Labelled Reductions, Runtime Errors, and Operational Subsumption

Dami, Laurent

Labelled Reductions, Runtime Errors, and Operational Subsumption [1]

Laurent Dami

Abstract

When combining modules it may be the case that the assembly is partially incorrect (error-prone), but nevertheless useful in some contexts. However usual type system will reject such assemblies as soon as they detect a potential error. We propose a more liberal approach: an error in a software configuration is tolerated as long as there are contexts which can use it without reaching the error; as a result, software reusability is improved. We introduce a general framework which defines in a language-independent way what it means to be "erroneous", and under which conditions a component may "subsume" another (i.e. replace it in any context). This new semantics, based on the observation of errors, is then applied to a comparison of various lambda-calculi, and shown to be close to the well-known approximation semantics. An interesting application is to use the subsumption semantics for a simple term model interpretation of subtyping. The framework also proposes a language-independent specification of labelled reduction, which is used as a technical tool to syntactically characterize finite approximation. This generalizes the work of Mason, Smith and Talcott on getting denotational structures through operational techniques, and furthermore provides an operational way to interpret recursive type definitions.

1 Introduction

In the classical theory of the λ-calculus [4], the divergent term Ω is equal to $\lambda x.\Omega$, i.e. a function returning a divergent term. By contrast, in the lazy theory [1] $\lambda x.\Omega$ is a value different from Ω. This theoretical distinction has important practical consequences, since a family of modern programming languages are now based on the lazy theory [18]. We are interested in similar distinctions with respect to another form of unsolvable terms, namely runtime errors such as `(1+ "foo")` or `aWindow.computeSalary()`. Let ε denote such errors. Most type systems will equally reject programs (ε), $(\lambda x.\varepsilon)$ or $((\lambda xy.y)\varepsilon)$. However the latter two do not actually generate a runtime error when executed; in a lazy system the last program will even produce a "good" value (the identity function $(\lambda y.y)$). This shows that the common notion of "erroneous" terms, as implemented in most typed languages, is over-restrictive. If, instead, we take a lazy approach to errors, then even a typed language can accept a term like $((\lambda xy.y)\varepsilon)$. Admittedly, this is not a strikingly useful example, but more realistic situations can be displayed with modular programming, where some software assemblies may be partially incorrect but nevertheless useful. As a simple example, consider a function like

$$T \stackrel{\text{def}}{=} \lambda x.\{\texttt{imprime} = x.\texttt{print}, \texttt{affiche} = x.\texttt{display}, \texttt{ferme} = x.\texttt{close}\}$$

[1] An extended abstract of this paper will appear in the *Proceedings of the 24th International Colloquium on Automata, Languages, and Programming, ICALP'97*, Bologna, Italy, July 7-11, LNCS, Springer-Verlag.

in a λ-calculus with records. This is a "translating" function, taking an "english" record with three fields as argument, and returning a corresponding "french" record. Now consider record $R \stackrel{\text{def}}{=} \{\texttt{display} = \texttt{"hello"}\}$ and program $(T\ R).\texttt{affiche}$. It makes sense to apply the translating function to a single-field record, and then select the french version of that single field; yet this program would be rejected by most existing type systems because the type of T is

$$\forall X, Y, Z, \{\texttt{print} : X, \texttt{display} : Y, \texttt{close} : Z\} \to \{\texttt{imprime} : X, \texttt{affiche} : Y, \texttt{ferme} : Z\}$$

and $\{\texttt{display} : \mathbf{String}\}$ (the type of R) cannot be unified with the left-hand side of the arrow. In other words, the partial assembly $(T\ R)$ is rejected because of a partial mismatch between the function and its argument. However the rejection decision does not take into account the fact that the context surrounding $(T\ R)$ only accesses field `affiche` and therefore yields no error. This small example is an illustration of situations which may arise frequently in modular or distributed systems, or even more in mobile object systems: software components are called to cooperate into a global task, and must be able to interact in a useful way even if their interfaces do not necessarily match perfectly.

Here we propose a more liberal approach to errors. In this approach, errors (written ε) can be passed around as any other value, sometimes in a lazy way, and therefore an error occurring inside a term is not necessarily propagated to the top level; a term is considered "erroneous" if and only if it *always* generates ε after a finite number of interactions with its context. We define a general framework for studying the semantics of programs containing errors, and a language-independent classification of error propagation properties; we also define an operational ordering of terms, called "subsumption", which gives a formal foundation for the notion of "substitutability" or "safe replacement" often used informally in the object-oriented literature: a term subsumes another iff it generates fewer errors in all program contexts. Subsumption often implies and sometimes equals the usual approximation ordering (Theorems 4.6, 5.6); its main interest is to directly interpret subtyping in a term model, which is simpler than the partial equivalence relations (PERs) of [8] or the coercion functions of [7]. This is illustrated in Section 5 where the abstract framework is applied to a comparison of various λ-calculi.

Our framework only applies to "uncatchable" errors, i.e. languages in which there is no way to recover from an error once it has been raised. This is a necessary condition in order to have a meaningful notion of subsumption: otherwise since error generation is the basic observation, an exception handling construct would make all programs incomparable; similar requirements are made for example in [10, 22]. Because of this requirement, ε is subsumed by any term and therefore is the *top* element in the subsumption ordering. In consequence the semantic structure is a lattice, like in the original work of Scott [31]. As noted in [6], lattices have a simple structure which often eases mathematical treatement; nevertheless the majority of semantic studies now uses more complex models (various forms of cpo constructions) because of criticisms [28, 6] related to the "over-defined" top element, which somehow implies the presence of multivalues (for example what is the meaning of an upper bound of the truth values `tt` and `ff`?). Without entering the discussion – see also [5] – we only mention that here uncatchable errors behave as a black hole above any value and therefore give a natural justification for coming back to a lattice structure with a top element.

In order to accept terms such as $((\lambda xy.y)\varepsilon)$, we also take a different approach to typing. Section 6 defines a type system which includes a type $\top$ that *contains the error element*. This is a radical departure from most type systems in the literature, esp. from object-oriented type systems in which the top type usually only contains all "good" elements. However, when combined with parametric polymorphism, this approach can increase reusability: the example above involving the translating function T now becomes typable, because the type of T can be instanciated to

$$\forall Y.\{\texttt{display} : Y\} \to \{\texttt{affiche} : Y\}$$

by unifying the X, Z type variables with $\top$. Of course this approach implies that every term is typable, but static prevention of errors is not lost, because erroneous terms can only have types from a small subset which we call *trivial* types and which are easily identifiable (Section 6).

For the technical development below we make heavy use *labelled reductions*, an old idea used in the λ-calculus to restrict the interaction behaviour of a term to a finite number of steps. Here this is generalised in an abstract way to other rewriting systems. Labelled reductions allow us to classify both terms and contexts according to the number of interaction steps they can perform, and therefore introduce an operational notion of finite approximation. This is quite similar to the approach of Mason, Smith and Talcott [26] who use syntactic projections to derive denotational results from operational specifications, except that our finite approximations are not bound to a particular language. Furthermore, we also propose in Section 6 an innovative application of such approximations as an alternative to the embedding-projection pairs of [9] for solving recursive type equations.

Note. This is an expanded version of the extended abstract published in [13]. Remarks from Scott Smith, Manuel Serrano and from anonymous referees on a previous draft of this paper were highly valuable and are gratefully acknowledged.

2 Basic definitions: error generation and preservation

This section defines a number of abstract notions, independent of any particular language. However, since some concepts need illustrations, informal examples will be drawn from the standard λ-calculus extended with constants and records. Precise definitions for this calculus and other calculi will be given later in Section 5. Prior knowledge of the λ-calculus and the notions of call-by-name (CBN), call-by-value (CBV) and lazy evaluation is assumed; standard references are [4, 27, 1]. As a reminder, common abbreviations for λ-terms are $\mathbf{I} \stackrel{\text{def}}{=} \lambda x.x, \mathbf{K} \stackrel{\text{def}}{=} \lambda xy.x, \Delta \stackrel{\text{def}}{=} \lambda x.xx, \Omega \stackrel{\text{def}}{=} \Delta\Delta, \mathbf{Y} \stackrel{\text{def}}{=} \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$; furthermore $\mu x.a$ abbreviates $\mathbf{Y}(\lambda x.a)$.

