Definition of the Lingay programming language (Version 0.2)

John Longley

August 26, 2008

Abstract

This document formally defines Lingay, a higher order object oriented programming language inspired by game semantics and intended as a kernel for the proposed Eriskay programming language. The present definition serves as a formal basis for metatheoretical study of the language and for the development of prototype implementations.

Introduction

This document formally defines Lingay — a strongly typed, higher order, class-based object oriented programming language whose design has been closely guided by ideas from game semantics. Lingay is proposed as a kernel for *Eriskay*, a full-scale programming language currently under development and envisaged as a framework for future work in program verification. We intend Lingay as a research language embodying many of the innovative aspects of Eriskay, and suitable for pedagogical purposes, metatheoretical study, programming experiments and case studies in program verification. The motivations for the project and its semantic underpinnings, along with an informal introduction to the main features of the language, are presented in a companion document.[1] A prototype implementation of Lingay is also available from the Eriskay project website.[2]

The present version of the definition of Lingay supersedes a substantially different one which appeared as an appendix to our contribution to the GaLoP III workshop (ETAPS, Budapest, April 2008), and which we now refer to as Version 0.1.

The author is grateful to Nicholas Wolverson for the research collaboration out of which this language definition has grown, and for his work on the implementation which generated much valuable feedback on the definition. This work was supported by EPSRC Grant GR/T08791: "A programming language based on game semantics".

[1]Longley, J. and Wolverson, N., *Eriskay: a programming language based on game semantics.* To appear as an Informatics Research Report, University of Edinburgh, 2008.

[2]`http://homepages.inf.ed.ac.uk/jrl/Eriskay`

1 Lexical matters

A Lingay program text is a sequence of ASCII characters. At the lexical level, a program text is analysed as a sequence of *input elements*, each of which is a *whitespace element*, a *comment*, or a *token*.

A whitespace element is a sequence consisting of a single whitespace character (space, horizontal tab, form feed, newline, or return).

A comment is either an *end-of-line* comment, of the form `//` *text* and running to the end of the line, or a *traditional* comment, of the form `/*` *text* `*/` where *text* is non-empty and possibly spreads over several lines. Formally, the set of possible comments is defined via the following regular expression:

$$(\texttt{//}\ \textit{non-line-term}^*) \ + \ (\texttt{/*}\ \textit{non-star}\ (\textit{non-star}^*\ \texttt{*}\ \texttt{*}^*\ \textit{non-star-slash})^*\ \textit{non-star}^*\ \texttt{*}\ \texttt{*}^*\ \texttt{/})$$

where *non-line-term* is the set of all ASCII characters except newline and return; *non-star* is the set of all ASCII characters except `*`; and *non-star-slash* is the set of all ASCII characters except `*` and `/`. Note that in Lingay, as in Java, traditional comments do not nest.

Tokens are certain non-empty sequences of non-whitespace ASCII characters, and are of two kinds: *reserved* and *non-reserved*. The reserved tokens in Lingay are as follows:[3]

```
{ } [| |] ( ) : ; . , ? ! -o -> => :< | = := $ ** +> + - < _ ...
as bool case class classimpl constr der deref else end extend extending false
fields fn fold for funclass if in int linear linclass methods new of oncefn
prom rec recval ref reftype root split tag then tok true unfold val unit with
```

Each non-reserved token belongs to one of five 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:

- *Identifiers*, ranged over by the metavariables x, y, k, f, m, t according to the context. In general, an identifier in Lingay is any finite sequence of alphabetic characters, numeric digits, underscores (`_`) and primes (`'`), beginning with an alphabetic character, which is not a reserved token.
- *Infixes*, which in Lingay are just the reserved tokens `+`, `-` and `<`, ranged over by $\imath$.
- *Boolean literals*, which are just the reserved tokens `true` and `false`.
- *Integer literals* such as `0`, `5` and `~23`. In general, an integer literal is either the token `0`, or a non-zero digit followed by zero or more further digits, or the 'minus' character `~` followed by a non-zero digit followed by zero or more further digits. Literals representing integers of arbitrary absolute size are admitted.
- *Type constants*, which in Lingay are just the reserved tokens `bool` and `int`, ranged over by κ.

[3]Strictly speaking, for the sake of compatibility with the future definition of Eriskay, the entire present definition of Lingay may be regarded as parameterized by a set of reserved tokens which must include those listed here.

We also define the class of *literals* (ranged over by *lit*) to consist of the boolean literals and integer literals.

As usual, the source text of a program is lexically analysed into a sequence of input elements according to the principle of longest match. Whitespace elements and comments are then discarded from this sequence, so that for the purpose of the remainder of the definition, a Lingay program may be regarded simply as a sequence of tokens.

2 Types

2.1 Surface and underlying type expressions

The syntax of *surface type expressions* is defined by the LALR(1) grammar given in Figure 1. Here, as elsewhere, the symbol ϵ indicates the empty sequence of tokens. If X is any phrase category, we write *Clist*(X) for the phrase category of (possibly empty) comma-separated lists of phrases of type X:

$$\begin{array}{rcl} \mathit{Clist}(X) & ::= & \epsilon \quad | \quad \mathit{Cbody}(X)\ X \\ \mathit{Cbody}(X) & ::= & \epsilon \quad | \quad \mathit{Cbody}(X)\ X\ \texttt{,} \end{array}$$

Phrases generated by the productions labelled † in Figure 1 are called *derived forms.* We let $\mathtt{t}$ range over surface type expressions (strictly speaking, over phrases of category *type-expr* or *type-expr1* in the grammar of Figure 1).

For the purpose of the formal definition, we distinguish between such surface type expressions and *underlying type expressions*, the latter being syntax trees for the (ambiguous) grammar given in Figure 2. We let τ, σ, ζ, ξ range over underlying type expressions (that is, syntax trees of category *type-expr* under the grammar of Figure 2), and π over underlying labelled product type expressions of the form $\{$*Clist* (*comp-type*)$\}$. We also let *lo* range over the phrase category *lin-opt.*

The following operations are associated with underlying type expressions.

- If $\pi = \{k_0 : \tau_0\,\texttt{,}\cdots\texttt{,}k_{n-1} : \tau_{n-1}\}$ and $\pi' = \{k'_0 : \tau'_0\,\texttt{,}\cdots\texttt{,}k'_{n'-1} : \tau_{n'-1}\}$ where the k_i and k'_j are all distinct, we write $\pi + \pi'$ for the type

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

  Otherwise $\pi + \pi'$ is undefined.

- If $\pi = \{k_0 : \tau_0\,\texttt{,}\cdots\texttt{,}k_{n-1} : \tau_{n-1}\}$ and $\pi' = \{k'_0 : \tau'_0\,\texttt{,}\cdots\texttt{,}k'_{n'-1} : \tau_{n'-1}\}$, and whenever $k_i = k'_j$ we have $\tau_i = \tau'_j$, we say π' is *compatible* with π. In this case we define $\pi \oplus \pi'$ to be the type

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

  where $i_0 < \cdots < i_{r-1}$ and $\{k_{i_0}, \ldots, k_{i_{r-1}}\} = \{k_0, \ldots, k_{n-1}\} - \{k'_0, \ldots, k'_{n'-1}\}$. If π' is not compatible with π, $\pi \oplus \pi'$ is undefined.

We say π' *extends* π if $\pi' = \pi + \pi''$ for some π'', or equivalently if $\pi' = \pi \oplus \pi''$ for some π'' (see rule 41 of Section 4).

```
        type-expr  ::=  type-expr1
                    |   type-expr1 -o  type-expr       (linear function type)
†                   |   type-expr1 ->  type-expr       (reusable function type)
†                   |   type-expr1 ** type-expr        (merged product type)
                    |   rectype t => type-expr         (recursive type)
       type-expr1  ::=  κ                              (type constant)
†                   |   unit                           (unit type)
†                   |   t                              (type name)
                    |   { Clist (comp-type) }          (labelled product)
                    |   VS (type-expr, type-expr)      (value-state product)
                    |   [| Clist (summand-type) |]     (labelled sum)
                    |   ! type-expr1                   (reusable type)
                    |   lin-opt classimpl type-expr,   (class implementation type)
                            type-expr, type-expr end
†                   |   lin-opt classimpl              (class implementation type,
                            fields type-expr            sugared version)
                            methods type-expr
                            constr type-expr end
†                   |   ( type-expr )                  (parsing brackets)
        comp-type  ::=  k : type-expr                  (component typing)
     summand-type  ::=  k of type-expr                 (summand typing)
†                   |   k                              (singleton summand)
          lin-opt  ::=  ϵ  |  linear
```

Figure 1: Context-free grammar for surface type expressions

```
        type-expr  ::=  κ
                    |   { Clist (comp-type) }
                    |   [| Clist (summand-type) |]
                    |   ! type-expr
                    |   type-expr -o  type-expr
                    |   rectype t => type-expr
                    |   lin-opt classimpl type-expr,
                            type-expr, type-expr end
        comp-type  ::=  k : type-expr
     summand-type  ::=  k of type-expr
          lin-opt  ::=  ϵ  |  linear
```

Figure 2: Context-free grammar for underlying type expressions

2.2 Translation of derived forms

Surface type expressions may be translated to underlying ones in the presence of a *type abbreviation environment*. Formally, a type abbreviation environment Θ is a finite partial function mapping identifiers t to underlying type expressions τ. If Θ is a type abbreviation environment, we define a function $-^{\Theta}$ from surface to underlying type expressions by induction on the structure of surface type expressions as follows. Note the distinction between object-level brackets `(`, `)` and meta-level parentheses $($, $)$.

- For the derived forms in Figure 1, we define

$$\begin{array}{rcl}
(\mathtt{t\ \text{->}\ t'})^{\Theta} & = & \mathtt{!\,(t^{\Theta}\ \text{-o}\ t'^{\Theta})} \\
(\mathtt{t\,**\,t'})^{\Theta} & = & \mathtt{t}^{\Theta} + \mathtt{t'}^{\Theta} \\
\mathtt{unit}^{\Theta} & = & \{\,\} \\
t^{\Theta} & = & \left[\begin{array}{ll} \Theta(\tau) & \text{if } \tau \in \mathrm{dom}\ \Theta \\ t & \text{otherwise} \end{array}\right. \\
\mathtt{VS\,(t,t')}^{\Theta} & = & \{\,\mathtt{value=t^{\Theta},\ state=t'^{\Theta}}\} \\
(\mathit{lo}\ \mathtt{classimpl\ fields\ t}_f & & \\
\mathtt{methods\ t}_m\ \mathtt{constr\ t}_k\ \mathtt{end})^{\Theta} & = & \mathit{lo}\ \mathtt{classimpl\ t}_f\,\mathtt{,t}_m\,\mathtt{,t}_k\ \mathtt{end}^{\Theta} \\
\mathtt{(t)}^{\Theta} & = & \mathtt{t}^{\Theta}
\end{array}$$

- For recursive types, we define

$$(\mathtt{rectype}\ t\ \mathtt{=>}\ \mathtt{t})^{\Theta} \quad = \quad \mathtt{rectype}\ t\ \mathtt{=>}\ \mathtt{t}^{\Theta\backslash t}$$

  where $(\Theta\backslash t)(t') = \Theta(t')$ for $t' \neq t$, and $(\Theta\backslash t)(t)$ is undefined.

- The remaining forms in Figure 1 are translated in the evident way, *e.g.*:

$$\begin{array}{rcl}
\{k_0\,\mathtt{:t}_0\,\mathtt{,}\,\cdots\,\mathtt{,}k_{n-1}\,\mathtt{:t}_{n-1}\}^{\Theta} & = & \{k_0\,\mathtt{:t}_0^{\Theta}\,\mathtt{,}\,\cdots\,\mathtt{,}k_{n-1}\,\mathtt{:t}_{n-1}^{\Theta}\} \\
\mathtt{[|}\,sm_0\,\mathtt{,}\,\cdots\,\mathtt{,}sm_{n-1}\,\mathtt{|]}^{\Theta} & = & \mathtt{[|}\,sm_0^{\Theta}\,\mathtt{,}\,\cdots\,\mathtt{,}sm_{n-1}^{\Theta}\,\mathtt{|]}
\end{array}$$

  In this last clause, we make use of an auxiliary translation on phrases *sm* of category *summand-type* defined by

$$\begin{array}{rcl}
(k\ \mathtt{of\ t})^{\Theta} & = & k\ \mathtt{of\ t}^{\Theta} \\
k^{\Theta} & = & k\ \mathtt{of}\ \{\,\}
\end{array}$$

2.3 Meta-level type abbreviations

As an aid to readability, we shall freely use the above derived forms in meta-level expressions intended to denote underlying type expressions. Thus, for example, the meta-level expression $\tau\,\text{->}\,\tau'$ serves as an abbreviation for the meta-level expression $!\,(\tau\,\text{-o}\,\tau')$. We also introduce the following meta-level abbreviations for type expressions:

$$\mathsf{VR}\,(\tau_0, \tau_1) \quad = \quad \{\,\mathtt{value:}\tau_0\mathtt{,\ residue:}\tau_1\,\}$$

$$\begin{array}{rcl}
\mathsf{Catchcont}\,(\zeta,\xi,\tau_0,\tau_1) & = & \texttt{[| result : }\{\texttt{value:}\tau_0\texttt{, more: }(\zeta\,\texttt{->}\,\xi)\,\texttt{-o}\,\tau_1\}\texttt{,} \\
 & & \quad\texttt{query : }\{\texttt{arg:}\zeta\texttt{, resume:}\,\xi\,\texttt{-o}\,(\zeta\,\texttt{->}\,\xi)\,\texttt{-o}\,\mathsf{VR}\,(\tau_0,\tau_1)\}\ \texttt{|]} \\
\mathsf{Root}_{lo} & = & lo\ \texttt{classimpl unit,unit,unit end} \\
\mathsf{Read}(\sigma) & = & \texttt{unit}\,\texttt{->}\,\sigma \\
\mathsf{Write}(\sigma) & = & \sigma\,\texttt{->}\,\texttt{unit} \\
\mathsf{RW}\,(\sigma) & = & \{\,\texttt{read:}\mathsf{Read}(\sigma)\texttt{, write:}\mathsf{Write}(\sigma)\}
\end{array}$$

For a labelled product type $\pi = \{k_0\!:\!\sigma_0\,\texttt{,}\,\ldots\,\texttt{,}k_{n-1}\!:\!\sigma_{n-1}\}$, we then define

$$\mathsf{RW\_all}\,(\pi) \;=\; \{k_0\!:\!\mathsf{RW}\,(\sigma_0)\texttt{,}\ \cdots\ \texttt{,}k_{n-1}\!:\!\mathsf{RW}\,(\sigma_{n-1})\}$$

We also introduce a notation $\pi\natural_{lo}\pi'$, whose definition depends on the value of *lo*:

- If $\pi = \{k_0\!:\texttt{!}\tau_0\,\texttt{,}\,\cdots\,\texttt{,}k_{n-1}\!:\texttt{!}\tau_{n-1}\}$ and π' is an arbitrary labelled product type expression, we define

$$\pi\natural_{\epsilon}\pi' \;=\; \{k_0\!:\!\mathsf{RW\_all}\,(\pi')\,\texttt{->}\,\tau_0\texttt{,}\ \cdots\ \texttt{,}k_{n-1}\!:\!\mathsf{RW\_all}\,(\pi')\,\texttt{->}\,\tau_{n-1}\}$$

- If $\pi = \{k_0\!:\!\rho_0\,\texttt{->}\,\rho'_0\,\texttt{,}\,\cdots\,\texttt{,}k_{n-1}\!:\!\rho_{n-1}\,\texttt{->}\,\rho'_{n-1}\}$, we define

$$\pi\natural_{\texttt{linear}}\pi' \;=\; \{k_0\!:\!\texttt{VS}\,(\rho_0,\pi')\,\texttt{->}\,\texttt{VS}\,(\rho'_0,\pi')\texttt{,}\ \cdots\ \texttt{,}k_{n-1}\!:\!\texttt{VS}\,(\rho_{n-1},\pi')\,\texttt{->}\,\texttt{VS}\,(\rho'_{n-1},\pi')\}$$

In each case, if π is not of the appropriate form then $\pi\natural_{lo}\pi'$ is undefined.

2.4 Well-formedness and status for types

The rules below generate judgements of the form $\Delta \vdash \tau \propto sd$, meaning "$\tau$ is a well-formed type expression with status *sd* in status environment Δ". Here *sd* ranges over the *status descriptors* ordinary, reusable, ground, which we take to be linearly ordered as follows:

$$\mathsf{ordinary} \sqsubseteq \mathsf{reusable} \sqsubseteq \mathsf{ground}$$

A status environment Δ has the form $\bullet\, t_0 \propto sd_0 \cdots \bullet\, t_{n-1} \propto sd_{n-1}$, where the identifiers t_i play the role of free type names. We write $\Delta[t]$ for the status associated with the rightmost appearance of t in Δ (if there is no such appearance then $\Delta[t]$ is undefined).

