr* `end`	(record separation)
	$\mid$	`%` k *expr*	(labelled sum injection)
	$\mid$	`case` *expr* `of` *Blist*(*clause*) `end`	(labelled sum elimination)
	$\mid$	`prom` *expr*	(reusable promotion)
	$\mid$	`der` *expr*	(non-reusable dereliction)
	$\mid$	`fn` *boundvar* `=>` *expr*	(linear function)
	$\mid$	*expr* `$` *expr*	(strict application)
	$\mid$	*expr* `=` *expr*	(equality test)
	$\mid$	`if` *expr* `then` *expr* `else` *expr*	(conditional)
	$\mid$	`rec` *boundvar* `=>` *expr*	(fixed point)
	$\mid$	`catchcont` *boundvar* `=>`*expr*	(continuation catching)
	$\mid$	*expr* `::` *type-expr*	(upcasting)
	$\mid$	*lin-opt* `triv`	(trivial class body)
	$\mid$	*lin-opt* `extend` *expr* `with`	
		expr`,` *expr*	(class extension)
	$\mid$	`new` *expr*	(new object)
comp-assign	::=	k `=` *expr*	(component assignment)
comp-bind	::=	k `=` *boundvar*	(component binding)
clause	::=	`%` k *boundvar* `=>` *expr*	(match clause)
boundvar	::=	x `:` *type-expr*	(bound variable)

Figure 5: Context-free grammar for expressions

$$
\begin{array}{rcl}
\textit{value} & ::= & \textit{ec} \\
& | & \texttt{(}\,\textit{value}\texttt{,}\,\textit{value}\,\texttt{)} \\
& | & \texttt{\{}\,\textit{Clist}(\textit{comp-value})\,\texttt{\}} \\
& | & \texttt{\%}\ k\ \textit{value} \\
& | & \texttt{prom}\ \textit{value} \\
& | & \texttt{fn}\ \textit{boundvar}\ \texttt{=>}\ \textit{expr} \\
& | & \textit{value}\ \texttt{::}\ \textit{type-expr} \\
& | & \texttt{new}\ \textit{value} \\
& | & \textit{lin-opt}\ \texttt{triv} \\
& | & \textit{lin-opt}\ \texttt{extend triv with}\ \textit{value}\texttt{,}\textit{value} \\
\textit{comp-value} & ::= & k\ \texttt{=}\ \textit{value}
\end{array}
$$

Figure 6: Context-free grammar for values

Notation and conventions

A (static) *environment* Γ is an ordered list of entries of the form $x : \boldsymbol{\tau}$ *safe-tag*, where x is an identifier, $\boldsymbol{\tau}$ an underlying type, and *safe-tag* is one of the following five *safety tags*:

$$\epsilon \quad \mathsf{safe} \quad \mathsf{halfsafe} \quad \mathsf{argsafe} \quad \mathsf{metasafe}$$

In writing environments we shall often use type expressions to stand for the underlying types they represent. We write $\Gamma[x]$ for the rightmost type (if any) associated with x in Γ. We say Γ is *reusable* if $\Gamma[x]$ is reusable whenever it is defined. If e is an expression, we write Γ_e for the environment consisting of just the entries in Γ pertaining to variables that occur free in Γ. We also write Safe(Γ) for the environment obtained from Γ by setting all safety tags to safe.

The following notation will be employed in many of the typing rules to ensure that linear variables (i.e. those of non-reusable type) do not appear more than once in well-typed expressions.[6] If Γ is an environment, $x_0, \ldots, x_{n-1}$ are variables, and e, e' are expressions, we say e, e' are Γ-*compatible modulo* $x_0, \ldots, x_{n-1}$, and write $e \sim_\Gamma^{x_0,\ldots,x_{n-1}} e'$, if for all $x \in \mathrm{dom}\ \Gamma - \{x_0, \ldots, x_{n-1}\}$ such that x occurs free in both ϕ and ϕ', the type $\Gamma[x]$ is reusable. In the case $n = 0$ we may simply write $\phi \sim_\Gamma \phi'$.

For rules which are declared to be subject to the *safety convention*, an auxiliary rule may be derived from it as follows. First, the tag safe is added to each variable environment entry explicitly displayed in the rule without a safety tag. (Entries appearing within variable environments denoted by metavariables such as Γ are unaffected.) Next, the tag safe is added to each judgement which is displayed in the rule without a safety tag. (We do not add safe to those environment entries or judgements in which the presence of the empty safety tag is explicitly indicated by the symbol ϵ, as in rules 33,35.)

Rules that are subject to the safety convention are identified by the letter **S** appearing to the left of the rule. For such rules, it is to be understood that both the original rule and the auxiliary rule derived from it are applicable.

[6]A similar effect could be achieved using a *dual-context* type system, except that this would not allow us to resolve name clashes between linear and reusable variables.

By exact analogy, we also have *halfsafety, argsafety* and *metasafety conventions* involving the tags halfsafe, argsafe and metasafe respectively. Rules that are subject to these conventions are identified by the letters **H**, **A**, **M**. The safety, halfsafety, argsafety and metasafety conventions will be referred to collectively as the SHAM conventions. Note that the same rule may be subject to several of these conventions.

In some rules, where we require two degrees of freedom with respect to safety tags, we make use of the SHAM conventions in conjunction with a metavariable *safe-tag* which plays the role of a second 'safety parameter', whose possible values may be specified by a side-condition. If this metavariable appears within angle brackets as $\langle$*safe-tag*$\rangle$, the intention is that this safety parameter is present in the auxiliary rules derived via the SHAM conventions, but not in the original rule itself, where the safety parameter is assumed to be ϵ. Thus, for example, rule 22 expands to three rules, of which two feature the metavariable *safe-tag* ranging over $\{\epsilon, \mathsf{safe}\}$, yielding a total of five possible variants of this rule.

In rule 35, we write $\mathsf{Catchcont}\,(\zeta, \xi, \tau_0, \tau_1)$ as an abbreviation for the type expression

$$\begin{array}{lcl} \texttt{[result} & : & \{\texttt{value:}\tau_0\texttt{, more: !(}\zeta\texttt{->}\xi\texttt{)->}\tau_1\} \\ \quad\texttt{query} & : & \{\texttt{arg:}\zeta\texttt{, resume:}\,\xi\texttt{->!(}\zeta\texttt{->}\xi\texttt{)->}\tau_0 * \tau_1\}\ \texttt{]} \end{array}$$

Finally, some specialized notation for use in rule 38. Suppose $\tau = \{k_0 : \tau_0\,,\cdots, k_{n-1} : \tau_{n-1}\}$ and $\tau' = \{k'_0 : \tau'_0\,,\cdots, k'_{n'-1} : \tau_{n-1}\}$.