Notation. We consider languages of the form $(\mathcal{T}, \mathcal{V}, \to)$ where

- $\mathcal{T}$ is a set of *terms*,
- $\mathcal{V} \subset \mathcal{T}$ is the set of *values*,

- $\rightarrow$ is a binary relation on terms (*one-step reduction*) satisfying $\forall v \in \mathcal{V}, v \rightarrow v' \implies v' \in \mathcal{V}$

The letters a, b, c range over arbitrary terms, v, u range over values. We assume that terms are built from a set $\mathcal{X}$ of *variables* and a set $\mathcal{F}$ of *function symbols* (also called sometimes *operators*) with fixed arities; letters x, y, z range over $\mathcal{X}$ and letter F is used for function symbols. We also assume standard notions of bound and free variables, and a function $FV : \mathcal{T} \rightarrow 2^{\mathcal{X}}$ giving the free variables of a term. Typical examples of relevant languages would be rewrite systems possibly dealing with bound variables, as in [23, 20, 32]; however for the rest of the presentation we do not need to stick to one particular formalism. $\mathcal{T}^C$ and $\mathcal{V}^C$ denote the sets of *closed* terms and values, i.e. those for which FV returns the empty set (we do not adopt the common notation $\mathcal{T}^0$ for closed terms, because natural number superscripts will be used for a different purpose – see Definition 3.11). *Substitutions* are partial functions $\sigma : \mathcal{X} \rightharpoonup \mathcal{T}$ associating terms to variables; $dom(\sigma)$ is the set of variables for which σ is defined. The application of a substitution σ to a term a, written $a\sigma$, denotes the term obtained by replacing all free occurrences of variables $x \in FV(a) \cap dom(\sigma)$ by their image $\sigma(x)$, while avoiding variable capture. The single substitution mapping x to a is written $[x := a]$. *Contexts* are terms possibly containing occurrences of a "hole" $[-]$; if $C[-]$ is a context, then $C[a]$ is the term obtained by filling the hole in $C[-]$ with a, possibly capturing variables. The set of contexts is written $\mathcal{T}[-]$; since there is no restriction on the number of occurrences of the hole, we have $\mathcal{T} \subset \mathcal{T}[-]$. A *subterm* of a is a term a' such that $a \equiv C[a']$ for some $C[-]$. The reflexive, transitive closure of $\rightarrow$ is written $\stackrel{*}{\rightarrow}$ and $\stackrel{*}{=}$ is its symmetric closure; $(a \rightarrow)$ is an abbreviation for $\exists b, a \rightarrow b$. Finally, if $\mathrel{\mathbb{R}}$ is one of the operational ordering relations defined below, with θ representing any collection of subscripts/superscripts, then $\stackrel{\theta}{=}$ is its symmetric closure and $\mathrel{\mathbb{R}}$ is its strict restriction, i.e. the relation $\mathrel{\mathbb{R}} \setminus \stackrel{\theta}{=}$.

2.1 Convergence, relevance, solvability

Definition 2.1 (Reduction properties)
For a language $\mathcal{L} \stackrel{\text{def}}{=} (\mathcal{T}, \mathcal{V}, \rightarrow)$ *we say that*

- a *is* stuck *iff* $a \notin \mathcal{V}$ *and* $\neg(a \rightarrow)$
- a diverges *(written* $a \Uparrow$*) iff, for each* b *such that* $a \stackrel{*}{\rightarrow} b$, *we have* $((b \notin \mathcal{V}) \wedge (b \rightarrow))$. *Conversely,* a converges *(*$a \Downarrow$*) iff* $\exists v \in \mathcal{V}, a \stackrel{*}{\rightarrow} v$.
- $\rightarrow$ *is* Church-Rosser *(CR) iff* $((a \stackrel{*}{\rightarrow} b) \wedge (a \stackrel{*}{\rightarrow} c)) \implies \exists d.((b \stackrel{*}{\rightarrow} d) \wedge (c \stackrel{*}{\rightarrow} d))$
- $\rightarrow$ *is* compatible *iff* $a \rightarrow b \implies C[a] \stackrel{*}{\rightarrow} C[b]$ *for any context* $C[-]$

Stuck terms like $1 + \texttt{"foo"}$ cannot compute further, but are not values, so they neither diverge nor converge: these are the terms that typically should be ruled out by a type system. Our definition of compatibility is slightly different from [4]: this is because we allow contexts with several occurences of the hole, so several steps may be needed to reduce all occurrences of a in $C[a]$ before reaching $C[b]$.

Definition 2.2 (Relevant contexts)
A context $C[-]$ is relevant *iff $a \Uparrow \implies C[a] \Uparrow$ and there is a term b such that $C[b] \Downarrow$.*

The idea here is that a context is irrelevant when no observation can be made about the term filling the hole. Since the basic observation is divergence, the first condition ensures that divergence at the hole is propagated to the outer level. The second condition rules out the contexts which are always divergent, because these also hide any observation from the hole.

Example 2.3
Contexts without any hole are irrelevant. Contexts $[-], ([-]ab), ((\lambda x.[-])ab), [-].l$ are relevant. The context $(\mathbf{K}[-]a)$ is relevant with CBV evaluation, but not with CBN. The context $\lambda x.[-]$ is relevant with both CBV and CBN, but not with lazy evaluation.

Definition 2.4 (Solvable terms)
A term a is solvable *iff, for every term b, there is a relevant context $C[-]$ such that $C[a] \stackrel{*}{\to} b$.*

2.2 Error properties

Definition 2.5 (Language properties)
A language $(\mathcal{T}, \mathcal{V}, \to)$

- has divergence *iff there is at least a term $\Omega \in \mathcal{T} \setminus \mathcal{V}$ such that $\Omega \Uparrow$.*
- *is* stuck-free *iff $\mathcal{T}^C$ contains no stuck terms.*
- has errors *iff there is a nonempty subset $\mathcal{E} \subset \mathcal{V}$ of* error values *satisfying $v \in \mathcal{E} \implies \neg(v \to)$. Most often we will consider a singleton set and write ε to denote the single error value. We write $a\dagger^0$ if $a \stackrel{*}{\to} v \in \mathcal{E}$ (the motivation for the 0 superscript will be made clear later in Definition 4.1).*
- *is* error-generating *iff there is an $a \in \mathcal{T}$ such that $a\dagger^0$ and for every subterm a' of a, $a' \notin \mathcal{E}$.*
- *is* error-complete *iff, for every value $v \in \mathcal{V}^C$, there is a relevant context $C[-]$ such that $C[v]\dagger^0$.*
- *is* error-preserving *iff, for every error value $\varepsilon \in \mathcal{E}$, there is no relevant context $C[-]$ such that $C[\varepsilon] \Uparrow$.*

Some comments are of order. Absence of stuck terms is easily obtained by adding an error term ε and completing the reduction relation so that stuck terms explicitly reduce to ε. In that case the language is also error-generating, which means that errors can be created dynamically. Error-completeness is a closely related, but different property, saying that every value can be turned into an error by some context; we will show examples of languages which are error-generating but not error-complete, or vice-versa. Finally, error-preservation ensures that errors are not observable internally; in other words, there is no "catch" construct to recover from errors. The definition of error-sensitivity in [10] or axiom 2 of [22] for meaningless terms in rewriting capture the same idea. Observe that error-preservation implies that ε is unsolvable.

Example 2.6
The pure λ-calculus with an added error constant ε has stuck terms: (εa) does not reduce and is not a value. With an added reduction rule $\forall a. \varepsilon a \to \varepsilon$ the language becomes stuck-free; however it is not error-generating. Error-completeness varies with the evaluation strategy: with CBN evaluation, all values are solvable, and therefore can become errors in some context. By contrast, lazy evaluation admits values which are unsolvable, so then the language is not error-complete: there is no relevant context which can turn $\lambda x.\Omega$ into an error.

Example 2.7
The λ-calculus with integers and integer operators is not stuck-free: for example $(3 + \lambda x.x)$ is stuck. Again it can be made stuck-free by adding an error constant ε and rules for reducing all stuck terms to ε; in that case it obviously becomes error-generating. Moreover it also becomes error-complete, independently of the evaluation strategy: this is because there are contexts such as $([-][-])$ and $([-] + [-])$ which discriminate between functional values and integer values, even if they are unsolvable.

Example 2.8
Assume a language with some exception constants $\epsilon_1 \ldots \epsilon_n$ and an exception handling construct

$$\mathbf{try}\ a\ \mathbf{catch}\ \epsilon_{i_1} \to b_1 \ldots \epsilon_{i_k} \to b_k\ \mathbf{end}$$

If exceptions are considered as errors, i.e. if $\{\epsilon_1 \ldots \epsilon_n\} \subseteq \mathcal{E}$, then this language is obviously *not* error-preserving. However exception handling does not necessarily always impair the error-preservation condition: both can be combined if "errors" and "exceptions" are distinguished. Typically errors should correspond to what one usually calls "type errors", like a mismatch between an operator and its operands, or a wrong message sent to an object who cannot understand it. By contrast, exceptions correspond to other abnormal conditions such as failure to open a file, or indexing an array out of its declared bounds. This distinction is exactly the one of the Java language [16], where the `Error` class characterizes uncatchable errors, while `RuntimeException` characterizes catchable exceptions.

Example 2.9
A language containing constructs $\mathbf{isnat}$, $\mathbf{islam}$, $\mathbf{ispr}$, ... for identifying various syntactic classes of values such as numbers, λ-abstractions or pairs, is in principle not error-preserving: for example the context

$$\mathbf{if}\ (\mathbf{islam}([-]))\ \mathbf{then}\ +1\ \mathbf{else}\ -1$$

returns -1 for all values which are not λ-abstractions, including ε. Such constructs were studied in [26]; however in this particular case the language nevertheless is error-preserving because of specific design choices: since evaluation in [26] is CBV and errors (stuck terms) are not values, $\mathbf{islam}(\varepsilon)$ is a stuck term. For the same purpose of syntactic discrimination, [3] take a different approach, based on a single construct covering all syntactic classes:

$$\mathbf{cases}\ a\ \mathbf{nat} : a_1\ \mathbf{fun} : a_2\ \mathbf{pair} : a_3 \ldots\ \mathbf{end}$$

In that case the language is naturally error-preserving, provided of course that the cases construct has no "default" clause and no clause to recognize errors.

Example 2.10
The λ-calculus extended with ε, with records $\{l_1 = a_1 \ldots l_n = a_n\}$ and with a field selection construct $a.l$, together with the obvious reduction and error generation rules, is stuck-free, error-generating, error-complete and error-preserving. This language will be studied below in Section 5.