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

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

for some $\sigma'_0,\ldots,\sigma'_{q-1}$ and distinct identifiers $m_0,\ldots,m_{q-1}$. We say simply that π_m is *of class form* if it is of class form relative to some $\sigma_0,\ldots,\sigma_{q-1}$.

$$\frac{\Delta \vdash \tau \propto sd}{\Delta \vdash \tau \propto sd'}\ \ sd' \sqsubseteq sd \qquad (1)$$

$$\frac{}{\Delta \vdash \kappa \propto \mathsf{ground}} \tag{2}$$

$$\frac{}{\Delta \vdash t \propto sd} \quad \Delta[t] = sd \tag{3}$$

$$\frac{\Delta \vdash \tau_0 \propto sd \qquad \cdots \qquad \Delta \vdash \tau_{n-1} \propto sd}{\Delta \vdash \texttt{\{}\ k_0 \texttt{:} \tau_0 \texttt{,} \ldots \texttt{,} k_{n-1} \texttt{:} \tau_{n-1}\ \texttt{\}} \propto sd} \quad k_0, \ldots, k_{n-1} \text{ distinct} \tag{4}$$

$$\frac{\Delta \vdash \tau_0 \propto sd \qquad \cdots \qquad \Delta \vdash \tau_{n-1} \propto sd}{\Delta \vdash \texttt{[|}\ k_0 \texttt{:} \tau_0 \texttt{,} \ldots \texttt{,} k_{n-1} \texttt{:} \tau_{n-1}\ \texttt{|]} \propto sd} \quad \begin{array}{l} n > 0 \\ k_0, \ldots, k_{n-1} \text{ distinct} \end{array} \tag{5}$$

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

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

$$\frac{\Delta \vdash \tau \propto \mathsf{ordinary} \qquad \Delta \vdash \tau' \propto \mathsf{ordinary}}{\Delta \vdash \tau \,\texttt{-o}\, \tau' \propto \mathsf{ordinary}} \tag{8}$$

$$\frac{\Delta \bullet t \propto sd \vdash \tau \propto sd}{\Delta \vdash \texttt{rectype}\ t\ \texttt{=>}\ \tau \ \propto sd} \quad t \notin \operatorname{dom} \Delta \tag{9}$$

$$\frac{\Delta \vdash \pi_f \propto \mathsf{reusable} \qquad \Delta \vdash \pi_m \propto \mathsf{reusable} \qquad \Delta \vdash \tau_k \propto \mathsf{ordinary}}{\Delta \vdash \texttt{classimpl}\ \pi_f \texttt{,} \pi_m \texttt{,} \tau_k\ \texttt{end} \propto \mathsf{reusable}} \quad \pi_m \text{ of class form} \tag{10}$$

$$\frac{\begin{array}{c} \Delta \vdash \pi_f \propto \mathsf{ordinary} \qquad \Delta \vdash \pi_m \propto \mathsf{reusable} \qquad \Delta \vdash \tau_k \propto \mathsf{ordinary} \\ \Delta \vdash \sigma_0 \propto \mathsf{ground} \qquad \cdots \qquad \Delta \vdash \sigma_{q-1} \propto \mathsf{ground} \end{array}}{\Delta \vdash \texttt{linear classimpl}\ \pi_f \texttt{,} \pi_m \texttt{,} \tau_k\ \texttt{end} \propto \mathsf{reusable}} \quad \begin{array}{l} \pi_m \text{ of class} \\ \text{form relative to} \\ \sigma_0, \ldots, \sigma_{q-1} \end{array} \tag{11}$$

We say τ is a *well-formed* type expression in Δ if $\Delta \vdash \tau \propto \mathsf{ordinary}$. We say that well-formed type expressions τ, τ' in Δ are *equivalent*, and write $\tau \approx \tau'$, if they are the same up to permutations of components in labelled product and sum types and renaming of bound type names t in type expressions `rectype` t `=>` τ; we omit the formal definition. For the purpose of Lingay, we may define *underlying types* in Δ to be the well-formed types in Δ modulo equivalence. We let $\boldsymbol{\tau}, \boldsymbol{\sigma}$ range over underlying types, and $\boldsymbol{\pi}$ over underlying labelled product types. As a slight abuse of notation, we sometimes write $\tau \approx \boldsymbol{\tau}$ to mean that τ is an element of the equivalence class $\boldsymbol{\tau}$. Likewise, we say $\boldsymbol{\pi}'$ extends π if some element π' of $\boldsymbol{\pi}'$ extends π.

We write $\mathrm{FTN}(\tau)$ for the set of type names that occur free in a type expression τ (where `reftype` is treated as a binder), and similarly for underlying types. We write $\tau[\tau'/t]$ for the result of replacing each free occurrence of t within τ by τ' (no renaming of bound type names is necessary).

We also define the notion of a *groundless* type expression inductively as follows:

- `int` and `bool` are not groundless.
- Type names t are not groundless.
- $\{k_0 : \tau_0\,,\cdots,k_{n-1} : \tau_{n-1}\}$ is groundless iff all the τ_i are groundless.
- `[|`$k_0 : \tau_0\,,\cdots,k_{n-1} : \tau_{n-1}$`|]` is groundless iff $n = 1$ and τ_0 is groundless.
- `!`τ is groundless iff τ is groundless.
- τ `-o` τ' is groundless.
- `rectype` t `=>` τ is groundless iff τ is groundless.
- `classimpl` $\pi_f\,,\pi_m\,,\tau_k$ `end` and `linear classimpl` $\pi_f\,,\pi_m\,,\tau_k$ `end` are groundless.

Finally, we define the notion of a *first-order* type expression in Δ inductively as follows:

- If $\Delta \vdash \tau \propto \mathsf{ground}$ then τ is first-order in Δ.
- $\{k_0 : \tau_0\,,\cdots,k_{n-1} : \tau_{n-1}\}$ and `[|`$k_0 : \tau_0\,,\cdots,k_{n-1} : \tau_{n-1}$`|]` are first-order in Δ iff all the τ_i are first-order in Δ.
- `!`τ is first-order in Δ iff τ is.
- If $\Delta \vdash \tau \propto \mathsf{ground}$ and τ' is first-order in Δ, then τ `-o` τ' is first-order in Δ.
- If τ is first-order in $\Delta \bullet\ t \propto \mathsf{ground}$, then `rectype` t `=>` τ is first-order in Δ.

Note that type expressions of the form τ `-o` τ' or *lo* `classimpl` π_f, π_m, τ_k `end` are never first-order.

2.5 Subtyping

Next we define a *subtyping* relation on well-formed type expressions. The following rules generate assertions of the form $\Sigma \vdash \tau <: \tau'$, where Σ is a *subtyping environment* of the form $\bullet\, t_0 <: t'_0 \cdots \bullet\ t_{n-1} <: t'_{n-1}$. Here S_n denotes the set of permutations of $\{0, \ldots, n-1\}$.

$$\frac{}{\Sigma \vdash \kappa <: \kappa} \tag{12}$$

$$\frac{}{\Sigma \vdash t <: t'} \quad \begin{array}{l} t = t' \text{ or} \\ \bullet\, t <: t' \in \Sigma \end{array} \tag{13}$$

$$\frac{\Sigma \vdash \tau_0 <: \tau'_0 \quad \cdots \quad \Sigma \vdash \tau_{n-1} <: \tau'_{n-1}}{\Sigma \vdash \{k_0 : \tau_0\,,\cdots,k_{n'-1} : \tau_{n'-1}\} <: \{k_{p0} : \tau'_{p0}\,,\cdots,k_{p(n-1)} : \tau'_{p(n-1)}\}} \quad \begin{array}{c} n' \geq n \\ p \in S_n \\ \tau_n, \ldots, \tau_{n'-1} \\ \text{groundless} \end{array} \tag{14}$$

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

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

$$\frac{\Sigma \vdash \tau_0' <: \tau_0 \qquad \Sigma \vdash \tau_1 <: \tau_1'}{\Sigma \vdash \tau_0 \texttt{-o}\, \tau_1 <: \tau_0' \texttt{-o}\, \tau_1'} \tag{17}$$

$$\frac{\Sigma \bullet t_0' <: t_1' \;\vdash\; \tau_0[t_0'/t_0] <: \tau_1[t_1'/t_1]}{\Sigma \vdash \texttt{rectype}\ t_0\ \texttt{=>}\ \tau_0 <: \texttt{rectype}\ t_1\ \texttt{=>}\ \tau_1} \quad \begin{array}{l} t_0' \notin \tau_0,\ t_1' \notin \tau_1 \\ t_0' \neq t_1' \end{array} \tag{18}$$

$$\frac{\Sigma \vdash \tau_k' <: \tau_k}{\Sigma \vdash \mathit{lo}\ \texttt{classimpl}\ \pi_f\texttt{,}\pi_m\texttt{,}\tau_k\ \texttt{end} <: \mathit{lo}\ \texttt{classimpl}\ \pi_f\texttt{,}\pi_m\texttt{,}\tau_k'\ \texttt{end}} \tag{19}$$

If τ, τ' are well-formed type expressions in some Δ, we write simply $\tau <: \tau'$ for $\emptyset \vdash \tau <: \tau'$, where $\emptyset$ denotes the empty subtyping environment. 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 in any given Δ.

2.6 Types for literals and infixes

To each literal *lit* we associate a type $\mathit{ty}(\mathit{lit})$ as follows: boolean literals have type `bool`, and integer literals have type `int`. To each infix $\imath$ we associate a result type $\mathit{ty}(\imath)$ as follows: $\mathit{ty}(\texttt{+}) = \mathit{ty}(\texttt{-}) = \texttt{int}$, and $\mathit{ty}(\texttt{<}) = \texttt{bool}$.

3 Expressions

3.1 Surface and underlying expressions

The surface syntax of *expressions* is given by the grammar of Figures 3 and 4, in conjunction with that of Figure 1. For any phrase category X, we define $\mathit{Clist}(X)$ as in Section 2, and define $\mathit{Blist}(X)$ to be the phrase category of (possibly empty) bar-separated lists over X:

$$\begin{array}{rcl} \mathit{Blist}(X) & ::= & \epsilon \quad | \quad \mathit{Bbody}(X)\ X \\ \mathit{Bbody}(X) & ::= & \epsilon \quad | \quad \mathit{Bbody}(X)\ X\ \texttt{|} \end{array}$$

Phrases generated by the productions labelled † are called *derived forms*. We let `e` range over surface expressions — that is, over phrases of any of the categories

expr expr1 expr2 expr3 expr4 expr0

	expr	::=	*expr1*	
		|	`oncefn` *match*	(linear function)
†		|	`fn` *match*	(reusable function)
		|	`rec` *match*	(fixed point)
		|	`catchcont` *match*	(continuation catching)
		|	`if` *expr* `then` *expr* `else` *expr*	(conditional)
†		|	*k* `:=` *expr*	(field update)
	expr1	::=	*expr2*	
		|	*expr2* `=` *expr2*	(equality test)
		|	*expr2* `\` *expr2*	(value-state pair)
	expr2	::=	*expr3*	
		|	*expr2* `$` *expr3*	(strict application)
		|	*expr3* $\imath$ *expr3*	(infix expression)
		|	*expr3* `+>` *expr3*	(record merge/override)
		|	*expr3* `:<` *type-expr*	(upcasting)
	expr3	::=	*expr4*	
		|	`tag` *k* *expr4*	(summand labelling)
†		|	`tok` *k*	(singleton summand)
†		|	*expr3* `.` *k*	(component extraction)
		|	`prom` *expr4*	(reusable promotion)
		|	`der` *expr4*	(non-reusable dereliction)
		|	`force` *expr4*	(promoted thunk forcing)
		|	`fold` *type-expr1* *expr4*	(rectype introduction)
		|	`unfold` *expr4*	(rectype elimination)
†		|	*expr3* *expr4*	(derelicted application)
		|	`new` *expr4*	(new object)
		|	`make` *expr4* *expr4*	(special linear object)
		|	`ref` *t* *expr4*	(referencing)
		|	`deref` *expr4*	(dereferencing)
	expr4	::=	*x*	(variable)
		|	*lit*	(literal)
†		|	`()`	(unit value)
†		|	`?` *k*	(field access)
		|	{ *Clist* (*comp-assign*) }	(labelled record)
		|	`split` *expr0* `as` *record-pat* `in` *expr0* `end`	(record separation)
†		|	`split` *expr0* `as` *record-pat1* `in` *expr0* `end`	(sugared record separation)
		|	`case` *expr0* `of` *Blist* (*case-clause*) `end`	(sum elimination)
		|	*lin-opt* `root`	(trivial class body)
		|	*lin-opt* `extend` *expr* `with` *expr*`,` *expr* `end`	(class extension)
†		|	*class-expr*	(sugared class expression)
		|	`reftype` *t* `for` *type-expr* `in` *expr0* `end`	(reference type expression)
†		|	`(` *expr0* `)`	(parsing brackets)

Figure 3: Context-free grammar for surface expressions, part 1

	expr0	::=	*expr*	
†		|	*decl expr0*	(local declaration)
	decl	::=	`val` *pattern* `=` *expr* `;`	(simple declaration)
		|	`recval` *x* `:` *type-expr* `=` *expr* `;`	(recursive declaration)
		|	*expr* `;`	(`it` declaration)
	match	::=	*x* `:` *type-expr* `=>` *expr*	(simple match clause)
†		|	*pattern1* `:` *type-expr* `=>` *expr*	(derived match clause)
†		|	`fold` *type-expr1 pattern* `=>` *expr*	(rectype match clause)
	pattern	::=	*pattern1*	
		|	*x*	(atomic pattern)
		|	*record-pat*	(record pattern)
		|	`fold` *type-expr1 pattern*	(rectype pattern)
	pattern1	::=	`_`	(wildcard pattern)
		|	*record-pat1*	(delimited record pattern)
		|	`(` *pattern* `)`	(parsing brackets)
	record-pat	::=	{ *comp-pat-list* } `+` *x*	(general record binding)
	record-pat1	::=	{ *comp-pat-list* }	(rigid record binding)
		|	{ *comp-pat-list* `...` }	(flexible record binding)
		|	`(` *pattern* `\` *pattern* `)`	(value-state pattern)
	comp-pat-list	::=	*Clist* (*comp-pat*)	
	comp-pat	::=	*k* `=` *pattern*	(component pattern)
	comp-assign	::=	*k* `=` *expr*	(component assignment)
	case-clause	::=	*k x* `=>` *expr0*	(simple case clause)
†		|	*k pattern1* `=>` *expr0*	(derived case clause)
†		|	*k* `=>` *expr0*	(singleton case clause)
	class-expr	::=	*class-kind* `::` *type-expr* *extend-clause* `with` *class-details* `end`	(sugared class expression)
	class-kind	::=	`class` | `funclass` | `linclass`	
	extend-clause	::=	ϵ	
		|	`extending` *expr* `:` *type-expr*	
		|	`extending` *x*	
	class-details	::=	`fields` *type-expr* `methods` { *Clist* (*method-body*) } `constr` *match*	
	method-body	::=	*m pattern1* `=` *expr*	

Figure 4: Context-free grammar for surface expressions, part 2

$$
\begin{array}{rcl}
\mathit{expr} & ::= & x \\
 & | & \mathit{lit} \\
 & | & \{\, \mathit{Clist}\,(\mathit{comp\text{-}assign})\,\} \\
 & | & \mathit{expr}\ \texttt{+>}\ \mathit{expr} \\
 & | & \texttt{split}\ \mathit{expr}\ \texttt{as}\ \mathit{record\text{-}pat}\ \texttt{in}\ \mathit{expr}\ \texttt{end} \\
 & | & \texttt{tag}\ k\ \mathit{expr} \\
 & | & \texttt{case}\ \mathit{expr}\ \texttt{of}\ \mathit{Blist}(\mathit{case\text{-}clause})\ \texttt{end} \\
 & | & \texttt{prom}\ \mathit{expr} \\
 & | & \texttt{der}\ \mathit{expr} \\
 & | & \texttt{force}\ \mathit{expr} \\
 & | & \texttt{fold}\ \mathit{type\text{-}expr}\ \mathit{expr} \\
 & | & \texttt{unfold}\ \mathit{expr} \\
 & | & \texttt{oncefn}\ \mathit{match} \\
 & | & \mathit{expr}\ \texttt{\$}\ \mathit{expr} \\
 & | & \mathit{expr}\ \texttt{=}\ \mathit{expr} \\
 & | & \mathit{expr}\ \imath\ \mathit{expr} \\
 & | & \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr} \\
 & | & \texttt{rec}\ \mathit{match} \\
 & | & \texttt{catchcont}\ \mathit{match} \\
 & | & \mathit{expr}\ \texttt{:<}\ \mathit{type\text{-}expr} \\
 & | & \mathit{lin\text{-}opt}\ \texttt{root} \\
 & | & \mathit{lin\text{-}opt}\ \texttt{extend}\ \mathit{expr}\ \texttt{with}\ \mathit{expr}\texttt{,}\mathit{expr}\ \texttt{end} \\
 & | & \texttt{new}\ \mathit{expr} \\
 & | & \texttt{make}\ \mathit{expr}\ \mathit{expr} \\
 & | & \texttt{reftype}\ t\ \texttt{for}\ \mathit{type\text{-}expr}\ \texttt{in}\ \mathit{expr} \\
 & | & \texttt{ref}\ t\ \mathit{expr} \\
 & | & \texttt{deref}\ \mathit{expr} \\
\mathit{match} & ::= & x\ \texttt{:}\ \mathit{type}\ \texttt{=>}\ \mathit{expr} \\
\mathit{comp\text{-}assign} & ::= & k\ \texttt{=}\ \mathit{expr} \\
\mathit{record\text{-}pat} & ::= & \{\, \mathit{Clist}\,(\mathit{comp\text{-}bind})\,\}\ \texttt{+}\ x \\
\mathit{comp\text{-}bind} & ::= & k\ \texttt{=}\ x \\
\mathit{case\text{-}clause} & ::= & k\ x\ \texttt{=>}\ \mathit{expr}
\end{array}
$$

Figure 5: Context-free grammar for underlying expressions

$$
\begin{array}{rcl}
\mathit{value} & ::= & x \\
 & \mid & \mathit{lit} \\
 & \mid & \{\ \mathit{Clist}\ (\mathit{comp\text{-}value})\ \} \\
 & \mid & \mathtt{tag}\ k\ \mathit{value} \\
 & \mid & \mathtt{prom}\ \mathit{value} \\
 & \mid & \mathtt{fold}\ \mathit{type\text{-}expr}\ \mathit{value} \\
 & \mid & \mathtt{oncefn}\ x\ \mathtt{:}\ \mathit{type}\ \mathtt{=>}\ \mathit{expr} \\
 & \mid & \mathit{value}\ \mathtt{:<}\ \mathit{type\text{-}expr} \\
 & \mid & \mathit{lin\text{-}opt}\ \mathtt{root} \\
 & \mid & \mathit{lin\text{-}opt}\ \mathtt{extend}\ \mathit{lin\text{-}opt}\ \mathtt{root}\ \mathtt{with}\ \mathit{value}\mathtt{,}\mathit{value}\ \mathtt{end} \\
 & \mid & \mathtt{reftype}\ t\ \mathtt{for}\ \mathit{type\text{-}expr}\ \mathtt{in}\ \mathit{value}\ \mathtt{end} \\
\mathit{comp\text{-}value} & ::= & k\ \mathtt{=}\ \mathit{value}
\end{array}
$$

Figure 6: Context-free grammar for values

We also introduce metavariables ranging over other phrase categories as follows.

mt	$\in$	*match*
cs	$\in$	*case-clause*
csl	$\in$	*Blist* (*case-clause*)
ck	$\in$	*class-kind*
dtls	$\in$	*class-details*
mb	$\in$	*method-body*
pt	$\in$	*pattern*
pt1	$\in$	*pattern1*
rp	$\in$	*record-pat*
rp1	$\in$	*record-pat1*

We say `e` is an expression of the *core language* if `e` does not contain any of the tokens `reftype`, `ref` or `deref`.

