A Family Of Syntactic Logical Relations For The Semantics Of Haskell-Like Languages.

By: **Patricia Johann** & Janis Voigtlaender

Abstract

Logical relations are a fundamental and powerful tool for reasoning about programs in languages with parametric polymorphism. Logical relations suitable for reasoning about observational behavior in polymorphic calculi supporting various programming language features have been introduced in recent years. Unfortunately, the calculi studied are typically idealized, and the results obtained for them offer only partial insight into the impact of such features on observational behavior in implemented languages. In this paper we show how to bring reasoning via logical relations closer to bear on real languages by deriving results that are more pertinent to an intermediate language for the (mostly) lazy functional language Haskell like GHC Core. To provide a more fine-grained analysis of program behavior than is possible by reasoning about program equivalence alone, we work with an abstract notion of relating observational behavior of computations which has among its specializations both observational equivalence and observational approximation. We take selective strictness into account, and we consider the impact of different kinds of computational failure, e.g., divergence versus failed pattern matching, because such distinctions are significant in practice. Once distinguished, the relative definedness of different failure causes needs to be considered, because different orders here in-duce different observational relations on programs (including the choice between equivalence and approximation). Our main contribution is the construction of an entire family of logical relations, parameterized over a definedness order on failure causes, each member of which characterizes the corresponding observational relation. Although we deal with properties very much tied to types, we base our results on a type-erasing semantics since this is more faithful to actual implementations.

Patricia, Johann; Janis, Voigtlaender. (2009) "A Family Of Syntactic Logical Relations For The Semantics Of Haskell-Like Languages". *Information and Computation*, vol.207, no. 2 , pp. 341 - 368. ISSN: 0890-5401 Version Of Record Available At www.sciencedirect.com

1 Introduction

Typeful programming as identified by Cardelli [1] is currently one of the key approaches to producing safe and reusable code. Types serve as documentation of functionality (even as partial specifications) and can help to rule out whole classes of errors before a program is ever run. Typeful programming is particularly effective for pure functional languages such as Haskell [2], where it comes with powerful reasoning techniques connecting the types of functions to their possible observable behaviors. One such technique is the use of logical relations to reason about polymorphic programs.

Polymorphism is essential for reconciling strong static typing, which attempts to prevent the use of code in unfit contexts by assigning types that are as precise and descriptive as possible, with the goal of flexible reuse. Of the two kinds of polymorphism identified by Strachey [3] — namely, *parametric polymorphism* and *ad-hoc polymorphism* — we are interested in the former here; see the survey [4] for a refined taxonomy. Parametric polymorphism expresses the requirement that a certain functionality is offered for arbitrary types in a uniform manner. Intuitively, this means that the same algorithm is employed in instantiations of a polymorphically typed function at different concrete types. This intuitive uniformity condition was first formally captured by Reynolds [5] through the introduction of the notion of *relational parametricity*, which in turn rests on the concept of *logical relations* [6,7].

The fundamental idea underlying logical relations is to interpret types as relations (rather than as sets, possibly with additional structure). These relational interpretations are built by induction, starting from specific relations for a language's base types (if any), and obtaining interpretations for compound types by propagating relations along the type structure in an "extensional" manner. The key result to be proved for every logical relation constructed in this way is that every function expressible in the underlying language is related to itself by the relational interpretation of its type. This *parametricity theorem*, or certain generalizations of it, can then be used, for example, to derive useful algebraic laws (so-called "free theorems") about polymorphic functions solely from their types [8], or to establish the semantic correctness of efficiency-improving program transformations [9–13]. But for all such applications, the usefulness in practice depends on a good fit between the semantics of the functional language of interest and that of the typically reduced formal calculus for which parametricity results are proved.

Indeed, the applicability to real programming languages of parametricity results obtained for idealized calculi cannot be taken for granted. Simply assuming such applicability is actually quite dangerous, as can be seen from experience with the *selective strictness* feature of Haskell, a language which is otherwise nonstrict. Denotationally specified via the polymorphic primitive

$$\begin{aligned} &\mathit{seq} :: \forall\alpha\ \beta.\ \alpha \to \beta \to \beta \\ &\mathit{seq}\ \bot\ b = \bot \\ &\mathit{seq}\ \ a\ b = b \quad \text{if } a \neq \bot \end{aligned}$$

in the language definition [2], and routinely used by programmers to control the time and space behavior of their programs, selective strictness was determined early on to have detrimental effects on parametricity. Nevertheless, reasoning about Haskell programs typically took place as if this were not an issue. In fact, Haskell programs were automatically optimized by a compiler using parametricity-based program transformations whose correctness in the presence of selective strictness was a conjecture at best. In the worst case, this means that the compiler can "optimize" a perfectly functioning program into one that fails to terminate or terminates with a runtime error. Conditions under which this can be avoided were first established in [14,15]. These conditions were derived from a new logical relation for which the parametricity theorem holds even with respect to (a naive, but standardly accepted denotational model of) a sublanguage of Haskell which includes selective strictness. A more thorough account in terms of a polymorphic lambda calculus similar to that used as intermediate language in the Glasgow Haskell Compiler (GHC) was recently given in [16]. With the current paper we further advance a line of research whose ultimate goal is the development of appropriate tools for reasoning about parametricity properties of real programming languages rather than toy calculi.

The first new aspect we consider is that of distinguishing different causes of program failure. The notion of "undefined value" is, in some form, fundamental to any semantic treatment of both fixpoint recursion (which is actually the first challenge when extending relational parametricity from Reynolds' original setting to more realistic languages; it is met by Wadler [8] and Pitts [17]) and selective strictness. For example, the notion of "undefined value" is captured by the notation $\bot$ in the above specification for *seq*, where it stands for a nonterminating computation or a runtime error, such as might be obtained as the result of a failed pattern match. Indeed, it is quite common to conflate these different failure causes into a single denotation or observation, but in practice this is not satisfactory. For example, conflating different failure causes means that a program transformation that is claimed to be semantics-preserving may very well transform a nonterminating program into one that instead terminates with a runtime error, and vice versa, or may confuse different kinds of runtime errors. If this happens automatically in a compiler, a

debugging nightmare ensues. And this issue is very real. In particular, Haskell examples similar to the ones in [14,15] can be given for which the classical *foldr*/*build*-fusion rule of Gill et al. [9] exhibits this behavior of transforming one kind of error into another one. So it is completely unclear whether the preconditions (on the arguments to *foldr*) found in earlier work to guarantee total correctness of *foldr*/*build*-fusion in the presence of *seq* still do so when considering different failure causes as semantically different. And even partial correctness of *foldr*/*build*-fusion, in the sense that the program after transformation at least semantically approximates the original one, is no longer guaranteed. While in [14–16] it was established to hold unconditionally, the aforementioned examples show that *foldr*/*build*-fusion may transform arbitrary different failures into each other in either direction. So, no matter how different failure causes are ordered by our notion of semantic approximation, some instance of *foldr*/*build*-fusion will violate that order, and thus not even be partially correct.

Actually, this last observation raises an important question. Assuming we consider different failure causes as semantically different, should there at least be some semantic approximation order between them? For example, one might have the intuition that nonterminating programs are strictly less defined than programs that terminate with a runtime error, and that these are in turn strictly less defined than those that terminate after computing a proper value. On the other hand, one might prefer another order between different kinds of failure or wish to leave them completely incomparable to each other with respect to the approximation order. So the answer to the above question is a conscious design decision that must precede any study of, say, partial correctness of parametricity-based program transformations, and very much depends on the usage scenario (e.g., debugging vs. production cycle). Since we do not want to predetermine this design decision, we leave the precise ordering between different kinds of failure as abstract as possible throughout our technical development. Thus, rather than developing a single new logical relation, we instead develop *an entire family of logical relations* which is parameterized by a suitable preorder embodying the various choices to be made here.

The parameterization of our family of logical relations has a further advantage, since it allows us to deal with semantic equivalence and (either direction of) semantic approximation in a unified manner. Previous work on logical relations for polymorphic languages has dealt exclusively with either an equational setting [17–21] or an inequational one [14–16,22]. This has led to a certain repetition in proofs of the key results about the relevant logical relations, as well as in proofs of applications of these results. For example, two separate but similar proofs are given in [16] for the two directions of semantic approximation in the *foldr*/*build*-fusion rule. This inequational treatment is preferable to the single (again similar) proof of semantic equivalence in [20], because it establishes one of the two directions of semantic equivalence without preconditions,

and so more insight is gained. But it comes with a cost in the form of proof repetition. This cost can be avoided using the tools from the present paper, given that a single proof parameterized by an abstract preorder suffices, and results for equivalence and (different choices of) approximation can then be read off by just instantiating this parameter in different ways according to the above design decisions. The proof for *foldr*/*build*-fusion is given explicitly in Section 6 precisely to allow this comparison. The same observations could, of course, also be made for other applications of parametricity results, such as Wadler's free theorems.

The main technical innovation of this paper is the construction of our parameterized logical relation. To more faithfully model Haskell-like programming languages, this relation is constructed on top of a type-erasing semantics. Let us explain. We are interested in languages with strong static typing. For such languages, types naturally play an important role both during programming and in compilation. But at runtime, types serve no purpose at all, precisely because "(originally) well-typed programs do not go wrong". So there is no need to carry type information through to the execution phase, and indeed each and every Haskell compiler erases all type information (and language constructs only dealing with types) from a program somewhere in the compilation process. What might be perceived as just an implementation detail actually has important semantic consequences with respect to polymorphic functions. In Haskell, type generalization and specialization are implicit. That is, they neither occur in the term syntax of the language, nor carry computational content. In contrast, parametricity theorems are typically (even necessarily) proved for extensions of the Girard-Reynolds polymorphic lambda calculus [23,24], in which type generalization and specialization are explicit term formers. A semantic mismatch occurs as soon as the calculus allows fixpoint recursion and termination is made observable at polymorphic types. In this case, a distinction can be made between two equally typed polymorphic terms, the first of which is diverging, and the second of which is converging, even while its instantiation at every type diverges. But such a distinction is not observable in Haskell, not even with *seq* present. As a consequence of this, using the logical relation from [16] to derive statements about functions in the calculus under study there enforces extra convergence conditions on polymorphic arguments — conditions that were not found to be necessary on the primarily intuitive level in [14,15], and indeed are not justifiable with respect to the semantics of Haskell.[2] To prevent such a discrepancy in the current paper, we work with a semantics defined on the type-erasure of terms. This is in line with the treatment of the intermediate language found in GHC, which

[2] Note that these conditions are not particular to selective strictness: the same kind of extra conditions also surface in the purely strict settings of [19,21,22,25]. They would also surface in the purely nonstrict setting of [17] if choosing to make whole program termination observable at arbitrary, and thus also at polymorphic, types.

is an extension of the Girard-Reynolds calculus (which has full and explicit typing), but whose dynamic semantics is type-erasing [26].

Specifically, the central contribution of this paper is the construction of a family of logical relations for Core, a polymorphic lambda calculus which supports fixpoint recursion, an algebraic data type with pattern matching, a strict-let construct, and an explicit error primitive. The members of this family characterize different notions, induced by a preorder parameter, of observational equivalence or approximation with respect to a type-erasing operational semantics. This result includes, even generalizes, the parametricity theorem for each such logical relation. The general approach is similar to that in Pitts' characterization of observational equivalence for the calculus PolyPCF [17]. However, it is not at all obvious that his machinery can be brought to bear here. Apart from handling the additional language features, a particular challenge is posed by the interaction between the low-level semantics on the untyped level and the intended reasoning on a higher, typed level.

The remainder of this paper is structured as follows. Section 2 introduces the syntax and (type-erasing operational) semantics of the calculus Core that is the object of our study, and illustrates the use of its selective strictness and finite failure constructs in example programs. It also introduces the (parameterized) notion of "respecting observable program behavior" that is central to specifying observational relations on Core programs. Section 3 studies the observable behavior of (type-erasures of) Core programs in context, and in particular establishes key properties of selective strictness and fixpoint recursion. Section 4 introduces (preorder-parameterized) restrictions on relations to accommodate the fixpoint, selective strictness, and finite failure primitives, and examines their interplay. It also defines our family of logical relations and highlights one particularly fruitful source of appropriately restricted relations to be used in applications. Section 5 proves our main technical result, namely that the logical relation obtained for each preorder parameter does indeed give rise to the intended observational relation. Section 6 proves an abstract correctness result for *foldr*/*build*-fusion and looks at several interesting instantiations of this result. Section 7 discusses related work. Section 8 concludes. Throughout, proofs which are too technical for the main part of the paper are deferred to Appendix A. The proofs in the appendix can safely be omitted without disrupting the main ideas of the paper, but are included here for the sake of completeness.

2 The Core Language

2.1 Syntax and Typing

Let $\mathbb{N}$ be the set of natural numbers including 0. We set $\mathbb{N}_+ = \mathbb{N} \setminus \{0\}$ and, for a new element ∞, $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$ and $\mathbb{N}_{+,\infty} = \mathbb{N}_+ \cup \{\infty\}$.

The syntax of Core *types* and *terms* is given in Figure 1, where α and x range over disjoint countably infinite sets of *type variables* and *term variables*, respectively, and i ranges over $\mathbb{N}_+$. To reduce the need for brackets, function types and function applications are read right- and left-associative, respectively, so that $\tau_1 \to \tau_2 \to \tau_3$ means $\tau_1 \to (\tau_2 \to \tau_3)$, while $F\ A\ B$ means $(F\ A)\ B$. The constructions $\forall\alpha.-$, $\lambda x :: \tau.-$, $\Lambda\alpha.-$, $\mathbf{case}\ M\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M';\ x : x' \Rightarrow -\}$, and $\mathbf{let!}\ x = M\ \mathbf{in}\ -$ are binders for α, x, and x'. We identify types and terms up to renaming of bound (type and term) variables. The concept of a free variable in a type or term is defined in the usual way. We write *Typ* for the set of closed types, that is, those having no free variables. We use standard notation for capture-avoiding substitution of types and/or terms for free occurrences of variables.

Types	$\tau ::= \alpha \mid \tau \to \tau \mid \forall\alpha.\tau \mid \tau\text{-}list$		
Terms	$M ::= x$		term variable
	$\mid\ \lambda x :: \tau.M$		function abstraction
	$\mid\ M\ M$		function application
	$\mid\ \Lambda\alpha.M$		type generalization
	$\mid\ M_\tau$		type specialization
	$\mid\ \mathbf{nil}_\tau$		empty list
	$\mid\ M : M$		non-empty list
	$\mid\ \mathbf{case}\ M\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M;\ x : x \Rightarrow M\}$		pattern matching
	$\mid\ \mathbf{let!}\ x = M\ \mathbf{in}\ M$		selective strictness
	$\mid\ \mathbf{fix}(M)$		fixpoint recursion
	$\mid\ \mathbf{error}_\tau(i)$		finite failure

Fig. 1. Syntax of the Core language.

Types are assigned to (some) terms according to the axioms and rules in Figure 2, where Γ ranges over *typing environments* of the form $\vec{\alpha}, x_1 :: \tau_1, \ldots, x_m :: \tau_m$ for a finite list $\vec{\alpha}$ of distinct type variables, $m \in \mathbb{N}$, a list $\vec{x} = x_1, \ldots, x_m$ of distinct term variables, and types $\tau_1, \ldots, \tau_m$ whose free variables are in $\vec{\alpha}$. In a typing judgement of the form $\Gamma \vdash M :: \tau$, with Γ as above, we require that the free variables of the term M are in $\vec{\alpha}, \vec{x}$ and that the free variables of the type τ are in $\vec{\alpha}$. The well-formedness conditions for typing environments and typing

judgements ensure that in the rule for **let**! in Figure 2, x does not occur in Γ and thus is also not free in A. That is, the strict-let construct is nonrecursive. The explicit type information in the syntax of function abstractions, empty lists, and finite failures ensures that for every Γ and M there is at most one τ with $\Gamma \vdash M :: \tau$. Given $\tau \in \mathit{Typ}$, we write $\mathit{Term}(\tau)$ for the set of terms M for which $\varnothing \vdash M :: \tau$ is derivable, where $\varnothing$ is the empty typing environment. Further, we set $\mathit{Term} = \bigcup_{\tau \in \mathit{Typ}} \mathit{Term}(\tau)$.

$$\Gamma, x :: \tau \vdash x :: \tau \qquad \Gamma \vdash \mathbf{nil}_\tau :: \tau\text{-}\mathit{list} \qquad \Gamma \vdash \mathbf{error}_\tau(i) :: \tau$$

$$\frac{\Gamma, x :: \tau \vdash M :: \tau'}{\Gamma \vdash (\lambda x :: \tau.M) :: \tau \to \tau'} \qquad \frac{\alpha, \Gamma \vdash M :: \tau}{\Gamma \vdash \Lambda\alpha.M :: \forall\alpha.\tau} \qquad \frac{\Gamma \vdash G :: \forall\alpha.\tau}{\Gamma \vdash G_{\tau'} :: \tau[\tau'/\alpha]}$$

$$\frac{\Gamma \vdash F :: \tau \to \tau' \qquad \Gamma \vdash A :: \tau}{\Gamma \vdash F\ A :: \tau'} \qquad \frac{\Gamma \vdash H :: \tau \qquad \Gamma \vdash T :: \tau\text{-}\mathit{list}}{\Gamma \vdash (H : T) :: \tau\text{-}\mathit{list}}$$

$$\frac{\Gamma \vdash L :: \tau\text{-}\mathit{list} \qquad \Gamma \vdash M_1 :: \tau' \qquad \Gamma, h :: \tau, t :: \tau\text{-}\mathit{list} \vdash M_2 :: \tau'}{\Gamma \vdash \mathbf{case}\ L\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1;\ h : t \Rightarrow M_2\} :: \tau'}$$

$$\frac{\Gamma \vdash A :: \tau \qquad \Gamma, x :: \tau \vdash B :: \tau'}{\Gamma \vdash \mathbf{let!}\ x = A\ \mathbf{in}\ B :: \tau'} \qquad \frac{\Gamma \vdash F :: \tau \to \tau}{\Gamma \vdash \mathbf{fix}(F) :: \tau}$$

Fig. 2. Core type assignment relation.