- If the k_i and k'_j are all distinct, we write $\tau + \tau'$ for the type

  $$\{k_0 : \tau_0\,,\cdots, k_{n-1} : \tau_{n-1}, k'_0 : \tau'_0\,,\cdots, k'_{n'-1} : \tau_{n-1}\}$$

  (otherwise $\tau + \tau'$ is undefined).

- We say τ' is *compatible* with τ if whenever $k_i = k'_j$, we have $\tau_i = \tau'_j$. In this case we define $\tau \oplus \tau'$ to be the type

  $$\{k_{i_0} : \tau_{i_0}\,,\cdots, k_{i_{r-1}} : \tau_{i_{r-1}}, k'_0 : \tau'_0\,,\cdots, k'_{n'-1} : \tau_{n-1}\}$$

  where $\{k_{i_0}, \ldots, k_{i_{r-1}}\} = \{k_0, \ldots, k_{n-1}\} - \{k'_0, \ldots, k'_{n'-1}\}$.

- If τ is as above with $\tau_i = \texttt{!(}\zeta_i\texttt{->}\xi_i\texttt{)}$ for each i, and σ is an arbitrary type, we write $\tau\natural\sigma$ for the type expression

  $$\{k_0 : \texttt{!(}\zeta_0 * \sigma\texttt{->}\xi_0 * \sigma\texttt{)}, \ldots, k_{n-1} : \texttt{!(}\zeta_{n-1} * \sigma\texttt{->}\xi_{n-1} * \sigma\texttt{)}\}$$

Typing rules

We give a proof system for deriving judgements of the form

$$\Gamma \vdash e : \boldsymbol{\tau}\ \textit{safe-tag}$$

where *safe-tag* is one of the five possible safety tags. For convenience, we freely use type expressions τ in place of the underlying types they represent.

The following rules are associated with typing judgements of the form $\Gamma \vdash e : \tau$.

$$\textbf{SHAM}\ \frac{}{\Gamma \vdash x : \tau}\ \tau = \Gamma[x] \qquad (19)$$

$$\textbf{SHAM}\ \ \frac{}{\Gamma \vdash ec : \tau}\ \ \tau = ty(ec) \tag{20}$$

$$\textbf{S}\ \ \frac{\Gamma \vdash e_0 : \tau_0 \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \mathtt{(}\,e_0\mathtt{,}e_1\,\mathtt{)}\ :\ \tau_0\mathtt{*}\tau_1}\ \ e_0 \sim_\Gamma e_1 \tag{21}$$

$$\textbf{SH}\ \ \frac{\begin{array}{c}\Gamma \vdash e : \tau_0\mathtt{*}\tau_1\ \langle \textit{safe-tag}\rangle \\ \Gamma \bullet x_0 : \tau_0\ \langle \textit{safe-tag}\rangle \bullet x_1 : \tau_1\ \langle \textit{safe-tag}\rangle \vdash e' : \tau\end{array}}{\begin{array}{c}\Gamma \vdash \mathtt{split}\ e\ \mathtt{as}\ \mathtt{(}\,x_0\mathtt{:}\tau_0\mathtt{,}x_1\mathtt{:}\tau_1\,\mathtt{)} \\ \mathtt{in}\ e'\ \mathtt{end}\ : \tau\end{array}}\ \ \begin{array}{l} e \sim_\Gamma^{x_0,x_1} e' \\ \textit{safe-tag} \in \{\epsilon, \mathsf{safe}\}\end{array} \tag{22}$$

$$\textbf{SA}\ \ \frac{\Gamma \vdash e_0 : \tau_0 \qquad \cdots \qquad \Gamma \vdash e_{n-1} : \tau_{n-1}}{\Gamma \vdash \mathtt{\{}\,k_0 = e_0\mathtt{,}\ldots\mathtt{,}k_{n-1} = e_{n-1}\,\mathtt{\}}\ :\ \mathtt{\{}\,k_0 : \tau_0\mathtt{,}\ldots\mathtt{,}k_{n-1} : \tau_{n-1}\,\mathtt{\}}}\ \ \forall i \neq j.\ e_i \sim_\Gamma e_j \tag{23}$$

$$\textbf{SA}\ \ \frac{\Gamma \vdash e : \pi \qquad \Gamma \vdash e' : \pi'}{\Gamma \vdash e\ \mathtt{+>}\ e' : \pi\sharp\pi'}\ \ \pi, \pi'\ \text{compatible} \tag{24}$$

$$\textbf{SHA}\ \ \frac{\begin{array}{c}\Gamma \vdash e : \mathtt{\{}\,k_0 : \tau_0\mathtt{,}\ldots\mathtt{,}k_{n-1} : \tau_{n-1}\,\mathtt{\}}\ \langle \textit{safe-tag}\rangle \\ \Gamma \bullet x_0 : \tau_0\ \langle \textit{safe-tag}\rangle \ldots \bullet x_{n-1} : \tau_{n-1}\ \langle \textit{safe-tag}\rangle \vdash e' : \tau\end{array}}{\begin{array}{c}\Gamma \vdash \mathtt{split}\ e\ \mathtt{as}\ \mathtt{\{}\,k_0\mathtt{=}x_0\mathtt{:}\tau_0\mathtt{,}\ldots\mathtt{,}k_{n-1}\mathtt{=}x_{n-1}\mathtt{:}\tau_{n-1}\,\mathtt{\}} \\ \mathtt{in}\ e'\ \mathtt{end}\ : \tau\end{array}}\ \ \begin{array}{l} e \sim_\Gamma^{x_0,\ldots,x_{n-1}} e' \\ \textit{safe-tag} \in \{\epsilon, \mathsf{safe}\}\end{array} \tag{25}$$

$$\textbf{S}\ \ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathtt{\%}\ k\ e : \mathtt{[}\,k : \tau\,\mathtt{]}} \tag{26}$$