As with type expressions, we distinguish between surface expressions and *underlying expressions*, the latter being syntax trees for the (ambiguous) grammar given in Figure 5 (in conjunction with that of Figure 2). We let e range over underlying expressions. The set $\mathrm{FV}(e)$ of *free variables* of e is defined as usual, where `split`, `case`, `oncefn`, `rec` and `catchcont` are treated as binders; we omit the formal definition.

Certain underlying 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 rules 27 and 41 of Section 4. The grammar for values is given in Figure 6. We let v range over values.

3.2 Translation of derived forms

A surface expression is translated into an underlying expression in two stages. The first stage, which is the only 'type-aware' part of the translation, simply consists of expanding each extending clause of the form `extending` x to `extending` x `: t`, where `t` is (any surface form for) the type associated with x in the current top-level environment Γ (see Section 7). This translation stage may be seen as a concession to the absence of any type inference mechanism in Lingay, and greatly reduces the amount of type annotation required in typical class definitions.

The second stage makes use of a type abbreviation environment Θ, and extends the translation $-^{\Theta}$ defined on type expressions in Section 2. Although rather complex, this section stage can be presented as simply a transformation on parse trees.

Formally, we define a translation $-^{\Theta}$ which maps surface expressions to underlying expressions; surface match clauses to underlying match clauses; and surface case clauses to underlying case clauses. In the following definition, we adopt the convention that the metavariables z, z' stand for *fresh variables.* More precisely, a clause of the form

$$(\mathit{lhs})^{\Theta} \;=\; \mathit{rhs}$$

asserts that for any surface expression lhs^* obtained as an instance of lhs via some meta-substitution μ, the translation $(\mathit{lhs}^*)^{\Theta}$ is given as the value of the meta-expression rhs^* obtained from rhs by first applying the meta-substitution μ, and then further meta-substituting for z and z' two distinct identifiers not appearing in lhs^*. The choice of these identifiers is immaterial; they may, if desired, be specified explicitly by a suitable choice function.

- The translation on expressions is defined as follows.
  - For the derived forms in Figures 3 and 4, we define

$$
\begin{array}{rcl}
(\texttt{fn}\ \mathit{mt})^{\Theta} & = & \texttt{prom}\ (\texttt{oncefn}\ \mathit{mt}^{\Theta}) \\
(k\ \texttt{:=}\ \texttt{e})^{\Theta} & = & (\texttt{rw.}k\texttt{.write e})^{\Theta} \\
(\texttt{tok}\ k)^{\Theta} & = & \texttt{tag}\ k\ \{\,\} \\
(\texttt{e.}k)^{\Theta} & = & \texttt{split e}^{\Theta}\ \texttt{as}\ \{k\texttt{=x}\}\texttt{+y in x end} \\
(\texttt{e e}')^{\Theta} & = & (\texttt{der e}^{\Theta})\ \texttt{\$ e}'^{\Theta} \\
\texttt{()}^{\Theta} & = & \{\} \\
(\texttt{?}\ k)^{\Theta} & = & (\texttt{rw.}k\texttt{.read \{\}})^{\Theta} \\
(\texttt{split e as}\ \{\mathit{cbl}\ldots\}\ \texttt{in e}'\ \texttt{end})^{\Theta} & = & (\texttt{split e as}\ \{\mathit{cbl}\}\texttt{+}z\ \texttt{in e}'\ \texttt{end})^{\Theta} \\
(\texttt{split e as}\ \{\mathit{cbl}\}\ \texttt{in e}'\ \texttt{end})^{\Theta} & = & (\texttt{split e as}\ \{\mathit{cbl}\}\texttt{+}z\ \texttt{in} \\
 & & \quad \texttt{val}\ z'\ \texttt{=}\ z\texttt{=() ; e}' \\
 & & \ \texttt{end})^{\Theta} \\
(\texttt{e})^{\Theta} & = & \texttt{e}^{\Theta} \\
(\texttt{val}\ \mathit{pt}\ \texttt{= e ; e}')^{\Theta} & = & \texttt{split}\ \{\texttt{k=e}^{\Theta}\}\ \texttt{as}\ \{\texttt{k=}\mathit{pt}\}\texttt{+}z\ \texttt{in e}'^{\Theta}\ \texttt{end} \\
(\texttt{recval}\ x\texttt{:t = e ; e}')^{\Theta} & = & (\texttt{val}\ x\ \texttt{= rec}\ x\texttt{:t => e ; e}')^{\Theta} \\
(\texttt{e ; e}')^{\Theta} & = & (\texttt{val it = e ; e}')^{\Theta}
\end{array}
$$

It remains to give the translation for sugared class expressions. For class definitions with empty extending clause, we define

$$(\mathit{ck}\ \texttt{::}\ \mathsf{t}\ \texttt{with}\ \mathit{dtls}\ \texttt{end})^{\Theta} \;=\; \begin{array}{l}(\mathit{ck}\ \texttt{::}\ \mathsf{t}\\ \quad \texttt{extending}\ \mathit{lo}\ \texttt{root : }\mathsf{Root}_{lo}\\ \quad \texttt{with}\ \mathit{dtls}\ \texttt{end})^{\Theta}\end{array}$$

where $lo = \texttt{linear}$ if $ck = \texttt{linclass}$, and $lo = \epsilon$ otherwise. For class definitions with an explicitly typed extending clause, we define

```
(ck class :: $\mathsf{t}'_m$ extending $\mathsf{e}_c$ : $\mathsf{t}_c$ with
     fields $\mathsf{t}''_f$
     methods { $mb_0$, ··· , $mb_{n-1}$ }
     constr $x$:$\mathsf{t}'_k$ => $\mathsf{e}_k$
 end)$^{\Theta}$
```

to be the underlying expression

```
lo extend $\mathsf{e}^{\Theta}$ with
     fn supervar : $\tau_{super}$ => fn selfvar : $\tau_{self}$ =>
          selfvar +>
          {trans ($mb_0, ck, \pi'_f, \pi_m, \pi'_m, \Theta$),
           ··· ,
           trans ($mb_{n-1}, ck, \pi'_f, \pi_m, \pi'_m, \Theta$)} ,
     fn super : $\tau_k$ -> $\pi_f$ => fn $x$:$\tau'_k$ => $\mathsf{e}_k^{\Theta}$
end
```

where

$lo = \texttt{linear}$ if $ck = \texttt{linclass}$, $lo = \epsilon$ otherwise
$\mathsf{t}_c{}^{\Theta} = lo\ \texttt{classimpl}\ \pi_f\texttt{,}\pi_m\texttt{,}\tau_k\ \texttt{end}$
$\mathsf{t}'_m{}^{\Theta} = \pi'_m, \quad \pi'_m$ extends π_m
$\mathsf{t}''_f{}^{\Theta} = \pi''_f, \quad \pi'_f = \pi_f + \pi''_f, \quad \mathsf{t}'_k{}^{\Theta} = \tau_k$
$\tau_{super} = \pi_m \natural_{lo} \pi'_f, \quad \tau_{self} = \pi'_m \natural_{lo} \pi'_f$

Here we define

```
trans ($m\ pt$=e, class, $\pi'_f, \pi_m, \pi'_m, \Theta$)  =
    $m$ = fn rw: RW_all ($\pi'_f$) =>
        (super = {$m_0$ = prom (supervar.$m_0$ rw), ···} ;
         self = {$m'_0$ = prom (supervar.$m_0$ rw), ···} ;
        (oncefn  $pt$ : Arg($\pi'_m$, $m$) => e)$^{\Theta}$)


trans ($m$ ($v\backslash s$) = e, funclass, $\pi'_f, \pi_m, \pi'_m, \Theta$)  =
    $m$ = fn rw: RW_all ($\pi'_f$) =>
        (super = {$m_0$ = prom (supervar.$m_0$ rw), ···} ;
         self = {$m'_0$ = prom (supervar.$m_0$ rw), ···} ;
        (oncefn  $v$ : Arg($\pi'_m, m$) =>
             (val $s$ = reads ($\pi'_f$); val ($v'\backslash s'$) = $e$;
             writes ($\pi'_f, s'$); $v'$))$^{\Theta}$)
```

$$\begin{array}{l}\text{trans}\ (m\ \texttt{(}v\backslash s\texttt{)}\ \texttt{=}\,\texttt{e},\ \texttt{linclass}, \pi'_f, \pi_m, \pi'_m, \Theta)\ = \\ \quad m\ \texttt{= (super = supervar ; self = selfvar ;} \\ \qquad \texttt{(fn (}v\backslash s\texttt{) :}\ \{\,\texttt{value:}\text{Arg}(\pi'_m, m),\ \texttt{state:}\pi'_f\,\}\ \texttt{=> e)}^{\Theta}\texttt{)}\end{array}$$

where we write

$$\begin{array}{rcl}\pi_m & = & \{m_0\texttt{:}\tau_0\texttt{,}\ \cdots\} \\ \pi'_m & = & \{m'_0\texttt{:}\tau'_0\texttt{,}\ \cdots\} \\ \text{Arg}(\pi'_m, m) & = & \rho \text{ if } m = m'_i \text{ and } \tau'_i = \rho\,\texttt{->}\,\rho'\end{array}$$

and if $\pi'_f = \{k_0\texttt{:}\sigma_0\texttt{,}\ \cdots\}$ then

$$\begin{array}{rcl}\text{reads}\,(\pi'_f) & = & \{k_0\texttt{=rw.}k_0\texttt{.read\{\},}\ \cdots\} \\ \text{writes}\,(\pi'_f, s) & = & \texttt{(rw.}k_0\texttt{.write (}s\texttt{.}k_0\texttt{);}\ \cdots\texttt{)}\end{array}$$

– The remaining forms in Figure 3 are translated in the evident way, *e.g.*:

$$\texttt{(case e of}\ \mathit{csl}\texttt{)}^{\Theta} \quad = \quad \texttt{case e}^{\Theta}\ \texttt{of}\ \mathit{csl}^{\Theta}$$

- The translation on match clauses is defined by

$$\begin{array}{rcl}\texttt{(}x\texttt{:t => e)}^{\Theta} & = & x\ \texttt{: t}^{\Theta}\ \texttt{=> e}^{\Theta} \\ \texttt{(\_:t => e)}^{\Theta} & = & z\texttt{:t}^{\Theta}\ \texttt{=> e}^{\Theta} \\ \texttt{(}\mathit{rp1}\texttt{:t => e)}^{\Theta} & = & \texttt{(}z\texttt{:t => split}\ z\ \texttt{as}\ \mathit{rp1}\ \texttt{in e end)}^{\Theta} \\ \texttt{((}\mathit{pt1}\texttt{):t => e)}^{\Theta} & = & \texttt{(}\mathit{pt1}\texttt{:t => e)}^{\Theta} \\ \texttt{((}x\texttt{):t => e)}^{\Theta} & = & \texttt{(}x\texttt{:t => e)}^{\Theta} \\ \texttt{((}\mathit{rp}\texttt{):t => e)}^{\Theta} & = & \texttt{(}z\texttt{:t => split}\ z\ \texttt{as}\ \mathit{rp}\ \texttt{in e end)}^{\Theta} \\ \texttt{((fold t}'\ \mathit{pt}\texttt{):t => e)}^{\Theta} & = & \texttt{(}z\texttt{:t => (val}\ \mathit{pt}\ \texttt{= unfold}\ z\ \texttt{; e))}^{\Theta} \\ \texttt{(fold t}\ \mathit{pt}\ \texttt{=> e)}^{\Theta} & = & \texttt{((fold t}\ \mathit{pt}\texttt{):t => e)}^{\Theta}\end{array}$$

- The translation on case clauses is defined by

$$\begin{array}{rcl}\texttt{(}k\ x\ \texttt{=> e)}^{\Theta} & = & k\ x\ \texttt{=> e}^{\Theta} \\ \texttt{(}k\ \mathit{pt1}\ \texttt{=>}\ e\texttt{)}^{\Theta} & = & k\ z\ \texttt{=> (val}\ \mathit{pt1}\ \texttt{=}\ z\ \texttt{; e))}^{\Theta} \\ \texttt{(}k\ \texttt{=> e)}^{\Theta} & = & \texttt{(}k\ \{\,\}\ \texttt{=> e)}^{\Theta}\end{array}$$

4 Static semantics

4.1 Environments and judgement forms

We now introduce some notation and conventions which will be employed in our static semantics for expressions.

A *reftype environment* Υ is a finite (ordered) list of the form

$$\bullet\, t_0 \mapsto \boldsymbol{\tau}_0 \ \cdots \ \bullet\, t_{n-1} \mapsto \boldsymbol{\tau}_{n-1}$$

where for each $i < n$, $\boldsymbol{\tau}_i$ is an underlying type in the status environment

$$\Delta_i \;=\; \bullet\; t_0 \propto \mathsf{ground} \ \cdots \ \bullet\; t_i \propto \mathsf{ground}$$

(Note that t_i may appear free in $\boldsymbol{\tau}_i$.) We write $\Delta_0(\Upsilon)$ for the status environment denoted by Δ_{n-1} under the above notation conventions. We also write $\Upsilon[t]$ for the underlying type associated with the rightmost appearance of t in Υ (if any).

A (static) *environment* Γ is a finite list of entries each of the form

$$\bullet\, x : \boldsymbol{\tau} \ \ st_1 \ st_2$$

where x is an identifier, $\boldsymbol{\tau}$ an underlying type, st_1 is one of the following *environment first safety tags*:[4]

$$\epsilon \quad \mathsf{safe} \quad \mathsf{rwvar} \quad \mathsf{argsafe}$$

and st_2 is one of the following *environment second safety tags*:

$$\epsilon \quad \mathsf{newsafe} \quad \mathsf{halfnewsafe} \quad \mathsf{weaknewsafe}$$

We write $\Gamma[x]$ (resp. $\Gamma^1[x]$, $\Gamma^2[x]$) for the type (resp. first or second safety tag) associated with the rightmost appearance of x in Γ, if any.

The following more specialized notions concerning environments Γ will also be required at certain points.

- Γ is called *reusable* if $\Gamma[x]$ is reusable for all x appearing in Γ (shadowed entries in Γ are not required to be reusable).
- Γ is called *weakly newsafe* if for all x appearing in Γ, $\Gamma^2[x]$ is either $\mathsf{newsafe}$ or $\mathsf{weaknewsafe}$ or else $\Gamma[x]$ is a ground type.
- Γ is called *rw-free* if $\Gamma^2[x] \neq \mathsf{rwvar}$ for all x in Γ.
- A variable x is called *weakly safe* in Γ if $\Gamma^1[x] \in \{\mathsf{safe}, \mathsf{rwvar}\}$.
- 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 e.
- We write Safe(Γ) for the environment obtained from Γ as follows: all first safety tags in Γ are replaced by safe; all second safety tags ϵ are replaced by $\mathsf{weaknewsafe}$; and all second safety tags $\mathsf{halfnewsafe}$ are replaced by $\mathsf{newsafe}$.

[4]Strictly speaking, a safety tag is either the empty sequence or a sequence consisting of a single keyword.