2.2 The New Features by Example

A typical example for the use of selective strictness in Haskell is the following function:

$$\begin{aligned}
&\mathit{foldl}' :: \forall\alpha\ \beta.\ (\beta \to \alpha \to \beta) \to \beta \to [\alpha] \to \beta \\
&\mathit{foldl}'\ f\ z\ [] \qquad\quad = z \\
&\mathit{foldl}'\ f\ z\ (h : t) = \mathbf{let}\ z' = f\ z\ h\ \mathbf{in}\ \mathit{seq}\ z'\ (\mathit{foldl}'\ f\ z'\ t)
\end{aligned}$$

Here *seq* ensures that the accumulating parameter is computed immediately in each recursive step rather than constructing a complex closure which would be computed only at the very end.

The above function definition can be expressed in Core as the following element of $\mathit{Term}(\forall\alpha.\forall\beta.(\beta \to \alpha \to \beta) \to \beta \to \alpha\text{-}\mathit{list} \to \beta)$:

$$\begin{aligned}
&\mathbf{fix}(\lambda \mathit{foldl}' :: \forall\alpha.\forall\beta.(\beta \to \alpha \to \beta) \to \beta \to \alpha\text{-}\mathit{list} \to \beta. \\
&\quad \Lambda\alpha.\Lambda\beta.\lambda f :: \beta \to \alpha \to \beta.\lambda z :: \beta.\lambda l :: \alpha\text{-}\mathit{list}. \\
&\quad \mathbf{case}\ l\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow z;\ h : t \Rightarrow \mathbf{let!}\ z' = f\ z\ h\ \mathbf{in}\ (\mathit{foldl}'_\alpha)_\beta\ f\ z'\ t\}).
\end{aligned}$$

This corresponds closely to the intermediate code produced for *foldl′* by GHC. In particular, it is in line with the latter (and in contrast to the treatment in [16]) regarding the avoidance of a duplication of the expression "$f\ z\ h$" during translation of the call to *seq*. It also agrees with the modelling of selective strictness in [27] and, for the Clean language, in [28].

A typical form of program failure other than nontermination is that of incomplete pattern matching, as in the following function:

$$\begin{aligned} &\mathit{init} :: \forall\alpha.\ [\alpha] \to [\alpha] \\ &\mathit{init}\ [h] \quad\ = [] \\ &\mathit{init}\ (h : t) = h : \mathit{init}\ t \end{aligned}$$

This function can be expressed in Core using the explicit error primitive as follows:

$$\begin{aligned} &\mathbf{fix}(\lambda \mathit{init} :: \forall\alpha.\alpha\text{-}\mathit{list} \to \alpha\text{-}\mathit{list}.\Lambda\alpha.\lambda l :: \alpha\text{-}\mathit{list}. \\ &\quad \mathbf{case}\ l\ \mathbf{of}\ \{\mathbf{nil}\ \ \Rightarrow \mathbf{error}_{\alpha\text{-}\mathit{list}}(1); \\ &\qquad\qquad h : t \Rightarrow \mathbf{case}\ t\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow \mathbf{nil}_\alpha;\ h' : t' \Rightarrow h : \mathit{init}_\alpha\ t\}\}). \end{aligned}$$

This is again very similar to what GHC produces internally, except that in Haskell more descriptive string arguments are used for finite failures (such as "Prelude.init: empty list" instead of the 1 above). Of course, the abstraction from strings to positive integers in our language does not change the nature of the phenomena under study.

2.3 A Type-Erasing Operational Semantics

Our semantics for Core is defined on the type-erasure of terms. For this, we need a notion of *untyped terms*, given by the following grammar:

$$\begin{aligned} M ::= {} & x \mid \lambda x.M \mid M\ M \mid \mathbf{nil} \mid M : M \mid \mathbf{case}\ M\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M;\ x : x \Rightarrow M\} \mid \\ & \mathbf{let!}\ x = M\ \mathbf{in}\ M \mid \mathbf{fix}(M) \mid \mathbf{error}(i). \end{aligned}$$

The construction $\lambda x.-$ (now without the type information present in Core) is an additional binder for x. As for typed terms, we identify untyped terms up to renaming of bound (term) variables, and define the concept of a free variable in the usual way. Capture-avoiding substitution on untyped terms is also defined as in the typed case. Given a set X of term variables, we write $\mathit{Untyped}(X)$ for the set of untyped terms whose free variables are in X. Further, we set $\mathit{Untyped} = \mathit{Untyped}(\varnothing)$, where $\varnothing$ is the empty set.

Typed terms are mapped to untyped terms using the *type-erasure transformation* $[\![\cdot]\!]$. It drops the type annotations in the binding occurrences of vari-

ables in function abstractions, eliminates all type generalizations and specializations, omits the type subscripts of empty lists and finite failures, but leaves the input term otherwise unchanged. Note that type-erasure distributes over term substitution and is invariant under type substitution. That is, $[\![M[A/x]]\!] = [\![M]\!][[\![A]\!]/x]$ and $[\![M[\tau/\alpha]]\!] = [\![M]\!]$.

The subset of *Untyped* whose elements (called *values*) respect the following grammar is denoted by *Value*:

$$V ::= \lambda x.M \mid \mathbf{nil} \mid M : M.$$

We use a small-step approach to structural operational semantics, so we need *redex/reduct-pairs* and a notion of reduction in context. The former are written $R \rightsquigarrow R'$ (with $R, R' \in \mathit{Untyped}$) and listed exhaustively in the following table:

R	R'
$(\lambda x.N)\ A$	$N[A/x]$
$\mathbf{case\ nil\ of}\ \{\mathbf{nil} \Rightarrow M;\ h : t \Rightarrow M'\}$	M
$\mathbf{case}\ H : T\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M;\ h : t \Rightarrow M'\}$	$M'[H/h, T/t]$
$\mathbf{let!}\ x = V\ \mathbf{in}\ N$	$N[V/x]$
$\mathbf{fix}(F)$	$F\ \mathbf{fix}(F)$

Here x, h, and t are term variables, $N \in \mathit{Untyped}(\{x\})$, $A, H, M, T, F \in \mathit{Untyped}$, $M' \in \mathit{Untyped}(\{h, t\})$, and $V \in \mathit{Value}$. It is essential that V is a value in the next-to-last pair, since the intended semantics of selective strictness could not otherwise be ensured.

To describe reduction in context, we use the notions of *evaluation frames* and *evaluation stacks*, given by the grammars

$$E ::= (-\ M) \mid (\mathbf{case}\ -\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M;\ x : x \Rightarrow M\}) \mid (\mathbf{let!}\ x = -\ \mathbf{in}\ M)$$

and

$$S ::= \mathit{Id} \mid S \circ E\,,$$

respectively, where each M ranges over untyped terms. If an evaluation stack comprises a single evaluation frame E, then we denote it by E rather than $\mathit{Id} \circ E$. Moreover, given an evaluation frame E and an untyped term M, we write $E\{M\}$ for the untyped term that results from replacing "$-$" by M in E. The concept of a free (term) variable in an evaluation frame or evaluation stack is defined in the obvious way.

Now a *transition* $(S_1, M_1) \rightarrowtail (S_2, M_2)$ (with $M_1, M_2 \in \mathit{Untyped}$ and S_1, S_2 being evaluation stacks without free variables) is possible for exactly the fol-

lowing combinations:

(S_1, M_1)	(S_2, M_2)	if
$(S, E\{N\})$	$(S \circ E, N)$	$N \notin \mathit{Value}$
$(S \circ E, V)$	$(S, E\{V\})$	$V \in \mathit{Value}$
(S, R)	(S, R')	$R \rightsquigarrow R'$

Here S is an evaluation stack, E is an evaluation frame, and the untyped terms that occur in the table are subject to the restrictions recorded on the right. Note that $\rightarrowtail$ is deterministic, but not terminating (due to **fix**). We denote by $\rightarrowtail^t$, with $t \in \mathbb{N}$, the t-fold composition of $\rightarrowtail$, and by $\rightarrowtail^*$ its reflexive, transitive closure. The latter is used to describe (potential) evaluation of typed terms as follows.

Definition 2.1. Given $M \in \mathit{Term}$, we write:

- $M{\Downarrow}$ if there is some $V \in \mathit{Value}$ with $(\mathit{Id}, [\![M]\!]) \rightarrowtail^* (\mathit{Id}, V)$,
- $M{\lightning}\, i$ if there is some evaluation stack S with $(\mathit{Id}, [\![M]\!]) \rightarrowtail^* (S, \mathbf{error}(i))$, and
- $M{\Uparrow}$ otherwise.

In the first case we say that M *converges*, in the second case that it *fails finitely*, and in the last case that it *diverges*.

A pair consisting of an evaluation stack without free variables and an element of *Untyped* is called an *end configuration* if it has one of the two forms reached by $\rightarrowtail^*$ in the first two items of Definition 2.1. We say that (S, M) *leads to an end configuration* if there is an end configuration (S', M') with $(S, M) \rightarrowtail^* (S', M')$. Note that this is not the case for every (S, M).

Observation 2.2. *Let $\Omega = \mathbf{fix}(\lambda x.x) \in \mathit{Untyped}$. There is no evaluation stack S such that (S, Ω) leads to an end configuration.*

The following lemma is proved in the appendix.

Lemma 2.3. *For every evaluation stack S and $M \in \mathit{Untyped}$, if (S, M) leads to an end configuration, then so does (Id, M).*

Given an evaluation stack S, we define for every evaluation stack S' their concatenation $S \mathbin{@} S'$ by induction on the structure of S' via the equations $S \mathbin{@} \mathit{Id} = S$ and $S \mathbin{@} (S'' \circ E) = (S \mathbin{@} S'') \circ E$. Then the following observation is straightforward from the definition of $\rightarrowtail$.

Observation 2.4. *For every triple S_1, S_2, S_3 of evaluation stacks without free variables and $M_1, M_2 \in \mathit{Untyped}$, if $(S_1, M_1) \rightarrowtail^* (S_2, M_2)$, then $(S_3 \mathbin{@} S_1, M_1) \rightarrowtail^* (S_3 \mathbin{@} S_2, M_2)$.*

Any reasonable notion of program equivalence or approximation should at least be a precongruence. More precisely, it should fulfill all four restrictions introduced in the following definition.

Definition 2.5. Let the relation $\mathcal{E}$ comprise 4-tuples of the form (Γ, M, M', τ) with $\Gamma \vdash M :: \tau$ and $\Gamma \vdash M' :: \tau$. We write $\Gamma \vdash M \;\mathcal{E}\; M' :: \tau$ when the tuple (Γ, M, M', τ) is in $\mathcal{E}$, and we abbreviate this to $M \;\mathcal{E}\; M'$ if $\Gamma = \varnothing$ since τ is then uniquely determined as the closed type of both M and M'.

- If for every Γ, M, and τ with $\Gamma \vdash M :: \tau$, we have $\Gamma \vdash M \;\mathcal{E}\; M :: \tau$, then $\mathcal{E}$ is called *reflexive*.
- If $\mathcal{E}; \mathcal{E} \subseteq \mathcal{E}$, where *relation composition* $\mathcal{E}_1; \mathcal{E}_2$ is defined by

$$\Gamma \vdash M \;(\mathcal{E}_1; \mathcal{E}_2)\; M' :: \tau \;\Leftrightarrow\; \exists M''.\; \Gamma \vdash M \;\mathcal{E}_1\; M'' :: \tau \wedge \Gamma \vdash M'' \;\mathcal{E}_2\; M' :: \tau\,,$$

  then $\mathcal{E}$ is called *transitive*.
- If $\mathcal{E}$ is closed under the axioms and rules in Figure 3, then it is called *compatible*.
- If $\mathcal{E}$ is closed under the rules in Figure 4, where $\Gamma[\tau'/\alpha]$ is the typing environment obtained from Γ by replacing every $x :: \sigma$ therein by $x :: \sigma[\tau'/\alpha]$, then it is called *substitutive*.

Note that every compatible relation is also reflexive.

To further specify a notion of program equivalence or approximation, we need to express the observations that can be made about a program and decide on a concept of adequacy that somehow restricts the relation between the observed behaviors for two programs supposed to be equivalent or in an approximating relationship. The possible observations can be captured as follows.

Definition 2.6. For every $M \in \mathit{Term}$, we define $\omega(M) \in \mathbb{N}_\infty$ by

$$\omega(M) = \begin{cases} 0 & \text{if } M{\Downarrow} \\ i & \text{if } M \not\downarrow i \\ \infty & \text{if } M{\Uparrow}. \end{cases}$$

Note that ω is well-defined. Indeed, the determinism of $\rightarrowtail$ ensures that at most one of the potential end configurations under consideration in Definition 2.1 can actually be reached from $(\mathit{Id}, [\![M]\!])$. Also note that if $[\![M]\!] \in \mathit{Value}$, then $\omega(M) = 0$. Moreover, for every $M, M' \in \mathit{Term}$, if $[\![M]\!] = [\![M']\!]$, then $\omega(M) = \omega(M')$.

Instead of deciding on a particular notion of adequacy now, and thus restricting the further development to a specific notion of program equivalence or approximation, we leave that choice as abstract as possible for the moment. That is, given any binary relation $\preccurlyeq$ on $\mathbb{N}_\infty$, we say that a relation $\mathcal{E}$ as in

$$\Gamma, x :: \tau \vdash x \;\mathcal{E}\; x :: \tau \qquad \Gamma \vdash \mathbf{error}_\tau(i) \;\mathcal{E}\; \mathbf{error}_\tau(i) :: \tau$$

$$\frac{\Gamma \vdash F \;\mathcal{E}\; F' :: \tau \to \tau}{\Gamma \vdash \mathbf{fix}(F) \;\mathcal{E}\; \mathbf{fix}(F') :: \tau} \qquad \frac{\Gamma, x :: \tau \vdash M \;\mathcal{E}\; M' :: \tau'}{\Gamma \vdash (\lambda x :: \tau.M) \;\mathcal{E}\; (\lambda x :: \tau.M') :: \tau \to \tau'}$$

$$\frac{\Gamma \vdash F \;\mathcal{E}\; F' :: \tau \to \tau' \qquad \Gamma \vdash A \;\mathcal{E}\; A' :: \tau}{\Gamma \vdash (F\ A) \;\mathcal{E}\; (F'\ A') :: \tau'}$$

$$\frac{\alpha, \Gamma \vdash M \;\mathcal{E}\; M' :: \tau}{\Gamma \vdash \Lambda\alpha.M \;\mathcal{E}\; \Lambda\alpha.M' :: \forall\alpha.\tau} \qquad \frac{\Gamma \vdash G \;\mathcal{E}\; G' :: \forall\alpha.\tau}{\Gamma \vdash G_{\tau'} \;\mathcal{E}\; G'_{\tau'} :: \tau[\tau'/\alpha]}$$

$$\Gamma \vdash \mathbf{nil}_\tau \;\mathcal{E}\; \mathbf{nil}_\tau :: \tau\text{-}list \qquad \frac{\Gamma \vdash H \;\mathcal{E}\; H' :: \tau \qquad \Gamma \vdash T \;\mathcal{E}\; T' :: \tau\text{-}list}{\Gamma \vdash (H : T) \;\mathcal{E}\; (H' : T') :: \tau\text{-}list}$$

$$\frac{\Gamma \vdash L \;\mathcal{E}\; L' :: \tau\text{-}list \quad \Gamma \vdash M_1 \;\mathcal{E}\; M_1' :: \tau' \quad \Gamma, h :: \tau, t :: \tau\text{-}list \vdash M_2 \;\mathcal{E}\; M_2' :: \tau'}{\begin{array}{l}\Gamma \vdash (\mathbf{case}\ L\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1;\ h : t \Rightarrow M_2\}) \\ \quad \mathcal{E}\ (\mathbf{case}\ L'\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1';\ h : t \Rightarrow M_2'\}) :: \tau'\end{array}}$$

$$\frac{\Gamma \vdash A \;\mathcal{E}\; A' :: \tau \qquad \Gamma, x :: \tau \vdash B \;\mathcal{E}\; B' :: \tau'}{\Gamma \vdash (\mathbf{let!}\ x = A\ \mathbf{in}\ B) \;\mathcal{E}\; (\mathbf{let!}\ x = A'\ \mathbf{in}\ B') :: \tau'}$$

Fig. 3. Compatibility properties.

$$\frac{\alpha, \Gamma \vdash M \;\mathcal{E}\; M' :: \tau}{\Gamma[\tau'/\alpha] \vdash M[\tau'/\alpha] \;\mathcal{E}\; M'[\tau'/\alpha] :: \tau[\tau'/\alpha]}$$

$$\frac{\Gamma, x :: \tau \vdash M \;\mathcal{E}\; M' :: \tau' \qquad \Gamma \vdash N \;\mathcal{E}\; N' :: \tau}{\Gamma \vdash M[N/x] \;\mathcal{E}\; M'[N'/x] :: \tau'}$$

Fig. 4. Substitutivity properties.