2.3 Operational Approximation

Inspired by the operational simulation techniques of process calculi, several operational orderings for sequential languages have been studied recently. Good explanations can be found in [1] and [21]; as a matter of fact, most of these orderings turn out to be equal, as discussed in [26]. So in the following we define approximation from a general comparison technique based on observation of convergence:

Definition 2.11 (Operational approximation)
Operational approximation $\underline{\sqsubset}$ *is defined as:*

$$(a \mathrel{\underline{\sqsubset}} b) \iff (\forall C[-], C[a]\Downarrow \implies C[b]\Downarrow)$$

Proposition 2.12

$$a \mathrel{\underline{\sqsubset}} b \implies \forall C[-], C[a] \mathrel{\underline{\sqsubset}} C[b]$$

Proof. Pick any context $C[-]$. For any context $D[-]$, if $D[C[a]]\Downarrow$ then $D[C[b]]\Downarrow$ because $a \mathrel{\underline{\sqsubset}} b$, so the conclusion follows immediately. □

Proposition 2.13
In error-preserving languages:

- $\forall a, \Omega \mathrel{\underline{\sqsubset}} a \mathrel{\underline{\sqsubset}} \varepsilon$
- $a\dagger^0 \implies a \stackrel{\Downarrow}{=} \varepsilon$
- $a \Uparrow \implies a \stackrel{\Downarrow}{=} \Omega$
- *if reduction is Church-Rosser and compatible, then* $a \stackrel{*}{=} b \implies a \stackrel{\Downarrow}{=} b$

Proof. Direct from definitions. □

A major question often related to operational approximation is whether the language under consideration satisfies a *context lemma*, which allows one to only inspect a restricted class of contexts, instead of arbitrary contexts, and therefore greatly simplifies proofs. This will not be discussed here, since proofs of the context lemma usually require the language to be fixed and cannot be done at an abstract level of presentation. However recent progress has been made towards generalizations of the context lemma; interested readers may consult [21, 24].

This completes the abstract presentation of the language family. In the following we will restrict our attention to languages satisfying most of the "good" properties defined above, namely stuck-free, error-preserving and Church-Rosser.

3 Labelled reduction

This section borrows from Chapter 14 of [4] the idea of *labelled reductions*, originally due to Hyland and Wadsworth, which is used here as a technical tool for reasoning syntactically about finite approximations. The motivation for introducing this notion is twofold: first, it will be used to formalize the concept of "interaction" between a term and its surrounding context, which is a necessary step towards our definition of "erroneous" terms; second, syntactic approximations can replace denotational constructions such as the embedding-projection pairs of [9] to interpret recursive types. Our presentation here generalizes [4] in the sense that it abstracts from any reference to a particular syntax.

3.1 Labelled languages

Labelled terms are obtained from usual terms by decorating subterms with natural numbers. For example

$$(((\lambda x.x^4)^{3^1})(\lambda yz.y^3))^{2^0}$$

is a labelled λ-term. A label can be seen intuitively as a finite amount of computing power, or rather "interaction power"; when this amount has been totally consumed, the corresponding term becomes divergent. Subterms without any label therefore have an infinite interaction power. Since it is sometimes convenient to consider that every subterm has a label (esp. in definition $(lab4)$ below), we also allow explicit decorations with label ∞, which are obviously equivalent to absent labels. We write $\tilde{a}, \tilde{b}, \ldots, \tilde{C}[-], \tilde{D}[-], \ldots$ for labelled terms and contexts, and $\tilde{\mathcal{T}}$ for the set of labelled terms. Clearly $\mathcal{T} \subset \tilde{\mathcal{T}}$ because a labelling may be empty. We write $\tilde{a} \leftarrow\!\!\!\!\!\sim\; \tilde{b}$ iff $\tilde{a}$ is obtainable by adding labels into the (possibly already labelled) term $\tilde{b}$. Finally, $|\tilde{a}|$ is the unlabelled term obtained by erasing all labels in $\tilde{a}$.

Given a set $\mathcal{V} \subset \mathcal{T}$ of values, we define the set of labelled values $\tilde{\mathcal{V}}$ as

$$\{\tilde{v} \mid \ |\tilde{v}| \in \mathcal{V} \wedge (\tilde{v} \equiv \tilde{C}[(\tilde{a})^0] \implies |C|[\Omega] \in \mathcal{V})\}$$

In other words, labelled values can contain 0 labels only in places where the corresponding subterm, replaced by a divergent term, still yields a value in the original language: this is typically the case in lazy computation systems [20], in which the outermost term constructor is enough to determine whether a term is a value or not.

Definition 3.1
Labelled reduction $\tilde{\to}$ *is the least binary relation on* $\tilde{\mathcal{T}}$ *which includes* $\to$ *and furthermore satisfies:*

$$\begin{array}{lrcl}
(lab1) & \tilde{C}[(\tilde{a}^m)^n] & \tilde{\to} & \tilde{C}[\tilde{a}^{\min(m,n)}] \\
(lab2) & \tilde{C}[\tilde{a}^0] & \tilde{\to} & \tilde{C}[\Omega] \\
(lab3) & \tilde{C}[\tilde{a}^n] & \tilde{\to} & \tilde{C}[\tilde{b}^n] \quad \textit{if } \tilde{C}[\tilde{a}] \tilde{\to} \tilde{C}[\tilde{b}]
\end{array}$$

$$(lab4) \quad \tilde{C}[F(\tilde{a}_1^{n_1+1} \ldots \tilde{a}_k^{n_k+1})] \ \tilde{\to} \ \tilde{C}[(F(\tilde{a}_1 \ldots \tilde{a}_k))^{\min(\{n_1 \ldots n_k\})}] \quad \textit{if}$$

$$\begin{cases} \tilde{C}[F(\tilde{a}_1 \ldots \tilde{a}_k)] \tilde{\to} \tilde{C}[\tilde{b}] \\ \neg \exists \tilde{b}' \leftarrow\!\!\!\!\!\sim\; \tilde{b}, \tilde{C}[F(\tilde{a}_1^{n_1+1} \ldots \tilde{a}_k^{n_k+1})] \tilde{\to} \tilde{C}[\tilde{b}'] \end{cases}$$

where symbols $m, n, \ldots$ in the rules above range over natural numbers or ∞. According to conditions $(lab1 - 2)$, terms with multiple labels select the smallest one, and terms with 0 label become divergent. Condition $(lab3)$ is just a contextual closure, specifying that labels around a redex do not prevent the reduction step to occur. The difficult part of the definition is rule $(lab4)$, which has been designed to cover cases where a label crosses a redex; such cases precisely correspond to the situation where a function symbol "interacts" with its context, and therefore the label should decrease by one. After application of the rule, the blocking label(s) has (have) moved upwards; the redex becomes a usual, unlabeled redex and can contract as normal. In most practical cases the number of subterms of F participating in the reduction step will be just one, which greatly simplifies the rule: either $k = 1$ or most labels in $\{n_1 \ldots n_k\}$ are "absent" (i.e. equal to ∞). In particular, reduction steps in all extended λ-calculi considered in this paper depend on one single syntactic construct (function symbol) below F. However the general formulation of $(lab4)$ as given above would be required to study for example process calculi; Example 3.4 below is a short illustration of a case where a reduction step depends on two function symbols.

Example 3.2
In order to satisfy the specification above, labelled β-reduction on the pure λ-calculus can be expressed by the usual β-rule

$$(\lambda x.\tilde{a})\tilde{b} \stackrel{\cdot}{\to} \tilde{a}[x := \tilde{b}]$$

together with a labelled β-rule

$$(\lambda x.\tilde{a})^{n+1}\tilde{b} \stackrel{\cdot}{\to} ((\lambda x.\tilde{a})\tilde{b})^n$$

which then reduces to $(\tilde{a}[x := \tilde{b}])^n$. The correspondence with the definition above is not immediate, since the function symbol corresponding to F is hidden by the usual applicative notation of the λ-calculus. To clarify things, we could adopt for example the notation of of higher-order rewrite systems [23], where β-reduction is expressed as $@(\lambda([x]Z(x)), Z') \to Z(Z')$. In that case the labelled β-rule is written as:

$$@(\lambda([x]Z(x))^{n+1}, Z') \stackrel{\cdot}{\to} (@(\lambda([x]Z(x)), Z'))^n \to (Z(Z'))^n$$

and it becomes easier to see that this formulation satisfies the $(lab4)$ specification. Two additional points are important to note about this labelled β-rule:

- The argument to the function, i.e. the metaterm $\tilde{b}$, has no label. This is because $\tilde{b}$ is just a passive participant in the reduction step. A rule

  $$(\lambda x.\tilde{a})^{n+1}\tilde{b}^{m+1} \stackrel{\cdot}{\to} (\tilde{a}[x := \tilde{b}])^{\min(n,m)}$$

  would violate the side condition of $(lab4)$ in case $n = \infty$, and furthermore would destroy confluence, as can be shown through example $(\lambda xy.y)\tilde{b}^1$ which would reduce to both $\mathbf{I}$ and Ω.

  The fact that metaterm $\tilde{b}$ in the rule has no label induces no loss of generality, since labels may appear inside $\tilde{b}$, even at the root of the syntax tree; however such labels will simply not participate actively in the reduction step.

- Our definition of labelled reduction is sligthly different from the one in [4], which reads:

$$(\lambda x.a)^{n+1}b \to (a[x := b^n])^n$$

There the argument b is also labelled by n after the reduction step. In the case of [4] the main goal is to prove strong normalization and therefore it is helpful to limit the computational power of b. In our case the goal is to study interactions between terms and contexts; for this purpose the fact that b could be divergent is just appropriate, since a divergent term no longer interacts with anything.