$$\textbf{SH}\ \ \frac{\begin{array}{c}\Gamma \vdash e : \mathtt{[}\,k_0 \mathtt{:}\ \tau_0\mathtt{,}\ldots\mathtt{,}k_{n-1}\mathtt{:}\ \tau_{n-1}\,\mathtt{]}\ \langle \textit{safe-tag}\rangle \\ \Gamma \bullet x_0 : \tau_0\ \langle \textit{safe-tag}\rangle \vdash e_0 : \tau' \\ \cdots \\ \Gamma \bullet x_{n-1} : \tau_{n-1}\ \langle \textit{safe-tag}\rangle \vdash e_{n-1} : \tau'\end{array}}{\begin{array}{c}\Gamma \vdash \mathtt{case}\ e\ \mathtt{of}\ \mathtt{\%}\,k_0 x_0\mathtt{:}\tau_0\mathtt{=>}e_0\ \mathtt{|}\ \cdots \\ \mathtt{|}\ \mathtt{\%}\,k_{n-1} x_{n-1}\mathtt{:}\tau_{n-1}\mathtt{=>}e_{n-1}\ \mathtt{end}\ : \tau'\end{array}}\ \ \begin{array}{l} \forall i.\ e \sim_\Gamma^{x_i} e_i \\ \textit{safe-tag} \in \{\epsilon, \mathsf{safe}\}\end{array} \tag{27}$$

$$\textbf{SA}\ \ \frac{\Gamma_v \vdash v : \tau}{\Gamma \vdash \mathtt{prom}\ v : \mathtt{!}\tau}\ \ \Gamma_v\ \text{reusable} \tag{28}$$

$$\textbf{SA}\ \ \frac{\Gamma \vdash e : \mathtt{!}\tau}{\Gamma \vdash \mathtt{der}\ e : \tau} \tag{29}$$

$$\textbf{M}\ \ \frac{\Gamma \bullet x : \sigma \vdash e : \tau}{\Gamma \vdash \mathtt{fn}\ x\mathtt{:}\sigma\mathtt{=>}e\ : \sigma\mathtt{->}\tau} \tag{30}$$

$$\textbf{SM}\ \frac{\Gamma\vdash e:\sigma\texttt{->}\tau \qquad \Gamma\vdash e':\sigma}{\Gamma\vdash e\,\texttt{\$}\,e':\tau}\quad e\sim_\Gamma e' \tag{31}$$

$$\textbf{S}\ \frac{\Gamma\vdash e:\tau \qquad \Gamma\vdash e':\tau}{\Gamma\vdash e\,\texttt{=}\,e':\texttt{bool}}\quad \begin{array}{l} e\sim_\Gamma e' \\ \tau\ \text{ground}\end{array} \tag{32}$$

$$\textbf{SH}\ \frac{\Gamma\vdash e:\texttt{bool}\ \epsilon \qquad \Gamma\vdash e_0:\tau \qquad \Gamma\vdash e_1:\tau}{\Gamma\vdash \texttt{if}\ e\ \texttt{then}\ e_0\ \texttt{else}\ e_1\ :\tau}\quad e\sim_\Gamma e_0,\ e\sim_\Gamma e_1 \tag{33}$$

$$\textbf{SH}\ \frac{\Gamma\bullet x:\tau\vdash e:\tau}{\Gamma\vdash \texttt{rec}\ x\texttt{:}\tau\texttt{=>}e\ :\tau}\quad \begin{array}{l} e'=\texttt{rec}\ x\texttt{:}\tau\texttt{=>}e \\ \Gamma_{e'}\ \text{reusable}\end{array} \tag{34}$$

$$\textbf{S}\ \frac{\Gamma\bullet x:\sigma\ \epsilon\vdash e:\tau_0\texttt{*}\tau_1}{\Gamma\vdash \texttt{catchcont}\ x\texttt{:}\sigma\texttt{=>}e:\mathsf{Catchcont}\,(\zeta,\xi,\tau_0,\tau_1)}\quad \begin{array}{l} \zeta\ \text{ground} \\ \tau_0\ \text{ground} \\ x\notin\operatorname{dom}\Gamma \\ \sigma=\texttt{!}(\zeta\texttt{->}\xi)\end{array} \tag{35}$$

$$\textbf{SHAM}\ \frac{\Gamma\vdash e:\tau}{\Gamma\vdash e\texttt{::}\tau'\ :\tau'}\quad \tau<:\tau' \tag{36}$$

$$\textbf{S}\ \frac{}{\Gamma\vdash lo\ \texttt{triv}:lo\ \texttt{classimpl}\ \{\}\texttt{,}\{\}\texttt{,}\{\}} \tag{37}$$

$$\frac{\begin{array}{c}\Gamma\vdash e_0:lo'\ \texttt{classimpl}\ \tau_1\texttt{,}\tau_2\texttt{,}\tau_3 \\ \mathrm{Safe}(\Gamma)\vdash e_1:\texttt{!}(\tau_{super}\texttt{->}\tau_{self}\texttt{->}\tau_{self})\ \ \mathsf{metasafe} \\ \Gamma\vdash e_2:\texttt{!}(\tau_3''\texttt{->}\tau_3\texttt{*}(\tau_1\texttt{->}\tau_1''))\end{array}}{\Gamma\vdash lo\ \texttt{extend}\ e_0\ \texttt{with}\ e_1\texttt{,}e_2\ :lo\ \texttt{classimpl}\ \tau_1''\texttt{,}\tau_2''\texttt{,}\tau_3''}\quad\cdots \tag{38}$$

$$\cdots\quad \begin{array}{l} e_0\sim_\Gamma e_1,\ e_0\sim_\Gamma e_2,\ e_1\sim_\Gamma e_2 \\ \tau_{super}=\tau_2\natural(\tau_1''\texttt{**}\rho),\ \tau_{self}=\tau_2'\natural(\tau_1''\texttt{**}\rho),\ \rho\ \text{fresh} \\ \tau_1''=\tau_1+\tau_1',\ \tau_2''=\tau_2\oplus\tau_2' \\ lo'=\epsilon\ \text{or}\ lo=\texttt{linear}\end{array}$$

$$\textbf{S}\ \frac{\Gamma\vdash e:lo\ \texttt{classimpl}\ \tau_1\texttt{,}\tau_2\texttt{,}\tau_3}{\Gamma\vdash \texttt{new}\ e:\tau_3\texttt{->}\tau_2} \tag{39}$$

Rules associated with judgements $\Gamma\vdash e:\tau$ safe. In addition to the auxiliary rules derived from the above via the safety convention, we have the following:

$$\frac{}{\Gamma\vdash x:\tau\ \mathsf{safe}}\quad \begin{array}{l}\tau=\Gamma[x] \\ x\ \text{argsafe or metasafe in}\ \Gamma\end{array} \tag{40}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau \ \mathsf{safe}} \quad \tau \text{ ground} \tag{41}$$

$$\frac{\Gamma \bullet x : \sigma \ \mathsf{safe} \vdash e : \tau \ \mathsf{safe}}{\Gamma \vdash \texttt{fn}\ x\texttt{:}\sigma\texttt{=>}e\ : \sigma\texttt{->}\tau \ \mathsf{safe}} \quad \begin{array}{l}\text{all } y \in \mathrm{FV}(e) - \{x\} \text{ safe,}\\ \text{argsafe or metasafe in } \Gamma\end{array} \tag{42}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{lo}\ \texttt{extend}\ e_0\ \texttt{with}\ e_1\texttt{,}e_2\ : \tau_{extend}\\ \Gamma \vdash e_0 : \tau_{superclass}\ \mathsf{safe}\\ \Gamma \vdash e_1 : \tau_{methods}\ \mathsf{safe}\\ \Gamma \vdash e_2 : \tau_{constr}\ \mathsf{safe}\end{array}}{\Gamma \vdash \mathit{lo}\ \texttt{extend}\ e_0\ \texttt{with}\ e_1\texttt{,}e_2\ : \tau_{extend}\ \mathsf{safe}} \tag{43}$$