Definition 2.5 is $\preccurlyeq$-*adequate* if for every $\tau \in \mathit{Typ}$ and $M, M' \in \mathit{Term}(\tau)$ with $M \;\mathcal{E}\; M'$, we have $\omega(M) \preccurlyeq \omega(M')$.

There is a vast richness of choices available for the relation $\preccurlyeq$; some interesting and natural ones, together with the computational intuition underlying them, are given in Example 2.7 below. Our parameterized approach to adequacy thus provides a uniform framework for studying a broad array of (potential) observational relations induced by choices for $\preccurlyeq$. The design decisions they capture range from whether to observe computational equivalence or approximation, to whether (and how) to distinguish or unify different causes of failure.

Example 2.7. Some interesting choices for $\preccurlyeq$ are given as follows.

(a) $\preccurlyeq = \{(n, n) \mid n \in \mathbb{N}_\infty\}$, depicted as:

$$0 \quad 1 \quad 2 \quad 3 \quad \cdots \quad \infty$$

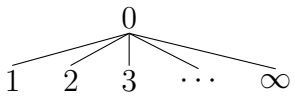
(b) $\preccurlyeq = \{(0, 0)\} \cup \mathbb{N}_{+,\infty} \times \mathbb{N}_{+,\infty}$, depicted as:

$$0 \quad 1 = 2 = 3 = \cdots = \infty$$

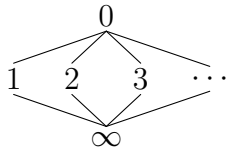
(c) $\preccurlyeq = \mathbb{N}_{+,\infty} \times \mathbb{N}_{+,\infty} \cup \mathbb{N}_\infty \times \{0\}$, depicted as:

$$\begin{array}{c} 0 \\ | \\ 1 = 2 = 3 = \cdots = \infty \end{array}$$

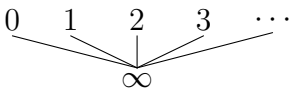
(d) $\preccurlyeq = \{(n,n) \mid n \in \mathbb{N}_\infty\} \cup \mathbb{N}_{+,\infty} \times \{0\}$, depicted as:



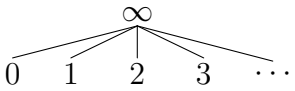
(e) $\preccurlyeq = \{(n,n) \mid n \in \mathbb{N}_\infty\} \cup \mathbb{N}_{+,\infty} \times \{0\} \cup \{\infty\} \times \mathbb{N}$, depicted as:



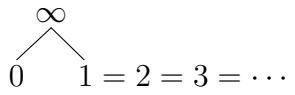
(f) $\preccurlyeq = \{(n,n) \mid n \in \mathbb{N}_\infty\} \cup \{\infty\} \times \mathbb{N}$, depicted as:



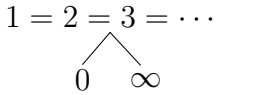
(g) $\preccurlyeq = \{(n,n) \mid n \in \mathbb{N}_\infty\} \cup \mathbb{N} \times \{\infty\}$, depicted as:



(h) $\preccurlyeq = \mathbb{N}_+ \times \mathbb{N}_{+,\infty} \cup \{(0,0),(0,\infty),(\infty,\infty)\}$, depicted as:



(i) $\preccurlyeq = \mathbb{N}_\infty \times \mathbb{N}_+ \cup \{(0,0),(\infty,\infty)\}$, depicted as:



(j) $\preccurlyeq = \mathbb{N}_\infty \times \mathbb{N}_+ \cup \{(0,0),(\infty,0),(\infty,\infty)\}$, depicted as:

$$\begin{array}{c} 1 = 2 = 3 = \cdots \\ | \\ 0 \\ | \\ \infty \end{array}$$

Intuitively, choice (a) will result in an observational equivalence relation $\equiv_1$ that distinguishes between convergence, divergence, and finite failure, and also distinguishes different instances of finite failure from one another. In contrast, choice (b) gives an observational equivalence relation $\equiv_2$ that unifies all different failure causes (but distinguishes them from convergence), while choice (c) defines the corresponding observational approximation relation $\sqsubseteq_2$. Choices (d) and (e) are intended to describe observational approximation re-

lations that distinguish all different failure causes and consider each kind of failure as strictly less defined than a converging behavior, where moreover in (d) different failure causes are considered pairwise incomparable, while in (e) a diverging term is considered strictly less defined than any finitely failing one. Choice (f) describes an observational approximation relation $\sqsubseteq_3$ that also distinguishes all different failure causes, with divergence considered least defined, but considers convergence and finite failures as pairwise incomparable. Choice (g) naturally gives the corresponding inverse $\sqsupseteq_3$. Choice (h) describes a variant $\sqsupseteq_4$ of $\sqsupseteq_3$ in which all instances of finite failure are unified. Finally, choice (i) describes an inverse observational approximation relation $\sqsupseteq_5$ that unifies all instances of finite failure and considers them less defined than both convergence and divergence, which in turn are incomparable. Choice (j) is perfectly possible, but quite unintuitive since it makes one sort of failure more defined, and at the same time makes another sort of failure less defined, than computations that terminate properly. Further somewhat "bizarre" choices are conceivable in which, for example, there is a strict linear order between different instances of finite failure.

Our overall aim is to characterize, for a given choice of $\preccurlyeq$, the largest reflexive, transitive, compatible, substitutive, and $\preccurlyeq$-adequate relation by induction on the type structure of Core. Clearly, it is not to be expected that this is possible, or gives a meaningful result, for arbitrary $\preccurlyeq$. In fact, one merit of our approach is to systematically identify "legal" choices of $\preccurlyeq$. Perhaps surprisingly, choices (d) and (e) from Example 2.7 will be outlawed (but rightly and explicably so, as later pointed out in Section 8). The next section, though, will first collect some technical material that is independent of the choice of $\preccurlyeq$.

3 Typed Stacks and the $\top$-Function

While our operational semantics is defined via untyped terms and evaluation frames, we want most of our reasoning to take place on the higher, typed level. Hence, we need also a typed notion for the context in which a computation takes place.

The grammars for *typed frames* and *typed stacks* are as those for evaluation frames and evaluation stacks, except that the M now range over typed terms and that there is an additional form $-_\tau$ of frames (where τ is a type). In other words, the typed frames are $(-\ M)$, $-_\tau$, $(\mathbf{case}\ -\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M;\ x : x \Rightarrow M\})$, and $(\mathbf{let!}\ x = -\ \mathbf{in}\ M)$. A type assignment relation $\Gamma \vdash S :: \tau \multimap \tau'$ (where Γ again ranges over typing environments, with well-formedness conditions similar to those for term typing judgements) assigns argument and result types to some typed stacks in such a way that for every Γ, S, and τ there is at most one suitable τ'. It is given in Figure 5. Given $\tau, \tau' \in \mathit{Typ}$, we write

$Stack(\tau, \tau')$ for the set of typed stacks S for which $\varnothing \vdash S :: \tau \multimap \tau'$ is derivable. Given such an S and $M \in Term(\tau)$, the application $(S\ M) \in Term(\tau')$ is defined by induction on the structure of S via the equations $Id\ M = M$ and $(S' \circ E)\ M = S'\ (E\{M\})$. We also use a typed version of the concatenation operation @, on typed stacks. And we set, for every $\tau \in Typ$, $Stack(\tau) = \bigcup_{\tau' \in Typ} Stack(\tau, \tau')$.

$$\Gamma \vdash Id :: \tau \multimap \tau$$

$$\frac{\Gamma \vdash S :: \tau' \multimap \tau'' \qquad \Gamma \vdash A :: \tau}{\Gamma \vdash S \circ (-\ A) :: (\tau \to \tau') \multimap \tau''} \qquad \frac{\Gamma \vdash S :: \tau[\tau'/\alpha] \multimap \tau''}{\Gamma \vdash S \circ -_{\tau'} :: (\forall \alpha.\tau) \multimap \tau''}$$

$$\frac{\Gamma \vdash S :: \tau' \multimap \tau'' \qquad \Gamma \vdash M_1 :: \tau' \qquad \Gamma, h :: \tau, t :: \tau\text{-}list \vdash M_2 :: \tau'}{\Gamma \vdash S \circ (\mathbf{case}\ -\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1;\ h : t \Rightarrow M_2\}) :: \tau\text{-}list \multimap \tau''}$$

$$\frac{\Gamma \vdash S :: \tau' \multimap \tau'' \qquad \Gamma, x :: \tau \vdash B :: \tau'}{\Gamma \vdash S \circ (\mathbf{let!}\ x = -\ \mathbf{in}\ B) :: \tau \multimap \tau''}$$

Fig. 5. Typing typed stacks.

A key ingredient of Pitts' approach to syntactic logical relations is a relation expressing whether or not a particular term put into a particular context described by a suitable stack leads to a (in some cases, particular) value in the empty context. Since in our setting there are more possibly observable behaviors than just convergence or divergence, we instead need a function with more possible outcomes than just the "yes" or "no" outcomes provided by relations. This motivates the following definition.

Definition 3.1. Let $\tau \in Typ$, $S \in Stack(\tau)$, and $M \in Term(\tau)$. We define $\top(S, M) \in \mathbb{N}_\infty$ to be:

- 0 if there is some $V \in Value$ with $([\![S]\!], [\![M]\!]) \rightarrowtail^* (Id, V)$,
- i if there is some evaluation stack S' with $([\![S]\!], [\![M]\!]) \rightarrowtail^* (S', \mathbf{error}(i))$, and
- ∞ otherwise.

Here the type-erasure transformation $[\![\cdot]\!]$ from typed stacks to evaluation stacks is the straightforward extension of the one on the term level. In particular, it omits all frames of the form $-_\tau$.

From the definitions of $\top$ and ω, and the determinism of $\rightarrowtail$, we obtain the following three observations.

Observation 3.2. *For every $M \in Term$, $\top(Id, M) = \omega(M)$.*

Observation 3.3. *Let $\tau \in Typ$, $S, S' \in Stack(\tau)$, and $M, M' \in Term(\tau)$. If $[\![S]\!] = [\![S']\!]$ and $[\![M]\!] = [\![M']\!]$, then $\top(S, M) = \top(S', M')$.*

Observation 3.4. *Let $\tau \in Typ$ and $S \in Stack(\tau)$.*

(a) For every $\tau' \in Typ$, $M \in Term(\tau')$, and typed frame E with $E\{M\} \in Term(\tau)$, we have $\top(S, E\{M\}) = \top(S \circ E, M)$.

(b) For every $R, R' \in Term(\tau)$ with $[\![R]\!] \rightsquigarrow [\![R']\!]$, we have $\top(S, R) = \top(S, R')$.

We also obtain the following two corollaries of Observations 3.2 and 3.4(a), and of Observations 3.2 and 3.4(b), respectively.

Corollary 3.5. *For every $\tau \in Typ$, $S \in Stack(\tau)$, and $M \in Term(\tau)$, we have $\top(S, M) = \omega(S\ M)$.*

Corollary 3.6. *For every $\tau \in Typ$ and $R, R' \in Term(\tau)$, if $[\![R]\!] \rightsquigarrow [\![R']\!]$, then $\omega(R) = \omega(R')$.*

In order to later establish the compatibility of the logical relation(s) developed in the next section, we need two key lemmas about the language constructs for selective strictness and fixpoint recursion. The first one is reminiscent of the denotational semantics definition of *seq* provided in the introduction. Its proof, which requires some care, is given in the appendix.

Lemma 3.7. *Let $\tau_1, \tau_2 \in Typ$, $A \in Term(\tau_1)$, and $S \in Stack(\tau_2)$. Let x be a term variable and B be a typed term with $x :: \tau_1 \vdash B :: \tau_2$. Then:*

$$\top(S, \mathbf{let!}\ x = A\ \mathbf{in}\ B) = \begin{cases} \omega(A) & \text{if } \omega(A) \neq 0 \\ \top(S, B[A/x]) & \text{otherwise.} \end{cases}$$

The second lemma, which is the key to properly handling fixpoint recursion, has the common "unwinding" flavor. To formulate it, we need the notation $(F^n\ A) \in Term(\tau)$ for the n-fold application of F to A, given $n \in \mathbb{N}$, $\tau \in Typ$, $F \in Term(\tau \to \tau)$, and $A \in Term(\tau)$. The following lemma is then proved in the appendix.

Lemma 3.8. *For every $\tau \in Typ$, $S \in Stack(\tau)$, and $F \in Term(\tau \to \tau)$, there exists an $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, we have $\top(S, \mathbf{fix}(F)) = \top(S, F^n\ \mathbf{fix}(\lambda x :: \tau.x))$.*

4 The Family of Logical Relations

Recall that so far we have left abstract the notion of adequacy inducing program equivalence or approximation by parameterizing it over a binary relation $\preccurlyeq$ on $\mathbb{N}_\infty$. We will continue to do so for the remainder of this and the next section (i.e., up to the end of Section 5). As we go through the technical development, though, we will notice that $\preccurlyeq$ cannot be completely arbitrary, but instead must fulfill certain restrictions. Two very natural requirements on $\preccurlyeq$, in order to ensure that the notion of program equivalence or approximation it induces is reflexive and transitive, are that for every $a \in \mathbb{N}_\infty$, $a \preccurlyeq a$, and that for every $a, b, c \in \mathbb{N}_\infty$, $a \preccurlyeq b$ and $b \preccurlyeq c$ imply $a \preccurlyeq c$. These two properties, making $\preccurlyeq$ itself a preorder, will henceforth be used without explicit

mention.[3] Two further restrictions are solely mandated by the presence of selective strictness, and will be given closer to the place where they first become relevant. Next, we discuss restrictions not *on* $\preccurlyeq$ but ones defined *in terms of* $\preccurlyeq$, restricting the relational interpretations of types.

4.1 $\preccurlyeq$-Compliance and $\preccurlyeq\!\!\preccurlyeq$-Closedness

That relational interpretations of types, in particular ones used for interpreting quantified type variables, must be restricted in certain ways is a recurring theme in the study of parametricity for extensions of the Girard-Reynolds calculus. The corresponding intuition for selective strictness, that is, for being able to force "out of order" evaluation of any subterm of any type, is as follows. The relative behavior as specified (for equally typed terms) by the relevant notion of adequacy, which induces the observational relation of interest, must also be reproduced by all relations (even ones between terms of different types) surfacing in the inductive construction of the logical relation. In our current abstract setting, this can be formalized as follows.

Given $\tau, \tau' \in \mathit{Typ}$, we define $\mathit{Rel}(\tau, \tau') = \mathcal{P}(\mathit{Term}(\tau) \times \mathit{Term}(\tau'))$. We say that $r \in \mathit{Rel}(\tau, \tau')$ is $\preccurlyeq$-*compliant* if for every $(M, M') \in r$, we have $\omega(M) \preccurlyeq \omega(M')$. The restriction of $\mathit{Rel}(\tau, \tau')$ to $\preccurlyeq$-compliant relations is denoted by $\mathit{Rel}^{\preccurlyeq}(\tau, \tau')$. We set $\mathit{Rel} = \bigcup_{\tau,\tau' \in \mathit{Typ}} \mathit{Rel}(\tau, \tau')$ and $\mathit{Rel}^{\preccurlyeq} = \bigcup_{\tau,\tau' \in \mathit{Typ}} \mathit{Rel}^{\preccurlyeq}(\tau, \tau')$.

Regarding an appropriate restriction for preserving parametricity in the presence of fixpoint recursion, we can follow the well-established closure operator approach of Pitts. So let $\tau, \tau' \in \mathit{Typ}$. Given $r \in \mathit{Rel}(\tau, \tau')$, we define $r^{\preccurlyeq} \subseteq \mathit{Stack}(\tau) \times \mathit{Stack}(\tau')$ by

$$(S, S') \in r^{\preccurlyeq} \text{ iff } \forall (M, M') \in r.\ \top(S, M) \preccurlyeq \top(S', M').$$

Similarly, given $s \subseteq \mathit{Stack}(\tau) \times \mathit{Stack}(\tau')$, we define $s^{\preccurlyeq} \in \mathit{Rel}(\tau, \tau')$ by

$$(M, M') \in s^{\preccurlyeq} \text{ iff } \forall (S, S') \in s.\ \top(S, M) \preccurlyeq \top(S', M').$$

A relation $r \in \mathit{Rel}$ is called $\preccurlyeq\!\!\preccurlyeq$-*closed* if $r^{\preccurlyeq\preccurlyeq} = r$. Note that one inclusion direction of the latter relation equality is always fulfilled by the first of the following three properties (which are standard for order-reversing Galois connections) for every $\tau, \tau' \in \mathit{Typ}$ and $r, r_1, r_2 \in \mathit{Rel}(\tau, \tau')$:

[3] Just for the record: the first property is needed for Observation 4.8, Lemmas 5.3, 5.4, and 4.16, Theorem 5.5, and Corollary 4.15, while the second property is needed for Lemmas 4.2 and 4.16 and Theorem 5.7.

$$r \subseteq r^{\preccurlyeq\preccurlyeq} \tag{1}$$
$$(r^{\preccurlyeq\preccurlyeq})^{\preccurlyeq} = r^{\preccurlyeq} \tag{2}$$
$$r_1 \subseteq r_2 \Rightarrow r_1^{\preccurlyeq\preccurlyeq} \subseteq r_2^{\preccurlyeq\preccurlyeq}. \tag{3}$$

The following preservation lemma establishes a crucial connection between $\preccurlyeq$-compliance and $\preccurlyeq\preccurlyeq$-closure.