Example 3.3
In a record calculus, the field extraction rule

$$\{l_1 = \tilde{a}_1 \ldots l_k = \tilde{a}_k\}.l \to \tilde{a}_i \quad \text{if } \exists i, l \equiv l_i$$

has corresponding labelled rule

$$\{l_1 = \tilde{a}_1 \ldots l_k = \tilde{a}_k\}^{n+1}.l \xrightarrow{\sim} \tilde{a}_i^n$$

Here again the usual record syntax does not immediately match the $(lab4)$ specification; however this is merely a matter of presentation. For any collection $l_1 \ldots l_k$ of field names we can have a corresponding k-ary function symbol $R_{l_1 \ldots l_k}$ for record construction; and the symbol corresponding to F in the rule is the field selection operation $(.l)$ of arity 1.

Example 3.4
Reduction rules in process calculi often have a shape like

$$c(x)?p \parallel c(v)!q \to (p[x := v] \parallel q)$$

Here both subterms under the $\parallel$ function symbol are "active" in the reduction, and therefore should be taken into account for labelled reduction. In consequence a corresponding labelled rule would be

$$(c(x)?\tilde{p})^{m+1} \parallel (c(v)!\tilde{q})^{n+1} \xrightarrow{\sim} (c(x)?\tilde{p} \parallel c(v)!\tilde{q})^{\min(m,n)} \to (\tilde{p}[x := v] \parallel \tilde{q})^{\min(m,n)}$$

Process calculi will not be studied in this paper; this was just intended as a different illustration of condition $(lab4)$ for labelled reduction.

Proposition 3.5
If $\mathcal{L} \equiv (\mathcal{T}, \mathcal{V}, \to)$ is stuck-free, and with Church-Rosser reduction, then so is its labelled extension. $(\tilde{\mathcal{T}}, \tilde{\mathcal{V}}, \xrightarrow{\sim})$.

Proof.

- $\tilde{\mathcal{T}}$ contains no stuck terms: every $\tilde{a} \in \tilde{\mathcal{T}}$ can be built as a finite sequence $\tilde{a}_0, \tilde{a}_1, \ldots$, where $\tilde{a}_0 \in \mathcal{T}$ is unlabelled, and each $\tilde{a}_{i+1}$ is obtained by adding a label decoration to exactly one subterm of $\tilde{a}_i$. We reason inductively on i. a_0 is not stuck because $\mathcal{L}$ is stuck-free, $\mathcal{V} \subseteq \tilde{\mathcal{V}}$ and $\to \subseteq \xrightarrow{\sim}$. Now suppose $\tilde{a}_i$ is not stuck, and take any decomposition $\tilde{a}_i \equiv \tilde{C}[\tilde{b}]$. If $\tilde{a}_i \in \tilde{\mathcal{V}}$, then by definition $\tilde{C}[\tilde{b}^{k+1}] \in \tilde{\mathcal{V}}$ for any k. Furthermore if $\tilde{C}[-]$ is relevant, then $\tilde{C}[\tilde{b}^0] \Uparrow$, or otherwise $\tilde{C}[\tilde{b}^0] \in \tilde{\mathcal{V}}$. So in any case $\tilde{a}_{i+1}$ is not stuck. If, on the other hand, $\tilde{a}_i \notin \tilde{\mathcal{V}}$, then it must reduce, and by inspection of the labelled rules $\tilde{C}[\tilde{b}^k]$ must also reduce for any k, so again $\tilde{a}_{i+1}$ is not stuck. Hence by induction no labelled term is stuck.

- $\tilde{\to}$ is Church-Rosser: like in [4], p.353, every labelled term has a unique $(lab1)$-normal form because the function $min(m, n)$ is associative and decreases the length of a term. Now since $\to \subseteq \tilde{\to}$ and is CR, we only need to consider new critical pairs introduced by rules $(lab2-4)$. Rule $(lab2)$ is defined on all terms and is the only rule erasing 0 labels; therefore whenever $\tilde{a}^0 \tilde{\to} \Omega$ and $\tilde{a}^0 \tilde{\to}^* \tilde{b}^0$ we also have $\tilde{b}^0 \tilde{\to} \Omega$. Rule $(lab3)$ is just a contextual closure rule and therefore overlaps with no other rule. Finally rule $(lab4)$ does not overlap with the other rules, because it only considers labels under a function symbol F, and the side condition prevents any overlap with the basic reduction relation $\to$.

Note that $\tilde{\mathcal{L}}$ is never error-preserving, as can be seen easily by a context like $([-]^1\mathbf{I})$ which diverges when filled with ε.

In the following we no longer need to precisely distinguish between the basic language and its labelled extension, so the $\tilde{}$ decorations will be dropped.

Lemma 3.6

1. $C[a^n] \Downarrow \implies C[a^{n+1}] \Downarrow$
2. $C[a^n] \Downarrow \implies C[a] \Downarrow$
3. $C[a] \Downarrow \implies \exists n. C[a^n] \Downarrow$
4. $a \Downarrow \implies a^1 \Downarrow$

Proof.

1. Induction on the length of reductions. If $C[a^n]$ is a value then $C[a^{n+1}]$ is a value too. If $n \equiv 0$ and $C[a^0] \Downarrow$, then $C[-]$ is irrelevant and therefore converges for any term, including a^1. Otherwise, suppose $C[a^n] \to b$. By inspection of the rules there are $D[-], c, m \leq n$ such that $b \equiv D[c^m]$, and $C[a^{n+1}] \to D[c^{m+1}]$, and by applying the induction hypothesis we reach the conclusion.
2. Like 1.
3. Like 1 (since the length l of the reduction of $C[a]$ to a value is finite, it suffices to pick n greater than l).
4. If $a \tilde{\to}^* v$ then $a^1 \tilde{\to}^* v^1$ which after reduction to $(lab1)$-normal form must be a value.

□

3.2 Finite relevance and interactivity

We will now use labelled reductions as as a general, abstract mechanism to replace the language-dependent finite projection functions of [3, 26, 1]. In this subsection we classify both contexts and terms according to the number of interaction steps they can perform, as measured by the labelled reduction relation.

Definition 3.7 (k-relevant contexts)

1. *A context* $C[-]$ *is* k-relevant *iff* $(a \Uparrow \implies C[a] \Uparrow)$ *and there is a term* b *such that* $C[b^{k+1}] \Downarrow$.
2. *The* relevance index *for* $C[-]$, *written* $RI(C[-])$, *is the smallest* k *such that* $C[-]$ *is* k*-relevant, or undefined if there is no such* k.
3. $\mathcal{C}^k$ *denotes the set* $\{C[-] \in \mathcal{T}[-] | RI(C[-]) = k\}$.

The notion of k-relevance captures the number of interaction steps between a context and the term filling it. 0-relevant contexts are contexts which only carry the hole around without interacting with it, like $[-]$, $(\mathbf{I}[-])$ or $(\{l = [-]\}.l)$; 1-relevant contexts include the 0-relevant ones, but in addition also include contexts like $([-]\mathbf{I})$ or $([-].l)$ which perform one single interaction step with the hole. More generally, we have:

Lemma 3.8

1. *Any* k*-relevant context is also* $(k+1)$*-relevant.*
2. *A context is relevant iff it is* k*-relevant for some* $k \geq 0$.

Proof. Direct consequences of Lemma 3.6. □

Lemma 3.9 (context decomposition)

$$C[-] \in \mathcal{C}^{k+1} \implies \exists C_1[C_2[-]] \stackrel{*}{=} C[-], C_1[-] \in \mathcal{C}^1 \wedge C_2[-] \in \mathcal{C}^k$$

Proof. If $k = 0$, there is an easy solution $C_1[-] \equiv C[-], C_2[-] \equiv [-]$. If $k > 0$, we know i) $\exists a, v, C[a^{k+2}] \stackrel{*}{\to} v$ and ii) $\forall b, C[b^{k+1}] \Uparrow$. Suppose $v \equiv C'[a'^{k+2}]$, with $C[-] \stackrel{*}{\to} C'[-], a \stackrel{*}{\to} a'$. Then by definition $C'[a'^{k+1}]$ must be a value, contradicting ii). So necessarily

$$C[a^{k+2}] \stackrel{*}{\to} D_1[D_2[a'^{k+2}]] \to D_1[b^{k+1}] \stackrel{*}{\to} v$$

where $D_2[a'^{k+2}] \to b^{k+1}$ is an instance of $(lab4)$. Now by rule $(lab1)$, $D_1[(D_2[a'^{k+2}])^{k+1}] \stackrel{*}{\to} D_1[(b^{k+1})^{k+1}] \stackrel{*}{\to} v$, so $D_1[-] \in \mathcal{C}^k$; moreover b^{k+1} must converge and $(b^{k+1})^1 \stackrel{*}{\to} b^1$, so by Lemma 3.6 b^1 must converge too, and therefore $D_2[a'^2] \to b^1 \Downarrow$, which implies $D_2 \in \mathcal{C}^1$. □

Corollary 3.10

$$C[-] \in \mathcal{C}^k \implies \exists C_1[...C_k[-]] \stackrel{*}{=} C[-], (\forall i, C_i[-] \in \mathcal{C}^1)$$

Now we can use relevance indices of contexts to measure the interactivity of terms; intuitively, a term is k-interactive if it can performe k interaction steps.

Definition 3.11 (k-interactivity)

1. *every term is* 0-interactive
2. a *is* $(k+1)$-interactive *iff* $\exists C[-] \in \mathcal{C}^k, C[a] \Downarrow$.