Rules for judgements $\Gamma \vdash e : \tau$ halfsafe.

$$\frac{\Gamma \vdash e_0 : \tau_0 \qquad \Gamma \vdash e_1 : \tau_1 \ \mathsf{safe}}{\Gamma \vdash \texttt{(}e_0\texttt{,}e_1\texttt{)} : \tau_0 * \tau_1 \ \mathsf{halfsafe}} \quad e_0 \sim_\Gamma e_1 \tag{44}$$

$$\frac{\Gamma \vdash e : \tau_0 * \tau_1 \ \mathsf{halfsafe} \qquad \Gamma \bullet x_0 : \tau_0 \bullet \mathsf{safe}\ x_1 : \tau_1 \vdash e' : \tau \ \textit{safe-tag}}{\Gamma \vdash \texttt{split}\ e\ \texttt{as}\ \texttt{(}x_0\texttt{,}x_1\texttt{)}\ \texttt{in}\ e'\ \texttt{end}\ : \tau\ \textit{safe-tag}} \quad \begin{array}{l} e \sim_\Gamma^{x_0,x_1} e' \\ \textit{safe-tag} \neq \epsilon\end{array} \tag{45}$$

Rules for judgements $\Gamma \vdash e : \tau$ argsafe.

$$\frac{\Gamma \bullet x : \sigma \ \mathsf{halfsafe} \vdash e : \tau \ \mathsf{halfsafe}}{\Gamma \vdash \texttt{fn}\ x\texttt{:}\sigma\texttt{=>}e\ : \sigma\texttt{->}\tau \ \mathsf{argsafe}} \tag{46}$$

$$\frac{\Gamma \vdash e : (\tau_0 * \tau_1)\texttt{->}(\tau_0' * \tau_1')\ \mathsf{argsafe} \qquad \Gamma \vdash e' : \tau_0 * \tau_1 \ \mathsf{halfsafe}}{\Gamma \vdash e\ \texttt{\$}\ e' : \tau_0' * \tau_1' \ \mathsf{halfsafe}} \tag{47}$$

Rules for judgements $\Gamma \vdash e : \tau$ metasafe. Only two very simple rules are needed here: the remaining work is done by the metasafety convention for the specified earlier rules.

$$\frac{\Gamma \vdash e : \tau \ \mathsf{argsafe}}{\Gamma \vdash e : \tau \ \mathsf{metasafe}} \tag{48}$$

$$\frac{\Gamma \vdash e : \pi \ \mathsf{metasafe}}{\Gamma \vdash e : \pi \ \mathsf{argsafe}} \tag{49}$$

Operational semantics

An extended language of expressions

For the purpose of the operational semantics, we need to add some auxiliary features and constructs to our language. These features do not appear in programs written by users of the language, but may arise at intermediate stages in the execution of programs.

First, we need to enlarge the set of identifiers that may be used as variable names. The class of identifiers employed so far will now be called the class of *standard identifiers*; to this we now add an infinite collection of *test symbols*, and the union of these two classes is called the class of *general identifiers*. We continue to use the metavariables $ident, x, y, k, f, m, t$ to range only over standard identifiers; we use $\hat{x}, \hat{y}$ to range over test symbols. We accordingly allow environments to contain type and safety ascriptions for test symbols as well as standard variables; in this section we use Γ to range over such generalized environments. By a *test symbol environment* we shall mean a variable environment all of whose variables are test symbols; we use $\hat{\Gamma}$ to range over such environments. We write $\mathsf{Sym}(\Gamma)$ for the test symbol environment consisting of just the variable entries for test symbols occurring in Γ.

We shall also require a notion of a reference to an object. Let $\mathcal{L}$ be an infinite set of *location symbols* (henceforth *locations*), disjoint from all other classes introduced so far. We use ℓ to range over $\mathcal{L}$. To facilitate the construction of canonical derivations, we shall suppose $\mathcal{L}$ comes equipped with a choice function for generating fresh locations. Explicitly, for any finite set $L \subset \mathcal{L}$, we suppose $\mathsf{freshLoc}(L)$ is some choice of location $\ell \notin L$.

A *decorated location* is an expression of the form $\ell^{\boldsymbol{\sigma},\boldsymbol{\tau}}_{\hat{\Gamma}}$. Informally, $\boldsymbol{\sigma}$ here is the type of the internal state of the object stored at the location ℓ; $\boldsymbol{\tau}$ will be the type of the object itself; and $\hat{\Gamma}$ records the test symbols which may potentially occur within the object. We use $\tilde{\ell}$ to range over decorated location symbols.

Finally, a few additional syntactic constructs will be needed. We therefore define an *augmented* version of our language by adding the following clauses to the grammar of Figure 5:

$$\begin{array}{rcl} expr & ::= & \hat{x} \\ & | & \tilde{\ell} \\ & | & \tilde{\ell}\texttt{.}m \\ & | & \texttt{fn}\ \hat{x}\,\texttt{:}\,type\text{-}expr\ \texttt{=>}\ expr \\ & | & \texttt{push}\ expr \\ & | & \texttt{update}\ \tilde{\ell}\ expr \end{array}$$

For the remainder of this section we use e to range over expressions of the augmented language.

We also need a suitable notion of a *value* for the augmented language. Here, by contrast with Section 6, we shall define a class of values that admits certain occurrences of free variables not underneath binders. Formally, the grammar for *open values* (of the augmented language) is obtained by adding the following clauses to the grammar of Figure 6:

$$value\ ::=\ \tilde{\ell}\ \mid\ \tilde{\ell}\texttt{.}m\ \mid\ \hat{x}\ \mid\ \texttt{der}\ \tilde{\ell}\texttt{.}m\ \mid\ \texttt{der}\ \hat{x}$$

Throughout this section we use v to range over open values.