Lemma 4.1. *For every $r \in Rel^{\preccurlyeq}$, we also have $r^{\preccurlyeq\preccurlyeq} \in Rel^{\preccurlyeq}$.*

Proof: For every $(M, M') \in r \in Rel^{\preccurlyeq}$, we have $\top(Id, M) \preccurlyeq \top(Id, M')$ by Observation 3.2, and thus $(Id, Id) \in r^{\preccurlyeq}$. Consequently, for every $(N, N') \in r^{\preccurlyeq\preccurlyeq}$, we have $\top(Id, N) \preccurlyeq \top(Id, N')$, which is equivalent to $\omega(N) \preccurlyeq \omega(N')$ by Observation 3.2. □

An important property of $\preccurlyeq\preccurlyeq$-closed relations is that they respect $\preccurlyeq$-adequate and compatible relations.

Lemma 4.2. *Let $\mathcal{E}$ be an $\preccurlyeq$-adequate and compatible relation, let $\tau, \tau' \in Typ$, and let $r \in Rel(\tau, \tau')$ be $\preccurlyeq\preccurlyeq$-closed. For every $M_1 \in Term(\tau)$ and $M_2, M_3 \in Term(\tau')$, if $(M_1, M_2) \in r$ and $M_2 \ \mathcal{E} \ M_3$, then $(M_1, M_3) \in r$.*

Proof: The desired $(M_1, M_3) \in r$ follows from the $\preccurlyeq\preccurlyeq$-closedness of r and the following reasoning for every $(S, S') \in r^{\preccurlyeq}$:

$$\begin{array}{rll} \top(S, M_1) & \preccurlyeq \top(S', M_2) & \text{by } (S, S') \in r^{\preccurlyeq} \text{ and } (M_1, M_2) \in r \\ & = \omega(S' \ M_2) & \text{by Corollary 3.5} \\ & \preccurlyeq \omega(S' \ M_3) & \\ & = \top(S', M_3) & \text{by Corollary 3.5.} \end{array}$$

Here $\omega(S' \ M_2) \preccurlyeq \omega(S' \ M_3)$ follows from $M_2 \ \mathcal{E} \ M_3$, because $\mathcal{E}$ is compatible and $\preccurlyeq$-adequate. □

The presence of the explicit error primitive in Core mandates no additional restriction on relations. It is already completely accounted for by $\preccurlyeq\preccurlyeq$-closedness, as we will see in the proof of Lemma 5.4.

4.2 The Relational Actions

The key to parametricity results is to build relational interpretations of types by induction on the type structure. Starting from an interpretation of type variables by relations (between typed terms), this requires defining a *relational action* for each of the ways of forming types. Such an action takes an appropriate number of relations and produces a new one as the interpretation for the compound type.

The main characteristic of all logical relations in the literature is that for two functions to be related they must map related arguments to related results. Since, as motivated in the previous subsection, in our setting all relations in the inductive construction should be $\preccurlyeq$-compliant, the following relational action additionally enforces this requirement.

Definition 4.3. Given $\tau_1, \tau_1', \tau_2, \tau_2' \in \mathit{Typ}$, $r_1 \in \mathit{Rel}(\tau_1, \tau_1')$, and $r_2 \in \mathit{Rel}(\tau_2, \tau_2')$, we define $(r_1 \to r_2) \in \mathit{Rel}(\tau_1 \to \tau_2, \tau_1' \to \tau_2')$ by

$$(F, F') \in (r_1 \to r_2) \;\text{ iff }\; \omega(F) \preccurlyeq \omega(F') \wedge \forall (A, A') \in r_1.\ (F\ A, F'\ A') \in r_2$$

for every $F \in \mathit{Term}(\tau_1 \to \tau_2)$ and $F' \in \mathit{Term}(\tau_1' \to \tau_2')$.

Interestingly, it turns out that no such explicit enforcement on the *result of the relational action* is necessary for the one corresponding to $\forall$-types. That is, the following definition is the standard one, except for the $\preccurlyeq$-compliance condition on the *quantified relations*.

Definition 4.4. Let τ_1 and τ_1' be types with at most a single free variable, α say. Suppose R is a function that maps every $\tau_2, \tau_2' \in \mathit{Typ}$ and $r \in \mathit{Rel}^{\preccurlyeq}(\tau_2, \tau_2')$ to an $R_{\tau_2,\tau_2'}(r) \in \mathit{Rel}(\tau_1[\tau_2/\alpha], \tau_1'[\tau_2'/\alpha])$. Then we define $(\forall R) \in \mathit{Rel}(\forall\alpha.\tau_1, \forall\alpha.\tau_1')$ by

$$(G, G') \in (\forall R) \;\text{ iff }\; \forall \tau_2, \tau_2' \in \mathit{Typ}, r \in \mathit{Rel}^{\preccurlyeq}(\tau_2, \tau_2').\ (G_{\tau_2}, G'_{\tau_2'}) \in R_{\tau_2,\tau_2'}(r)$$

for every $G \in \mathit{Term}(\forall\alpha.\tau_1)$ and $G' \in \mathit{Term}(\forall\alpha.\tau_1')$. We also write $\forall R$ as $\forall r.R(r)$, suppressing reference to τ_2 and τ_2'.

The relational action for list types is also the standard one by structural lifting, appropriately combined with $\preccurlyeq\preccurlyeq$-closure. That is, given $\tau, \tau' \in \mathit{Typ}$ and $r \in \mathit{Rel}(\tau, \tau')$, we define $\mathit{list}(r) \in \mathit{Rel}(\tau\text{-}\mathit{list}, \tau'\text{-}\mathit{list})$ as the greatest (post-)fixpoint (with respect to set inclusion) of the mapping $s \mapsto (1 + (r \times s))^{\preccurlyeq\preccurlyeq}$ for $s \in \mathit{Rel}(\tau\text{-}\mathit{list}, \tau'\text{-}\mathit{list})$, where

$$1 + (r \times s) = \{(\mathbf{nil}_\tau, \mathbf{nil}_{\tau'})\} \cup \{(H : T, H' : T') \mid (H, H') \in r \wedge (T, T') \in s\}$$

for every such s. The existence of the greatest fixpoint is guaranteed by monotonicity of the mapping $s \mapsto (1 + (r \times s))^{\preccurlyeq\preccurlyeq}$ with respect to set inclusion, which in turn follows from (3). Note that for every $r \in \mathit{Rel}$:

$$\mathit{list}(r) = (1 + (r \times \mathit{list}(r)))^{\preccurlyeq\preccurlyeq}. \tag{4}$$

Finally, the relational actions can be combined to define a logical relation by induction on the structure of Core types. It maps a type and a list containing relations as interpretations for the type's free variables to a new relation.

Definition 4.5. For every type τ, $n \in \mathbb{N}$, list $\vec{\alpha} = \alpha_1, \ldots, \alpha_n$ of distinct type variables containing the free variables of τ, lists $\vec{\tau} = \tau_1, \ldots, \tau_n$ and $\vec{\tau}' = \tau_1', \ldots, \tau_n'$ of closed types, and list $\vec{r} = r_1, \ldots, r_n$ with $r_i \in \mathit{Rel}(\tau_i, \tau_i')$

for every $1 \leq i \leq n$, we define $\Delta_\tau(\vec{r}/\vec{\alpha}) \in Rel(\tau[\vec{\tau}/\vec{\alpha}], \tau[\vec{\tau}'/\vec{\alpha}])$ by induction on the structure of τ as follows:

$$\Delta_{\alpha_i}(\vec{r}/\vec{\alpha}) = r_i \tag{5}$$
$$\Delta_{\tau' \to \tau''}(\vec{r}/\vec{\alpha}) = \Delta_{\tau'}(\vec{r}/\vec{\alpha}) \to \Delta_{\tau''}(\vec{r}/\vec{\alpha}) \tag{6}$$
$$\Delta_{\forall\alpha.\tau'}(\vec{r}/\vec{\alpha}) = \forall r.\Delta_{\tau'}(\vec{r}, r^{\preccurlyeq\preccurlyeq}/\vec{\alpha}, \alpha) \tag{7}$$
$$\Delta_{\tau'\text{-}list}(\vec{r}/\vec{\alpha}) = list(\Delta_{\tau'}(\vec{r}/\vec{\alpha})). \tag{8}$$

Note that without loss of generality the variable bound in the head of $\forall\alpha.\tau'$ in clause (7) can be assumed to not occur in $\vec{\alpha}$.

4.3 Preservation of Restrictions

The task of this subsection is to establish that $\Delta_\tau(\vec{r}/\vec{\alpha})$ is $\preccurlyeq$-compliant and $\preccurlyeq\preccurlyeq$-closed provided every relation in $\vec{r}$ is. This entails showing how these restrictions are pushed along the type structure by the relational actions. Due to the explicit enforcement of $\preccurlyeq$-compliance in Definition 4.3, the following observation is trivial.

Observation 4.6. *For every $r_1, r_2 \in Rel$, we have $(r_1 \to r_2) \in Rel^{\preccurlyeq}$.*

In contrast, since no explicit enforcement takes place in Definition 4.4, the corresponding statement regarding the relational action for $\forall$-types depends on a precondition and requires an explicit proof.

Lemma 4.7. *Let R be as in Definition 4.4. If $R_{\tau_2,\tau_2'}(r) \in Rel^{\preccurlyeq}$ for some $\tau_2, \tau_2' \in Typ$ and $r \in Rel^{\preccurlyeq}(\tau_2, \tau_2')$, then also $(\forall R) \in Rel^{\preccurlyeq}$.*

Proof: Let $(G, G') \in (\forall R)$. Then $(G_{\tau_2}, G'_{\tau_2'}) \in R_{\tau_2,\tau_2'}(r)$ and thus $\omega(G_{\tau_2}) \preccurlyeq \omega(G'_{\tau_2'})$, from which $\omega(G) \preccurlyeq \omega(G')$ follows by the final observation in Definition 2.6. □

Regarding the relational action for list types, it suffices to note that for every $r \in Rel$ the relation $1 + (r \times list(r))$ is $\preccurlyeq$-compliant since for every (L, L') contained in it, we have $\omega(L) = 0 = \omega(L')$. This immediately gives the following observation by Lemma 4.1 and (4).

Observation 4.8. *For every $r \in Rel$, we have $list(r) \in Rel^{\preccurlyeq}$.*

Now we turn to $\preccurlyeq\preccurlyeq$-closedness.

Lemma 4.9. *For every $r_1, r_2 \in Rel$, if r_2 is $\preccurlyeq\preccurlyeq$-closed, then so is $r_1 \to r_2$.*

Proof: We have to show that $(F, F') \in (r_1 \to r_2)^{\preccurlyeq\preccurlyeq}$ implies $(F, F') \in (r_1 \to r_2)$, i.e., $\omega(F) \preccurlyeq \omega(F')$ and for every $(A, A') \in r_1$, $(F\ A, F'\ A') \in r_2$. The former holds because $(r_1 \to r_2)^{\preccurlyeq\preccurlyeq}$ is $\preccurlyeq$-compliant by Observation 4.6 and Lemma 4.1. The latter follows from $\preccurlyeq\preccurlyeq$-closedness of r_2 if we can show that $(A, A') \in r_1$ and $(F, F') \in (r_1 \to r_2)^{\preccurlyeq\preccurlyeq}$ together imply $(F\ A, F'\ A') \in r_2^{\preccurlyeq\preccurlyeq}$. To

do so, we reason for every $(S, S') \in r_2^{\preccurlyeq}$ as follows:

$$\begin{aligned} \top(S, F\ A) &= \top(S \circ (-\ A), F) && \text{by Observation 3.4(a)} \\ &\preccurlyeq \top(S' \circ (-\ A'), F') \\ &= \top(S', F'\ A') && \text{by Observation 3.4(a).} \end{aligned}$$

Here $\top(S \circ (-\ A), F) \preccurlyeq \top(S' \circ (-\ A'), F')$ holds by $(F, F') \in (r_1 \to r_2)^{\preccurlyeq\preccurlyeq}$ and $(S \circ (-\ A), S' \circ (-\ A')) \in (r_1 \to r_2)^{\preccurlyeq}$. The latter is established by reasoning for every $(N, N') \in (r_1 \to r_2)$ as follows:

$$\begin{aligned} \top(S \circ (-\ A), N) &= \top(S, N\ A) && \text{by Observation 3.4(a)} \\ &\preccurlyeq \top(S', N'\ A') \\ &= \top(S' \circ (-\ A'), N') && \text{by Observation 3.4(a).} \end{aligned}$$

Here $\top(S, N\ A) \preccurlyeq \top(S', N'\ A')$ holds by $(S, S') \in r_2^{\preccurlyeq}$ and $(N\ A, N'\ A') \in r_2$. The latter follows from $(N, N') \in (r_1 \to r_2)$ and $(A, A') \in r_1$ by the definition of $r_1 \to r_2$. □

Lemma 4.10. *Let R be as in Definition 4.4. If $R_{\tau_2,\tau_2'}(r)$ is $\preccurlyeq\preccurlyeq$-closed for every $\tau_2, \tau_2' \in Typ$ and $r \in Rel^{\preccurlyeq}(\tau_2, \tau_2')$, then $\forall R$ is also $\preccurlyeq\preccurlyeq$-closed.*

We omit the proof, which is very similar to that of Lemma 4.9.

Now, the desired statement about the propagation of $\preccurlyeq$-compliance and $\preccurlyeq\preccurlyeq$-closedness can easily be proved by induction on the structure of τ, using Lemmas 4.1, 4.7, 4.9, and 4.10, Observations 4.6 and 4.8, and (2) and (4).

Lemma 4.11. *Let τ, $\vec{\alpha}$, and $\vec{r}$ be as in Definition 4.5. If every relation in $\vec{r}$ is $\preccurlyeq$-compliant and $\preccurlyeq\preccurlyeq$-closed, then so is $\Delta_\tau(\vec{r}/\vec{\alpha})$.*

By similar inductions we easily obtain the following two results as well.

Observation 4.12. *Let τ, $\vec{\alpha}$, and $\vec{r}$ be as in Definition 4.5. Moreover, let $r' \in Rel$ and let α' be a type variable not occurring in $\vec{\alpha}$ (and hence not occurring free in τ). Then $\Delta_\tau(\vec{r}, r'/\vec{\alpha}, \alpha') = \Delta_\tau(\vec{r}/\vec{\alpha})$.*

Observation 4.13. *Let τ, $\vec{\alpha}$, and $\vec{r}$ be as in Definition 4.5. Moreover, let α' be a type variable not occurring in $\vec{\alpha}$ and let τ' be a type with free variables in $\vec{\alpha}, \alpha'$. Then $\Delta_{\tau'[\tau/\alpha']}(\vec{r}/\vec{\alpha}) = \Delta_{\tau'}(\vec{r}, \Delta_\tau(\vec{r}/\vec{\alpha})/\vec{\alpha}, \alpha')$.*

4.4 Manufacturing Permissible Relations

For later applications of the logical relation we need a source of appropriately restricted relations, that is, ones that are $\preccurlyeq$-compliant and $\preccurlyeq\preccurlyeq$-closed. Such a source is obtained by considering two dual notions of graphs of typed stacks up to the logical relation. For every $\tau, \tau' \in Typ$ and $S \in Stack(\tau, \tau')$, we define

left-graph$_S \in \mathit{Rel}(\tau, \tau')$ by

$$(M, M') \in \textit{left-graph}_S \text{ iff } (S\ M, M') \in \Delta_{\tau'}()$$

and *right-graph*$_S \in \mathit{Rel}(\tau', \tau)$ by

$$(M', M) \in \textit{right-graph}_S \text{ iff } (M', S\ M) \in \Delta_{\tau'}().$$

To investigate whether (or when) *left-graph*$_S$ and/or *right-graph*$_S$ is $\preccurlyeq$-compliant — that is, whether $(S\ M, M') \in \Delta_{\tau'}()$ implies $\omega(M) \preccurlyeq \omega(M')$ in the first case, and dually for the second case — it seems necessary to establish some $\preccurlyeq$-relationship between $\omega(M)$ and $\omega(S\ M)$. (Note that $\Delta_{\tau'}()$ is $\preccurlyeq$-compliant by Lemma 4.11, so a corresponding relation between $\omega(S\ M)$ and $\omega(M')$ is already known in both cases.) For the case that M is not converging, the following lemma is helpful.

Lemma 4.14. *Let $\tau, \tau' \in \mathit{Typ}$, $S \in \mathit{Stack}(\tau, \tau')$, and $M \in \mathit{Term}(\tau)$. If $\omega(M) \neq 0$, then $\omega(S\ M) = \omega(M)$.*

Proof: By Corollary 3.5, we have $\omega(S\ M) = \top(S, M)$. So we have to show that for every $i \in \mathbb{N}_+$, $\omega(M) = i$ implies $\top(S, M) = i$, and that $\omega(M) = \infty$ implies $\top(S, M) = \infty$. The former follows easily by combining Definitions 2.1, 2.6, and 3.1 with Observation 2.4. For the latter, assume $\top(S, M) \neq \infty$. Then $([\![S]\!], [\![M]\!])$ must lead to an end configuration. By Lemma 2.3 this contradicts $\omega(M) = \infty$. □

Note that with $S = (-\ A)$ and $M = F$, Lemma 4.14 implies the following corollary, to be used later in Section 6.