A well-formed typing judgement has the form

$$\Upsilon, \Gamma \vdash e : \boldsymbol{\tau}\ st_1\ st_2$$

where Υ is a reftype environment as above, Γ is a static environment as above, $\boldsymbol{\tau}$ and all types appearing in Γ are underlying types in $\Delta_0(\Upsilon)$, st_1 is one of the *judgement first safety tags*

$$\epsilon \quad \mathsf{safe} \quad \mathsf{writesafe} \quad \mathsf{argsafe} \quad \mathsf{metasafe}_\epsilon \quad \mathsf{metasafe}_{\mathtt{linear}}$$

and st_2 is one of the *judgement second safety tags*

$$\epsilon \quad \mathsf{newsafe} \quad \mathsf{halfnewsafe}$$

To each judgement first safety tag st_1 we associate an environment first safety tag $\widetilde{st_1}$ as follows: $\widetilde{st_1} = \mathsf{safe}$ if $st_1 = \mathsf{safe}$, and $\widetilde{st_1} = \epsilon$ otherwise.

4.2 Linear variables and compatibility

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. (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.) 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 e and e', the type $\Gamma[x]$ is reusable. In the case $n = 0$ we may simply write $e \sim_\Gamma e'$.

4.3 Rule conventions

A *rule instance* in general has the form

$$\frac{J_0 \quad \cdots \quad J_{r-1}}{J}$$

where the J_i and J are well-formed typing judgements. The purpose of the typing rules given below is to define a set of *valid* rule instances; the set of *derivable* typing judgements is then the smallest set closed under deductions via valid rule instances. However, for readability, our typing rules are presented with the help of several notational conventions, which we now explain.

The following steps, in order, are notionally applied to each of our typing rules in order to generate a set of valid instances:

1. First, any derived syntactic forms for types and expressions that feature in the rules (such as $\sigma \,\texttt{->}\, \tau$ and $e\,e'$) are expanded in accordance with the translation rules of Sections 2.2 and 3.2. In addition, the abbreviation $\langle st \rangle$ in rules 24, 26 and 41 is expanded to $st_1\ st_2$.

2. Next, for all rules in which reftype environments play no active role (that is, all rules except 44, 45 and 46), each judgement form appearing in the rule is prefixed by "Υ,". (Note that reftype environments are not required at all in order to give a static semantics for core language expressions.)

3. There are four special conventions that apply selectively to certain typing rules, known as the *safety, writesafety, argsafety* and *newsafety* conventions. Rules that are subject to these conventions are identified by the letters $\mathbf{S}, \mathbf{W}, \mathbf{A}, \mathbf{N}$ respectively.

   For each rule subject to the safety [resp. writesafety, argsafety, newsafety] convention, an auxiliary rule is derived by adding the tag safe [resp. writesafe, argsafe, newsafe] to the right hand side each judgement which is displayed in the rule without a non-empty safety tag. (The first premises of rules 24 and 26, which carry *variable* safety tags, require more specialized treatment. For the writesafety and newsafety conventions, we include in the auxiliary rule both the original judgement and the corresponding judgement with $st_1\ st_2$ replaced by writesafe ϵ, whilst for the safety and argsafety conventions we include only the original judgement featuring $st_1\ st_2$.)

   In each case, it is to be understood that both the original rule and the auxiliary rules derived via any applicable conventions each give rise to a set of valid rule instances in accordance with step 4 below.

4. A rule now serves as a template for valid rule instances as follows. By a *valuation* ν for a rule we mean an assignment of (mathematical) values $\nu(X)$ to the metavariables X appearing in the rule such that

   - ν is compatible with the declared range of each metavariable (for example, $\nu(x)$ is indeed an identifier, and $\nu(\tau)$ is an underlying type expression);
   - any side-conditions appearing in the rule are satisfied under the assignment ν;
   - for each judgement form appearing in the rule, the result of instantiating each metavariable according to ν and then replacing each (outermost) underlying type expression by the corresponding underlying type is a well-formed typing judgement as defined in Section 4.1).

   Thus, each valuation for a rule generates a rule instance, and we define the valid instances of the rule to be precisely the instances so generated.

We now proceed to the typing rules themselves.

4.4 Basic typing rules

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

$$\frac{}{\Gamma \vdash x : \tau} \quad \tau \approx \Gamma[x] \tag{20}$$

$$\mathbf{SWN} \quad \frac{}{\Gamma \vdash \mathit{lit} : \tau} \quad \tau \approx \mathit{ty}(\mathit{lit}) \tag{21}$$

$$\textbf{SWAN}\ \frac{\Gamma \vdash e_0 : \tau_0 \qquad \cdots \qquad \Gamma \vdash e_{n-1} : \tau_{n-1}}{\Gamma \vdash \{\, k_0\texttt{=}e_0\,,\ldots,k_{n-1}\texttt{=}e_{n-1}\,\} \;:\; \{\, k_0 : \tau_0\,,\ldots,k_{n-1} : \tau_{n-1}\,\}} \quad \begin{array}{l}\forall\, i \neq j.\\ e_i \sim_\Gamma e_j\end{array} \tag{22}$$

$$\textbf{SWAN}\ \frac{\Gamma \vdash e : \pi \qquad \Gamma \vdash e' : \pi'}{\Gamma \vdash e \;\texttt{+>}\; e' : \pi \oplus \pi'} \quad \pi' \text{ compatible with } \pi \tag{23}$$

$$\textbf{SWAN}\ \frac{\begin{array}{c}\Gamma \vdash e : \{\, k_0 : \tau_0\,,\ldots,k_{n-1} : \tau_{n-1}\,\} + \tau'\ \langle st\rangle\\ \Gamma \bullet x_0 : \tau_0\ \langle st\rangle \ldots \bullet x_{n-1} : \tau_{n-1}\ \langle st\rangle \bullet y : \tau'\ \langle st\rangle \vdash e' : \tau\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{split}\ e\ \texttt{as}\ \{\, k_0\texttt{=}x_0\,,\ldots,k_{n-1}\texttt{=}x_{n-1}\,\}\ \texttt{+}\, y\\ \texttt{in}\ e'\ \texttt{end}\ : \tau\end{array}} \quad \ldots \tag{24}$$

$$\ldots \quad \begin{array}{l} e \sim_\Gamma^{x_0,\ldots,x_{n-1}} e'\\ x_0,\ldots,x_{n-1}, y \text{ distinct}\\ st_1 \in \{\epsilon, \mathsf{safe}, \mathsf{argsafe}\}\\ st_2 \in \{\epsilon, \mathsf{newsafe}, \mathsf{halfnewsafe}\}\end{array}$$

$$\textbf{SWN}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{tag}\ k\ e : \texttt{[|}\ k : \tau\ \texttt{|]}} \tag{25}$$

$$\textbf{SWN}\ \frac{\begin{array}{c}\Gamma \vdash e : \texttt{[|}\ k_0 : \tau_0\,,\ldots,k_{n-1} : \tau_{n-1}\ \texttt{|]}\ \langle st\rangle\\ \Gamma \bullet x_0 : \tau_0\ \langle st\rangle \vdash e_0 : \tau'\\ \cdots\\ \Gamma \bullet x_{n-1} : \tau_{n-1}\ \langle st\rangle \vdash e_{n-1} : \tau'\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ k_0 x_0 \texttt{=>} e_0\ \texttt{|}\ \cdots\\ \texttt{|}\ k_{n-1} x_{n-1} \texttt{=>} e_{n-1}\ \texttt{end}\ : \tau'\end{array}} \quad \begin{array}{l}\forall i.\ e \sim_\Gamma^{x_i} e_i\\ st_1 \in \{\epsilon, \mathsf{safe}\}\\ st_2 \in \{\epsilon, \mathsf{newsafe}\}\end{array} \tag{26}$$

$$\textbf{SWN}\ \frac{\Gamma_v \vdash v : \tau}{\Gamma \vdash \texttt{prom}\ v : \texttt{!}\tau} \quad \Gamma_v \text{ reusable} \tag{27}$$

$$\textbf{SWN}\ \frac{\Gamma \vdash e : \texttt{!}\tau}{\Gamma \vdash \texttt{der}\ e : \tau} \tag{28}$$

$$\textbf{SWN}\ \frac{\Gamma \vdash e : \texttt{!(unit -o}\ \tau\texttt{)}}{\Gamma \vdash \texttt{force}\ e : \texttt{!}\tau} \tag{29}$$

$$\textbf{SWN}\ \frac{\Gamma \vdash e : \tau''}{\Gamma \vdash \texttt{fold}\ \tau\ e : \tau} \quad \begin{array}{l}\tau \approx \texttt{rectype}\ t\ \texttt{=>}\ \tau'\\ \tau'' <: \tau'[\tau/t]\end{array} \tag{30}$$

$$\textbf{SWN}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{unfold}\ e : \tau''} \quad \begin{array}{l}\tau \approx \texttt{rectype}\ t\ \texttt{=>}\ \tau'\\ \tau'' \approx \tau'[\tau/t]\end{array} \tag{31}$$

$$\frac{\Gamma \bullet x : \sigma \vdash e : \tau}{\Gamma \vdash \texttt{oncefn}\ x\texttt{:}\sigma\texttt{=>}e\ : \sigma \texttt{-o}\, \tau} \tag{32}$$

$$\textbf{SWN}\quad \frac{\Gamma \vdash e : \sigma \texttt{-o}\, \tau \qquad \Gamma \vdash e' : \sigma}{\Gamma \vdash e \,\texttt{\$}\, e' : \tau} \quad e \sim_\Gamma e' \tag{33}$$

$$\textbf{SWN}\quad \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash e \texttt{ = } e' : \texttt{bool}} \quad \begin{array}{l} e \sim_\Gamma e' \\ \tau \text{ ground} \end{array} \tag{34}$$

$$\textbf{SWN}\quad \frac{\Gamma \vdash e : \texttt{int} \qquad \Gamma \vdash e' : \texttt{int}}{\Gamma \vdash e\, \imath\, e' : \tau} \quad \tau \approx ty(\imath) \tag{35}$$

$$\textbf{SWN}\quad \frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash e_0 : \tau \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ e_0\ \texttt{else}\ e_1\ : \tau} \quad e \sim_\Gamma e_0,\ e \sim_\Gamma e_1 \tag{36}$$

$$\frac{\Gamma \bullet x : \tau\ st_1{}'\ st_2 \vdash e : \tau\ st_1\ st_2}{\Gamma \vdash \texttt{rec}\ x\texttt{:}\tau\texttt{=>}e\ : \tau\ st_1\ st_2} \quad \begin{array}{l} \tau \text{ groundless} \\ e' = \texttt{rec}\ x\texttt{:}\tau\texttt{=>}e \\ \Gamma_{e'} \text{ reusable} \\ st_1 \in \{\epsilon, \mathsf{safe}, \mathsf{writesafe}\} \\ st_2 \in \{\epsilon, \mathsf{newsafe}\} \\ st_1{}' = \widetilde{st_1} \end{array} \tag{37}$$

$$\frac{\Gamma \bullet x : \sigma\ st_1{}'\ st_2 \vdash e : \mathsf{VR}\,(\tau_0, \tau_1)\ st_1\ st_2}{\Gamma \vdash \texttt{catchcont}\ x\texttt{:}\sigma\texttt{=>}e : \mathsf{Catchcont}\,(\zeta, \xi, \tau_0, \tau_1)\ st_1\ st_2} \quad \begin{array}{l} \zeta, \tau_0 \text{ ground} \\ x \notin \operatorname{dom} \Gamma \\ \sigma \approx \zeta \texttt{->} \xi \\ st_1 \in \{\epsilon, \mathsf{safe}, \mathsf{writesafe}\} \\ st_2 \in \{\epsilon, \mathsf{newsafe}\} \\ st_1{}' = \widetilde{st_1} \end{array} \tag{38}$$

$$\textbf{SWAN}\quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e \texttt{ :<} \tau'\ : \tau'} \quad \tau <: \tau' \tag{39}$$

$$\textbf{SWN}\quad \frac{}{\Gamma \vdash lo\ \texttt{root} : lo\ \texttt{classimpl unit,unit,unit end}} \tag{40}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_0 : lo\ \texttt{classimpl}\ \pi_f \texttt{,} \pi_m \texttt{,} \tau_k\ \texttt{end}\ \langle st \rangle \\ \Gamma \vdash v_1 : \tau_{super} \texttt{->} \tau_{self} \texttt{->} \tau_{self}\ \langle st \rangle \\ \mathrm{Safe}(\Gamma) \vdash v_1 : \tau_{super} \texttt{->} \tau_{self} \texttt{->} \tau_{self}\ \ \mathsf{metasafe}_{lo}\ st_2 \\ \Gamma \vdash v_2 : (\tau_k \texttt{->} \pi_f) \texttt{->} \tau'_k \texttt{->} \pi'_f\ \langle st \rangle \end{array}}{\Gamma \vdash lo\ \texttt{extend}\ e_0\ \texttt{with}\ v_1 \texttt{,} v_2\ \texttt{end}\ : lo\ \texttt{classimpl}\ \pi'_f \texttt{,} \pi'_m \texttt{,} \tau'_k\ \texttt{end}\ \langle st \rangle} \quad \ldots \tag{41}$$

$$\ldots \quad \begin{array}{l} e_0 \sim_\Gamma v_1,\ e_0 \sim_\Gamma v_2,\ v_1 \sim_\Gamma v_2 \\ \pi'_f \text{ extends } \pi_f,\ \pi'_m \text{ extends } \pi_m,\ \pi'_m \not\approx \{\} \\ \tau_{super} \approx \pi_m \natural_{lo} \pi'_f,\ \tau_{self} \approx \pi'_m \natural_{lo} \pi'_f \\ st_1 \in \{\epsilon, \mathsf{safe}, \mathsf{writesafe}\},\ st_2 \in \{\epsilon, \mathsf{newsafe}\} \end{array}$$

$$\textbf{SW} \quad \frac{\Gamma \vdash e : lo\ \texttt{classimpl}\ \pi_f\,\texttt{,}\pi_m\,\texttt{,}\tau_k\ \texttt{end}}{\Gamma \vdash \texttt{new}\ e : \tau_k\ \texttt{->}\ \pi_m} \quad lo = \epsilon \text{ or } \tau_k \text{ ground} \tag{42}$$

$$\textbf{SW} \quad \frac{\Gamma \vdash e : \texttt{linear classimpl}\ \pi_f\,\texttt{,}\pi_m\,\texttt{,}\tau_k\ \texttt{end} \qquad \Gamma \vdash e' : \tau_k}{\Gamma \vdash \texttt{make}\ e\ e' : \pi_m} \tag{43}$$

$$\textbf{SWAN} \quad \frac{\Upsilon \bullet t \mapsto \tau,\, \Gamma \vdash e : \tau'}{\Upsilon, \Gamma \vdash \texttt{reftype}\ t\ \texttt{for}\ \tau\ \texttt{in}\ e\ \texttt{end}\ : \tau'} \quad \begin{array}{l} t \notin \mathrm{FTN}(\tau') \\ \tau' \text{ first-order in } \Upsilon \end{array} \tag{44}$$

$$\textbf{SWN} \quad \frac{\Upsilon, \Gamma \vdash e : \tau}{\Upsilon, \Gamma \vdash \texttt{ref}\ t\ e\ : t} \quad \tau \approx \Upsilon[t] \tag{45}$$

$$\textbf{SWN} \quad \frac{\Upsilon, \Gamma \vdash e : t}{\Upsilon, \Gamma \vdash \texttt{deref}\ e\ : \tau} \quad \tau \approx \Upsilon[t] \tag{46}$$

4.5 Safety rules for ordinary classes

The following additional rules are associated with judgements $\Gamma \vdash e : \tau$ safe:

$$\frac{}{\Gamma \vdash x : \tau\ \mathsf{safe}} \quad \begin{array}{l} \tau \approx \Gamma[x] \\ x \text{ weakly safe in } \Gamma \end{array} \tag{47}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau\ \mathsf{safe}} \quad \tau \text{ ground} \tag{48}$$

$$\frac{\Gamma \bullet x : \sigma\ \mathsf{safe} \vdash e : \tau\ \mathsf{safe}}{\Gamma \vdash \texttt{oncefn}\ x\texttt{:}\sigma\texttt{=>}e\ : \sigma\ \texttt{-o}\ \tau\ \mathsf{safe}} \quad \begin{array}{c} \text{all } y \in \mathrm{FV}(e) - \{x\} \\ \text{weakly safe in } \Gamma \end{array} \tag{49}$$

Additional rules for judgements $\Gamma \vdash e : \tau$ writesafe:

$$\frac{}{\Gamma \vdash x : \tau\ \mathsf{writesafe}} \quad \begin{array}{l} \tau \approx \Gamma[x] \\ \Gamma^1[x] \neq \mathsf{rwvar} \end{array} \tag{50}$$

$$\frac{}{\Gamma \vdash x.k.\texttt{read} : \mathsf{Read}(\sigma)\ \mathsf{writesafe}} \quad \begin{array}{l} \Gamma^1[x] = \mathsf{rwvar} \\ \Gamma[x] \text{ extends } \{k{:}\mathsf{RW}\,(\sigma)\} \end{array} \tag{51}$$

$$\frac{\Gamma \vdash e : \sigma\ \mathsf{safe}\ \mathsf{newsafe}}{\Gamma \vdash x.k.\texttt{write}\ e : \texttt{unit}\ \mathsf{writesafe}} \quad \begin{array}{l} \Gamma^1[x] = \mathsf{rwvar} \\ \Gamma[x] \text{ extends } \{k{:}\mathsf{RW}\,(\sigma)\} \\ \Gamma_e \text{ rw-free} \end{array} \tag{52}$$

$$\frac{\Gamma \vdash v : \sigma\ \mathsf{safe}}{\Gamma \vdash x.k.\texttt{write}\ v : \texttt{unit}\ \mathsf{writesafe}} \quad \begin{array}{l} \Gamma^1[x] = \mathsf{rwvar} \\ \Gamma[x] \text{ extends } \{k{:}\mathsf{RW}\,(\sigma)\} \\ \Gamma_v \text{ rw-free} \\ \Gamma_v \text{ weakly newsafe} \end{array} \tag{53}$$

$$\frac{\Gamma \bullet x : \sigma \vdash e : \tau\ \mathsf{writesafe}}{\Gamma \vdash \texttt{oncefn}\ x\texttt{:}\sigma\texttt{=>}e\ : \sigma\,\texttt{-o}\,\tau\ \mathsf{writesafe}} \tag{54}$$