3. *the* interactivity index *of a term* a, *written* $II(a)$, *is the biggest* k *such that* a *is* k*-interactive, or* ∞ *if* a *is* k*-interactive for every* k.

4. $\mathcal{T}^k$ *denotes the set* $\{a \in \mathcal{T} | II(a) \leq k\}$.

Example 3.12

- In the lazy λ-calculus [1] all λ-abstractions are values, so the term $\lambda x.\Omega$ is 1-interactive, as well as $(\lambda x.a)^1$ for any function $\lambda x.a$.

- In the classical call-by-name λ-calculus, the term $\lambda x.x\Omega$ is 1-interactive.

As demonstrated by these examples, the notion of k-interactivity not only applies to labelled terms, but also to unlabelled ones. Labels are used as an auxiliary study tool, but then the results can be extracted and give information about the unlabelled language.

Lemma 3.13

1. *if* a *is* $(k+1)$*-interactive then it is also* k*-interactive*

2. $\mathcal{T}^k \subseteq \mathcal{T}^{k+1}$

3. $II(a) = 0 \iff a \Uparrow$

Proof.

1. The case $k = 0$ is obvious. For $k > 0$, by definition there is a k-relevant context $C[-]$ with $C[a] \Downarrow$. By Lemma 3.9 $C[-] \xrightarrow{*} C_1[C_2[-]]$ where $C_2[-]$ is $(k-1)$-relevant and $C_2[a] \Downarrow$.

2. By definition.

3. By definition there is no relevant context $C[-]$ such that $C[a] \Downarrow$; hence a must be divergent. Conversely, if $a \Uparrow$ there can be no relevant context $C[-]$ such that $C[a] \Downarrow$, so the biggest interactivity level of a is 0. □

An interactive term is not necessarily solvable. For example the "ogre" $(\mathbf{YK})$ in the lazy λ-calculus is unsolvable, but has interactivity index ∞ because it can consume infinitely many arguments. Similarly, ε is also interactive.

Lemma 3.14

$$(\forall n, a^n \mathrel{\underline{\boxplus}} b^n) \iff (a \mathrel{\underline{\boxplus}} b)$$

Proof. ($\Leftarrow$): comes directly from Proposition 2.12, taking the context $[-]^n$. ($\Rightarrow$): For any context $C[-]$, if $C[a] \Uparrow$, then the Lemma holds vacuously. On the other hand if $C[a] \Downarrow$, then by Lemma 3.6(3) there is an n such that $C[a^n] \Downarrow$. Since $a^n \mathrel{\underline{\boxplus}} b^n$, we have $C[b^n] \Downarrow$ and therefore by Lemma 3.6(2) $C[b] \Downarrow$. □

Like in [26], syntactic approximations as expressed by labelled terms could now be used to prove a whole range of semantic properties usually requiring denotational techniques; a striking example in [26] is a fixpoint induction theorem. Since this is not the main purpose, it will not be developed further here; however we would like to point that labelled reduction is an appropriate and general technique to perform such kind of studies.

4 Erroneous Terms and Subsumption

We want to allow some errors to occur inside terms, because of the assumption that these will not necessarily be propagated to the top level. However, if a term contains *only* errors, then it is observationally not different from an error itself. For example, the term $\lambda x.\varepsilon$ is not β-equal to ε, but only yields errors in any context; therefore we are aiming at a semantics in which $\varepsilon = \lambda x.\varepsilon$. This does *not* mean that all unsolvable terms are errors: terms like $\lambda x.\Omega$, $\mu x.\lambda y.x$, $\mu x.\{l = x\}$ are all unsolvable, but can interact with some contexts without ever generating errors. Hence we come to define the erroneous terms are those which always yield errors after a *finite* number of interaction steps:

Definition 4.1 (Erroneous terms)
*A term a is k-*erroneous, *written $a\dagger^k$, iff $C[a] \xrightarrow{*} \varepsilon$ for every context $C[-] \in \mathcal{C}^k$. A term a is* erroneous, *written $a\dagger$, iff it is k-erroneous for some k.*

Clearly 0-erroneous terms must belong to the class $\{a | a \xrightarrow{*} \varepsilon\}$. Examples of 1-erroneous terms are $\lambda x.\varepsilon$ or $\{l = \varepsilon\}$.

Proposition 4.2
In Church-Rosser, error-preserving languages:

1. $[a\dagger \wedge a \xrightarrow{*} b] \implies b\dagger$
2. $a\dagger \implies \forall C[-] \in \mathcal{C}, C[a]\dagger$

Proof.

1. For any context $C[-]$ with $C[a] \xrightarrow{*} \varepsilon$, $C[b]$ and ε must have a common reduct. But ε has no other reduct than itself and hence $C[b] \xrightarrow{*} \varepsilon$.
2. By definition there must be some k, n such that a is k-erroneous and $C[-] \in \mathcal{C}^n$. If $k \leq n$ then $C[a] \xrightarrow{*} \varepsilon$; otherwise $C[a]$ must be $(k-n)$-erroneous and therefore is erroneous.

Corollary 4.3
Erroneous terms are unsolvable.

Now we are ready to define subsumption.

Definition 4.4 (Subsumption)
A term a subsumes *another term b, written $a \sqsubseteq b$, iff it generates fewer errors in all program contexts:*

$$a \sqsubseteq b \iff \forall C[-], C[a]\dagger \implies C[b]\dagger$$

Proposition 4.5
In error-preserving languages:

- $\forall a, \Omega \sqsubseteq_{\varepsilon} a \sqsubseteq_{\varepsilon} \varepsilon$
- $a\dagger \implies a \stackrel{\varepsilon}{=} \varepsilon$
- $a \Uparrow \implies a \stackrel{\varepsilon}{=} \Omega$

Proof. Direct from definitions.

All the properties above are also valid for $\sqsubseteq$ (see Proposition 2.13). The obvious question then is how the two orderings relate. This in general depends on the language properties, as shown through several examples in the next section. Nevertheless, a general result can be stated already:

Theorem 4.6
In an error-complete language, $a \sqsubseteq_{\varepsilon} b \implies a \sqsubseteq b$.

Proof. We will show $(a \sqsubseteq_{\varepsilon} b) \implies (\forall C[-], C[b] \Uparrow \implies C[a] \Uparrow)$, from which $(a \sqsubseteq b)$ directly follows by definition. Suppose $a \sqsubseteq_{\varepsilon} b$. For any context $C[-]$, furthermore suppose $C[b] \Uparrow$ and $C[a] \Downarrow$. If the language is error-complete, then there exists a relevant context $D[-]$ with $D[C[a]]\dagger^0$; but since $D[-]$ is relevant, $D[C[b]] \Uparrow$, contradicting $a \sqsubseteq_{\varepsilon} b$. Hence $C[b] \Downarrow \implies C[a] \Uparrow$. □

5 Comparing various lambda calculi

We will now apply our abstract framework to several languages, all related to the λ-calculus, but with various kinds of extensions, and with two different notions of values: *head normal forms* (terms withouth a head redex) or *lazy values* (terms with an outermost abstraction construct). In order to proceed compositionally, language fragments are all described by several categories of rules (for term formation, for the reduction relation, and for value determination); then these rules are assembled to form the various languages. Since most rules are fairly standard they are given in the appendix. Head and lazy versions of languages are distinguished by the superscripts ^{H} and ^{L}.

For the pure λ-calculus Λ we recall from [4] that:

- terms in *head normal form* (HNF) are terms of the shape $\lambda x_1 \ldots x_n . x a_1 \ldots a_k$, where n and k may be 0, x is any variable (equal or not to one of the x_is), and $a_1 \ldots a_k$ are arbitrary terms.
- a unsolvable $\iff a \notin HNF$;
- an equational theory of the λ-calculus is *inconsistent* if it equates all λ-terms;
- no consistent theory can equate two different $\beta\eta$-normal forms (Böhm theorem).

Hence the only room for variation in consistent λ-theories is the treatment of unsolvables. A theory is *sensible* iff it equates all the unsolvables, and it is *semi-sensible* iff it does not equate a solvable term with an unsolvable one.

Since the λ-calculus has no errors, $\stackrel{\varepsilon}{=}$ clearly is inconsistent. By contrast, $\sqsubseteq$ on Λ^H is the usual approximation relation, and its reflexive closure $\stackrel{\Downarrow}{=}$ is the sensible theory of [4]; $\stackrel{\Downarrow}{=}$ on Λ^L is the semi-sensible, *lazy* theory of [1], which equates unsolvable terms of the same order. An example where the two theories differ is the status of the "ogre" $\mathbf{YK}$, which is an unsolvable term of infinite order (it can consume an infinite number of arguments). In Λ^H we have $\Omega \stackrel{\Downarrow}{=} \mathbf{YK} \sqsubseteq a$ for every a, while in Λ^L we have $\Omega \sqsubseteq a \sqsubseteq \mathbf{YK}$. A detailed discussion of these different relations can be found in [1].