Corollary 4.15. *Let $\tau, \tau' \in \mathit{Typ}$, $F \in \mathit{Term}(\tau \to \tau')$, and $A \in \mathit{Term}(\tau)$. If $0 \preccurlyeq \omega(F\ A)$, then $0 \preccurlyeq \omega(F)$. If $\omega(F\ A) \preccurlyeq 0$, then $\omega(F) \preccurlyeq 0$.*

But the real worth of Lemma 4.14 here is that it allows us to restrict our attention in the above investigation to M with $\omega(M) = 0$. For *left-graph*$_S$, we then have to ensure that $\omega(M) = 0$ and $\omega(S\ M) \preccurlyeq \omega(M')$ imply $\omega(M) \preccurlyeq \omega(M')$. This motivates the following definition(s), as well as the following lemma, whose proof essentially assembles the whole line of reasoning above.

Given $\tau \in \mathit{Typ}$ and $S \in \mathit{Stack}(\tau)$, we say that S is $\preccurlyeq$-*upwards* (or $\preccurlyeq$-*downwards*) if for every $M \in \mathit{Term}(\tau)$ with $\omega(M) = 0$, we have $0 \preccurlyeq \omega(S\ M)$ (or $\omega(S\ M) \preccurlyeq 0$, respectively).

Lemma 4.16. *Let $\tau, \tau' \in \mathit{Typ}$ and $S \in \mathit{Stack}(\tau, \tau')$. Then left-graph$_S$ and right-graph$_S$ are $\preccurlyeq\preccurlyeq$-closed. Moreover, if S is $\preccurlyeq$-upwards, then left-graph$_S$ is $\preccurlyeq$-compliant. Also, if S is $\preccurlyeq$-downwards, then right-graph$_S$ is $\preccurlyeq$-compliant.*

Proof: To prove the statement regarding $\preccurlyeq\preccurlyeq$-closedness for *left-graph*$_S$, we have to show that $(M, M') \in (\textit{left-graph}_S)^{\preccurlyeq\preccurlyeq}$ implies $(S\ M, M') \in \Delta_{\tau'}()$. By Lemma 4.11, $\Delta_{\tau'}()$ is $\preccurlyeq\preccurlyeq$-closed, so it suffices to show that $(M, M') \in (\textit{left-graph}_S)^{\preccurlyeq\preccurlyeq}$ and $(S', S'') \in (\Delta_{\tau'}())^{\preccurlyeq}$ imply $\top(S', S\ M) \preccurlyeq \top(S'', M')$. But

this follows from Corollary 3.5 and the following reasoning for every $(N, N') \in$ *left-graph*$_S$, which establishes $(S' \mathbin{@} S, S'') \in (\textit{left-graph}_S)^{\preccurlyeq}$ from $(S', S'') \in (\Delta_{\tau'}())^{\preccurlyeq}$:

$$\begin{aligned} \top(S' \mathbin{@} S, N) &= \top(S', S\ N) && \text{by Corollary 3.5} \\ &\preccurlyeq \top(S'', N') && \text{by } (S', S'') \in (\Delta_{\tau'}())^{\preccurlyeq} \text{ and } (S\ N, N') \in \Delta_{\tau'}(), \end{aligned}$$

where $(S\ N, N') \in \Delta_{\tau'}()$ follows from $(N, N') \in$ *left-graph*$_S$ by definition. The proof of the statement regarding $\preccurlyeq\preccurlyeq$-closedness for *right-graph*$_S$ is completely analogous.

To prove the statement regarding $\preccurlyeq$-compliance for *left-graph*$_S$, let $(M, M') \in$ *left-graph*$_S$, that is, $(S\ M, M') \in \Delta_{\tau'}()$. By Lemma 4.11 we then immediately have $\omega(S\ M) \preccurlyeq \omega(M')$. So it suffices to show that $\omega(M) \preccurlyeq \omega(S\ M)$. But this follows from $\preccurlyeq$-upwardness of S by Lemma 4.14. The proof of the statement regarding $\preccurlyeq$-compliance for *right-graph*$_S$ is completely analogous. □

5 The Characterization Result

To characterize the largest reflexive, transitive, compatible, substitutive, and $\preccurlyeq$-adequate relation via the logical relation from Definition 4.5, we first have to lift the latter from closed terms to a relation on terms possibly containing free variables, that is, to a relation in the sense of Definition 2.5. This is done, as usual, via closing substitutions.

Definition 5.1. Let $n, m \in \mathbb{N}$, let $\vec{\alpha}$ be a list of n type variables, $\vec{x} = x_1, \ldots, x_m$ be a list of term variables, $\tau_1, \ldots, \tau_m$ be types, and $\Gamma = \vec{\alpha}, x_1 :: \tau_1, \ldots, x_m :: \tau_m$. Given typed terms M and M' and a type τ with $\Gamma \vdash M :: \tau$ and $\Gamma \vdash M' :: \tau$, we write

$$\Gamma \vdash M\ \Delta\ M' :: \tau$$

if for every pair of lists $\vec{\sigma} = \sigma_1, \ldots, \sigma_n$ and $\vec{\sigma'} = \sigma'_1, \ldots, \sigma'_n$ of closed types and every list $\vec{r} = r_1, \ldots, r_n$ of $\preccurlyeq\preccurlyeq$-closed $r_i \in \mathit{Rel}^{\preccurlyeq}(\sigma_i, \sigma'_i)$, we have that for every pair of lists $\vec{N} = N_1, \ldots, N_m$ and $\vec{N'} = N'_1, \ldots, N'_m$ with $(N_j, N'_j) \in \Delta_{\tau_j}(\vec{r}/\vec{\alpha})$ for every $1 \leq j \leq m$, the following membership holds:

$$(M[\vec{\sigma}/\vec{\alpha}, \vec{N}/\vec{x}], M'[\vec{\sigma'}/\vec{\alpha}, \vec{N'}/\vec{x}]) \in \Delta_\tau(\vec{r}/\vec{\alpha}).$$

The first step to be taken now is to prove the fundamental property of the logical relation, namely that it is reflexive. Actually, one proves the stronger statement that Δ is compatible. For this, a number of "compatibility lemmas" are needed. Here we do not discuss those for term formers already present in the Girard-Reynolds calculus and actually also omit the ones related to list types, which are just as standard.

So let us first consider selective strictness. What we want is the following lemma, where the choice of restrictions imposed on r_1 and r_2 is inspired by Lemma 4.19 in [16].

Lemma 5.2. *Let $\tau_1, \tau_1', \tau_2, \tau_2' \in Typ$, $r_1 \in Rel^{\preccurlyeq}(\tau_1, \tau_1')$, $r_2 \in Rel(\tau_2, \tau_2')$, and $(A, A') \in r_1$. Let x be a term variable and B and B' be typed terms such that $x :: \tau_1 \vdash B :: \tau_2$ and $x :: \tau_1' \vdash B' :: \tau_2'$. If $(B[A/x], B'[A'/x]) \in r_2$ and r_2 is $\preccurlyeq\preccurlyeq$-closed, then $(\mathbf{let!}\ x = A\ \mathbf{in}\ B, \mathbf{let!}\ x = A'\ \mathbf{in}\ B') \in r_2$.*

It turns out, however, that a proof does not succeed if we do not at least impose the following (further) two restrictions on $\preccurlyeq$:

$$\forall a \in \mathbb{N}_{+,\infty}.\ 0 \preccurlyeq a \ \Rightarrow\ \forall b \in \mathbb{N}_{\infty}.\ b \preccurlyeq a \tag{9}$$

$$\forall a \in \mathbb{N}_{+,\infty}.\ a \preccurlyeq 0 \ \Rightarrow\ \forall b \in \mathbb{N}_{\infty}.\ a \preccurlyeq b \tag{10}$$

Note that both quantifications for a exclude $a = 0$. Assuming from now on that (9) and (10) hold for the relation $\preccurlyeq$ under consideration, we can prove the above lemma as follows.

Proof of Lemma 5.2: Let $(S, S') \in r_2^{\preccurlyeq}$. By Lemma 3.7 we have

$$\top(S, \mathbf{let!}\ x = A\ \mathbf{in}\ B) = \begin{cases} \omega(A) & \text{if } \omega(A) \neq 0 \\ \top(S, B[A/x]) & \text{otherwise} \end{cases}$$

and

$$\top(S', \mathbf{let!}\ x = A'\ \mathbf{in}\ B') = \begin{cases} \omega(A') & \text{if } \omega(A') \neq 0 \\ \top(S', B'[A'/x]) & \text{otherwise.} \end{cases}$$

By $(A, A') \in r_1 \in Rel^{\preccurlyeq}$, $(S, S') \in r_2^{\preccurlyeq}$, and $(B[A/x], B'[A'/x]) \in r_2$, we have $\omega(A) \preccurlyeq \omega(A')$ and $\top(S, B[A/x]) \preccurlyeq \top(S', B'[A'/x])$. There are four cases to consider.

Case a: $\omega(A) \neq 0$ and $\omega(A') \neq 0$. Then $\top(S, \mathbf{let!}\ x = A\ \mathbf{in}\ B) = \omega(A)$ and $\top(S', \mathbf{let!}\ x = A'\ \mathbf{in}\ B') = \omega(A')$, and thus $\top(S, \mathbf{let!}\ x = A\ \mathbf{in}\ B) \preccurlyeq \top(S', \mathbf{let!}\ x = A'\ \mathbf{in}\ B')$.

Case b: $\omega(A) = 0$ and $\omega(A') = 0$. Then $\top(S, \mathbf{let!}\ x = A\ \mathbf{in}\ B) = \top(S, B[A/x])$ and $\top(S', \mathbf{let!}\ x = A'\ \mathbf{in}\ B') = \top(S', B'[A'/x])$, and thus $\top(S, \mathbf{let!}\ x = A\ \mathbf{in}\ B) \preccurlyeq \top(S', \mathbf{let!}\ x = A'\ \mathbf{in}\ B')$.

Case c: $\omega(A) = 0$ and $\omega(A') \neq 0$. Then $\top(S, \mathbf{let!}\ x = A\ \mathbf{in}\ B) = \top(S, B[A/x])$ and $\top(S', \mathbf{let!}\ x = A'\ \mathbf{in}\ B') = \omega(A')$. Moreover, $\omega(A) \preccurlyeq \omega(A')$ and (9) then imply that $\top(S, B[A/x]) \preccurlyeq \omega(A')$, and thus $\top(S, \mathbf{let!}\ x = A\ \mathbf{in}\ B) \preccurlyeq \top(S', \mathbf{let!}\ x = A'\ \mathbf{in}\ B')$.

Case d: $\omega(A) \neq 0$ and $\omega(A') = 0$. Then $\top(S, \mathbf{let!}\ x = A\ \mathbf{in}\ B) = \omega(A)$ and $\top(S', \mathbf{let!}\ x = A'\ \mathbf{in}\ B') = \top(S', B'[A'/x])$. Moreover, $\omega(A) \preccurlyeq \omega(A')$ and (10)

then imply that $\omega(A) \preccurlyeq \top(S', B'[A'/x])$, and thus $\top(S, \mathbf{let!}\ x = A\ \mathbf{in}\ B) \preccurlyeq \top(S', \mathbf{let!}\ x = A'\ \mathbf{in}\ B')$.

Since $\top(S, \mathbf{let!}\ x = A\ \mathbf{in}\ B) \preccurlyeq \top(S', \mathbf{let!}\ x = A'\ \mathbf{in}\ B')$ for every choice of $(S, S') \in r_2^{\preccurlyeq}$, we have $(\mathbf{let!}\ x = A\ \mathbf{in}\ B, \mathbf{let!}\ x = A'\ \mathbf{in}\ B') \in r_2^{\preccurlyeq\preccurlyeq}$, from which the desired statement follows by $\preccurlyeq\preccurlyeq$-closedness of r_2. □

The rationale behind the restrictions (9) and (10), apart from that they make the above proof go through, will be discussed in Section 8.

The key lemma for fixpoint recursion is the following induction principle.
Lemma 5.3. *Let $\tau, \tau' \in Typ$, $F \in Term(\tau \to \tau)$, $F' \in Term(\tau' \to \tau')$, and $r \in Rel(\tau, \tau')$. If r is $\preccurlyeq\preccurlyeq$-closed and for every $(A, A') \in r$, we also have $(F\ A, F'\ A') \in r$, then $(\mathbf{fix}(F), \mathbf{fix}(F')) \in r$.*
Proof: By the definitions of $\top$ and $[\![\cdot]\!]$ and Observation 2.2, we have $\top(S, \mathbf{fix}(\lambda x :: \tau.x)) \preccurlyeq \top(S', \mathbf{fix}(\lambda x :: \tau'.x))$ for every $(S, S') \in r^{\preccurlyeq}$. Thus, we have $(\mathbf{fix}(\lambda x :: \tau.x), \mathbf{fix}(\lambda x :: \tau'.x)) \in r^{\preccurlyeq\preccurlyeq}$. Using the preconditions of the lemma, it follows from this by induction on natural numbers that for every $n \in \mathbb{N}$, we have $(F^n\ \mathbf{fix}(\lambda x :: \tau.x), F'^n\ \mathbf{fix}(\lambda x :: \tau'.x)) \in r$. Now, let $(S, S') \in r^{\preccurlyeq}$. By Lemma 3.8 there exists $n \in \mathbb{N}$ such that $\top(S, \mathbf{fix}(F)) = \top(S, F^n\ \mathbf{fix}(\lambda x :: \tau.x))$ and $\top(S', \mathbf{fix}(F')) = \top(S', F'^n\ \mathbf{fix}(\lambda x :: \tau'.x))$. By the above, this implies $\top(S, \mathbf{fix}(F)) \preccurlyeq \top(S', \mathbf{fix}(F'))$. Since this is so for every choice of $(S, S') \in r^{\preccurlyeq}$, we have $(\mathbf{fix}(F), \mathbf{fix}(F')) \in r^{\preccurlyeq\preccurlyeq}$, from which the desired $(\mathbf{fix}(F), \mathbf{fix}(F')) \in r$ follows by $\preccurlyeq\preccurlyeq$-closedness of r. □

For finite failure, the required lemma is almost trivial.
Lemma 5.4. *Let $\tau, \tau' \in Typ$, $r \in Rel(\tau, \tau')$, and $i \in \mathbb{N}_+$. If r is $\preccurlyeq\preccurlyeq$-closed, then $(\mathbf{error}_\tau(i), \mathbf{error}_{\tau'}(i)) \in r$.*
Proof: For every $(S, S') \in r^{\preccurlyeq}$, we have $\top(S, \mathbf{error}_\tau(i)) = i = \top(S', \mathbf{error}_{\tau'}(i))$. Together with $\preccurlyeq\preccurlyeq$-closedness of r, this suffices. □

We can now prove the following important theorem.
Theorem 5.5. *The relation Δ is compatible and substitutive. In particular, for every $\tau \in Typ$ and $M \in Term(\tau)$, we have $(M, M) \in \Delta_\tau()$.*
Proof: We have to show that Δ is closed under each of the axioms and rules in Figures 3 and 4. The axiom $\Gamma, x :: \tau \vdash x\ \Delta\ x :: \tau$ is trivially satisfied due to the way Δ is defined. Also by that definition, to establish the rule

$$\frac{\Gamma \vdash A\ \Delta\ A' :: \tau \qquad \Gamma, x :: \tau \vdash B\ \Delta\ B' :: \tau'}{\Gamma \vdash (\mathbf{let!}\ x = A\ \mathbf{in}\ B)\ \Delta\ (\mathbf{let!}\ x = A'\ \mathbf{in}\ B') :: \tau'}$$

it suffices to show that for Γ as in Definition 5.1, a term variable x not among the $\vec{x}$, types τ and τ', typed terms A, A', B, and B' with $\Gamma \vdash A :: \tau$, $\Gamma \vdash A' :: \tau$, $\Gamma, x :: \tau \vdash B :: \tau'$, and $\Gamma, x :: \tau \vdash B' :: \tau'$, lists $\vec{\sigma} = \sigma_1, \ldots, \sigma_n$ and $\vec{\sigma}' = \sigma'_1, \ldots, \sigma'_n$ of closed types, list $\vec{r} = r_1, \ldots, r_n$ of $\preccurlyeq\preccurlyeq$-closed $r_i \in Rel^{\preccurlyeq}(\sigma_i, \sigma'_i)$, list $\vec{N} = N_1, \ldots, N_m$ of $N_j \in Term(\tau_j[\vec{\sigma}/\vec{\alpha}])$, and list $\vec{N'} = N'_1, \ldots, N'_m$ of

$N'_j \in \mathit{Term}(\tau_j[\vec{\sigma}'/\vec{\alpha}])$,

$$(A[\vec{\sigma}/\vec{\alpha}, \vec{N}/\vec{x}], A'[\vec{\sigma}'/\vec{\alpha}, \vec{N}'/\vec{x}]) \in \Delta_\tau(\vec{r}/\vec{\alpha})$$

and

$$\forall (N, N') \in \Delta_\tau(\vec{r}/\vec{\alpha}).\ (B[\vec{\sigma}/\vec{\alpha}, \vec{N}/\vec{x}, N/x], B'[\vec{\sigma}'/\vec{\alpha}, \vec{N}'/\vec{x}, N'/x]) \in \Delta_{\tau'}(\vec{r}/\vec{\alpha})$$

imply

$$((\mathbf{let!}\ x = A\ \mathbf{in}\ B)[\vec{\sigma}/\vec{\alpha}, \vec{N}/\vec{x}], (\mathbf{let!}\ x = A'\ \mathbf{in}\ B')[\vec{\sigma}'/\vec{\alpha}, \vec{N}'/\vec{x}]) \in \Delta_{\tau'}(\vec{r}/\vec{\alpha}).$$

But this is indeed so by Lemma 5.2, taking into account that by Lemma 4.11, $\Delta_\tau(\vec{r}/\vec{\alpha})$ is $\preccurlyeq$-compliant and $\Delta_{\tau'}(\vec{r}/\vec{\alpha})$ is $\preccurlyeq\preccurlyeq$-closed. The remaining axioms and rules are established in a similar fashion, additionally using (1) and (4), Observation 4.13, Lemmas 4.1, 5.3, and 5.4, and the aforementioned additional compatibility lemmas. □

One further important property that Δ should have is $\preccurlyeq$-adequacy.