Additional rules for judgements $\Gamma \vdash e : \tau$ newsafe and $\Gamma \vdash e : \tau$ halfnewsafe:

$$\frac{}{\Gamma \vdash x : \tau\ \mathsf{newsafe}} \quad \begin{array}{l} \tau \approx \Gamma[x] \\ \Gamma^2[x] = \mathsf{newsafe} \end{array} \tag{55}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{oncefn}\ x\texttt{:}\sigma\texttt{=>}e\ : \sigma\,\texttt{-o}\,\tau \\ \Gamma \bullet x : \sigma\ \mathsf{newsafe} \vdash e : \tau\ \mathsf{newsafe}\end{array}}{\Gamma \vdash \texttt{oncefn}\ x\texttt{:}\sigma\texttt{=>}e\ : \sigma\,\texttt{-o}\,\tau\ \mathsf{newsafe}} \tag{56}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{rec}\ x\texttt{:}\tau\texttt{=>}e\ : \tau \\ \Gamma \bullet x : \tau\ \mathsf{newsafe} \vdash e : \tau\ \mathsf{newsafe}\end{array}}{\Gamma \vdash \texttt{rec}\ x\texttt{:}\tau\texttt{=>}e\ : \tau\ \mathsf{newsafe}} \tag{57}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{catchcont}\ x\texttt{:}\sigma\texttt{=>}e : \mathsf{Catchcont}\,(\zeta, \xi, \tau_0, \tau_1) \\ \Gamma \bullet x : \tau\ \mathsf{newsafe} \vdash e : \mathsf{VR}\,(\tau_0, \tau_1)\ \mathsf{newsafe}\end{array}}{\Gamma \vdash \texttt{catchcont}\ x\texttt{:}\sigma\texttt{=>}e : \mathsf{Catchcont}\,(\zeta, \xi, \tau_0, \tau_1)\ \mathsf{newsafe}} \tag{58}$$

$$\frac{\Gamma \vdash e : \mathit{lo}\ \texttt{classimpl}\ \pi_f\,\texttt{,}\pi_m\,\texttt{,}\tau_k\ \texttt{end}\ \ \mathsf{newsafe}}{\Gamma \vdash \texttt{new}\ e : \tau_k\,\texttt{->}\,\pi_m\ \mathsf{halfnewsafe}} \quad \mathit{lo} = \epsilon \text{ or } \tau_k \text{ ground} \tag{59}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \texttt{linear classimpl}\ \pi_f\,\texttt{,}\pi_m\,\texttt{,}\tau_k\ \texttt{end}\ \ \mathsf{newsafe} \\ \Gamma \vdash e' : \tau_k\ \mathsf{newsafe}\end{array}}{\Gamma \vdash \texttt{make}\ e\ e' : \pi_m\ \mathsf{halfnewsafe}} \tag{60}$$

$$\frac{\Gamma \vdash e_0 : \tau_0 \ \mathsf{halfnewsafe} \quad \cdots \quad \Gamma \vdash e_{n-1} : \tau_{n-1} \ \mathsf{halfnewsafe}}{\Gamma \vdash \{\, k_0\texttt{=}e_0\,,\ldots,k_{n-1}\texttt{=}e_{n-1}\,\} \ : \ \{\, k_0 : \tau_0\,,\ldots,k_{n-1} : \tau_{n-1}\,\} \ \mathsf{halfnewsafe}} \quad \begin{array}{l} \forall\, i \neq j. \\ e_i \sim_\Gamma e_j \end{array} \tag{61}$$

$$\frac{\Gamma \vdash e : \tau \ st_1\,\epsilon \qquad \Gamma \vdash e : \tau \ \epsilon\, st_2}{\Gamma \vdash e : \tau \ st_1\, st_2} \tag{62}$$

Additional rules for judgements $\Gamma \vdash e : \tau$ $\mathsf{argsafe}$:

$$\frac{}{\Gamma \vdash x : \tau \ \mathsf{argsafe}} \quad \begin{array}{l} \tau \approx \Gamma[x] \\ \Gamma^1[x] = \mathsf{argsafe} \end{array} \tag{63}$$

$$\frac{\Gamma \ \bullet x \ : \mathsf{RW\_all}\,(\pi) \ \mathsf{rwvar} \vdash e : \tau \ \mathsf{writesafe}}{\Gamma \vdash \texttt{fn}\ x : \mathsf{RW\_all}\,(\pi) \ \texttt{=>}\ e \ : \mathsf{RW\_all}\,(\pi) \ \texttt{->}\ \tau \ \mathsf{argsafe}} \tag{64}$$

$$\frac{\begin{array}{c} \Gamma \vdash e : \{\, k_0 : \tau_0\,,\ldots,k_{n-1} : \tau_{n-1}\,\} \ + \tau' \ \langle st\rangle \\ \mathrm{Safe}(\Gamma \bullet x_0 : \tau_0 \ \langle st\rangle \ldots \bullet x_{n-1} : \tau_{n-1} \ \langle st\rangle \bullet y : \tau' \ \langle st\rangle) \vdash e' : \tau \ \mathsf{argsafe} \end{array}}{\begin{array}{c} \Gamma \vdash \texttt{split}\ e\ \texttt{as}\ \{\, k_0\texttt{=}x_0\,,\ldots,k_{n-1}\texttt{=}x_{n-1}\,\} \ \texttt{+}\, y \\ \texttt{in}\ e'\ \texttt{end} \ : \tau \ \mathsf{argsafe} \end{array}} \ \cdots \tag{65}$$

$$\cdots \quad \begin{array}{l} e \sim_\Gamma^{x_0,\ldots,x_{n-1}} e' \\ x_0,\ldots,x_{n-1}, y \text{ distinct} \end{array}$$

$$\frac{\Gamma \vdash e : \mathsf{RW}\,(\sigma) \ \texttt{->}\, \tau \ \mathsf{argsafe} \qquad \Gamma \vdash e' : \mathsf{RW}\,(\sigma) \ \mathsf{safe}}{\Gamma \vdash e\, e' : \tau \ \mathsf{writesafe}} \tag{66}$$

Rule for judgements $\Gamma \vdash e : \tau$ $\mathsf{metasafe}_\epsilon$:

$$\frac{\begin{array}{c} \Gamma \bullet \mathit{super} : \tau_{\mathit{super}} \ \mathsf{argsafe\ newsafe} \bullet \mathit{self} : \tau_{\mathit{self}} \ \mathsf{argsafe\ newsafe} \vdash \\ e : \tau_{\mathit{self}} \ \mathsf{argsafe} \end{array}}{\begin{array}{c} \Gamma \vdash \texttt{fn}\ \mathit{super} : \tau_{\mathit{super}} \ \texttt{=>}\ \texttt{fn}\ \mathit{self} : \tau_{\mathit{self}} \ \texttt{=>} e \\ : \tau_{\mathit{super}} \texttt{->} \tau_{\mathit{self}} \texttt{->} \tau_{\mathit{self}} \ \mathsf{metasafe}_\epsilon \end{array}} \quad \begin{array}{l} \tau_{\mathit{super}} \approx \pi \natural_\epsilon \pi'' \\ \tau_{\mathit{self}} \approx \pi' \natural_\epsilon \pi'' \\ \Gamma_e \text{ reusable} \end{array} \tag{67}$$

4.6 Safety rules for linear classes

For method implementations in linear classes, "safety" means that nested invocations of *super* and *self* are prohibited. This is accomplished by the following rule:

$$\frac{\begin{array}{c}\Gamma \bullet \mathit{super} : \tau_{\mathit{super}} \bullet \mathit{self} : \tau_{\mathit{self}} \vdash e : \tau_{\mathit{self}} \\ \Lambda \vdash e \propto 1\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{fn}\ \mathit{super} : \tau_{\mathit{super}}\ \texttt{=>}\ \texttt{fn}\ \mathit{self} : \tau_{\mathit{self}}\ \texttt{=>} e \\ : \tau_{\mathit{super}}\ \texttt{->}\ \tau_{\mathit{self}}\ \texttt{->}\ \tau_{\mathit{self}}\ \mathsf{metasafe}_{\texttt{linear}}\end{array}} \quad \begin{array}{l}\tau_{\mathit{super}} \approx \pi \natural_{\texttt{linear}} \pi'' \\ \tau_{\mathit{self}} \approx \pi' \natural_{\texttt{linear}} \pi'' \\ \Gamma_e \text{ reusable} \\ \Lambda = \Lambda_0(\Gamma), \mathit{super} : 1, \mathit{self} : 1\end{array} \tag{68}$$

Here we use a judgement form $\Lambda \vdash e \propto r$ asserting that an expression e is of *abstraction rank* r relative to a *rank environment* $\Lambda = x_0 \propto r_0, \ldots, x_{n-1} \propto r_{n-1}$, where the r_i are natural numbers. Informally, this expresses the idea that e will be rendered "safe" by applying it to r arguments (as in $e\, e_0 \ldots e_{r-1}$), assuming that each x_i may be rendered safe by applying it to r_i arguments. If Γ is an environment whose variables (in order) are $x_0, \ldots, x_{n-1}$, then $\Lambda_0(\Gamma)$ denotes the rank environment $x_0 \propto 0, \ldots, x_{n-1} \propto 0$.

Abstraction rank judgements are generated by the following rules. We write $\Lambda[x]$ for the number associated with the rightmost occurrence of x in Λ (if any), and say e is Λ-*sensitive* if some x with $\Lambda[x] > 0$ occurs free in e. We also write $\smile$ for subtraction truncated at zero.

$$\frac{}{\Lambda \vdash x \propto r} \quad r \geq \Lambda[x] \tag{69}$$

$$\frac{}{\Lambda \vdash \mathit{lit} \propto r} \tag{70}$$

$$\frac{\Lambda \vdash e_0 \propto r \qquad \cdots \qquad \Lambda \vdash e_{n-1} \propto r}{\Lambda \vdash \{k_0\texttt{=}e_0\texttt{,}\ \ldots\ \texttt{,}k_{n-1}\texttt{=}e_{n-1}\} \propto r} \tag{71}$$

$$\frac{\Lambda \vdash e \propto r \qquad \Lambda \vdash e' \propto r}{\Lambda \vdash e\ \texttt{+>}\ e' \propto r} \tag{72}$$

$$\frac{\Lambda \vdash e \propto r \qquad \Lambda, x_0 \propto r, \ldots, x_{n-1} \propto r, y \propto r \vdash e' \propto r'}{\Lambda \vdash \texttt{split}\ e\ \texttt{as}\ \{k_0\texttt{=}x_0\texttt{,}\ \ldots\ \texttt{,}k_{n-1}\texttt{=}x_{n-1}\}\ \texttt{+}\ y\ \texttt{in}\ e'\ \texttt{end}\ \propto r'} \tag{73}$$

$$\frac{\Lambda \vdash e \propto r}{\Lambda \vdash \texttt{tag}\ k\ e \propto r} \tag{74}$$

$$\frac{\Lambda \vdash e \propto r \qquad \Lambda, x_0 \propto r \vdash e_0 \propto r' \qquad \cdots \qquad \Lambda, x_{n-1} \propto r \vdash e_{n-1} \propto r'}{\Lambda \vdash \texttt{case}\ e\ \texttt{of}\ k_0 x_0\texttt{=>}e_0\ \texttt{|}\ \ldots\ \texttt{|}\ k_{n-1} x_{n-1}\texttt{=>}e_{n-1}\ \texttt{end}\ \propto r'} \tag{75}$$

$$\frac{\Lambda \vdash v \propto r}{\Lambda \vdash \texttt{prom}\ v \propto r} \tag{76}$$

$$\frac{\Lambda \vdash e \propto r}{\Lambda \vdash \texttt{der}\ e \propto r} \tag{77}$$

$$\frac{\Lambda \vdash e \propto r}{\Lambda \vdash \texttt{force}\ e \propto r \smile 1} \tag{78}$$

$$\frac{\Lambda \vdash e \propto r}{\Lambda \vdash \texttt{fold}\ \tau\ e \propto r} \tag{79}$$

$$\frac{\Lambda \vdash e \propto r}{\Lambda \vdash \texttt{unfold}\ e \propto r} \tag{80}$$

$$\frac{\Lambda, x \propto 0 \vdash e \propto r}{\Lambda \vdash \texttt{oncefn}\ x : \sigma \texttt{=>} e \propto r + 1} \tag{81}$$

$$\frac{}{\Lambda \vdash \texttt{oncefn}\ x : \sigma \texttt{=>} e \propto 0}\ \ \texttt{oncefn}\ x : \sigma \texttt{=>} e \text{ not } \Lambda\text{-sensitive} \tag{82}$$

$$\frac{\Lambda \vdash e \propto r \qquad \Lambda \vdash e' \propto 0}{\Lambda \vdash e\,\texttt{\$}\,e' \propto r \smile 1} \tag{83}$$

$$\frac{\Lambda \vdash e \propto r \qquad \Lambda \vdash e' \propto r}{\Lambda \vdash e \,\texttt{=}\, e' \propto r} \tag{84}$$

$$\frac{\Lambda \vdash e \propto r \qquad \Lambda \vdash e' \propto r}{\Lambda \vdash e \,\imath\, e' \propto r} \tag{85}$$

$$\frac{\Lambda \vdash e \propto r \qquad \Lambda \vdash e_1 \propto r \qquad \Lambda \vdash e_2 \propto r}{\Lambda \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \propto r} \tag{86}$$

$$\frac{\Lambda, x \propto r \vdash e \propto r}{\Lambda \vdash \texttt{rec}\ x : \tau \texttt{=>} e \propto r} \tag{87}$$

$$\frac{\Lambda, x \propto 0 \vdash e \propto r}{\Lambda \vdash \texttt{catchcont}\ x : \sigma \texttt{=>} e \propto r} \tag{88}$$

$$\frac{\Lambda \vdash e \propto r}{\Lambda \vdash e \,\texttt{:<}\, \tau \propto r} \tag{89}$$

$$\frac{}{\Lambda \vdash lo\ \texttt{root} \propto r} \tag{90}$$

$$\frac{}{\Lambda \vdash lo\ \texttt{extend}\ e_0\ \texttt{with}\ v_1, v_2\ \texttt{end}\ \propto r}\ e_0, v_1, v_2 \text{ not } \Lambda\text{-sensitive} \tag{91}$$

$$\frac{}{\Lambda \vdash \texttt{new}\ e \propto r}\ e \text{ not } \Lambda\text{-sensitive} \tag{92}$$

$$\frac{\Lambda \vdash e' \propto 0}{\Lambda \vdash \texttt{make}\ e\ e' \propto r}\ e \text{ not } \Lambda\text{-sensitive} \tag{93}$$

$$\frac{}{\Lambda \vdash \texttt{ref}\ t\ e \propto r} \tag{94}$$

5 Runtime expressions and evaluation contexts

5.1 Runtime expressions

In order to describe the dynamic semantics of Lingay, it is convenient to work with a language of *runtime expressions* which differs slightly from the language of expressions used above. On the one hand, the language of runtime expressions includes various auxiliary constructs which do not appear in source programs, but which arise at intermediate stages during evaluation. On the other hand, runtime expressions feature almost no explicit type information, reflecting the fact that the process of evaluation may be presented quite independently of the type system.

The language of runtime expressions involves the following new lexical categories and other finitary structures:

- A category of *locations*, ranged over by ℓ, playing the role of references to heap objects.
- A category of *temporary locations*, ranged over by $\jmath$, which play a technical role in the semantics of linear classes.
- A category of *proper references*, ranged over by ρ, used in the semantics of the `reftype` system.
- A category of *test variables*, ranged over by $\hat{x}$, used in the semantics of `catchcont`.
- A category of *force tokens*, ranged over by ξ, used in the semantics of `force`.
- Finite sets of identifiers, ranged over by F and M, playing the role of sets of field and method names respectively.
- Finite sets of force tokens, ranged over by Ξ.
- Heaps, ranged over by h, to be defined in Section 6.1 (note that the definitions of heaps and runtime expressions are mutually recursive).

Because runtime expressions are manipulated as abstract structures, it is not necessary to specify concretely what the new lexical categories consist of (other than to require that they are infinite), nor to devise a linear textual representation for runtime expressions involving finite sets and heaps (we are content to incorporate these as decorations attaching to the tokens `extend`, `constr`, `alloc`). The grammar for runtime expressions is given in Figure 7; auxiliary expression forms are marked with $\star$.