A couple of basic lemmas on the pure λ-calculus will be useful in the developments below:

Lemma 5.1
$\lambda x.a \sqsubseteq \lambda x.b \iff a \sqsubseteq b$

Proof. ($\Longleftarrow$) is direct from Proposition 2.12. For the other direction, consider any context $C[-]$, and build context $D[-] \equiv C[([-])x]$. By approximation $D[\lambda x.a] \Downarrow \implies D[\lambda x.b] \Downarrow$, but $D[\lambda x.a] \stackrel{*}{\to} C[a]$, so $C[a] \Downarrow \implies C[b] \Downarrow$. □

Lemma 5.2
$\lambda x.a \sqsubseteq b \implies (b \stackrel{*}{\to} \lambda x.b') \wedge (a \sqsubseteq b')$

Proof. For each c, if $((\lambda x.a)c) \Downarrow$, then (bc) must converge. This is only possible if i) $b \stackrel{*}{\to} \lambda x.b'$ and $b'[x := c] \Downarrow$, or ii) $b \stackrel{*}{\to} yb_1 \ldots b_n$ for some variable y. Assuming the second case, in context $C[-] \stackrel{\text{def}}{=} (\lambda y.[-])\Omega$ we would have $C[b] \Uparrow$ and $C[\lambda x.a] \Downarrow$, contradicting the initial assumption. So b must reduce to $\lambda x.b'$ and then $a \sqsubseteq b'$ by Lemma 5.1. □

5.1 Standard λ-calculus with ε

Λ_ε is the pure λ-calculus with an added constant ε and corresponding reduction rule $\varepsilon a \to \varepsilon$.

Lemma 5.3
In Λ_ε, $a\dagger \iff a \stackrel{*}{\to} \lambda x_1 \ldots x_n.\varepsilon$

Proof. ($\Longleftarrow$): easy, $\lambda x_1 \ldots x_n.\varepsilon$ is n-erroneous. ($\Longrightarrow$): a must be k-erroneous for some k, so we can use induction on k. If $k = 0$ then $a \stackrel{*}{\to} \varepsilon$; otherwise, by analysis of the rules, $a \stackrel{*}{\to} \lambda x.a'$ with $a'\dagger^{k-1}$. By induction hypothesis $a' \stackrel{*}{\to} \lambda x_1 \ldots x_n.\varepsilon$ for some n, and hence $a \stackrel{*}{\to} \lambda x.\lambda x_1 \ldots x_n.\varepsilon$. □

Lemma 5.4

1. *Λ_ε is not error-generating, but is error-preserving.*
2. *$\sqsubseteq^H \subseteq \sqsubseteq_\varepsilon^H$ and $\sqsubseteq^L \subseteq \sqsubseteq_\varepsilon^L$.*
3. *$\mathbf{YK} \stackrel{\Downarrow^L}{=}_\varepsilon \varepsilon$.*

4. Λ_ε^H *is error-complete, but not* Λ_ε^L.

5. $\overset{\Downarrow}{=}_\varepsilon^H$, $\overset{\varepsilon}{=}_\varepsilon^H$, $\overset{\Downarrow}{=}_\varepsilon^L$ *and* $\overset{\varepsilon}{=}_\varepsilon^L$ *are consistent.*

Proof.

1. Easy by inspection of rules β and ε.

2. Because of 1), $C[a] \in \Lambda$ converges in $\to_{\beta\varepsilon}$ only if it also converges in $\to_\beta$.

3. From Proposition 2.13 we already know $\mathbf{YK} \mathrel{\Box\!\!\!=}_\varepsilon \varepsilon$, so we only have to show $\forall C[-], C[\varepsilon]\Downarrow \implies C[\mathbf{YK}]\Downarrow$. First observe that $\mathbf{YK} \xrightarrow{*} \lambda x.\mathbf{YK}\Downarrow$, and $(\mathbf{YK})a \xrightarrow{*} (\mathbf{YK})$. So every reduction sequence starting at $C[\varepsilon]$ can be mirrored by a sequence starting at $C[\mathbf{YK}]$: for each instance $\varepsilon a \to \varepsilon$ of the ε reduction rule, there is a corresponding step $\mathbf{YK}a \to \mathbf{YK}$.

4. Values in Λ_ε^H are λ-terms in head normal form, or ε. Since HNFs are solvable, for every v there is always a context $C[-]$ such that $C[v]\dagger^0$. By contrast, value $\lambda x.\Omega$ in Λ_ε^L never reduces to an error.

5. Recall from [4] that it suffices to show that $\mathbf{K}$ and $\mathbf{F} \overset{\text{def}}{=} \lambda xy.y$ are not equated. This is shown easily by taking the contexts $[-]\Omega\varepsilon$ and $[-]\varepsilon\Omega$.

□

Lemma 5.5

$$\mathrel{\Box\!\!\!=}_\varepsilon^H = \mathrel{\Box\!\!\!=}_\varepsilon^L$$

Proof. By the Lemma 5.3 the error terms in both calculi are the same. □

Theorem 5.6

1. $\mathrel{\Box\!\!\!\Downarrow}_\varepsilon^H \subseteq \mathrel{\Box\!\!\!=}_\varepsilon^H$ *and* $\mathrel{\Box\!\!\!\Downarrow}_\varepsilon^L \subseteq \mathrel{\Box\!\!\!=}_\varepsilon^L$.

2. $\mathrel{\Box\!\!\!\Downarrow}_\varepsilon^H = \mathrel{\Box\!\!\!=}_\varepsilon^H$.

Proof.

1. Suppose $a \mathrel{\Box\!\!\!\Downarrow} b$. By Lemma 5.3, for any context $C[-]$, if $C[a]\dagger$ then $C[a] \xrightarrow{*} \lambda x_1 \ldots x_n.\varepsilon$. Therefore by Lemma 5.2 $C[b] \xrightarrow{*} \lambda x_1 \ldots x_n.b'$ with $\varepsilon \mathrel{\Box\!\!\!=} b'$, so $C[b]\dagger$.

2. $(\subseteq)$:preceding part of the theorem. $(\supseteq)$: from Theorem 4.6, knowing that Λ_ε^H is error-complete. □

5.2 λ-calculus with records

The λ-calculus is now extended with records, i.e. collections of bindings from *names* to terms. As usual, these are written with curly braces; we use the vector notation $\{\overline{l_i = a_i}\}$ to denote the record with finite list of fields $l_1 = a_1, \ldots, l_n = a_n$, with all l_i distinct.

The value rule ρ in the appendix specifies that any record is a value in both the head- and the lazy-calculus. This is a debatable choice: obviously in a lazy calculus all records should be values, but in a head calculus one could imagine to require record fields to be in head normal form. We leave this question open for further studies; the choice made here seemed the most natural, as we know of no system which evaluates inside record fields. Similar choices have often been taken in calculi with tuples: for example in [26] even if evaluation is call-by-value, it stops when reaching a pairing construct.

In addition to the record values, records introduce new forms of head normal forms through the field selection construct: for example $\lambda x.((x\ a).l\ b)$ must be a value. This is expressed here by value rule σ, which interacts with rule χ of the basic calculus for the definition of head normal forms.

Lemma 5.7

1. $\Lambda_{\{\}}$ *is error-generating, error-complete and error-preserving for both the head and the lazy calculus.*
2. $\underline{\underline{\varepsilon}}^H_{\{\}} = \underline{\underline{\varepsilon}}^L_{\{\}}$.
3. $\underline{\underline{\Downarrow}}^H_{\varepsilon} \subseteq \underline{\underline{\Downarrow}}^H_{\{\}}$ *and* $\underline{\underline{\Downarrow}}^L_{\varepsilon} \subseteq \underline{\underline{\Downarrow}}^L_{\{\}}$.

Proof.

1. Error-generating: obvious. Error-complete: each closed value is either of record shape or of functional shape. In each case there is a context $([-]a)$ or $[-].l$ which generates an error. Error-preserving: easy by inspection of the reduction rules.
2. As for Λ_ε (Lemma 5.5): the error terms are the same (although the proof here is slightly more complex, as error terms may also be of record shape).
3. Induction on the shape of contexts. □

Since now even the lazy calculus is error-complete, the "ogre" $\mathbf{YK}$ has a different status than in Λ_ε:

Proposition 5.8
In $\Lambda_{\{\}}$, $\neg(\mathbf{YK} \stackrel{\Downarrow}{=} \varepsilon)$

Proof. Because $\Lambda^L_{\{\}}$ is error-complete and because of Theorem 4.6, it suffices to show $\neg(\mathbf{YK} \stackrel{\varepsilon}{=} \varepsilon)$. In the empty context $[-]$, there is no k such that $\mathbf{YK}$ is k-erroneous, because it can consume an infinite number of arguments without yielding an error. □

On the other hand there is a new term which is erroneous, namely the empty record:

Proposition 5.9
In $\Lambda_{\{\}}$. $\{\} \stackrel{\epsilon}{=} \varepsilon$

Proof. By inspection of the reduction rules, $\{\}$ cannot interact without yielding an error, so it is 1-erroneous. □

5.3 Extensible records

If objects are modelled as records of functions, modelling inheritance requires an operation to extend or modify some fields. We write $a \Leftarrow l = b$ for the record a in which field l has been added or overwritten with value b; this is exactly like the **with** construct of [29, 33]. The set of record values does not change, but the set of head normal forms does: the extension construct introduces new head normal forms like $\lambda x.x \Leftarrow l = b$.