We extend our type system to the augmented language as follows. All the typing rules of Section 6 still stand (with the proviso that any metavariables Γ, e, v or variants thereof appearing in the rules now have the ranges appropriate for the augmented language). In addition, we have the following typing rules for the new phrase forms:

$$\mathbf{S}\ \frac{}{\Gamma \vdash \ell_{\hat{\Gamma}}^{\boldsymbol{\sigma},\boldsymbol{\tau}} : \boldsymbol{\tau}}\ \hat{\Gamma} = \mathsf{Sym}(\Gamma) \tag{50}$$

$$\mathbf{S}\ \frac{}{\Gamma \vdash \ell_{\hat{\Gamma}}^{\boldsymbol{\sigma},\boldsymbol{\tau}}.m : \boldsymbol{\tau}'}\ \begin{array}{l}\hat{\Gamma} = \mathsf{Sym}(\Gamma)\\ \boldsymbol{\tau} = \{m : \boldsymbol{\tau}', \cdots\}\end{array} \tag{51}$$

$$\mathbf{S}\ \frac{\Gamma \vdash e : \boldsymbol{\sigma} * (\boldsymbol{\tau} \natural \boldsymbol{\sigma})}{\Gamma \vdash \texttt{push}\ e : \boldsymbol{\tau}} \tag{52}$$

$$\mathbf{S}\ \frac{\Gamma \vdash e : \boldsymbol{\rho}' * \boldsymbol{\sigma}}{\Gamma \vdash \texttt{update}\ \ell_{\hat{\Gamma}}^{\boldsymbol{\sigma},\boldsymbol{\tau}}\ e : \boldsymbol{\rho}'} \tag{53}$$

Evaluation contexts

An evaluation context is, intuitively, a term with a single "hole" corresponding to the location of the subterm which would need to be supplied for evaluation to proceed. Our primary reason for introducing evaluation contexts is to give an operational semantics for `catchcont` expressions.

The grammar for evaluation contexts is given in Figure 7. (Like expressions, evaluation contexts are officially parse trees, and we use brackets to disambiguate where necessary.) We use the metavariable E to range over evaluation contexts, and write $E[e]$ for the expression obtained by replacing the hole $[-]$ in E by e. We also write $E \circ E'$ for the evident evaluation context $E[E'[-]]$. We say an evaluation context E is *transparent to* a variable x if it is not of the form $E_1[$`catchcont` $x : \tau$`=>`$E_2[-]]$ for any E_1, E_2. Finally, a *stuck term* is an expression of the form $E[$`der` $\hat{x}$ `$` $v]$. We use u to range over stuck terms, and w to range over open values plus stuck terms.

Heaps and evaluation judgements

Our operational semantics will make use of a simple model of memory provided by the following definition of a *heap*. For the purpose of the formal definition, we regard both the current state of an object and the code for its methods as being stored together as a pair on the heap.

Formally, a *heap* is a function h from some finite set of decorated locations to pairs of open values v_s, v_b, such that if $h(\ell_{\hat{\Gamma}}^{\boldsymbol{\sigma},\boldsymbol{\tau}}) = v_s, v_b$ then $\hat{\Gamma} \vdash v_s : \boldsymbol{\sigma}$ and $\hat{\Gamma} \vdash v_b : \boldsymbol{\tau} \natural \boldsymbol{\sigma}$. We write $\operatorname{dom} h$ for the set of $\tilde{\ell}$ for which $h(\tilde{\ell})$ is defined.

$$
\begin{array}{rcl}
\mathit{eval} & ::= & [-] \\
& | & \texttt{(}\ \mathit{eval}\ \texttt{,}\ \mathit{expr}\ \texttt{)} \\
& | & \texttt{(}\ \mathit{value}\ \texttt{,}\ \mathit{eval}\ \texttt{)} \\
& | & \texttt{split}\ \mathit{eval}\ \texttt{as}\ \texttt{(}x\texttt{,}y\texttt{)}\ \texttt{in}\ \mathit{expr}\ \texttt{end} \\
& | & \{\ \mathit{List}(\mathit{comp\text{-}comma})\ k\texttt{=}\mathit{eval}\ \mathit{List}(\mathit{comma\text{-}comp})\ \} \\
& | & \mathit{eval}\ \texttt{+>}\ \mathit{expr} \\
& | & \mathit{value}\ \texttt{+>}\ \mathit{eval} \\
& | & \texttt{split}\ \mathit{eval}\ \texttt{as}\ \{\ \mathit{Clist}(\mathit{comp\text{-}bind})\ \}\ \texttt{in}\ \mathit{expr}\ \texttt{end} \\
& | & \texttt{\%}\ k\ \mathit{eval} \\
& | & \texttt{case}\ \mathit{eval}\ \texttt{of}\ \mathit{Blist}(\mathit{clause})\ \texttt{end} \\
& | & \texttt{der}\ \mathit{eval} \\
& | & \mathit{eval}\ \texttt{\$}\ \mathit{expr} \\
& | & \mathit{value}\ \texttt{\$}\ \mathit{eval} \\
& | & \mathit{eval}\ \texttt{=}\ \mathit{expr} \\
& | & \mathit{value}\ \texttt{=}\ \mathit{eval} \\
& | & \texttt{if}\ \mathit{eval}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr} \\
& | & \texttt{rec}\ \mathit{boundvar}\ \texttt{=>}\ \mathit{eval} \\
& | & \texttt{catchcont}\ \mathit{boundvar}\ \texttt{=>}\ \mathit{eval} \\
& | & \mathit{eval}\ \texttt{::}\ \mathit{type\text{-}expr} \\
& | & \mathit{lin\text{-}opt}\ \texttt{extend}\ \mathit{eval}\ \texttt{with}\ \mathit{expr}, \mathit{expr} \\
& | & \mathit{lin\text{-}opt}\ \texttt{extend}\ \mathit{value}\ \texttt{with}\ \mathit{eval}, \mathit{expr} \\
& | & \mathit{lin\text{-}opt}\ \texttt{extend}\ \mathit{value}\ \texttt{with}\ \mathit{value}, \mathit{eval} \\
& | & \texttt{new}\ \mathit{eval} \\
& | & \texttt{push}\ \mathit{eval} \\
& | & \texttt{update}\ \tilde{\ell}\ \mathit{eval} \\
\mathit{comp\text{-}comma} & ::= & \mathit{comp\text{-}value}\ \texttt{,} \\
\mathit{comma\text{-}comp} & ::= & \texttt{,}\ \mathit{comp\text{-}assign}
\end{array}
$$