An occurrence of a variable x within a runtime expression e is *free* if it is not part of a subexpression $\mathtt{oncefn}\ x\mathtt{=>}e'$, $\mathtt{rec}\ x\mathtt{=>}e'$ or $\mathtt{catchcont}\ x\mathtt{=>}e'$. As usual, we write $e[v/x]$ for the result of substituting v for each free occurrence of x in e. (No renaming of bound variables is necessary, since variable capture situations cannot in fact arise in the course of evaluations of well-formed programs.) A runtime expression e is *closed* if it contains no free variables (note that it may contain unbound locations).

Certain runtime expressions are designated as *runtime values*. The grammar for runtime values is given in Figure 8. Throughout this section we use e and v to range over runtime expressions and runtime values respectively. We also write $\mathcal{V}$ for the set of runtime values.

It is useful to note that all runtime expressions that will arise in the course of evaluation of well-typed programs will satisfy the following syntactic constraint: in any subexpression of any of the forms

$$\begin{array}{l}
\mathtt{extend}^{F,M}\ e\ \mathtt{with}\ e_0\mathtt{,}e_1\ \mathtt{end} \\
\mathtt{restore}\ h\ \mathtt{in}\ e_0\ \mathtt{end} \\
\textit{lin-opt}\ \mathtt{constr}^{F}_{\Xi}\mathtt{(}e_0\mathtt{,}e_1\mathtt{)} \\
\textit{lin-opt}\ \mathtt{alloc}^{F}_{\Xi}\mathtt{(}e\mathtt{,}e_0\mathtt{)} \\
\mathtt{temp}\ x\mathtt{=}e_0\ \mathtt{in}\ e\ \mathtt{end} \\
\mathtt{writetemp}\ \mathtt{(}\jmath\mathtt{,}e_0\mathtt{)}
\end{array}$$

the subexpressions e_0, e_1 are always runtime values. This observation significantly simplifies the definition of evaluation contexts, to be presented in Section 5.3.

If $v = \{k_0\mathtt{=}v_0\mathtt{,}\ \ldots\mathtt{,}k_{n-1}\mathtt{=}v_{n-1}\}$ and $v' = \{k'_0\mathtt{=}v'_0\mathtt{,}\ \ldots\mathtt{,}k'_{n'-1}\mathtt{=}v'_{n'-1}\}$, we define $v \oslash v'$ to be the value

$$\{k_{i_0}\mathtt{=}v_{i_0}\mathtt{,}\ \cdots\mathtt{,}k_{i_{r-1}}\mathtt{=}v_{i_{r-1}}\mathtt{,}k'_0\mathtt{=}v'_0\mathtt{,}\ \cdots\mathtt{,}k'_{n'-1}\mathtt{=}v'_{n'-1}\}$$

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

For the purpose of defining the semantics of equality testing in the presence of typecasts, we define a function $\mathsf{strip}\colon \mathcal{V} \to \mathcal{V} \sqcup \{*\}$, which intuitively maps a value to its ground type content (if any):

$$\begin{array}{rcl}
\mathsf{strip}(\mathit{lit}) & = & \mathit{lit} \\
\mathsf{strip}(\{k_0\mathtt{=}v_0\mathtt{,}\ \cdots\mathtt{,}k_{n-1}\mathtt{=}v_{n-1}\}) & = & \{k_{i_0}\mathtt{=}\mathsf{strip}(v_{i_0})\mathtt{,}\ \cdots\mathtt{,}k_{i_{m-1}}\mathtt{=}\mathsf{strip}(v_{i_{m-1}})\} \\
 & & \quad \text{if } \{i_0, \cdots, i_{m-1}\} = \{i \mid \mathsf{strip}(v_i) \neq *\} \neq \emptyset \\
\mathsf{strip}(\{k_0\mathtt{=}v_0\mathtt{,}\ \cdots\mathtt{,}k_{n-1}\mathtt{=}v_{n-1}\}) & = & * \quad \text{if } \mathsf{strip}(v_i) = * \text{ for all } i \\
\mathsf{strip}(\mathtt{tag}\ k\ v) & = & \mathtt{tag}\ k\ \mathsf{strip}(v) \quad \text{if } \mathsf{strip}(v) \neq * \\
\mathsf{strip}(\mathtt{tag}\ k\ v) & = & * \quad \text{if } \mathsf{strip}(v) = * \\
\mathsf{strip}(\mathtt{prom}\ v) & = & \mathsf{strip}(v) \\
\mathsf{strip}(\mathtt{prom}\ \mathtt{restore}\ h\ \mathtt{in}\ v\ \mathtt{end}) & = & \mathsf{strip}(v) \\
\mathsf{strip}(\mathsf{fold}\ v) & = & \mathsf{fold}\ \mathsf{strip}(v) \quad \text{if } \mathsf{strip}(v) \neq *
\end{array}$$

$$
\begin{array}{llcl}
 & \mathit{expr} & ::= & x \\
\star & & | & \hat{x} \\
 & & | & \mathit{lit} \\
 & & | & \{\ \mathit{Clist}\ (\mathit{comp\text{-}assign})\ \} \\
 & & | & \mathit{expr}\ \texttt{+>}\ \mathit{expr} \\
 & & | & \texttt{split}\ \mathit{expr}\ \texttt{as}\ \mathit{record\text{-}pat}\ \texttt{in}\ \mathit{expr}\ \texttt{end} \\
 & & | & \texttt{tag}\ k\ \mathit{expr} \\
 & & | & \texttt{case}\ \mathit{expr}\ \texttt{of}\ \mathit{Blist}(\mathit{case\text{-}clause})\ \texttt{end} \\
 & & | & \texttt{prom}\ \mathit{expr} \\
 & & | & \texttt{der}\ \mathit{expr} \\
 & & | & \texttt{force}\ \mathit{expr4} \\
\star & & | & \texttt{saveheap}\ \texttt{(}\mathit{expr}\texttt{,}\xi\texttt{)} \\
\star & & | & \texttt{restore}\ h\ \texttt{in}\ \mathit{expr}\ \texttt{end} \\
 & & | & \texttt{fold}\ \mathit{expr} \\
 & & | & \texttt{unfold}\ \mathit{expr} \\
 & & | & \texttt{oncefn}\ x\ \texttt{=>}\ \mathit{expr} \\
\star & & | & \texttt{oncefn}\ \hat{x}\ \texttt{=>}\ \mathit{expr} \\
 & & | & \mathit{expr}\ \texttt{\$}\ \mathit{expr} \\
 & & | & \mathit{expr}\ \texttt{=}\ \mathit{expr} \\
 & & | & \mathit{expr}\ \imath\ \mathit{expr} \\
 & & | & \texttt{if}\ \mathit{expr}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr} \\
 & & | & \texttt{rec}\ x\ \texttt{=>}\ \mathit{expr} \\
 & & | & \texttt{catchcont}\ x\ \texttt{=>}\mathit{expr} \\
 & & | & \mathit{lin\text{-}opt}\ \texttt{root} \\
 & & | & \mathit{lin\text{-}opt}\ \texttt{extend}^{F,M}\ \mathit{expr}\ \texttt{with}\ \mathit{expr}\texttt{,}\mathit{expr}\ \texttt{end} \\
 & & | & \texttt{new}\ \mathit{expr} \\
 & & | & \texttt{make}\ \mathit{expr}\ \mathit{expr} \\
\star & & | & \mathit{lin\text{-}opt}\ \texttt{constr}^{F}_{\Xi}\texttt{(}\mathit{expr}\texttt{,}\mathit{expr}\texttt{)} \\
\star & & | & \mathit{lin\text{-}opt}\ \texttt{alloc}^{F}_{\Xi}\texttt{(}\mathit{expr}\texttt{,}\mathit{expr}\texttt{)} \\
\star & & | & \ell \\
\star & & | & \ell\ \texttt{.}\ k \\
\star & & | & \ell\ \texttt{.}\ \mathit{rw\text{-}op} \\
\star & & | & \texttt{temp}\ x\ \texttt{=}\ \mathit{expr}\ \texttt{in}\ \mathit{expr}\ \texttt{end} \\
\star & & | & \texttt{readtemp}\ \mathit{temploc} \\
\star & & | & \texttt{writetemp}\ \texttt{(}\mathit{temploc}\texttt{,}\mathit{expr}\texttt{)} \\
 & & | & \rho \\
 & & | & \texttt{ref}\ \mathit{expr} \\
 & & | & \texttt{deref}\ \mathit{expr} \\
 & \mathit{comp\text{-}assign} & ::= & k\ \texttt{=}\ \mathit{expr} \\
 & \mathit{record\text{-}pat} & ::= & \{\ \mathit{Clist}\ (\mathit{comp\text{-}bind})\ \}\ \texttt{+}\ x \\
 & \mathit{comp\text{-}bind} & ::= & k\ \texttt{=}\ x \\
 & \mathit{case\text{-}clause} & ::= & k\ x\ \texttt{=>}\ \mathit{expr} \\
\star & \mathit{rw\text{-}op} & ::= & k\ \texttt{.}\ \texttt{read}\ \mid\ k\ \texttt{.}\ \texttt{write}\ \mid\ \texttt{readAll}\ \mid\ \texttt{writeAll} \\
\star & \mathit{temploc} & ::= & x\ \mid\ \jmath
\end{array}
$$

Figure 7: Context-free grammar for runtime expressions

$$
\begin{array}{rcl}
\textit{value} & ::= & \hat{x} \mid \texttt{der}\ \hat{x} \\
& \mid & \textit{lit} \\
& \mid & \{\ \textit{Clist}\ (\textit{comp-value})\ \} \\
& \mid & \texttt{tag}\ k\ \textit{value} \\
& \mid & \texttt{prom}\ \textit{value} \\
& \mid & \texttt{prom restore}\ h\ \texttt{in}\ \textit{value}\ \texttt{end} \\
& \mid & \texttt{fold}\ \textit{value} \\
& \mid & \texttt{oncefn}\ x\texttt{=>}\ \textit{expr} \\
& \mid & \textit{lin-opt}\ \texttt{root} \\
& \mid & \textit{lin-opt}\ \texttt{extend}^{F,M}\ \textit{lin-opt}\ \texttt{root with}\ \textit{value}, \textit{value}\ \texttt{end} \\
& \mid & \textit{lin-opt}\ \texttt{constr}^{F}_{\Xi}\ \texttt{(}\textit{value}, \textit{value}\texttt{)} \\
& \mid & \texttt{der}\ \textit{lin-opt}\ \texttt{constr}^{F}_{\Xi}\ \texttt{(}\textit{value}, \textit{value}\texttt{)} \\
& \mid & \ell \mid \texttt{der}\ \ell \\
& \mid & \ell\ \texttt{.}\ k \mid \texttt{der}\ \ell\ \texttt{.}\ k \\
& \mid & \ell\ \texttt{.}\ \textit{rw-op} \\
& \mid & \rho \\
\textit{comp-value} & ::= & k\ \texttt{=}\ \textit{value}
\end{array}
$$

Figure 8: Context-free grammar for runtime values

$$
\begin{array}{rcll}
\mathsf{strip}(\mathsf{fold}\ v) & = & * & \text{if } \mathsf{strip}(v) = * \\
\mathsf{strip}(\rho) & = & \rho & \\
\mathsf{strip}(v) & = & * & \text{for all other values } v
\end{array}
$$

We also write $\approx$ for the equivalence relation on values generated by permutations of record components; we omit the formal definition.

Finally, for each infix $\imath = \texttt{+}, \texttt{-}, \texttt{<}$, we write $\phi_\imath$ for the evident associated operation on literals: for example, $\phi_{\texttt{+}}(\texttt{3}, \texttt{5}) = \texttt{8}$ and $\phi_{\texttt{<}}(\texttt{3}, \texttt{5}) = \texttt{true}$.

5.2 Translation of well-typed expressions to runtime expressions

By an *expression abbreviation environment* we shall mean a finite partial function Ω mapping identifiers x to closed runtime expressions e. In the presence of an expression abbreviation environment Ω, a well-typed expression e as defined in the preceding sections may be translated into a runtime expression as follows:

- Firstly, each `extend` subexpression is annotated with sets F, M recording the field and method names featuring in the parent class being extended (this is the only type-related information we shall require at runtime). More precisely, if in some typing derivation for e, we have an occurrence of a subexpression

  $$\textit{lin-opt}\ \texttt{extend}\ \textit{expr}_1\ \texttt{with}\ \textit{expr}_2\texttt{,}\textit{expr}_3\ \texttt{end}$$

  in which the subexpression $\textit{expr}_1$ is assigned the type $\textit{lin-opt}\ \texttt{classimpl}\ \pi_f\ \texttt{,}\pi_m\ \texttt{,}\tau_k\ \texttt{end}$, where

$$\pi_f = \{f_0 : \sigma_0\,\texttt{,}\, \ldots\,\texttt{,}\, f_{p-1} : \sigma_{p-1}\}, \qquad \pi_m = \{m_0 : \sigma_0\,\texttt{,}\, \ldots\,\texttt{,}\, m_{n-1} : \sigma_{n-1}\}$$

then the relevant occurrence of `extend` is annotated with the sets

$$F = \{f_0, \ldots, f_{p-1}\}, \qquad M = \{m_0, \ldots, m_{n-1}\}$$

Note that F and M are independent of the typing derivation chosen.

- Secondly, all other explicit type information in e is erased by applying the following rewrite rules (the first of which applies to phrases of category *match*):

$$\begin{aligned} x\texttt{:}\tau\texttt{=>}e &\rightsquigarrow x\texttt{=>}e \\ \texttt{fold}\ \tau\ e &\rightsquigarrow \texttt{fold}\ e \\ e\ \texttt{:<}\ \tau &\rightsquigarrow e \\ \texttt{reftype}\ t\ \texttt{for}\ \tau\ \texttt{in}\ e\ \texttt{end} &\rightsquigarrow e \\ \texttt{ref}\ t\ e &\rightsquigarrow \texttt{ref}\ e \end{aligned}$$

- Thirdly, we substitute closed runtime expressions for any free variables in e as determined by Ω. More formally, if $\mathrm{FV}(e) = \{x_0, \ldots, x_{n-1}\}$ and $\Omega(x_i) = e_i$ for each i, we replace e by the runtime expression $e[e_0/x_0] \cdots [e_{n-1}/x_{n-1}]$.

5.3 Evaluation contexts

An evaluation context is, intuitively, a runtime expression with a single "hole" corresponding to the location of the subterm which would need to be supplied for evaluation to proceed. Evaluation contexts are used in our operational semantics for `catchcont` expressions.

The grammar for evaluation contexts is given in Figure 9. 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 $E_1[\texttt{catchcont}\ x : \tau\texttt{=>}E_2[-]]$ for any E_1, E_2. Finally, a *stuck term* is an expression of the form $E[\texttt{der}\ \hat{x}\ \texttt{\$}\ v]$. We let u range over stuck terms, and w over open values plus stuck terms.

If $v = \{k_0\texttt{=}v_0\,\texttt{,}\, \ldots\,\texttt{,}\, k_{n-1}\texttt{=}v_{n-1}\}$ and $u = \{k'_0\texttt{=}w_0\,\texttt{,}\, \ldots\,\texttt{,}\, k'_{n'-1}\texttt{=}w_{n'-1}\}$, we define $v \oslash u$ to be the stuck term

$$\{k_{i_0}\texttt{=}v_{i_0}\,\texttt{,}\, \cdots\,\texttt{,}\, k_{i_{r-1}}\texttt{=}v_{i_{r-1}}\,\texttt{,}\, k'_0\texttt{=}w_0\,\texttt{,}\, \cdots\,\texttt{,}\, k'_{n'-1}\texttt{=}w_{n'-1}\}$$

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

6 Dynamic semantics

6.1 Heaps and evaluation judgements

A *heap* h is defined to be a partial function defined on some finite set of locations ℓ, temporary locations $\jmath$ and references ρ, such that

$$
\begin{array}{rcl}
\mathit{eval} & ::= & \mathit{eval1} \\
 & | & \texttt{fold}\ \mathit{eval} \\
\mathit{eval1} & ::= & [-] \\
 & | & \{\ \mathit{List}(\mathit{comp\text{-}comma})\ k\texttt{=}\mathit{eval}\ \mathit{List}(\mathit{comma\text{-}comp})\ \} \\
 & | & \mathit{eval}\ \texttt{+>}\ \mathit{expr} \\
 & | & \mathit{value}\ \texttt{+>}\ \mathit{eval} \\
 & | & \texttt{split}\ \mathit{eval}\ \texttt{as}\ \{\ \mathit{Clist}\ (\mathit{comp\text{-}bind})\,\}\ \texttt{in}\ \mathit{expr}\ \texttt{end} \\
 & | & \texttt{tag}\ k\ \mathit{eval} \\
 & | & \texttt{case}\ \mathit{eval}\ \texttt{of}\ \mathit{Blist}(\mathit{case\text{-}clause})\ \texttt{end} \\
 & | & \texttt{der}\ \mathit{eval} \\
 & | & \texttt{force}\ \mathit{eval} \\
 & | & \texttt{saveheap}\ \texttt{(}\mathit{eval},\xi\texttt{)} \\
 & | & \texttt{unfold}\ \mathit{eval1} \\
 & | & \mathit{eval}\ \texttt{\$}\ \mathit{expr} \\
 & | & \mathit{value}\ \texttt{\$}\ \mathit{eval} \\
 & | & \mathit{eval}\ \texttt{=}\ \mathit{expr} \\
 & | & \mathit{value}\ \texttt{=}\ \mathit{eval} \\
 & | & \mathit{eval}\ \imath\ \mathit{expr} \\
 & | & \mathit{value}\ \imath\ \mathit{eval} \\
 & | & \texttt{if}\ \mathit{eval}\ \texttt{then}\ \mathit{expr}\ \texttt{else}\ \mathit{expr} \\
 & | & \texttt{rec}\ x\ \texttt{=>}\ \mathit{eval} \\
 & | & \texttt{catchcont}\ x\ \texttt{=>}\ \mathit{eval} \\
 & | & \mathit{lin\text{-}opt}\ \texttt{extend}\ \mathit{eval}\ \texttt{with}\ \mathit{value}, \mathit{value}\ \texttt{end} \\
 & | & \texttt{new}\ \mathit{eval} \\
 & | & \texttt{make}\ \mathit{eval}\ \mathit{expr} \\
 & | & \texttt{make}\ \mathit{value}\ \mathit{eval} \\
 & | & \mathit{lin\text{-}opt}\ \texttt{alloc}^{F}_{\Xi}\texttt{(}\mathit{eval}, \mathit{value}\texttt{)} \\
 & | & \texttt{ref}\ t\ \mathit{eval} \\
 & | & \texttt{deref}\ \mathit{eval} \\
\mathit{comp\text{-}comma} & ::= & \mathit{comp\text{-}value}\ \texttt{,} \\
\mathit{comma\text{-}comp} & ::= & \texttt{,}\ \mathit{comp\text{-}assign}
\end{array}
$$

Figure 9: Context-free grammar for evaluation contexts

- for all $\ell \in \text{dom } h$, $h(\ell)$ is a tuple (lo, Ξ, v_s, v_b), where lo is either ϵ or `linear`, Ξ is a finite set of force tokens, v_s is a labelled record value (recording the current state of the fields of the object at ℓ), and v_b is a labelled record value (giving the method bodies of the object at ℓ);
- for all $\jmath \in \text{dom } h$, $h(\jmath)$ is a pair (Ξ, v), where Ξ is a finite set of force tokens, and v is a value;
- for all $\rho \in \text{dom } h$, $h(\rho)$ is a value v.