Interestingly, the status of the empty record changes when records are extensible:

Proposition 5.10
In $\Lambda_{\{\}\Leftarrow}$, $\neg(\{\} \stackrel{\epsilon}{=} \varepsilon)$

Proof. Through the extension construct the empty record is solvable: for any value v there is a relevant context $([-] \Leftarrow l = v).l$ yielding that value when filled with the empty record. □

We state without proof the following laws for records and record extensions

Proposition 5.11

$$\begin{array}{rcll} \{\overline{l_i = a_i}\} & \sqsubseteq & \{\overline{l_j = b_j}\} & \text{if } \overline{l_j} \subseteq \overline{l_i} \text{ and } a_i \stackrel{\epsilon}{=} b_j \text{ when } l_i = b_j \\ \{l = \varepsilon, \overline{l_i = a_i}\} & \stackrel{\epsilon}{=} & \{\overline{l_i = a_i}\} & \end{array}$$

$$\begin{array}{rcl} n((a \Leftarrow l_1 = b_1) \Leftarrow l_2 = b_2) & \stackrel{\epsilon}{=} & ((a \Leftarrow l_2 = b_2) \Leftarrow l_1 = b_1) \\ ((a \Leftarrow l = b_1) \Leftarrow l = b_2) & \stackrel{\epsilon}{=} & (a \Leftarrow l = b_2) \\ \{\} \Leftarrow l = b & \sqsubseteq & \{\} \end{array}$$

Showing these laws requires induction on the size of contexts, which might be difficult for arbitrary contexts. However, one can show that all calculi above satisfy the context lemma (operational extensionality, "ciu theorem" of [26]); the proof proceeds by induction on the length of computations – see for example [1]. Thanks to this lemma, it then suffices to check a restricted class of "applicative contexts" and the proof of the laws above becomes easy. Complete developments and more detailed discussions for record extension constructs can be found in [12, 14].

6 Interpreting Types

This section illustrates the usefulness of both subsumption and labelled reductions for the semantics of types : subsumption is a natural foundation for interpreting subtyping, and labelled

$$
\begin{array}{rcl}
\mathbf{Ti}[T]^0_\eta & - & \{a | a \sqsubseteq \Omega\} \\
\mathbf{Ti}[\top]^{n+1}_\eta & - & \mathcal{T}^{n+1} \\
\mathbf{Ti}[X]^{n+1}_\eta & = & \eta(X)^{n+1} \\
\mathbf{Ti}[T \to U]^{n+1}_\eta & = & \{a \in \mathcal{T}^{n+1} | b \in \mathbf{Ti}[T]^n_\eta \implies a(b) \in \mathbf{Ti}[U]^n_\eta\} \\
\mathbf{Ti}[\{\overline{l_i : T_i}\}]^{n+1}_\eta & = & \{a \in \mathcal{T}^{n+1} | \forall i, a.l_i \in \mathbf{Ti}[T_i]^n_\eta\} \\
\mathbf{Ti}[\mu X.T]^{n+1}_\eta & = & \mathbf{Ti}[T]^{n+1}_{\eta[X \mapsto \mathbf{Ti}[\mu X.T]^n_\eta]} \\
\mathbf{Ti}[T]_\eta & = & \{a | \forall n \in \omega. a^n \in \mathbf{Ti}[T]^n_\eta\}.
\end{array}
$$

Figure 1: Type interpretation for functions and records

terms are a natural foundation for interpreting recursive types, following the approach of [9]. This is just an appetizer, as lack of space prevents us from going through full technical developments. Nevertheless the general approach borrows well-known techniques and therefore should be easy to follow.

Types are interpreted as non-empty, downward-closed subsets of terms in the $\sqsubseteq$ ordering. Let $\mathbf{Tset}$ denote the set of such subsets. For any $t \in \mathbf{Tset}$, t^n denotes the set $\{a^n | a \in t\}$ (finite projection). A *type environment* η is a mapping from $\mathbf{Tvar}$ to $\mathbf{Tset}$. Given a type environment, a type interpretation function $\mathbf{Ti}[-]$ maps types to members of $\mathbf{Tset}$. We will illustrate this approach on the $\Lambda_{()}$ calculus of the previous section, considering types of the following syntax.

$$T, U ::= \top \mid X \mid T \to U \mid \{\overline{l_i : T_i}\} \mid \mu X.T$$

For the sake of simplicity we do not consider extensible records, which require a more complex system with so-called row variables (see [29, 33]). Type assignment rules and subtyping rules are not displayed here: standard rules are assumed (see for example [11]). We also assume a rule (*top*) assigning type $\top$ to *any term*. Figure 1 gives the type interpretation. A well-known difficulty associated with recursive types is the fact that arrow types are contravariant on the left. The ideal model of [25] solves the problem through contractive maps on ideals in the semantic domain; this requires some conditions on the syntax of type expressions to enforce contractiveness. By contrast we follow here the idea of [9], using a family of indexed type interpretations, where the index denotes finite approximations. In this approach non-contractive type expressions are naturally mapped to the bottom type (the one containing only divergent terms), without any syntactic constraints. With labelled terms this can be done in an operational way, without needing to resort to denotational semantics.

Lemma 6.1
$\forall T, \eta, \mathbf{Ti}[T]_\eta \in \mathbf{Tset}$.

Proof. Induction on T □

Usually the goal of a type system is to prevent runtime errors. Here because of the *top* rule

every term is typable, including erroneous ones. However, simple inspection of the type syntax can rule out the types containing errors:

Definition 6.2 (Trivial types)
The set Triv *of trivial types is defined inductively as:*

- $\top \in \mathbf{Triv}$
- $U' \in \mathbf{Triv} \implies T \to U' \in \mathbf{Triv}$
- *if* $T \equiv \{\overline{l_i : T_i}\}$ *and* $\forall i, T_i \in \mathbf{Triv}$, *then* $T \in \mathbf{Triv}$
- *if* $T \in \mathbf{Triv}$, *then* $\mu X.T \in \mathbf{Triv}$

Lemma 6.3
In any non-trivial type environment, non-trivial types do not contain erroneous terms. (η is non-trivial iff $\varepsilon \notin \eta(X)$ for each type variable X in dom(η)).

Proof. Induction on the shape of types. □

Lemma 6.4
The following equality between record types is sound:

$$\{l : \top, \overline{l_i : T_i}\} = \{\overline{l_i : T_i}\}$$

Proof. Since $\varepsilon \in \mathbf{Ti}[\top]$, the condition $a.l_i \in \mathbf{Ti}[T_i]$ on field l is always satisfied, even for records where field l is absent. □

Example 6.5
The example of the introduction

$$(T\ R).\texttt{affiche}, \quad \text{where} \quad \begin{cases} T \stackrel{\text{def}}{=} (\lambda x.\{\texttt{imprime} = x.\texttt{print}, \texttt{affiche} = x.\texttt{display}, \texttt{ferme} = x.\texttt{close}\}) \\ R \stackrel{\text{def}}{=} \{\texttt{display} = \texttt{"hello"}\} \end{cases}$$

has type **String** because $(T\ R)$ has type $\{\texttt{imprime} : \top, \texttt{affiche} : \mathbf{String}, \texttt{ferme} : \top\}$, which is non-trivial and equal to $\{\texttt{affiche} : \mathbf{String}\}$.

7 Conclusion

We have designed a framework accounting for some "lazyness" in the treatment of errors. This results in increased flexibility for typing modular systems: as demonstrated by our example, it may be sometimes useful to combine modules even if they do not match perfectly. The combination may contain errors, but also "good" values; in contexts for which we can prove that only the good values are accessed, the overall software configuration is valid. Being able to support such increased flexibility is an important benefit for dynamic software configurations. In contrast with traditional software systems where the whole configuration is known statically,

many recent systems involve distribution and mobility of code; in such cases the components assembled for a global computation must be able to cooperate even if they were not necessarily written originally to perfectly fit together.

In order to develop this semantic framework for errors, several steps were needed. Labelled reductions were used to formalize the notions of finite relevance for contexts, and finite interactivity for terms. On this basis we gave a definition of "erroneous" terms as those terms which always generate an error after a finite number of interaction steps. Then the subsumption ordering, stating when a term can "safely replace" another one, was defined as a simulation based on the observation of erroneous terms. This ordering is close to the more usual approximation ordering; both have Ω as bottom element and ε as top element, and in most languages (the ones which satisfy the "error-completeness" condition), subsumption is a coarser relation than approximation. These technical devices were brought together in an interpretation of types as ideals in the subsumption ordering; subsumption interprets subtyping while labelled reductions give an basis for induction in the interpretation of recursive types.

One fundamental limitation of this work is the basic assumption that errors are "uncatchable". The reason for this limitation is not technical: it is rather conceptual, and deeply bound to the very idea of subsumption, which is at the core of object-oriented programming. As soon as a language has a catch construct, it has an internal way of observing errors, and therefore an object with added methods can no longer be "safely" substituted for an object with fewer methods. For example in a context like

$$\mathbf{if}\ ([-].\mathtt{meth} = \varepsilon)\ \mathbf{then}\ 1\ \mathbf{else}\ -1$$

objects with or without the `meth` method are clearly incomparable, and inheritance is no longer a "safe" or "harmless" extension of previous software. Example 2.8 however discusses the fact that errors can be distinguished from exceptions, and that it is perfectly conceivable to have catchable exceptions together with uncatchable type errors. At a practical level the Java language [16] made such a choice; at a theoretical level the errors of [10], which are used to observe sequentiality, are treated like exceptions, so their *catch* construct is not incompatible with our approach.

References

[1] Samson Abramsky and C.-H. Luke Ong. Full Abstraction in the Lazy Lambda Calculus. *Information and Computation*, 105:159-267, 1993.

[2] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, Monographs in Computer Science, 1996.

[3] Martin Abadi, Benjamin Pierce and Gordon Plotkin. Faithful Ideal Models for Recursive Polymorphic Types. *Int. J. of Foundations for Computer Science*, 2(1):1-21, 1991.

[4] Henk Barendregt. *The Lambda-Calculus, its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics, North-Holland, 1984.

[5] Gérard Boudol. Lambda-Calculi for (Strict) Parallel Functions. *Information and Computation* 108:51-127, 1994.