Figure 7: Context-free grammar for evaluation contexts

If $\tilde{\ell} = \ell^{\boldsymbol{\sigma},\boldsymbol{\tau}}_{\hat{\Gamma}}$, $\hat{\Gamma} \vdash v_s : \boldsymbol{\sigma}$ and $\hat{\Gamma} \vdash v_b : \boldsymbol{\tau}\natural\boldsymbol{\sigma}$, we write $h(\ell \mapsto v_s, v_b)$ for the heap defined by

$$h(\ell \mapsto v_s, v_b)(\ell') \;=\; \left[\begin{array}{ll} v_s, v_b & \text{if } \ell' = \ell \\ h(\ell') & \text{if } \ell' \neq \ell \text{ and } \ell' \in \text{dom } h \\ \text{undefined} & \text{otherwise} \end{array}\right.$$

We shall give a proof system for deriving *evaluation judgements* of the form:

$$\hat{\Gamma} \;\vdash\; h, e \;\Downarrow\; h', w$$

where: $\hat{\Gamma}$ is a test symbol environment, such that no test symbol appears twice in Γ; each $\tilde{\ell} \in \text{dom } h \cup \text{dom } h'$ is of the form $\ell^{\boldsymbol{\sigma};\boldsymbol{\tau}}_{\hat{\Gamma}'}$ where $\hat{\Gamma}'$ extends $\hat{\Gamma}$; and e, w are well-typed terms in $\hat{\Gamma}$.

Some auxiliary programs

We next introduce a few auxiliary expression contexts that will be useful in our operational rules. For readability, we shall write unit for the type expression {}, and void for the expression {} (so that void has type unit).

The following provides what is needed for passing from the "approximation operator" appearing in a class body to the method implementations for a particular object of that class. If e is of type

$$\mathsf{unit}\ \texttt{->}\ (\tau\natural(\sigma\texttt{**}\rho))\ \texttt{->}\ (\tau\natural(\sigma\texttt{**}\rho))$$

then we define

$$\mathsf{fold}_{\tau,\sigma}[e] \;:\; \tau\natural\sigma$$

to be the expression

$$\texttt{rec}\ x : \tau\natural\sigma\ \texttt{=>}\ e\ [\rho := \mathsf{unit}]\ \mathsf{void}\ x$$

Note that if e is explicitly typed then τ, σ may be inferred from e, so that in rule 75 below we write simply fold$[e]$.

Next, some machinery for combining the method or constructor implementations from a subclass and a superclass to yield appropriate method or constructor implementations for the class as a whole; this will be required for in rule 71 below. We give the machinery for constructors first. Suppose $\tau_1'' = \tau_1 + \tau_1'$ (see Section 6). Then given $v_2 : \tau_3\texttt{->}\mathsf{unit}\texttt{*(}\mathsf{unit}\texttt{->}\tau_1\texttt{)}$ and $v_2' : \tau_3''\texttt{->}\tau_3\texttt{*(}\tau_1\texttt{->}\tau_1''\texttt{)}$ we may define

$$\mathsf{comb\_constrs}_{\tau_1,\tau_1',\tau_3,\tau_3''}[v_2, v_2'] \;:\; \tau_3''\texttt{->}\mathsf{unit}\texttt{*}\tau_1''$$

to be the expression

```
fn $x : \tau_3''$ =>
    split $v_2'\, x$ as ($y' : \tau_3$, $z' : \tau_1'$) in
        split $v_2\, y'$ as ($y$ : unit, $z : \tau_1$) in
            (void, fn $a$ : unit =>$z'(z\ a)$))
        end
    end
```

For method implementations, we require some simple isomorphisms between types. If $\sigma, \sigma', \sigma''$ are such that $\sigma + \sigma'$ is defined, we take

$$\mathsf{merge}_{\sigma,\sigma'} \;:\; \sigma\texttt{*}\sigma'\texttt{->}\sigma + \sigma' \qquad \mathsf{unmerge}_{\sigma,\sigma'} \;:\; \sigma + \sigma'\texttt{->}\sigma\texttt{*}\sigma'$$

to be expressions defining the evident conversions; we omit the tedious definitions.

Now suppose $\tau_1'' = \tau_1 + \tau_1'$ and $\tau_2'' = \tau_2 \oplus \tau_2'$, and take $\tau_2^-, \tau_2'^-, \tau_d$ such that $\tau_2'' = \tau_2^- + \tau_2' = \tau_2 + \tau_2'^-$ and $\tau_2 = \tau_2^- + \tau_d$, $\tau_2' = \tau_2'^- + \tau_d$. Then given

$$\begin{array}{lcl} v_1 & : & \tau_2\natural(\tau_1\texttt{**}\rho)\ \texttt{->}\ \tau_2\natural(\tau_1\texttt{**}\rho) \\ v_1' & : & \tau_2\natural(\tau_1''\texttt{**}\rho')\ \texttt{->}\ \tau_2'\natural(\tau_1''\texttt{**}\rho')\ \texttt{->}\ \tau_2'\natural(\tau_1''\texttt{**}\rho') \end{array}$$

we may define

$$\mathsf{comb\_meths}_{\tau_1,\tau_1',\tau_2,\tau_2'}\,[v_1, v_1'] \;:\; \texttt{polytype}\ \rho\ \texttt{=>}\ \mathsf{unit}\ \texttt{->}\ \tau_2''\natural(\tau_1''\texttt{**}\rho')\ \texttt{->}\ \tau_2''\natural(\tau_1''\texttt{**}\rho')$$

to be the expression

```
fn $super$:unit => fn $self$:$\tau_2''\natural(\tau_1''$**$\rho')$ =>
    split unmerge $self$ as
        ($oldself$:$\tau_2\natural(\tau_1''$**$\rho)$, $newself$:$\tau_2'^-\natural(\tau_1''$**$\rho)$) in
    split unmerge $self$ as
        ($oldself'$:$\tau_2^-\natural(\tau_1''$**$\rho)$, $newself'$:$\tau_2'\natural(\tau_1''$**$\rho)$) in
    let val $oldimpl$:$\tau_2\natural(\tau_1''$**$\rho)$ = $v_1$ $[\rho := \tau_1'$**$\rho']$ void $oldself$ in
    let val $newimpl$:$\tau_2'\natural(\tau_1''$**$\rho)$ = $v_1'$ $oldself$ $newself'$ in
    split unmerge $oldimpl$ as
        ($oldimpl'$:$\tau_2^-\natural(\tau_1''$**$\rho')$, $diff$:$\tau_d\natural(\tau_1''$**$\rho')$) in
        merge ($oldimpl'$, $newimpl$)
    end end end end end
```

Here we have omitted the type subscripts on the occurrences of unmerge, unflatten$'$, merge since these may easily be inferred from the other type information present. Likewise, in rule 71 below we omit the type subscripts on comb_meths and comb_constrs, since these may be inferred from the types of the arguments.