For any $z \in \text{dom } h$, we write $h_\Xi(z)$ for the set Ξ of force tokens appearing in $h(z)$.

If $v_s = \{f_0\texttt{=}v_0\texttt{,} \ldots \texttt{,} f_{p-1}\texttt{=}v_{p-1}\}$ and $k = k_i$, we define

$$v_s[k\texttt{=}v] \;=\; \{f_0\texttt{=}v_0\texttt{,} \ldots \texttt{,} f_i\texttt{=}v\texttt{,} \ldots \texttt{,} f_{p-1}\texttt{=}v_{p-1}\}$$

If $h(\ell) = (lo, \Xi, v_s, v_b)$, we define updated heaps $h[\ell \mapsto v]$, $h[\ell\texttt{.}k \mapsto v]$ by

$$\begin{array}{rcl} h[\ell \mapsto v](z) & = & \left[\begin{array}{ll} h(z) & \text{if } z \neq \ell \\ (lo, \Xi, v, v_b) & \text{if } z = \ell \end{array}\right. \\ h[\ell\texttt{.}k \mapsto v](z) & = & \left[\begin{array}{ll} h(z) & \text{if } z \neq \ell \\ (lo, \Xi, v_s[k\texttt{=}v], v_b) & \text{if } z = \ell \end{array}\right. \end{array}$$

Similarly, if $h(\jmath) = (\Xi, v_0)$, we define the heap $h[\jmath \mapsto v]$ by

$$h[\jmath \mapsto v](z) \;=\; \left[\begin{array}{ll} h(z) & \text{if } z \neq \jmath \\ (\Xi, v) & \text{if } z = \jmath \end{array}\right.$$

We also write $h \upharpoonright \xi$ for the restriction of h to the set $\{z \mid \xi \in h_\Xi(z)\}$, and $h \oslash h'$ for the heap defined by

$$\begin{array}{rcl} (h \oslash h')(\ell) & = & \left[\begin{array}{ll} h'(\ell) & \text{if } \ell \in \text{dom } h' \\ h(\ell) & \text{if } \ell \in \text{dom } h - \text{dom } h' \end{array}\right. \\ (h \oslash h')(\jmath) & = & \left[\begin{array}{ll} h'(\jmath) & \text{if } \jmath \in \text{dom } h' \\ h(\jmath) & \text{if } \jmath \in \text{dom } h - \text{dom } h' \end{array}\right. \\ (h \oslash h')(\rho) & = & h(\rho) \end{array}$$

Our operational semantics consists of a system of rules for deriving *evaluation judgements* of the form $\Xi \vdash h, e \;\Downarrow\; h', w$, where w may be either a value or a stuck term. Here the portion $h, e \;\Downarrow\; h', w$ is called the *body* of the judgement; we use Φ as a metavariable ranging over bodies.

The operational rules presented below are to be understood with reference to the following conventions:

- Where only the bodies of judgements are given, a rule of the form

$$\frac{\Phi_0 \qquad \cdots \qquad \Phi_{r-1}}{\Phi} \;\text{side-conditions}$$

  is understood as an abbreviation for the rule

$$\frac{\Xi \vdash \Phi_0 \qquad \cdots \qquad \Xi \vdash \Phi_{r-1}}{\Xi \vdash \Phi} \;\text{side-conditions}$$

- Where mention of heaps is omitted, a rule of the form

$$\frac{\Xi_0 \vdash e_0 \Downarrow w_0 \qquad \cdots \qquad \Xi_{r-1} \vdash e_{r-1} \Downarrow w_{r-1}}{e \Downarrow w}\ \text{side-conditions}$$

is understood as an abbreviation for the rule

$$\frac{\Xi_0 \vdash h_0, e_0 \Downarrow h_1, w_0 \qquad \cdots \qquad \Xi_{r-1} \vdash h_{r-1}, e_{r-1} \Downarrow h_r, w_{r-1}}{h_0, e \Downarrow h_r, w}\ \text{side-conditions}$$

6.2 Auxiliary programs

We next define certain runtime expression contexts, for use in particular operational rules below. Officially these are contexts for the language defined by the grammar of Figure 7; we use meta-level parentheses (,) to disambiguate where necessary. For readability, we also adapt some of the derived forms from the grammar of Figure 3 as meta-level abbreviations:

$$\begin{array}{rcl}
\texttt{fn}\ x\ \texttt{=>}\ e & \equiv & \texttt{prom}\ (\texttt{oncefn}\ x\ \texttt{=>}\ e) \\
e\ \texttt{;}\ e' & \equiv & \texttt{split}\ \{\texttt{k=}e\}\ \texttt{as}\ \{\}\texttt{+}z\ \texttt{in}\ e'\ \texttt{end} \quad (z \notin e') \\
e\ e' & \equiv & (\texttt{der}\ e)\ \texttt{\$}\ e' \\
e\texttt{.}k & \equiv & \texttt{split}\ e\ \texttt{as}\ \{k\texttt{=}x\}\texttt{+}y\ \texttt{in}\ x\ \texttt{end}
\end{array}$$

The following runtime expression contexts are required for rule 117 below.

$$\begin{array}{rcl}
\mathsf{initializer}\,[v] & = & v\ \texttt{(fn u => \{\})} \\
\mathsf{selfbind}\,[v] & = & \texttt{rec f =>}\ (v\ \texttt{\{\}})\ \texttt{f}
\end{array}$$

For rule 115 we require contexts $\mathsf{comb\_constrs}\,[v, v']$ and $\mathsf{comb\_meths}^{F,M}_{lo}\,[v, v']$. The former is defined by

$$\begin{array}{rcl}
\mathsf{comb\_constrs}\,[v, v'] & = & \texttt{fn supercon => fn arg =>} \\
& & \quad v'\ (v\ \texttt{(fn u => \{\})) arg}
\end{array}$$

The latter is defined using an auxiliary context $\mathsf{extendImpl}^{F,M}_{lo}\,[v, \mathit{self}, k]$, which is itself defined separately for $lo = \epsilon, \texttt{linear}$. In the following definitions, we take $F = \{f_0, \ldots, f_{p-1}\}$ and $M = \{m_0, \ldots, m_{n-1}\}$.

$$\begin{array}{rcl}
\mathsf{extendSelf}^{M}\,[\mathit{self}, \mathit{rest}\,] & = & \{m_0\ \texttt{= fn rw =>}\ \mathit{self}\texttt{.}m_0\texttt{(rest +> rw), ... ,} \\
& & \ m_{n-1}\ \texttt{= fn rw =>}\ \mathit{self}\texttt{.}m_{n-1}\texttt{(rest +> rw)}\} \\
\mathsf{extendImpl}^{F,M}_{\epsilon}\,[v, \mathit{self}, k] & = & \texttt{fn rw' =>} \\
& & \quad \texttt{split rw' as \{}f_0\texttt{=}rw_0\texttt{,}\ldots\texttt{,}f_{p-1}\texttt{=}rw_{p-1}\texttt{\} + rest in} \\
& & \qquad v\ \texttt{(}\mathsf{extendSelf}^{M}\,[\mathit{self}, \texttt{rest}]\texttt{)}\texttt{.}k \\
& & \qquad\quad \{f_0\texttt{=}rw_0\texttt{,}\ldots\texttt{,}f_{p-1}\texttt{=}rw_{p-1}\} \\
& & \quad \texttt{end}
\end{array}$$

```
$\mathsf{contract}^{F}[e,j]$ = fn {value=x, state=s} =>
    split $e$ (readtemp $j$ +>s) as {value=y, state=t} in
    split t as {$f_0$=$s_0$, ... ,$f_{p-1}$=$s_{p-1}$} + rest in
      writetemp($j$,rest) ;
      {value=y, state={$f_0$=$s_0$, ... ,$f_{p-1}$=$s_{p-1}$}}
    end end
$\mathsf{contractAll}^{F,M}[e,j]$ = {$m_0$=$\mathsf{contract}^{F}[e.m_0,j]$, ... ,$m_{n-1}$=$\mathsf{contract}^{F}[e.m_{n-1},j]$}
$\mathsf{extendImpl}^{F,M}_{\mathtt{linear}}[v, self, k]$ = fn {value=x, state=t} =>
    split t as {$f_0$=$s_0$, ... ,$f_{p-1}$=$s_{p-1}$} + rest in
    temp j=rest in
      ($v$ ($\mathsf{contractAll}^{F,M}[self,$ j$]$)).$k$
        {value=x, state=$f_0$=$s_0$, ... ,$f_{p-1}$=$s_{p-1}$}}
    end end

$\mathsf{extendImplAll}^{F,M}_{lo}[v, self]$ = {$m_0$ = $\mathsf{extendImpl}^{F,M}_{lo}[v, self, m_0]$, ... ,
                                $m_{n-1}$ = $\mathsf{extendImpl}^{F,M}_{lo}[v, self, m_{n-1}]$}
$\mathsf{comb\_meths}^{F,M}_{lo}[v, v']$ = fn super => fn self =>
    split self as {$m_0$=$v_0$, ... ,$m_{n-1}$=$v_{n-1}$} + d in
      $v'$ ($\mathsf{extendImplAll}^{F,M}_{lo}[v$ {}, {$m_0$=$v_0$, ... ,$m_{n-1}$=$v_{n-1}$}$]$)
        self
    end
```

Finally, for rule 121 we require the meta-notation $\mathsf{rw\_bind}_{lo}(v, \ell, F)$. This is defined separately for $lo = \epsilon,$ `linear` as follows, supposing $v = \{m_0\texttt{=}v_0, \ldots, m_{n-1}\texttt{=}v_{n-1}\}$.

```
$\mathsf{rw\_suite}^{F}[\ell]$ = {$f_0$ = {read=(prom $\ell$).$f_0$.read,
              write=(prom $\ell$).$f_0$.write}, ... ,
          $f_{p-1}$ = {read=(prom $\ell$).$f_{p-1}$.read,
              write=(prom $\ell$).$f_{p-1}$.write}}
$\mathsf{rw\_bind}_{\epsilon}(v, \ell, F)$ = {$m_0$ = $v_0$ ($\mathsf{rw\_suite}^{F}(\ell)$), ... ,$m_{n-1}$ = $v_{n-1}$ ($\mathsf{rw\_suite}^{F}(\ell)$)}

$\mathsf{funToImp}[v, \ell]$ = fn x =>
    split $v$ {value=x, state=$\ell$.readAll $ {}}
    as {value=y, state=s'} in
      $\ell$.writeAll $ s' ; y
    end
$\mathsf{rw\_bind}_{\mathtt{linear}}(v, \ell, F)$ = {$m_0$ = $\mathsf{funToImp}[v_0, \ell]$, ... ,$m_{n-1}$ = $\mathsf{funToImp}[v_{n-1}, \ell]$}
```

6.3 Operational rules for successful evaluation

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

$$\frac{}{v \Downarrow v} \tag{95}$$

$$\frac{e_0 \Downarrow v_0 \quad \cdots \quad e_{n-1} \Downarrow v_{n-1}}{\{k_0\texttt{=}e_0\texttt{,}\ \ldots\texttt{,}k_{n-1}\texttt{=}e_{n-1}\} \Downarrow \{k_0\texttt{=}v_0\texttt{,}\ \ldots\texttt{,}k_{n-1}\texttt{=}v_{n-1}\}} \tag{96}$$

$$\frac{e \Downarrow v \quad e' \Downarrow v'}{e \ \texttt{+>}\ e' \Downarrow v \oslash v'} \tag{97}$$

$$\frac{\begin{array}{c} e \Downarrow \{k_{p0}\texttt{=}v_{p0}\texttt{,}\ \ldots\texttt{,}k_{p(n'-1)}\texttt{=}v_{p(n'-1)}\} \\ e'[v_0/x_0,\ldots,v_{n-1}/x_{n-1},\{k_n\texttt{=}v_n\texttt{,}\ \ldots\texttt{,}k_{n'-1}\texttt{=}v_{n'-1}\}/y] \Downarrow w \end{array}}{\texttt{split}\ e\ \texttt{as}\ \{k_0\texttt{=}x_0\texttt{,}\ \ldots\texttt{,}k_{n-1}\texttt{=}x_{n-1}\}\ \texttt{+}y\ \texttt{in}\ e'\ \texttt{end} \Downarrow w} \quad \begin{array}{l} p \in S_{n'} \\ pn < p(n+1) < \\ \cdots < p(n'-1) \end{array} \tag{98}$$

$$\frac{e \Downarrow w}{\texttt{tag}\ k\ e \Downarrow \texttt{tag}\ k\ w} \tag{99}$$

$$\frac{e \Downarrow \texttt{tag}\ k_i\ v \quad e_i[v/x_i] \Downarrow w}{\texttt{case}\ e\ \texttt{of}\ k_0x_0\texttt{=>}e_0\ \texttt{|}\ \cdots\ \texttt{|}\ k_{n-1}x_{n-1}\texttt{=>}e_{n-1}\ \texttt{end} \Downarrow w} \tag{100}$$

$$\frac{e \Downarrow \texttt{prom}\ e' \quad e' \Downarrow w}{\texttt{der}\ e \Downarrow w} \tag{101}$$

$$\frac{\Xi \vdash e \Downarrow v \quad \Xi \bullet \xi \vdash \texttt{saveheap (der}\ v\ \texttt{\$}\ \{\}\texttt{,}\ \xi\texttt{)} \Downarrow w}{\Xi \vdash \texttt{force}\ e \Downarrow w} \quad \xi = \mathsf{fresh}(\Xi) \tag{102}$$

$$\frac{h_0, e \Downarrow h_1, v}{h_0, \texttt{saveheap(}e\texttt{,}\xi\texttt{)} \Downarrow h_1,\ \texttt{prom restore}\ h_2\ \texttt{in}\ v\ \texttt{end}} \quad h_2 = h_1 \upharpoonright \xi \tag{103}$$

$$\frac{}{h, \texttt{restore}\ h'\ \texttt{in}\ v\ \texttt{end} \Downarrow h'', v} \quad h'' = h \oslash h' \tag{104}$$

$$\frac{e \Downarrow \texttt{fold}\ w}{\texttt{unfold}\ e \Downarrow w} \tag{105}$$

$$\frac{e \Downarrow \texttt{oncefn}\ x\texttt{=>}e_0 \qquad e' \Downarrow v \qquad e_0[v/x] \Downarrow w}{e\ \texttt{\$}\ e' \Downarrow w} \tag{106}$$

$$\frac{e \Downarrow v \qquad e' \Downarrow v'}{e\ \texttt{=}\ e' \Downarrow v''} \quad v'' = \left[\begin{array}{ll} \texttt{true} & \text{if } \mathsf{strip}(v) \approx \mathsf{strip}(v') \\ \texttt{false} & \text{otherwise} \end{array}\right. \tag{107}$$

$$\frac{e \Downarrow v \qquad e' \Downarrow v'}{e\ \imath\ e' \Downarrow v''} \quad v'' = \phi_\imath(v, v') \tag{108}$$

$$\frac{e_0 \Downarrow \texttt{true} \qquad e_1 \Downarrow w}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow w} \tag{109}$$

$$\frac{e_0 \Downarrow \texttt{false} \qquad e_2 \Downarrow w}{\texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow w} \tag{110}$$

$$\frac{e\,[\texttt{rec}\ x\texttt{=>}e\,/\,x] \Downarrow w}{\texttt{rec}\ x\texttt{=>}e \Downarrow w} \tag{111}$$