Lemma 5.6. *The relation Δ is $\preccurlyeq$-adequate.*

Proof: Let $\tau \in \mathit{Typ}$ and $M, M' \in \mathit{Term}(\tau)$. If $M\ \Delta\ M'$, then $(M, M') \in \Delta_\tau()$ by the definition of Δ. Since $\Delta_\tau()$ is $\preccurlyeq$-compliant by Lemma 4.11, this implies $\omega(M) \preccurlyeq \omega(M')$. □

Our main theorem is that Δ is not just any compatible, substitutive, and $\preccurlyeq$-adequate relation, but is actually exactly the one we are interested in.

Theorem 5.7. *The relation Δ is the largest compatible, substitutive, and $\preccurlyeq$-adequate relation. It is also reflexive and transitive.*

Proof: By Theorem 5.5 and Lemma 5.6, Δ is compatible, substitutive, and $\preccurlyeq$-adequate. Since it is compatible, it is reflexive as well.

For the first statement of the theorem, it remains to prove that Δ subsumes every compatible, substitutive, and $\preccurlyeq$-adequate relation. Let $\mathcal{E}$ be such a relation, let Γ, M, M', and τ be as in Definition 5.1, and assume $\Gamma \vdash M\ \mathcal{E}\ M' :: \tau$. Further, let $\vec{\sigma} = \sigma_1, \ldots, \sigma_n$ and $\vec{\sigma}' = \sigma'_1, \ldots, \sigma'_n$ be lists of closed types, $\vec{r} = r_1, \ldots, r_n$ be a list of $\preccurlyeq\preccurlyeq$-closed $r_i \in \mathit{Rel}^{\preccurlyeq}(\sigma_i, \sigma'_i)$, and $\vec{N} = N_1, \ldots, N_m$ and $\vec{N}' = N'_1, \ldots, N'_m$ be lists with $(N_j, N'_j) \in \Delta_{\tau_j}(\vec{r}/\vec{\alpha})$ for every $1 \leq j \leq m$. Since Δ is reflexive, we have $\Gamma \vdash M\ \Delta\ M :: \tau$, which by Definition 5.1 implies that

$$(M[\vec{\sigma}/\vec{\alpha}, \vec{N}/\vec{x}], M[\vec{\sigma}'/\vec{\alpha}, \vec{N}'/\vec{x}]) \in \Delta_\tau(\vec{r}/\vec{\alpha}). \tag{11}$$

Moreover, $\Gamma \vdash M\ \mathcal{E}\ M' :: \tau$ and the substitutivity of $\mathcal{E}$ imply that

$$M[\vec{\sigma}'/\vec{\alpha}, \vec{N}'/\vec{x}]\ \mathcal{E}\ M'[\vec{\sigma}'/\vec{\alpha}, \vec{N}'/\vec{x}]. \tag{12}$$

Since (11) and (12) combine into $(M[\vec{\sigma}/\vec{\alpha}, \vec{N}/\vec{x}], M'[\vec{\sigma}'/\vec{\alpha}, \vec{N}'/\vec{x}]) \in \Delta_\tau(\vec{r}/\vec{\alpha})$ by Lemmas 4.2 and 4.11, and since this is obtained independently of the

choice of the lists $\vec{\sigma}$, $\vec{\sigma}'$, $\vec{r}$, $\vec{N}$, and $\vec{N}'$ above, we indeed have the desired $\Gamma \vdash M \;\Delta\; M' :: \tau$ by Definition 5.1.

For the second statement of the theorem, it remains to prove that Δ is transitive. It is easy to see that the collection of compatible, substitutive, and $\preccurlyeq$-adequate relations is closed under relation composition. This implies that $\Delta;\Delta$ is compatible, substitutive, and $\preccurlyeq$-adequate, and is thus subsumed by the largest such relation, i.e., $\Delta;\Delta \subseteq \Delta$. $\square$

Henceforth, we will use the reflexivity and transitivity of Δ without explicit mention.

Before moving on to a larger application in the next section, let us derive some useful consequences of what we have learned about Δ. The first one tells us that any notion of program equivalence or approximation induced by some $\preccurlyeq$ is closed under equality and small-step reductions (in either direction) of type erasures.

Lemma 5.8. *For every $M, M' \in \mathit{Term}$ of the same type, if $[\![M]\!] = [\![M']\!]$ or $[\![M]\!] \rightsquigarrow [\![M']\!]$, then $M \;\Delta\; M'$ and $M' \;\Delta\; M$.*

Proof: By definition, we have to show that $(M, M') \in \Delta_\tau()$ and $(M', M) \in \Delta_\tau()$, where τ is the type of both M and M'. By Lemma 4.11, $\Delta_\tau()$ is $\preccurlyeq\preccurlyeq$-closed and thus equal to $(\Delta_\tau())^{\preccurlyeq\preccurlyeq}$. Then it suffices to note that for every $(S, S') \in (\Delta_\tau())^{\preccurlyeq}$, $\top(S, M) = \top(S, M')$ and $\top(S', M') = \top(S', M)$ by Observation 3.3 or Observation 3.4(b), as well as $\top(S, M') \preccurlyeq \top(S', M')$ by $(S, S') \in (\Delta_\tau())^{\preccurlyeq}$ and $(M', M') \in \Delta_\tau()$, where the latter holds by Theorem 5.5. $\square$

Another interesting direction is to look at extensionality principles for arbitrary Δ. To consider the extensionality principle for $\forall$-types, let τ be a type with at most a single free variable, α say. Then for every $G, G' \in \mathit{Term}(\forall\alpha.\tau)$ we have $G \;\Delta\; G'$ if and only if for every $\tau' \in \mathit{Typ}$ it holds that $G_{\tau'} \;\Delta\; G'_{\tau'}$. The proof, which uses Lemmas 4.1, 4.2, and 4.11 and Theorem 5.5, is identical to that of Theorem 5.4 in [17]. In particular, no extra condition relating $\omega(G)$ and $\omega(G')$ appears in either the statement or the proof.

This is in contrast to Lemma 7.7 in [16], and also to the extensionality principle for function types in the present setting, which goes as follows. Let $\tau_1, \tau_2 \in \mathit{Typ}$. Then for every $F, F' \in \mathit{Term}(\tau_1 \to \tau_2)$ we have $F \;\Delta\; F'$ if and only if $\omega(F) \preccurlyeq \omega(F')$ and for every $A \in \mathit{Term}(\tau_1)$ it holds that $F\ A \;\Delta\; F'\ A$. The proof, which uses the $\preccurlyeq$-adequacy of Δ, Lemmas 4.2 and 4.11, and Theorem 5.5, is analogous to that of Lemma 7.6 in [16].

6 Application to Short Cut Fusion

The Core equivalents of the Haskell functions *foldr* and *build* from [9] are as follows:

$$\begin{aligned} \mathit{foldr} = \Lambda\alpha.\Lambda\beta.\lambda c :: \alpha \to \beta \to \beta.\lambda n :: \beta.\mathbf{fix}(\lambda f :: \alpha\text{-}\mathit{list} \to \beta.\lambda l :: \alpha\text{-}\mathit{list}. \\ \mathbf{case}\ l\ \mathbf{of}\ \{\mathbf{nil}\ \Rightarrow n; \\ h : t \Rightarrow c\ h\ (f\ t)\}) \end{aligned}$$

and

$$\mathit{build} = \Lambda\alpha.\lambda g :: \forall\beta.(\alpha \to \beta \to \beta) \to \beta \to \beta.g_{\alpha\text{-}\mathit{list}}\ (\lambda h :: \alpha.\lambda t :: \alpha\text{-}\mathit{list}.h : t)\ \mathbf{nil}_\alpha.$$

Note that $\mathit{foldr} \in \mathit{Term}(\forall\alpha.\forall\beta.(\alpha \to \beta \to \beta) \to \beta \to \alpha\text{-}\mathit{list} \to \beta)$ and $\mathit{build} \in \mathit{Term}(\forall\alpha.(\forall\beta.(\alpha \to \beta \to \beta) \to \beta \to \beta) \to \alpha\text{-}\mathit{list})$.

The $\mathit{foldr}/\mathit{build}$-rule for Core now says that for every $\tau, \tau' \in \mathit{Typ}$, $G \in \mathit{Term}(\forall\beta.(\tau \to \beta \to \beta) \to \beta \to \beta), C \in \mathit{Term}(\tau \to \tau' \to \tau')$, and $N \in \mathit{Term}(\tau')$, $(\mathit{foldr}_\tau)_{\tau'}\ C\ N\ (\mathit{build}_\tau\ G)$ should be transformed into $G_{\tau'}\ C\ N$. Our concern when considering (partial or total) correctness of this rule with respect to a given notion of program equivalence or approximation is thus to find out whether these two expressions are related by that notion, or whether there can at least be given preconditions on N and C that guarantee the two expressions to be so related.

To underscore the need for a very careful study here, and to set the scene for eventual interpretation and evaluation of our results, we want to give a few, perhaps surprising, examples of how selective strictness and different failure causes interact with $\mathit{foldr}/\mathit{build}$-fusion. To this end, consider the Haskell function *lastThatOrEmpty*, defined as follows in terms of foldl' from Section 2.2:

$$\begin{aligned} &\mathit{lastThatOrEmpty} :: \forall\alpha.\ (\alpha \to \mathit{Bool}) \to [\alpha] \to [\alpha] \\ &\mathit{lastThatOrEmpty}\ p = \mathit{foldl}'\ (\lambda z\ h \to \mathbf{if}\ p\ h\ \mathbf{then}\ [h]\ \mathbf{else}\ z)\ [] \end{aligned}$$

Given a predicate p and an input list l, it returns the singleton list containing the last element of l that satisfies p, or the empty list if no such element exists. It can be expressed in Core as an element of $\mathit{Term}(\forall\alpha.(\alpha \to (\forall\beta.\beta \to \beta \to \beta)) \to \alpha\text{-}\mathit{list} \to \alpha\text{-}\mathit{list})$ as follows, where we use the Church encoding of the boolean type, and at the same time employ *build* to abstract from list constructors:

$$\begin{aligned} &\Lambda\alpha.\lambda p :: \alpha \to (\forall\beta.\beta \to \beta \to \beta). \\ &\lambda l :: \alpha\text{-}\mathit{list}.\mathit{build}_\alpha\ (\Lambda\beta.\lambda c :: \alpha \to \beta \to \beta.\lambda n :: \beta. \\ &\qquad\qquad (\mathit{foldl}'_\alpha)_\beta\ (\lambda z :: \beta.\lambda h :: \alpha.(p\ h)_\beta\ (c\ h\ n)\ z)\ n\ l) \end{aligned}$$

However, for the sake of intuitive reading, we will use the Haskell version in the discussion here. In particular, we want to consider instances of fusion involving the following two producer functions (with *even* being the obvious predicate on the type *Int* of integers):

$$\begin{aligned} &\mathit{lastEvenOrEmpty} :: [\mathit{Int}] \to [\mathit{Int}] \\ &\mathit{lastEvenOrEmpty} = \mathit{lastThatOrEmpty}\ \mathit{even} \end{aligned}$$

and

$$\begin{aligned} &\mathit{lastOrEmpty} :: \forall\alpha.\ [\alpha] \to [\alpha] \\ &\mathit{lastOrEmpty} = \mathit{lastThatOrEmpty}\ (\lambda x \to \mathit{True}) \end{aligned}$$

On the consumer side, we will use two functions expressed via *foldr*:

$$\begin{aligned} &\mathit{headOr} :: \forall\alpha.\ [\alpha] \to [\alpha] \to [\alpha] \\ &\mathit{headOr} = \mathit{foldr}\ (\lambda h\ t \to [h]) \end{aligned}$$

and

$$\begin{aligned} &\mathit{assertEmptyElse} :: \forall\alpha.\ (\alpha \to [\alpha]) \to [\alpha] \to [\alpha] \\ &\mathit{assertEmptyElse}\ f = \mathit{foldr}\ (\lambda h\ t \to f\ h)\ [] \end{aligned}$$

The first of these can be used to return a singleton list containing the head element of a given list, with an alternative provided for the case that the input list is empty. For example,

$$\begin{aligned} &\mathit{headOrEmpty} :: \forall\alpha.\ [\alpha] \to [\alpha] \\ &\mathit{headOrEmpty} = \mathit{headOr}\ [] \end{aligned}$$

returns the empty list in that case, while

$$\begin{aligned} &\mathit{headOrError} :: \forall\alpha.\ [\alpha] \to [\alpha] \\ &\mathit{headOrError} = \mathit{headOr}\ (\mathit{error}\ \text{“empty list”}) \end{aligned}$$

produces an explicit error. The function *assertEmptyElse* checks that a given list is empty, if so, returns the empty list, and otherwise does whatever the argument function f tells it to do with the first list element. For example, the function

$$\begin{aligned} &\mathit{assertEmptyElseError} :: \forall\alpha.\ \mathit{Show}\ \alpha \Rightarrow [\alpha] \to [\alpha] \\ &\mathit{assertEmptyElseError} = \mathit{assertEmptyElse}\ (\mathit{error} \cdot \mathit{show}) \end{aligned}$$

then produces an explicit error mentioning the first element of the supposedly empty list, while the function

$$\begin{aligned} &\mathit{assertEmpty} :: \forall\alpha.\ [\alpha] \to [\alpha] \\ &\mathit{assertEmpty} = \mathit{assertEmptyElse}\ (\lambda h \to []) \end{aligned}$$

consumer · producer	before	after
headOrEmpty · *lastEvenOrEmpty*	[2]	[2]
headOrError · *lastEvenOrEmpty*	[2]	*error* "empty list"
assertEmptyElseError · *lastOrEmpty*	*error* "2"	*error* "1"
assertEmpty · *lastOrEmpty*	[]	[]

Table 1
Results on the input list $[1, 2]$ before and after *foldr*/*build*-fusion.

simply returns the empty list anyway. Let us now look at some combinations of consumers and producers. We apply each of these combinations to the input list $[1, 2]$ of type $[Int]$, and report the computed result before and after performing *foldr*/*build*-fusion in Table 1. The second and third lines are probably not what the reader would have expected, certainly not without performing *foldr*/*build*-fusion "by hand" and very carefully simulating the resulting program while taking the subtle use of *seq* in the definition of $foldl'$ into account.

These two lines show that it is both possible that an erstwhile normally terminating program suddenly leads to a runtime error after fusion, and that different runtime errors might get confused. Moreover, replacing *headOrError* by *headOr loop* for any nonterminating computation *loop* in the second line shows that it is also possible that a terminating program becomes nonterminating. Similarly, examples can be given where runtime errors and nontermination get confused. So at least two questions arise:

(a) Under what preconditions is *foldr*/*build*-fusion semantics-preserving, even when taking different kinds of failures into account and considering them as separate? Ideally, these conditions would explain why the first and fourth lines in Table 1 are unproblematic.
(b) Under what preconditions do we at least get partial correctness with respect to a chosen semantic approximation order, determined by its relative treatment of different failure causes? Ideally, this would allow us to formulate guarantees such as that the use of *headOrError*, rather than *headOr loop*, in the second line ensures that at least no nontermination will be introduced.

Once one has some experience with Haskell's selective strictness semantics, it might be easy to answer questions like these for specific instances like those in Table 1 by taking the definition of the producer function into account. But we aim at *general* statements formulated in terms of the arguments to the consuming *foldr* only. For later reference, Table 2 lists those arguments for the fusion instances considered in Table 1.

consumer	first arg. of *foldr*	second arg. of *foldr*
headOrEmpty	$\lambda h\ t \to [h]$	$[]$
headOrError	$\lambda h\ t \to [h]$	*error* "empty list"
assertEmptyElseError	$\lambda h\ t \to \mathit{error}\ (\mathit{show}\ h)$	$[]$
assertEmpty	$\lambda h\ t \to []$	$[]$

Table 2
Arguments to *foldr* for different consumers.

For our formal investigation we switch back to using Core rather than Haskell. And in the interest of maximal generality and reusability, we (again) refrain from restricting attention to a particular notion of program equivalence or approximation. Instead, we prove a more abstract statement that can then be instantiated for each such notion individually.

So let $\preccurlyeq$ be a preorder on $\mathbb{N}_\infty$ satisfying (9) and (10). To make more apparent that the logical relation we are dealing with below depends on $\preccurlyeq$, we will denote it as $\Delta^{\preccurlyeq}$. For the sake of readability, though, we avoid using similar indexing for the relational actions or the concept of left-graphs, even though these notions of course also depend on $\preccurlyeq$.