Operational rules

We now give the rules for deriving evaluation judgements. Where test variable environments are omitted, it is understood that each judgement in the rule should be prefixed with "$\hat{\Gamma} \vdash$". Where heaps are omitted, it is understood that the following *heap convention* applies: a rule given as

$$\frac{\hat{\Gamma}_0 \vdash e_0 \Downarrow v_0 \qquad \cdots \qquad \hat{\Gamma}_{r-1} \vdash e_{r-1} \Downarrow v_{r-1}}{\hat{\Gamma} \vdash e \Downarrow v} \ \text{(side-conditions)}$$

abbreviates the rule

$$\frac{\hat{\Gamma}_0 \vdash h_0, e_0 \Downarrow h_1, v_0 \qquad \cdots \qquad \hat{\Gamma}_{n-1} \vdash h_{n-1}, e_{n-1} \Downarrow h_n, v_{n-1}}{\hat{\Gamma} \vdash h_0, e \Downarrow h_n, v} \ \text{(side-conditions)}$$

The following rules are associated with successful evaluation to a value (though some of them also deal with evaluation to stuck terms).

$$\frac{}{v \Downarrow v} \tag{54}$$

$$\frac{e_0 \Downarrow v_0 \qquad e_1 \Downarrow w_1}{\texttt{(}e_0\texttt{,}e_1\texttt{)} \Downarrow \texttt{(}v_0\texttt{,}w_1\texttt{)}} \tag{55}$$

$$\frac{e \Downarrow \texttt{(}v_0\texttt{,}v_1\texttt{)} \qquad e'[v_0/x, v_1/y] \Downarrow w}{\texttt{split}\ e\ \texttt{as (}x\texttt{,}y\texttt{) in}\ e'\ \texttt{end} \Downarrow w} \tag{56}$$

$$\frac{e_0 \Downarrow v_0 \qquad \cdots \qquad e_{n-1} \Downarrow v_{n-1}}{\{k_0\texttt{=}e_0\texttt{,} \ldots \texttt{,}k_{n-1}\texttt{=}e_{n-1}\} \Downarrow \{k_0\texttt{=}v_0\texttt{,} \ldots \texttt{,}k_{n-1}\texttt{=}v_{n-1}\}} \tag{57}$$

$$\frac{e \Downarrow v \qquad e' \Downarrow v'}{e\ \texttt{+>}\ e' \Downarrow v \oplus v'} \tag{58}$$

$$\frac{e \Downarrow \{k_0\texttt{=}v_0\texttt{,} \ldots \texttt{,}k_{n-1}\texttt{=}v_{n-1}\} \qquad e'[v_0/x_0, \ldots, v_{n-1}/x_{n-1}] \Downarrow w}{\texttt{split}\ e\ \texttt{as}\ \{k_0\texttt{=}x_0\texttt{:}\tau_0\texttt{,} \ldots \texttt{,}k_{n-1}\texttt{=}x_{n-1}\texttt{:}\tau_{n-1}\}\ \texttt{in}\ e'\ \texttt{end} \Downarrow w} \tag{59}$$

$$\frac{e \Downarrow w}{\texttt{\%}\,k\ e \Downarrow \texttt{\%}\,k\ w} \tag{60}$$

$$\frac{e \Downarrow \texttt{\%}\,k_i\ v \qquad e_i[v/x_i] \Downarrow w}{\texttt{case}\ e\ \texttt{of}\ \cdots\ \texttt{\%}\,k_i x_i\texttt{:}\tau_i\texttt{=>}e_i\ \cdots\ \texttt{end} \Downarrow w} \tag{61}$$

$$\frac{e \Downarrow w}{\texttt{prom}\ e \Downarrow \texttt{prom}\ w} \tag{62}$$

$$\frac{e \Downarrow \texttt{prom}\ v}{\texttt{der}\ e \Downarrow v} \tag{63}$$

$$\frac{e \Downarrow ec \qquad e' \Downarrow v}{e\texttt{\$}e' \Downarrow \phi_{ec}(v)} \tag{64}$$

$$\frac{e \Downarrow \texttt{fn}\ x : \tau\texttt{=>}e_0 \qquad e' \Downarrow v \qquad e_0[v/x] \Downarrow w}{e\texttt{\$}e' \Downarrow w} \tag{65}$$

$$\frac{e_0 \Downarrow v_0 \qquad e_1 \Downarrow v_1}{e_0\texttt{=}e_1 \Downarrow v} \quad v = eq(v_0, v_1) \tag{66}$$

$$\frac{e_0 \Downarrow \texttt{true} \qquad e_1 \Downarrow w}{\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow w} \tag{67}$$

$$\frac{e_0 \Downarrow \texttt{false} \qquad e_2 \Downarrow w}{\texttt{if } e_0 \texttt{ then } e_1 \texttt{ else } e_2 \Downarrow w} \tag{68}$$

$$\frac{e\,[\texttt{rec } x : \tau\texttt{=>}e \,/\, x] \Downarrow w}{\texttt{rec } x : \tau\texttt{=>}e \Downarrow w} \tag{69}$$

$$\frac{e \Downarrow w}{e\texttt{::}\tau \Downarrow w\texttt{::}\tau} \tag{70}$$

$$\frac{\begin{array}{c}\hat{\Gamma} \vdash e_0 \Downarrow lo \texttt{ extend triv with } v_1\texttt{,}v_2 \\ \hat{\Gamma} \vdash e_1 \Downarrow v_1' \qquad \hat{\Gamma} \vdash e_2 \Downarrow v_2'\end{array}}{\begin{array}{c}\hat{\Gamma} \vdash lo \texttt{ extend } e_0 \texttt{ with } e_1\texttt{,}e_2 \\ lo \texttt{ extend triv with } \mathsf{comb\_meths}\,[v_1, v_1']\texttt{,} \\ \mathsf{comb\_constrs}\,[v_2, v_2']\end{array}} \tag{71}$$

$$\frac{\hat{\Gamma} \vdash h_0, e \Downarrow h_1, (v_s\texttt{,}v_b)}{\hat{\Gamma} \vdash h_0, \texttt{push } e \Downarrow h_2, \ell^{\boldsymbol{\sigma},\boldsymbol{\tau}}_{\hat{\Gamma}}} \quad \begin{array}{l}\hat{\Gamma} \vdash v_s : \boldsymbol{\sigma} \\ \hat{\Gamma} \vdash v_b : \boldsymbol{\tau}\natural\boldsymbol{\sigma} \\ \ell = \mathsf{freshLoc}\,(\mathrm{dom}\ h_1) \\ h_2 = h_1(\ell^{\boldsymbol{\sigma},\boldsymbol{\tau}}_{\hat{\Gamma}} \mapsto v_s, v_b)\end{array} \tag{72}$$