$$\frac{e[\hat{x}/x] \Downarrow \{\texttt{value=}v_0\texttt{,residue=}v_1\}}{\begin{array}{c}\texttt{catchcont}\ x\texttt{=>}e \Downarrow \\ \texttt{tag result}\ \{\texttt{value=}v_0\texttt{, more=oncefn}\ \hat{x}\texttt{=>}v_1\}\end{array}} \quad \hat{x} \text{ fresh} \tag{112}$$

$$\frac{e[\hat{x}/x] \Downarrow E[\texttt{der}\ \hat{x}\ \texttt{\$}\ v]}{\begin{array}{c}\texttt{catchcont}\ x\texttt{=>}e \Downarrow \\ \texttt{tag query}\ \{\texttt{arg=}v\texttt{, resume=oncefn}\ z\texttt{=>oncefn}\ \hat{x}\texttt{=>}E[z]\}\end{array}} \quad \begin{array}{l}\hat{x} \text{ fresh} \\ z = \mathsf{fresh}\,(E[-])\end{array} \tag{113}$$

$$\frac{h_0, e \Downarrow h_1, \texttt{oncefn}\ \hat{x}\texttt{=>}e_0 \qquad h_1, e' \Downarrow h_2, v \qquad h_2[v/\hat{x}], e_0[v/\hat{x}] \Downarrow h_3, w}{h_0,\ e\ \texttt{\$}\ e' \Downarrow h_3, w} \tag{114}$$

$$\frac{e \Downarrow lo\ \texttt{extend}^{F,M}\ \texttt{root with}\ v_1\texttt{,}v_2\ \texttt{end}}{\begin{array}{c} lo\ \texttt{extend}^{F',M'}\ e\ \texttt{with}\ v_1'\texttt{,}v_2'\ \texttt{end} \Downarrow \\ lo\ \texttt{extend}^{F',M'}\ lo\ \texttt{root with}\ \mathsf{comb\_meths}^{F,M}_{lo}\,[v_1, v_1']\texttt{,}\ \mathsf{comb\_constrs}\,[v_2, v_2']\ \texttt{end}\end{array}} \tag{115}$$

$$\frac{\Xi \vdash e \Downarrow lo\ \texttt{root}}{\Xi \vdash \texttt{new}\ e \Downarrow lo\ \texttt{constr}_\Xi\ \texttt{(fn u=>\{\}, \{\})}} \tag{116}$$

$$\frac{\Xi \vdash e \Downarrow lo\ \texttt{extend}^{F,M}\ lo\ \texttt{root with}\ v_1\texttt{,}v_2\ \texttt{end}}{\Xi \vdash \texttt{new}\ e \Downarrow lo\ \texttt{constr}^F_\Xi\ \texttt{(}\mathsf{initializer}[v_2]\texttt{,}\ \mathsf{selfbind}[v_1]\texttt{)}} \tag{117}$$

$$\frac{e \Downarrow lo\ \texttt{constr}^{F}_{\Xi'}\texttt{(}v_i\texttt{,}v_m\texttt{)} \qquad lo\ \texttt{alloc}^{F}_{\Xi'}\texttt{(der}\ v_i\ \texttt{\$}\ e'\texttt{,}\ v_m\texttt{)} \Downarrow w}{\texttt{der}\ e\ \texttt{\$}\ e' \Downarrow w} \tag{118}$$

$$\frac{\Xi \vdash e \Downarrow \texttt{linear root} \qquad \Xi \vdash \texttt{linear alloc}^{F}_{\Xi}\texttt{(\{\},\{\})} \Downarrow w}{\Xi \vdash \texttt{make}\ e\ e' \Downarrow w} \tag{119}$$

$$\frac{\begin{array}{c}\Xi \vdash e \Downarrow \texttt{linear extend}^{F,M}\ \texttt{linear root with}\ v_1\texttt{,}v_2\ \texttt{end} \\ \Xi \vdash \texttt{linear alloc}^{F}_{\Xi}\texttt{(der}\ v_i\ \texttt{\$}\ e'\texttt{,}\ v_m\texttt{)} \Downarrow w\end{array}}{\Xi \vdash \texttt{make}\ e\ e' \Downarrow w} \quad \begin{array}{l} v_i = \mathsf{initializer}[v_2] \\ v_m = \mathsf{selfbind}[v_1]\end{array} \tag{120}$$

$$\frac{h_0, e \Downarrow h_1, v_s}{h_1, lo\ \texttt{alloc}^{F}_{\Xi'}\texttt{(}e\texttt{,}v_m\texttt{)} \Downarrow h_2, \ell} \quad \begin{array}{l} \ell = \mathsf{freshLoc}\,(\mathrm{dom}\ h_1) \\ h_2 = h_1(\ell \mapsto (lo, \Xi', v_s, v_b)) \\ v_b = \mathsf{rw\_bind}_{lo}(v_m, \ell, F)\end{array} \tag{121}$$

$$\frac{}{h, \ell\texttt{.}k \Downarrow h, v} \quad h(\ell) = (lo, \Xi', v_s, \{k\texttt{=}v\texttt{,}\ \cdots\}) \tag{122}$$

$$\frac{h_0, e \Downarrow h_1, \ell\texttt{.}k\texttt{.read} \qquad h_1, e' \Downarrow h_2, \{\}}{h_0, e\ \texttt{\$}\ e' \Downarrow h_2, v} \quad h_2(\ell) = (\epsilon, \Xi', \{k\texttt{=}v\texttt{,}\ \cdots\}, v_b) \tag{123}$$

$$\frac{h_0, e \Downarrow h_1, \ell\texttt{.}k\texttt{.write} \qquad h_1, e' \Downarrow h_2, v}{h_0, e\ \texttt{\$}\ e' \Downarrow h_3, \{\}} \quad h_3 = h_2[\ell\texttt{.}k \mapsto v] \tag{124}$$

$$\frac{h_0, e \Downarrow h_1, \ell\texttt{.readAll} \qquad h_1, e' \Downarrow h_2, \{\}}{h_0, e\ \texttt{\$}\ e' \Downarrow h_2, v_s} \quad h_2(\ell) = (\epsilon, \Xi', v_s, v_b) \tag{125}$$

$$\frac{h_0, e \Downarrow h_1, \ell\texttt{.writeAll} \qquad h_1, e' \Downarrow h_2, v}{h_0, e\ \texttt{\$}\ e' \Downarrow h_3, \{\}} \quad h_3 = h_2[\ell \mapsto v] \tag{126}$$

$$\frac{\Xi \vdash h_1, e[\jmath/x] \Downarrow h_1', w}{\Xi \vdash h, \texttt{temp}\ x\texttt{=}v\ \texttt{in}\ e\ \texttt{end} \Downarrow h', w} \quad \begin{array}{l} \jmath = \mathsf{freshTemp}(h) \\ h_1 = h + [\jmath \mapsto (\Xi, v)] \\ h_1' = h' + [\jmath \mapsto (\Xi, v')]\end{array} \tag{127}$$

$$\frac{}{h, \texttt{readtemp}\ \jmath \Downarrow h, v} \quad h(\jmath) = (\Xi', v) \tag{128}$$

$$\frac{}{h, \texttt{writetemp (}\jmath\texttt{,}v\texttt{)} \Downarrow h', \{\}} \quad h' = h[\jmath \mapsto v] \tag{129}$$

$$\frac{h_0, e \Downarrow h_1, v}{h_0, \texttt{ref}\ e \Downarrow h_2, \rho} \quad \begin{array}{l} \rho = \mathsf{freshRef}\,(\mathrm{dom}\ h_1) \\ h_2 = h_1(\rho \mapsto v)\end{array} \tag{130}$$

$$\frac{h_0, e \Downarrow h_1, \rho}{h_0, \texttt{deref}\ e \Downarrow h_1, v} \quad h_1(\rho) = v \tag{131}$$

6.4 Operational rules for stuck evaluation

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

$$\frac{}{\texttt{der}\ \hat{x}\ \texttt{\$}\ v \ \Downarrow\ \texttt{der}\ \hat{x}\ \texttt{\$}\ v} \tag{132}$$

$$\frac{e \ \Downarrow\ E[\texttt{der}\ \hat{x}\ \texttt{\$}\ v]}{E'[e] \ \Downarrow\ (E' \circ E)[\texttt{der}\ \hat{x}\ \texttt{\$}\ v]} \quad E, E' \text{ transparent to } x \tag{133}$$

$$\frac{e_0 \ \Downarrow\ v_0 \qquad \cdots \qquad e_{i-1} \ \Downarrow\ v_{i-1} \qquad e_i \ \Downarrow\ u_i}{\{k_0\texttt{=}e_0\texttt{,}\ \ldots\ \texttt{,}k_{n-1}\texttt{=}e_{n-1}\} \ \Downarrow\ \{k_0\texttt{=}v_0\texttt{,}\ \ldots\ \texttt{,}k_i\texttt{=}u_i\texttt{,}\ \ldots\ \texttt{,}k_{n-1}\texttt{=}e_{n-1}\}} \quad i < n \tag{134}$$

$$\frac{e \ \Downarrow\ v \qquad e' \ \Downarrow\ u}{e\ \texttt{+>}\ e' \ \Downarrow\ v \oslash u} \tag{135}$$

$$\frac{e \ \Downarrow\ v \qquad e' \ \Downarrow\ u}{e\ \texttt{\$}\ e' \ \Downarrow\ v\ \texttt{\$}\ u} \tag{136}$$

$$\frac{e_0 \ \Downarrow\ v_0 \qquad e_1 \ \Downarrow\ u_1}{e_0\ \texttt{=}\ e_1 \ \Downarrow\ v_0\ \texttt{=}\ u_1} \tag{137}$$

$$\frac{e_0 \ \Downarrow\ v_0 \qquad e_1 \ \Downarrow\ u_1}{e_0\ \imath\ e_1 \ \Downarrow\ v_0\ \imath\ u_1} \tag{138}$$

$$\frac{e_0 \ \Downarrow\ v_0 \qquad e_1 \ \Downarrow\ u_1}{\texttt{make}\ e_0\ e_1 \ \Downarrow\ \texttt{make}\ v_0\ u_1} \tag{139}$$

$$\frac{e[\hat{x}/x] \ \Downarrow\ E[\texttt{der}\ \hat{y}\ \texttt{\$}\ v]}{\texttt{catchcont}\ x\texttt{=>}e \ \Downarrow\ \texttt{catchcont}\ x\texttt{=>}E'[\texttt{der}\ \hat{y}\ \texttt{\$}\ v']} \quad \begin{array}{l} \hat{y} \neq \hat{x} \\ E' = E[x/\hat{x}] \\ v' = v[x/\hat{x}] \end{array} \tag{140}$$

7 Top-level issues

At top level, a Lingay session is an interactive process which proceeds in a series of *read-eval cycles.* On each cycle, the programmer submits a top level declaration, which the system attempts to typecheck and (if appropriate) evaluate relative to certain environment information maintained by the system. If all these phases complete successfully, the system displays the result of the computation and the If the parsing or typechecking phases fail, an error is reported.

The environment information maintained by the system consists of the following:

- a type abbreviation environment Θ as defined in Section 2,
- a static environment Γ as defined in Section 4, in which all first safety tags are ϵ and all second safety tags are ϵ or newsafe,
- a heap h as defined in Section 6,
- an expression abbreviation environment Ω as defined in Section 5, such that dom $\Omega =$ dom Γ and such that whenever $\Omega(x) = e$, all locations and temporary locations appearing in e are in dom h.

Each of these may be potentially updated from one cycle to the next. At the start of the session, these are set to certain *initial* values to be specified by the implementor (for example, they may all be empty).

On each cycle, a *top-level declaration* is submitted to the session. A top-level declaration is a phrase of category *top-decl*, as defined by the following grammar in conjunction with the surface grammars for types and expressions (Figures 1, 3 and 4):

$$\begin{array}{rcl} \textit{top-decl} & ::= & \texttt{type}\ t\ \texttt{=}\ \textit{type-expr}\ \texttt{;} \\ & | & \texttt{datatype}\ t\ \texttt{=}\ \texttt{[|}\ \textit{Clist}\ (\textit{summand-type})\ \texttt{|]}\ \texttt{;} \\ & | & \textit{decl} \end{array}$$

We leave the implementor free to specify exactly how a sequence of top-level declarations is extracted from a stream of input characters (for instance, at what point lexing and parsing errors are detected and how the system recovers from them); the only firm requirement is that an input stream consisting of a sequence of *correctly formed* phrases of category *top-decl*, each of which is immediately followed (modulo whitespace) by at least one return character, should indeed be processed as a series of top-level declarations as follows.

We now describe how each of the three forms of top-level declaration are processed in the presence of the environment information $(\Theta, \Gamma, h, \Omega)$.

- A top-level declaration of the form `type` t `= t ;` is processed by first attempting to translate `t` to an underlying type expression $\mathtt{t}^{\Theta}$ as detailed in Section 2.
  - If $\mathtt{t}^{\Theta}$ is undefined (because `t` involves some type name not appearing in Θ), an error is reported and the cycle ends with the environment information unchanged.
  - If $\mathtt{t}^{\Theta} = \tau$, we define $\Theta' = \Theta[t \mapsto \tau]$. The successful binding of τ to t is reported, and the cycle ends with the updated environment information $(\Theta, \Gamma, h, \Omega)$.
- A top-level declaration `datatype` t `= [|` sm_0 `,` $\cdots$ `,` sm_{n-1} `|] ;` is processed as the sequence of declarations

  $$\begin{array}{l} \textit{type-expr}\ t\ \texttt{=}\ \texttt{rectype}\ t\ \texttt{=>}\ \texttt{[|}\ sm_0\ \texttt{,}\ \cdots\ \texttt{,}\ sm_{n-1}\ \texttt{|]}\ \texttt{;} \\ \text{inj-decl}(sm_0, t) \\ \cdots \\ \text{inj-decl}(sm_{n-1}, t) \end{array}$$

  where we define

  $$\begin{array}{rcl} \text{inj-decl}(k\ \texttt{of}\ \texttt{t}, t) & = & \texttt{val}\ k\ \texttt{=}\ \texttt{fn}\ x\texttt{:t}\ \texttt{=>}\ \texttt{fold}\ t\ \texttt{(tag}\ k\ x\texttt{)}\ \texttt{;} \\ \text{inj-decl}(k, t) & = & \texttt{val}\ k\ \texttt{=}\ \texttt{fold}\ t\ \texttt{(tok}\ k\texttt{)}\ \texttt{;} \end{array}$$

- A top-level declaration `val` x `= e ;` is processed as follows.
  - Firstly, we attempt to translate `e` to a surface expression e' by replacing each subphrase `extending` t (of category *extend-clause*) by `extending` t `: t`, where `t` is some surface type expression representing $\Gamma[t]$ (which may be chosen arbitrarily). If this translation fails because some relevant identifier t does not appear in Γ, an error is reported and the cycle ends with the environment information unchanged.
  - Secondly, we attempt to translate e' to an underlying type expression e^{Θ} as detailed in Section 3.
    - If e^{Θ} is undefined, an error is reported and the cycle ends with the environment information unchanged.
    - Otherwise, we define $e = \mathsf{e}^{\Theta}$.
  - Thirdly, we attempt to typecheck e relative to Γ and the reftype environment $\emptyset$.
    - If either $\emptyset, \Gamma \vdash e : \tau \; \epsilon \; \mathsf{newsafe}$ or $\emptyset, \Gamma \vdash e : \tau \; \epsilon \; \mathsf{halfnewsafe}$ is derivable for some (necessarily unique) underlying type τ, we record the type τ and set $st_2 = \mathsf{newsafe}$, and the value of τ is reported at this point.
    - Otherwise, if $\emptyset, \Gamma \vdash e : \tau \; \epsilon \; \epsilon$ is derivable for some (necessarily unique) underlying type τ we record the type τ and set $st_2 = \epsilon$, and the value of τ is reported at this point, along with the fact that the result of evaluating e will be "not newsafe".
    - If $\emptyset, \Gamma \vdash e : \tau \; \epsilon \; \epsilon$ is not derivable for any τ, a type error is reported and the cycle ends with the environment information unchanged.
  - Next, we translate e to a closed runtime expression e^* using the environment Ω as detailed in Section 5. Given that e has already been typechecked relative to Γ, this translation will always succeed.
  - Lastly, we attempt to evaluate e^* relative to the heap h: that is, we attempt to compute some (necessarily unique) h', v' such that the evaluation judgement $\emptyset \vdash h, e^* \;\Downarrow\; h', v'$ is derivable. The evaluation procedure must be such that if a suitable h', v' exist they will eventually be found, given sufficient time and memory.
    - If h' and v' are found, then the fact of termination is reported, and if τ is a ground type then the value v' is also reported. (An implementation may also report ground type components of non-ground values.) The cycle then ends with the environment information updated to

      $$(\Theta,\ \Gamma' \bullet x : \tau \; \epsilon \; st_2,\ h',\ \Omega[x \mapsto v'])$$

      where Γ' is obtained from Γ by deleting all entries $\bullet\, y : \tau' \; st_1' \; st_2'$ such that $y \in \mathrm{FV}(e)$ and $\Gamma[y]$ is not reusable.
- A top-level declaration `recval` x`:t = e` is processed precisely as `val` x `= rec` x`:t => e`.
- A top-level declaration `e ;` is processed precisely as `val it = e ;`.

Finally, if the processing of a top-level declaration is aborted (*e.g.* by an interrupt from the user) before it is complete, this fact is reported and the cycle ends with the environment information unchanged.