In principle, the proof goes along similar lines as earlier proofs for more specific (equational or inequational) statements for smaller extensions of the Girard-Reynolds calculus; see, e.g., [16–18]. First, by Theorem 5.5 and Definition 4.5, we obtain from the type of G alone that

$$(G, G) \in (\forall r.(\Delta_\tau^{\preccurlyeq}(r^{\preccurlyeq\preccurlyeq}/\beta) \to (r^{\preccurlyeq\preccurlyeq} \to r^{\preccurlyeq\preccurlyeq})) \to (r^{\preccurlyeq\preccurlyeq} \to r^{\preccurlyeq\preccurlyeq})).$$

By Definitions 4.3 and 4.4 and Observation 4.12 this implies that for every $r \in \mathit{Rel}^{\preccurlyeq}(\tau\text{-}\mathit{list}, \tau')$, $C_1 \in \mathit{Term}(\tau \to \tau\text{-}\mathit{list} \to \tau\text{-}\mathit{list})$, $C_2 \in \mathit{Term}(\tau \to \tau' \to \tau')$, $N_1 \in \mathit{Term}(\tau\text{-}\mathit{list})$, and $N_2 \in \mathit{Term}(\tau')$:

$$\begin{aligned}
&\omega(C_1) \preccurlyeq \omega(C_2)\\
&\wedge (\forall (A_1, A_2) \in \Delta_\tau^{\preccurlyeq}().\ \omega(C_1\ A_1) \preccurlyeq \omega(C_2\ A_2)\\
&\qquad\qquad \wedge \forall (B_1, B_2) \in r^{\preccurlyeq\preccurlyeq}.\ (C_1\ A_1\ B_1, C_2\ A_2\ B_2) \in r^{\preccurlyeq\preccurlyeq})\\
&\wedge (N_1, N_2) \in r^{\preccurlyeq\preccurlyeq}\\
\Rightarrow\ &(G_{\tau\text{-}\mathit{list}}\ C_1\ N_1, G_{\tau'}\ C_2\ N_2) \in r^{\preccurlyeq\preccurlyeq}.
\end{aligned}$$

From the form of the *foldr*/*build*-rule (and from earlier experience) we know that we want to instantiate $C_1 = \lambda h :: \tau.\lambda t :: \tau\text{-}\mathit{list}.h : t$, $C_2 = C$, $N_1 = \mathbf{nil}_\tau$, $N_2 = N$, and

$$r = \{(L, M) \mid L \in \mathit{Term}(\tau\text{-}\mathit{list}) \wedge M \in \mathit{Term}(\tau') \wedge ((\mathit{foldr}_\tau)_{\tau'}\ C\ N\ L)\ \Delta^{\preccurlyeq}\ M\}\,,$$

where the latter relation should be $\preccurlyeq\preccurlyeq$-closed and $\preccurlyeq$-compliant. To establish that it is, we first note that by the definition of *foldr*, Lemma 5.8, and the compatibility of $\Delta^{\preccurlyeq}$, it equals *left-graph*$_S$, where

$$\begin{aligned} S = \mathbf{case}\ -\ \mathbf{of}\ \{\mathbf{nil}\ &\Rightarrow N; \\ h : t &\Rightarrow C\ h\ (\mathbf{fix}(\lambda f :: \tau\text{-}\mathit{list} \to \tau'.\lambda l :: \tau\text{-}\mathit{list}. \\ &\qquad \mathbf{case}\ l\ \mathbf{of}\ \{\mathbf{nil}\ \Rightarrow N; \\ &\qquad\quad h' : t' \Rightarrow C\ h'\ (f\ t')\})\ t)\}. \end{aligned}$$

To successfully apply Lemma 4.16, we need to know that S is $\preccurlyeq$-upwards. For this, we need to consider $\omega(S\ M)$ for M with $\omega(M) = 0$. Intuitively, it is clear that for such M, that is, for converging M, the pattern matching in the only frame of S will successfully reduce to one of its two branches. So to guarantee $0 \preccurlyeq \omega(S\ M)$ then (as required for S to be $\preccurlyeq$-upwards), it suffices to require that $0 \preccurlyeq \omega(N)$ and that for every $A \in \mathit{Term}(\tau)$ and $B \in \mathit{Term}(\tau')$, $0 \preccurlyeq \omega(C\ A\ B)$. A more formal counterpart to this intuitive reasoning can be found as Lemma A.4 in the appendix. Since under the specified conditions on N and C we now know from Lemma 4.16 that r is $\preccurlyeq$-compliant and $\preccurlyeq\preccurlyeq$-closed, we also know that the choice of r was justified (i.e., it really is in $\mathit{Rel}^{\preccurlyeq}(\tau\text{-}\mathit{list}, \tau')$) and that occurrences of $r^{\preccurlyeq\preccurlyeq}$ in the implication displayed above can be replaced by r itself. Then the desired $((\mathit{foldr}_\tau)_{\tau'}\ C\ N\ (\mathit{build}_\tau\ G))\ \Delta^{\preccurlyeq}\ (G_{\tau'}\ C\ N)$ follows from the definition of *build*, Corollary 3.6, Lemma 5.8, and the compatibility of $\Delta^{\preccurlyeq}$, provided we can establish that

$$0 \preccurlyeq \omega(C)\,,$$

that

$$\forall A_1 \in \mathit{Term}(\tau).\ 0 \preccurlyeq \omega(C\ A_1)\,,$$

that

$$\begin{aligned} &\forall (A_1, A_2) \in \Delta^{\preccurlyeq}_\tau(), B_1 \in \mathit{Term}(\tau\text{-}\mathit{list}), B_2 \in \mathit{Term}(\tau'). \\ &\quad ((\mathit{foldr}_\tau)_{\tau'}\ C\ N\ B_1)\ \Delta^{\preccurlyeq}\ B_2 \\ &\Rightarrow ((\mathit{foldr}_\tau)_{\tau'}\ C\ N\ ((\lambda h :: \tau.\lambda t :: \tau\text{-}\mathit{list}.h : t)\ A_1\ B_1))\ \Delta^{\preccurlyeq}\ (C\ A_2\ B_2)\,, \end{aligned}$$

and that

$$((\mathit{foldr}_\tau)_{\tau'}\ C\ N\ \mathbf{nil}_\tau)\ \Delta^{\preccurlyeq}\ N.$$

But the first two conditions follow by Corollary 4.15 from the condition imposed on C above, whereas the other two statements follow from the definition of *foldr*, Lemma 5.8, and the compatibility of $\Delta^{\preccurlyeq}$. So altogether we have proved the following theorem.

Theorem 6.1. *If* $0 \preccurlyeq \omega(N)$ *and for every* $A \in \mathit{Term}(\tau)$ *and* $B \in \mathit{Term}(\tau')$, $0 \preccurlyeq \omega(C\ A\ B)$, *then* $((\mathit{foldr}_\tau)_{\tau'}\ C\ N\ (\mathit{build}_\tau\ G))\ \Delta^{\preccurlyeq}\ (G_{\tau'}\ C\ N)$.

By simply instantiating $\preccurlyeq$ to particular preorders on $\mathbb{N}_\infty$ satisfying (9) and (10), and using Theorem 5.7, we can now obtain partial and total correctness results

for *foldr*/*build*-fusion with respect to a variety of observational approximation and equivalence relations without having to repeat any proof. We illustrate this for some of the observational relations presented in Example 2.7.

Consider the observational equivalence relation $\equiv_1$ that semantically distinguishes between every pair of different failure causes. The underlying choice for $\preccurlyeq$ is the one given under (a) in Example 2.7. For this $\preccurlyeq$, the conditions $0 \preccurlyeq \omega(N)$ and $0 \preccurlyeq \omega(C\ A\ B)$ in Theorem 6.1 are obviously equivalent to $\omega(N) = 0$ and $\omega(C\ A\ B) = 0$, and thus to $N{\Downarrow}$ and $(C\ A\ B){\Downarrow}$, respectively. So we immediately get the following corollary.

Corollary 6.2. *If* $N{\Downarrow}$ *and for every* $A \in \mathit{Term}(\tau)$ *and* $B \in \mathit{Term}(\tau')$, $(C\ A\ B){\Downarrow}$, *then* $((\mathit{foldr}_\tau)_{\tau'}\ C\ N\ (\mathit{build}_\tau\ G)) \equiv_1 (G_{\tau'}\ C\ N)$.

From choice (b) in Example 2.7 we get, via exactly the same reasoning, the same result for the observational equivalence relation $\equiv_2$ that unifies all different failure causes, and thus essentially corresponds to the one from [20]. The corresponding partial correctness result from [16] is obtained from the inverse of choice (c), that is, from $\preccurlyeq = \mathbb{N}_{+,\infty} \times \mathbb{N}_{+,\infty} \cup \{0\} \times \mathbb{N}_\infty$, by simply observing that in this case $0 \preccurlyeq \omega(N)$ and $0 \preccurlyeq \omega(C\ A\ B)$ are fulfilled unconditionally.

Corollary 6.3. $((\mathit{foldr}_\tau)_{\tau'}\ C\ N\ (\mathit{build}_\tau\ G)) \sqsupseteq_2 (G_{\tau'}\ C\ N)$

Of the remaining observational relations from Example 2.7, we only give here the immediate consequences of Theorem 6.1 for the $\preccurlyeq$-choices (g) and (i).

Corollary 6.4. *If not* $N \not\downarrow i$ *for any* $i \in \mathbb{N}_+$ *and for every* $A \in \mathit{Term}(\tau)$ *and* $B \in \mathit{Term}(\tau')$, *not* $(C\ A\ B) \not\downarrow i$ *for any* $i \in \mathbb{N}_+$, *then* $((\mathit{foldr}_\tau)_{\tau'}\ C\ N\ (\mathit{build}_\tau\ G)) \sqsupseteq_3 (G_{\tau'}\ C\ N)$.

Corollary 6.5. *If not* $N{\Uparrow}$ *and for every* $A \in \mathit{Term}(\tau)$ *and* $B \in \mathit{Term}(\tau')$, *not* $(C\ A\ B){\Uparrow}$, *then* $((\mathit{foldr}_\tau)_{\tau'}\ C\ N\ (\mathit{build}_\tau\ G)) \sqsupseteq_5 (G_{\tau'}\ C\ N)$.

So what have we gained, in addition to capturing in a single framework previously separate-but-related proofs of equational and inequational statements for observational relations that unify different failure causes? What we have gained is that we can now answer questions like those raised earlier in this section. For example, question (a) from earlier in this section is answered by Corollary 6.2:

- That corollary gives preconditions on *foldr*'s arguments under which *foldr*/*build*-fusion is totally correct even when considering different failure causes as semantically different. Checking those preconditions for the entries in the first and fourth lines in Table 2 would have allowed us to predict, without having to look at the definitions of producer functions, that the first and fourth lines in Table 1 are unproblematic instances of fusion.

Similarly, Corollaries 6.4 and 6.5 provide answers for question (b), which asked for sufficient preconditions for partial correctness of *foldr*/*build*-fusion with respect to semantic approximation orders that differ with respect to their

relative treatment of different failure causes:

- Corollary 6.5 allows us to formulate the desired guarantee, in advance, that for the consumer/producer-combination in the second line in Table 1 no nontermination can possibly be introduced via *foldr*/*build*-fusion. We simply need to check the entries in the second line in Table 2 against the preconditions in Corollary 6.5, and observe from the definition of $\sqsupseteq_5$ that $M \sqsupseteq_5 M'$ with diverging M' is only possible if M is already diverging as well. Of course, we have seen in Table 1 that for the same consumer/producer-combination an introduction of finite failure is very well possible. But now we also know that we could not have expected otherwise, given that the consuming *foldr*'s arguments do not satisfy the preconditions of Corollary 6.4.
- Corollary 6.4 lets us establish that *foldr*/*build*-fusion cannot possibly introduce or confuse finite failures when applied to *headOr loop* (*lastEvenOrEmpty l*) for any $l :: [Int]$ and nonterminating computation *loop*.
- The same obviously is not true for the consumer/producer-combination in the third line in Table 1. Again, this is easily explained by checking the consuming *foldr*'s arguments against the preconditions of Corollary 6.4. Moreover, it turns out that for that combination it is not even possible to guarantee nonintroduction of divergence. Starting from the observation that the first argument in the third line in Table 2 does not fulfill the precondition it should for successfully applying Corollary 6.5, it is not hard to find an input list showing this.

The above kinds of analyses are exactly what we sought to enable with our study of the interaction between selective strictness, different failure causes, and *foldr*/*build*-fusion or, more generally, parametricity as such.

7 Related Work

This paper further advances a line of research whose ultimate goal is the development of appropriate tools for reasoning about parametricity properties of real programming languages rather than toy calculi. It builds on [14–17,25], both in terms of technical approach and insights obtained. In this section we describe in some detail the relationship of the present paper to these papers. We also discuss other related work.

The language Core introduced in this paper is an extension of Pitts' PolyPCF [17] with a selective strictness construct and an explicit error primitive that can be used at all types. But whereas our most closely related paper [16] uses a **seq**-primitive to model selective strictness (and does not include any error primitives), this paper instead uses a nonrecursive strict-let construct. As mentioned in Section 2.2, the strict-let formulation avoids duplicating expressions

during the translation of calls to *seq* in Haskell. A more essential difference from [16] (and also [17]) is that the operational semantics given in Section 2.3 of this paper has no value of the form $\Lambda\alpha.M$ and no redex/reduct-pair for type instantiation. This is, of course, because our semantics is type-erasing, so that neither type generalization nor specialization carries computational content. This is just as in Haskell. It also accords with the situation in the more informal (and finite-failure-unaware) setting of [14,15]. Since our semantics thus allows no externally observable difference between a polymorphic term and its instantiation at any type, the $\preccurlyeq$-compliance requirement in Definition 4.3 has no counterpart in Definition 4.4. The proof of Lemma 4.7 captures the essence of why none is needed. That no explicit enforcement of $\preccurlyeq$-compliance on the result of the relational action is necessary for Definition 4.4 is analogous to the situation in [14,15], but contrasts with that in [16]. Similarly, the extensionality principle for $\forall$-types formulated after Lemma 5.8 does not refer to $\preccurlyeq$, again in contrast with [16], but reflecting what we had intuitively in the setting of [14,15]. The mismatch between what one has for $\forall$-types in Haskell-like languages and what we had for $\forall$-types in PolySeq in [16] is thus fully accounted for by the type-erasing semantics of Core.

A key ingredient in our technical development is the function $\top(\cdot,\cdot)$ expressing whether (the type-erasure of) a particular term in (the type-erasure of the stack representation of) a particular context terminates in a value, terminates in an error, or diverges. This function plays an important role in restricting attention to relations that play well with parametricity in the presence of fixpoint recursion. The key concept in this context is $\preccurlyeq\preccurlyeq$-closure, a generalization of Pitts' well-known notion of $\top\top$-closure which relates the outcomes of $\top$ for different stack-term pairs by $\preccurlyeq$, rather than by bidirectional implication as in [17] or unidirectional implication as in [16].

The source of fully appropriate relations for quantification in the relational action for $\forall$-types — i.e., of $\preccurlyeq$-compliant and $\preccurlyeq\preccurlyeq$-closed relations — that we establish in Section 4.4 is inspired by [17]. There, such a source (with respect to the relevant restriction) was identified by considering graphs of typed stacks up to observational equivalence. In the inequational setting of [16] two dual (to each other) graph notions were derived from this concept. Here we similarly use two graph notions, but we define them directly in terms of the logical relation. The $\preccurlyeq$-upwards and $\preccurlyeq$-downwards properties used to establish $\preccurlyeq$-compliance of left- and right-graphs, respectively, in Lemma 4.16 are abstract versions of the totality restriction on stacks from [16].

Many of the results reported in this paper are more abstract versions of corresponding ones in [16], and the same is true of their proofs. Lemma 4.2, for example, generalizes Lemma 4.10 in [16], and its proof is structurally identical to the corresponding proof given there. The differences between the proofs lie entirely in the nature of $\top$ as a function here versus as a relation in [16], in the

use of $\preccurlyeq$ here versus the use of unidirectional implication in [16], and in the use of equality (on $\mathbb{N}_\infty$) here versus bidirectional implication in [16]. Statements and proofs of some other lemmas in the present paper are similarly analogous to proofs in [16] or in [17].

Our small-step and type-erasing semantics approach makes it especially easy to modularize the constructions necessary to accommodate additional language features. As a proof-of-concept, we have worked out the details of adding general existential types [29] into our framework without breaking parametricity or compromising our results in Section 6. Other logical relations for (purely strict) polymorphic calculi with existential types built-in are given in, for example, [22] and [25]. The main difference between our (analyses and thus our) results and theirs is that ours are parameterized over $\preccurlyeq$. For the extension of Core with existential types, as for Core itself, parameterization over $\preccurlyeq$ gives generally applicable results that can be instantiated for particular observational relations of interest. We ultimately obtain an analogue of Theorem 5.7 for the extension of Core by existential types. Moreover, because our approach is parameterized over $\preccurlyeq$, we can derive an abstract extensionality principle for existential types that is in the spirit of the one from [25], but can be instantiated for different notions of program equivalence or approximation.