$$\frac{h_0, e \Downarrow h_1, (v\texttt{,}v')}{h_0, \texttt{update } \tilde{\ell}\ e \Downarrow h_2, v} \quad \begin{array}{l}h_1(\tilde{\ell}) = v_s, v_b \\ h_2 = h_1(\tilde{\ell} \mapsto v', v_b)\end{array} \tag{73}$$

$$\frac{e \Downarrow lo \texttt{ new (}lo \texttt{ triv)} \qquad \texttt{push (}e'\texttt{,}\mathsf{void}\texttt{)} \Downarrow w}{e\texttt{\$}e' \Downarrow w} \tag{74}$$

$$\frac{\begin{array}{c}e \Downarrow lo \texttt{ new (}lo \texttt{ extend triv with } v_1\texttt{,}v_2\texttt{)} \\ \texttt{push (}v_2\,e'\texttt{,}\mathsf{fold}[v_1]\texttt{)} \Downarrow w\end{array}}{e\texttt{\$}e' \Downarrow w} \tag{75}$$

$$\frac{h_0, e \Downarrow h_1, \texttt{der } \tilde{\ell}\texttt{.}m \qquad h_1, \texttt{update } \tilde{\ell} \texttt{ (der } v_b\texttt{.}m \texttt{ \$ (}e'\texttt{,}v_s\texttt{))} \Downarrow h_2, w}{h_0, e\texttt{\$}e' \Downarrow h_2, w} \quad h_1(\tilde{\ell}) = v_s, v_b \tag{76}$$

$$\frac{\hat\Gamma \bullet \hat x : \sigma \vdash e[\hat x/x] \Downarrow \texttt{(}v_0\texttt{,}v_1\texttt{)}}{\begin{array}{c}\hat\Gamma \vdash \texttt{catchcont}\ x : \sigma\texttt{=>}e \Downarrow \\ \texttt{\%result \{value=}v_0\texttt{, more=fn}\ \hat x : \sigma\texttt{=>}v_1\texttt{\}}\end{array}} \quad \hat x \text{ fresh} \tag{77}$$

$$\frac{\hat\Gamma \bullet \hat x : \sigma \vdash e[\hat x/x] \Downarrow E[\texttt{der}\ \hat x\ \texttt{\$}\ v]}{\begin{array}{c}\hat\Gamma \vdash \texttt{catchcont}\ x : \sigma\texttt{=>}e \Downarrow \\ \texttt{\%query \{arg=}v\texttt{, resume=fn}\ z : \xi\texttt{=>fn}\ \hat x : \sigma\texttt{=>}E[z]\texttt{\}}\end{array}} \quad \begin{array}{l}\sigma = \zeta\texttt{->}\xi \\ \hat x \text{ fresh} \\ y = \mathsf{fresh}\,(E[-])\end{array} \tag{78}$$

$$\frac{h_0, e \Downarrow h_1, \texttt{fn}\ \hat x : \tau\texttt{=>}e_0 \qquad h_1, e' \Downarrow h_2, v \qquad h_2[v/\hat x], e_0[v/\hat x] \Downarrow h_3, w}{h_0,\ e\texttt{\$}e' \Downarrow h_3, w} \tag{79}$$

The following rules deal specifically with evaluation of expressions to stuck terms:

$$\frac{}{\texttt{der}\ \hat x\ \texttt{\$}\ v \Downarrow \texttt{der}\ \hat x\ \texttt{\$}\ v} \tag{80}$$

$$\frac{e \Downarrow E[\texttt{der}\ \hat x\ \texttt{\$}\ v]}{E'[e] \Downarrow (E' \circ E)[\texttt{der}\ \hat x\ \texttt{\$}\ v]} \quad E, E' \text{ transparent to } x \tag{81}$$

$$\frac{e_0 \Downarrow v_0 \qquad \cdots \qquad e_{i-1} \Downarrow v_{i-1} \qquad e_i \Downarrow u_i}{\texttt{\{}k_0\texttt{=}e_0\texttt{,}\ \ldots\texttt{,}k_{n-1}\texttt{=}e_{n-1}\texttt{\}} \Downarrow \texttt{\{}k_0\texttt{=}v_0\texttt{,}\ \ldots\texttt{,}k_i\texttt{=}u_i\texttt{,}\ \ldots\texttt{,}k_{n-1}\texttt{=}e_{n-1}\texttt{\}}} \quad i < n \tag{82}$$

$$\frac{e \Downarrow v \qquad e' \Downarrow u}{e\,\texttt{+>}\,e' \Downarrow v\,\texttt{+>}\,u} \tag{83}$$

$$\frac{e \Downarrow v \qquad e' \Downarrow u}{e\texttt{\$}e' \Downarrow v\,u} \tag{84}$$

$$\frac{e_0 \Downarrow v_0 \qquad e_1 \Downarrow u_1}{e_0\texttt{=}e_1 \Downarrow v_0\texttt{=}u_1} \tag{85}$$

$$\frac{e_0 \Downarrow v_0 \qquad e_1 \Downarrow w_1}{lo\ \texttt{extend}\ e_0\ \texttt{with}\ e_1\texttt{,}e_2 \Downarrow lo\ \texttt{extend}\ v_0\ \texttt{with}\ w_1\texttt{,}e_2} \tag{86}$$

$$\frac{e_0 \Downarrow v_0 \qquad e_1 \Downarrow v_1 \qquad e_2 \Downarrow w_2}{lo\ \texttt{extend}\ e_0\ \texttt{with}\ e_1\texttt{,}e_2 \Downarrow lo\ \texttt{extend}\ v_0\ \texttt{with}\ v_1\texttt{,}w_2} \tag{87}$$

$$\frac{\hat\Gamma \bullet \hat x : \sigma \vdash e[\hat x/x] \Downarrow E[\texttt{der}\ \hat y\ \texttt{\$}\ v]}{\hat\Gamma \vdash \texttt{catchcont}\ x : \sigma\texttt{=>}e \Downarrow \texttt{catchcont}\ x : \sigma\texttt{=>}E'[\texttt{der}\ \hat y\ \texttt{\$}\ v']} \quad \begin{array}{l}\hat y \neq \hat x \\ E' = E[x/\hat x] \\ v' = v[x/\hat x]\end{array} \tag{88}$$