[6] Baard Bloom. Can LCF Be Topped? Flat Lattice Models of Typed λ-calculus. *Information and Computation* 87:264-301, 1990.

[7] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as Implicit Coercion. *Information and Computation* 93:172-221, 1991. Also in [17], pp 197-245.

[8] Kim Bruce and Giuseppe Longo. A Modest Model of Records, Inheritance, and Bounded Quantification. *Information and Computation* 87:196-240, 1990. Also in [17], pp 151-195.

[9] Felice Cardone and Mario Coppo. Two extensions of Curry's Type Inference System. In *Logic and Computer Science*, P. Odifreddi(ed), pp 19-75. Academic Press, 1990.

[10] Robert Cartwright, Pierre-Louis Curien and Matthias Felleisen. Fully Abstract Semantics for Observably Sequential Languages. *Information and Computation* 111(2):297-401, 1994.

[11] Luca Cardelli and John Mitchell. Operations on Records. In [17], pp 295-350. First appeared in *Math. Structures in Comp. Sc.*, 1991, pp 3-48.

[12] Laurent Dami. A Lambda-Calculus for Dynamic Binding. To appear in *Theoretical Comp. Sc.*, special issue on Coordination, 1997.

[13] Laurent Dami. Labelled Reductions, Runtime Errors, and Operational Subsumption. To appear in *Proc. ICALP'97*, LNCS, Springer-Verlag, 1997.

[14] Laurent Dami. A Comparison of Record Calculi. In this report, ed. D. Tsichritzis, Université de Genève, 1997.

[15] Andrew Gordon and Gareth Rees. *Bisimilarity for a First-Order Calculus of Objects with Subtyping*. Technical Report 386, Computer Laboratory, University of Cambridge, January 1996, available at `ftp://ftp.cl.cam.ac.uk/papers/adg/TR386-adg-gdr-obj.ps.gz`. Technical summary in *Proceedings 23rd ACM POPL*, Jan 1996, pp 386-395.

[16] J. Gosling, B. Joy and G. Steele. *The Java™ Language Specification*. Addison-Wesley, 1996.

[17] Carl A. Gunter and John C. Mitchell, eds. *Theoretical aspects of object-oriented programming: types,semantics, and language design*. MIT Press, Foundations of computing series, 1994.

[18] P. Hudak et al. The Haskell Report and Haskell Tutorial. *ACM SIGPLAN Notices* 27(5), May 1992.

[19] Furio Honsell and Simonetta Ronchi della Rocca. An Approximation Theorem for Topological Lambda Models and the Topological Incompleteness of Lambda Calculus. *Journal of Compter and System Sciences*, 45:49-75, 1992.

[20] D. J. Howe. Equality in lazy computation systems. In *Proc. 4th IEEE Symp. on Logic in Comp. Sc.*, pp 198-203, 1989.

[21] Trevor Jim and Albert R. Meyer. Full Abstraction and the Context Lemma. *SIAM J. on Computing* 25(3):663-696, June 1996.

[22] Richard Kennaway, Vincent van Oostrom and Fer-Jan de Vries. Meaningless Terms in Rewriting. In *Algebraic and Logic Programming '96*, pp 254-268. LNCS 1139, Springer-Verlag, 1996.

[23] Jan W. Klop, Vincent van Oostrom and Femke van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121:279-308, 1993.

[24] Marina Lenisa. Semantic Techniques for Deriving Coinductive Characterizations of Observational Equivalences for λ-Calculi. Proc. *TLCA'97*, pp 248-266. LNCS 1210, Springer-Verlag, 1997.

[25] David MacQueen, Gordon Plotkin and Ravi Sethi. An Ideal Model for Recursive Polymorphic Types. *Information and Control*, 71:95-130, 1986.

[26] Ian A. Mason, Scott F. Smith and Carolyn L. Talcott. From Operational Semantics to Domain Theory. In *Information and Computation*, 128:26-47, 1996.

[27] Gordon Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125-159, 1975.

[28] Gordon Plotkin. T^{ω} as a universal domain. *J. Comput Systems Sci.*, 17:209-236, 1978.

[29] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proceedings ACM POPL'89*, pp 242-249. Also in [17], pp 67-96.

[30] Didier Rémy. Typing Record Concatenation for Free. In *Proceedings ACM POPL'92*, pp 166-176. ACM Press, 1992. Also in [17], pp 351-372.

[31] Dana Scott. Data types as lattices. *SIAM J. of Computing*, 5:522-587, 1976.

[32] C. Talcott, A Theory of Binding Structures and Applications to Rewriting. *Theoretical Computer Science*, 112:99-143, 1993.

[33] Mitchell Wand. Type Inference for Record Concatenation and Multiple Inheritance. *Information and Computation*, 93(1):1-15, 1991.

A Language Rules

Rules are named by greek letters reminding of the syntactic constructor involved, and the same letter is often reused for rules in different categories, e.g. there is a (β) reduction rule but also a (β) value rule or a (β) typing rule. Furthermore, letters decorated with a "'" indicate propagation or generation of errors; finally, letters enclosed in || are reduction rules for contextual closure.

A.1 Standard λ-calculus

Syntax	$(\xi)\dfrac{x \in \mathcal{X}}{x \in \mathcal{T}}$ $(\lambda)\dfrac{x \in \mathcal{X}, a \in \mathcal{T}}{\lambda x.a \in \mathcal{T}}$ $(\beta)\dfrac{a,b \in \mathcal{T}}{(ab) \in \mathcal{T}}$
Red. Rules	$(\beta)\dfrac{}{(\lambda x.a)b \to a[x := b]}$ $(\|\lambda\|)\dfrac{a \to b}{\lambda x.a \to \lambda x.b}$ $(\|\beta 1\|)\dfrac{a \to b}{(ac) \to (bc)}$ $(\|\beta 2\|)\dfrac{a \to b}{(ca) \to (cb)}$
Values	$(\xi)\dfrac{}{x \in \mathcal{H}}$ $(\beta)\dfrac{v \in \mathcal{H}, a \in \mathcal{T}}{(va) \in \mathcal{H}}$ $(\chi)\dfrac{v \in \mathcal{H}}{v \in \mathcal{V}}$ $(\lambda^H)\dfrac{v \in \mathcal{V}}{\lambda x.v \in \mathcal{V}}$ $(\lambda^L)\dfrac{a \in \mathcal{T}}{\lambda x.a \in \mathcal{V}}$

A.2 Standard λ-calculus with ε

Syntax	$(\varepsilon)\dfrac{}{\varepsilon \in \mathcal{T}}$
Red. Rules	$(\varepsilon)\dfrac{}{(\varepsilon a) \to \varepsilon}$
Values	$(\varepsilon)\dfrac{}{\varepsilon \in \mathcal{V}}$

A.3 λ-calculus with records

Syntax	$(\rho)\dfrac{\forall i, a_i \in \mathcal{T}}{\{\overline{l_i = a_i}\} \in \mathcal{T}} \qquad (\sigma)\dfrac{a \in \mathcal{T}}{a.l \in \mathcal{T}}$
Red. Rules	$(\sigma_\rho)\dfrac{\exists j, l \equiv l_j}{\{\overline{l_i = a_i}\}.l \to a_j} \qquad (\sigma_\varepsilon)\dfrac{\forall j, l \not\equiv l_j}{\{\overline{l_i = a_i}\}.l \to \varepsilon}$ $(\sigma_\lambda)\dfrac{}{(\lambda x.a).l \to \varepsilon} \qquad (\beta_\rho)\dfrac{}{(\{\overline{l_i = a_i}\}\ b) \to \varepsilon}$ $(\varepsilon_\sigma)\dfrac{}{\varepsilon.l \to \varepsilon} \qquad (\lvert\sigma\rvert)\dfrac{a \to a'}{a.l \to a'.l}$ $(\lvert\rho\rvert)\dfrac{a_i \to a_i'}{\{\ldots, l_i = a_i, \ldots\} \to \{\ldots, l_i = a_i', \ldots\}}$
Values	$(\rho 1)\dfrac{\forall i, a_i \in \mathcal{T}}{\{\overline{l_i = a_i}\} \in \mathcal{R}} \qquad (\rho 2)\dfrac{a \in \mathcal{R}}{a \in \mathcal{V}} \qquad (\sigma)\dfrac{a \in \mathcal{H}}{a.l \in \mathcal{H}}$

A.4 λ-calculus with records and record extensions

Syntax	$(\epsilon)\dfrac{a, b \in \mathcal{T}}{a \Leftarrow l = b \in \mathcal{T}}$
Red. Rules	$(\epsilon_\rho)\dfrac{}{\{\overline{l_i = a_i}\} \Leftarrow l = b \rightarrow \{l = b.(\overline{l_i = a_i} \setminus l)\}}$ $(\epsilon_\lambda)\dfrac{}{(\lambda x.a) \Leftarrow l = b \rightarrow \varepsilon}$ $(\lvert\epsilon 1\rvert)\dfrac{a \rightarrow a'}{a \Leftarrow l = b \rightarrow a' \Leftarrow l = b}$ $(\lvert\epsilon 2\rvert)\dfrac{b \rightarrow b'}{a \Leftarrow l = b \rightarrow a \Leftarrow l = b'}$ $(\varepsilon_\epsilon)\dfrac{}{\varepsilon \Leftarrow l = b \rightarrow \varepsilon}$
Values	$(\epsilon)\dfrac{a \in \mathcal{H}}{a \Leftarrow l = b \in \mathcal{H}}$