More practical related work can be found in [27] and [30]. There, a semantic setup is presented that deals with the pragmatic issues of allowing, distinguishing, and handling different kinds of failures in a compiler like GHC. Those papers' consideration of not only raising finite failures (in pure code), but also catching them (in monadic code), is well beyond the scope of our present work. Nevertheless, their treatment of the relative definedness of different failure causes allows for an interesting comparison with ours here. The driving force in [27] and [30] is the desire to build a semantics that justifies as many low-level compiler optimizations as possible, even when those optimizations potentially change the evaluation order of programs. For example, these papers want to consider a transformation from

$$\mathbf{let!}\ x = M\ \mathbf{in}\ (\mathbf{let!}\ y = N\ \mathbf{in}\ P)$$

to

$$\mathbf{let!}\ y = N\ \mathbf{in}\ (\mathbf{let!}\ x = M\ \mathbf{in}\ P)$$

as semantics-preserving, even though M and N may lead to different errors. The way to go then is to assign a *set* of failure causes to every nonconverging expression. For example, in contrast to what one might expect, the result of either of the above expressions in the nonconverging case is seen as the set of all errors that any of M, N, and P can lead to. This requires evaluation of P in a certain "exception-finding mode", in order to detect errors occurring in it independently of the (already failing) terms that get substituted for its potentially free variables x and y. This rather complicates the semantics,

be they operational as in [27] or denotational as in [30], and indeed makes it unclear how to integrate with the technical machinery we set up here to deal with parametricity. A less technical, but very important motivational difference is that [27] and [30] willingly trade precision of analysis for efficiency of compiled code. In particular, "fictitious" errors may surface in their setting: wherever a diverging term appears, one is forced to treat it, semantically, as if it could also lead to any kind of finite failure. Put differently, any diverging term is necessarily "below" any (finitely or infinitely) failing one of same type with respect to the refinement and approximation orders of [27] and [30]. This is an early commitment that we are not quite willing to make for our $\preccurlyeq$-induced observational relations, as we think it would ultimately prevent us from getting important results such as some of those reported in Section 6. There, we have, for example, given a partial correctness result for *foldr*/*build*-fusion in which we need not require convergence of *foldr*'s arguments and yet can show that fusion at least will not introduce divergence, even though it might introduce finite failure. Key to that result (in Corollary 6.5) was the availability of an appropriate choice for $\preccurlyeq$ that implies a semantic approximation order (or inverse thereof, namely $\sqsupseteq_5$) that would be outlawed by making the above commitment.

8 Conclusion

In this paper we have shown how to bring reasoning via logical relations closer to bear on real languages by deriving results that are pertinent to an intermediate language like GHC Core for the (mostly) lazy functional language Haskell. Specifically, we have constructed a family of logical relations for a polymorphic lambda calculus which models several aspects of intermediate languages for Haskell. This family is parameterized over a relative definedness preorder on different failure causes, and each of its members is shown to exactly characterize the notion of relating observational behavior of programs, with respect to a type-erasing operational semantics, that is induced by its preorder parameter. As a consequence, relational parametricity becomes available as reasoning principle in a much more realistic setting than before, and is endowed with a high reuse potential. We have capitalized on this potential to prove an abstract correctness result for short cut fusion which can be specialized to various concrete setups (both ones already considered in the literature and new ones).

What remains to be discussed is the role of the restrictions (9) and (10) on the preorder parameter, which were needed to accommodate selective strictness. They outlaw choices (d) and (e) from Example 2.7, both of which seemed to provide quite interesting ways of arranging the relative definedness of different failure causes. So what, if anything, is wrong with an observational approxi-

mation relation distinguishing at least two different notions of program failure that are both considered to approximate every term that evaluates to a proper value? The answer can actually be given independently of any concern for relational parametricity: such an observational approximation relation could no longer be compatible. To see this, assume that the two relevant kinds of program failure are nontermination, $A = \mathbf{fix}(\lambda x :: (\forall\alpha.\alpha)\text{-}\mathit{list}.x)$, and an explicit runtime error, $B = \mathbf{error}_{(\forall\alpha.\alpha)\text{-}\mathit{list}}(1)$. By assumption, we would have $A \sqsubseteq \mathbf{nil}_{\forall\alpha.\alpha}$ and $B \sqsubseteq \mathbf{nil}_{\forall\alpha.\alpha}$. But then by compatibility (or, equivalently, monotonicity such as one would expect from every notion of semantic approximation) we should also have that $\mathbf{let!}\ x = A\ \mathbf{in}\ B \sqsubseteq \mathbf{let!}\ x = \mathbf{nil}_{\forall\alpha.\alpha}\ \mathbf{in}\ B$ and $\mathbf{let!}\ x = B\ \mathbf{in}\ A \sqsubseteq \mathbf{let!}\ x = \mathbf{nil}_{\forall\alpha.\alpha}\ \mathbf{in}\ A$. Since in our semantic setup always $N \sqsubseteq \mathbf{let!}\ x = N\ \mathbf{in}\ M$ for nonconverging N and arbitrary M of same type, this would imply that $A \sqsubseteq B$ and $B \sqsubseteq A$, which would contradict the assumption that $\sqsubseteq$ distinguishes between A and B. So there is actually very good reason to disallow the kind of approximation relations mentioned above. For us, at least, that was an entirely unanticipated impact of selective strictness on the semantics of Haskell-like languages, and we consider bringing it to light an additional merit of our abstract approach. Of course, the phenomenon observed above does not prevent the definition and study of various notions of observational equivalence or approximation that relate different failure causes in other computationally interesting and intuitive ways.

Acknowledgement

We thank the anonymous reviewers for their helpful comments and suggestions for improving the paper.

References

[1] L. Cardelli, Typeful programming, in: E. Neuhold, M. Paul (Eds.), Formal Description of Programming Concepts, IFIP State of the Art Reports Series, Springer-Verlag, 1991, pp. 431–507.

[2] S. Peyton Jones (Ed.), Haskell 98 Language and Libraries: The Revised Report, Cambridge University Press, 2003.

[3] C. Strachey, Fundamental concepts in programming languages, Lecture notes for a course at the International Summer School in Computer Programming, 1967. Reprint appeared in Higher-Order and Symbolic Computation 13 (1–2) (2000) 11–49.

[4] L. Cardelli, P. Wegner, On understanding types, data abstraction, and polymorphism, ACM Computing Surveys 17 (4) (1985) 471–522.

[5] J. Reynolds, Types, abstraction and parametric polymorphism, in: Information Processing, Proceedings, Elsevier, 1983, pp. 513–523.

[6] H. Friedman, Equality between functionals, in: Logic Colloquium '72–73, Proceedings, Springer-Verlag, 1975, pp. 22–37.

[7] G. Plotkin, Lambda-definability and logical relations, memorandum SAI-RM-4, University of Edinburgh (1973).

[8] P. Wadler, Theorems for free!, in: Functional Programming Languages and Computer Architecture, Proceedings, ACM Press, 1989, pp. 347–359.

[9] A. Gill, J. Launchbury, S. Peyton Jones, A short cut to deforestation, in: Functional Programming Languages and Computer Architecture, Proceedings, ACM Press, 1993, pp. 223–232.

[10] O. Chitil, Type inference builds a short cut to deforestation, in: International Conference on Functional Programming, Proceedings, ACM Press, 1999, pp. 249–260.

[11] J. Svenningsson, Shortcut fusion for accumulating paramters & zip-like functions, in: International Conference on Functional Programming, Proceedings, ACM Press, 2002, pp. 124–132.

[12] J. Voigtländer, Concatenate, reverse and map vanish for free, in: International Conference on Functional Programming, Proceedings, ACM Press, 2002, pp. 14–25.

[13] N. Ghani, P. Johann, Monadic augment and generalised short cut fusion, Journal of Functional Programming 17 (6) (2007) 731–776.

[14] P. Johann, J. Voigtländer, Free theorems in the presence of *seq*, in: Principles of Programming Languages, Proceedings, ACM Press, 2004, pp. 99–110.

[15] P. Johann, J. Voigtländer, The impact of *seq* on free theorems-based program transformations, Fundamenta Informaticae 69 (1-2) (2006) 63–102.

[16] J. Voigtländer, P. Johann, Selective strictness and parametricity in structural operational semantics, inequationally, Theoretical Computer Science (2007), DOI:10.1016/j.tcs.2007.09.014.

[17] A. Pitts, Parametric polymorphism and operational equivalence, Mathematical Structures in Computer Science 10 (3) (2000) 321–359.

[18] P. Johann, Short cut fusion is correct, Journal of Functional Programming 13 (4) (2003) 797–814.

[19] R. Møgelberg, Interpreting polymorphic FPC into domain theoretic models of parametric polymorphism, in: International Colloquium on Automata, Languages and Programming, Proceedings, Vol. 4052 of LNCS, Springer-Verlag, 2006, pp. 372–383.

[20] J. Voigtländer, P. Johann, Selective strictness and parametricity in structural operational semantics, Tech. Rep. TUD-FI06-02, Technische Universität Dresden (2006).

[21] K. Crary, R. Harper, Syntactic logical relations for polymorphic and recursive types, in: Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin, Vol. 172 of ENTCS, Elsevier, 2007, pp. 259–299.

[22] A. Ahmed, Step-indexed syntactic logical relations for recursive and quantified types, in: European Symposium on Programming, Proceedings, Vol. 3924 of LNCS, Springer-Verlag, 2006, pp. 69–83.

[23] J.-Y. Girard, Interprétation functionelle et élimination des coupures dans l'arithmétique d'ordre supérieure, Ph.D. thesis, Université Paris VII (1972).

[24] J. Reynolds, Towards a theory of type structure, in: Colloque sur la Programmation, Proceedings, Springer-Verlag, 1974, pp. 408–423.

[25] A. Pitts, Typed operational reasoning, in: B. Pierce (Ed.), Advanced Topics in Types and Programming Languages, MIT Press, 2005, pp. 245–289.

[26] A. Tolmach, An external representation for the GHC core language, Draft (2001).

[27] A. Moran, S. Lassen, S. Peyton Jones, Imprecise exceptions, Co-inductively, in: Higher Order Operational Techniques in Semantics, Proceedings, Vol. 26 of ENTCS, Elsevier, 1999, pp. 122–141.

[28] M. Eekelen, M. Mol, Proof tool support for explicit strictness, in: Implementation and Application of Functional Languages 2005, Selected Papers, Vol. 4015 of LNCS, Springer-Verlag, 2006, pp. 37–54.

[29] J. Mitchell, G. Plotkin, Abstract types have existential type, ACM Transactions on Programming Languages and Systems 10 (3) (1988) 470–502.

[30] S. Peyton Jones, A. Reid, C. Hoare, S. Marlow, F. Henderson, A semantics for imprecise exceptions, in: Programming Language Design and Implementation, Proceedings, ACM Press, 1999, pp. 25–36.

A Proofs Appendix

We start with an auxiliary observation that is a consequence of $\rightarrowtail$ being deterministic.

Observation A.1. *For every triple S_1, S_2, S_3 of evaluation stacks without free variables, $M_1, M_2, M_3 \in$ Untyped, and $t, t' \in \mathbb{N}$, if $(S_1, M_1) \rightarrowtail^t (S_2, M_2)$, $(S_1, M_1) \rightarrowtail^{t'} (S_3, M_3)$, and (S_3, M_3) is an end configuration, then $t \leq t'$ and $(S_2, M_2) \rightarrowtail^{t'-t} (S_3, M_3)$.*

Proof of Lemma 2.3: Fix S. We prove the more general statement that for every $M \in \mathit{Untyped}$ and evaluation stack S', if $(S \mathbin{@} S', M)$ leads to an end configuration, then so does (S', M), by induction on the number t of steps required for the former. The induction base ($t = 0$) is straightforward, using that $(S \mathbin{@} S', M)$ being an end configuration implies either $S' = \mathit{Id}$ and $M \in \mathit{Value}$ or $M = \mathbf{error}(i)$ for some $i \in \mathbb{N}_+$. For the induction step ($t \to t+1$), assume that $(S \mathbin{@} S', M) \rightarrowtail (S'', M')$ and (S'', M') leads to an end configuration in t steps. If $S' = \mathit{Id}$ and $M \in \mathit{Value}$, then the induction claim holds trivially. Otherwise, a straightforward case distinction on the transition $(S \mathbin{@} S', M) \rightarrowtail (S'', M')$ yields that there exists an evaluation stack S''' with $(S', M) \rightarrowtail (S''', M')$ and $S'' = S \mathbin{@} S'''$. Together with the induction hypothesis for $(S'', M') = (S \mathbin{@} S''', M')$ this implies the induction claim. $\square$

Proof of Lemma 3.7: By Observation 3.4(a) we have $\top(S, \mathbf{let!}\ x = A\ \mathbf{in}\ B) = \top(S \circ (\mathbf{let!}\ x = -\ \mathbf{in}\ B), A)$. We proceed by case distinction on $\omega(A)$.

Case a: $\omega(A) = 0$. Then by Definitions 2.1 and 2.6 there exists a $V \in \mathit{Value}$ with $(\mathit{Id}, [\![A]\!]) \rightarrowtail^* (\mathit{Id}, V)$. Together with Observation 2.4, we thus have:

$$\begin{aligned}(([\![S]\!] \circ (\mathbf{let!}\ x = -\ \mathbf{in}\ [\![B]\!]), [\![A]\!]) &\rightarrowtail^* ([\![S]\!] \circ (\mathbf{let!}\ x = -\ \mathbf{in}\ [\![B]\!]), V) \\ &\rightarrowtail\ ([\![S]\!], \mathbf{let!}\ x = V\ \mathbf{in}\ [\![B]\!]) \\ &\rightarrowtail\ ([\![S]\!], [\![B]\!][V/x]).\end{aligned}$$

To establish $\top(S \circ (\mathbf{let!}\ x = -\ \mathbf{in}\ B), A) = \top(S, B[A/x])$, we perform a further case distinction on $\top(S, B[A/x])$. By Definition 3.1, it is:

- 0 if there is some $V' \in \mathit{Value}$ with $([\![S]\!], [\![B]\!][[\![A]\!]/x]) \rightarrowtail^* (\mathit{Id}, V')$,
- i if there is some evaluation stack S' with $([\![S]\!], [\![B]\!][[\![A]\!]/x]) \rightarrowtail^* (S', \mathbf{error}(i))$, and
- ∞ otherwise.

For the first two cases, the desired equality follows from the transition sequence displayed above and from Lemma A.2, to be given and proved below. For the last case, we show it by contradiction. Assume $\top(S \circ (\mathbf{let!}\ x = -\ \mathbf{in}\ B), A) \neq \infty$. Then $([\![S]\!] \circ (\mathbf{let!}\ x = -\ \mathbf{in}\ [\![B]\!]), [\![A]\!])$ must lead to an end configuration. Given that also $([\![S]\!] \circ (\mathbf{let!}\ x = -\ \mathbf{in}\ [\![B]\!]), [\![A]\!]) \rightarrowtail^* ([\![S]\!], [\![B]\!][V/x])$, Observation A.1 implies that so does $([\![S]\!], [\![B]\!][V/x])$. Combined with Lemma A.3, to be given and proved below, this contradicts $\top(S, B[A/x]) = \infty$.

Case b: $\omega(A) = i$ for some $i \in \mathbb{N}_+$. Then there exists an evaluation stack S' with $(\mathit{Id}, [\![A]\!]) \rightarrowtail^* (S', \mathbf{error}(i))$. By Observation 2.4 this implies $([\![S]\!] \circ (\mathbf{let!}\ x = -\ \mathbf{in}\ [\![B]\!]), [\![A]\!]) \rightarrowtail^* (([\![S]\!] \circ (\mathbf{let!}\ x = -\ \mathbf{in}\ [\![B]\!])) \mathbin{@} S', \mathbf{error}(i))$, and thus $\top(S \circ (\mathbf{let!}\ x = -\ \mathbf{in}\ B), A) = i$.

Case c: $\omega(A) = \infty$. We show $\top(S \circ (\mathbf{let!}\ x = -\ \mathbf{in}\ B), A) = \infty$ by contra-

diction. Assume $\top(S \circ (\mathbf{let!}\ x = -\ \mathbf{in}\ B), A) \neq \infty$. Then $([\![S]\!] \circ (\mathbf{let!}\ x = -\ \mathbf{in}\ [\![B]\!]), [\![A]\!])$ must lead to an end configuration. By Lemma 2.3 this contradicts $\omega(A) = \infty$. □

Lemma A.2. *Let x be a term variable and let $A \in$ Untyped and $V \in$ Value be such that*

$$(Id, A) \rightarrowtail^* (Id, V). \tag{13}$$

For every $B \in$ Untyped$(\{x\})$ and evaluation stack S without free variables, if $(S, B[A/x])$ leads to an end configuration, then $(S, B[V/x])$ leads to an end configuration of the same characteristic. [4]

Proof: The proof uses the fact that from (13) follows that for every $M \in$ Untyped$(\{x\})$:

$$M \neq x \wedge M[V/x] \in \mathit{Value} \Rightarrow M[A/x] \in \mathit{Value} \tag{14}$$

and

$$M[A/x] \in \mathit{Value} \Rightarrow M[V/x] \in \mathit{Value}. \tag{15}$$

Actually, we prove a slightly more general statement, namely that for every end configuration $(\overline{S}, \overline{M})$, $B \in$ Untyped$(\{x\})$, and evaluation stack S whose free variables are in $\{x\}$, if $(S[A/x], B[A/x])$ leads to $(\overline{S}, \overline{M})$, then $(S[V/x], B[V/x])$ leads to an end configuration of the same characteristic as $(\overline{S}, \overline{M})$. [5] For every end configuration $(\overline{S}, \overline{M})$ we prove it by (strong) induction on the number t of steps required for $(S[A/x], B[A/x])$ to lead to $(\overline{S}, \overline{M})$. The induction base ($t = 0$) is straightforward, using the facts that $(S[A/x], B[A/x]) = (\overline{S}, \overline{M})$ implies either $S = \overline{S} = Id$ and $B[A/x] = \overline{M} \in$ Value or $B[A/x] = \overline{M} = \mathbf{error}(i)$ for some $i \in \mathbb{N}_+$, that $B[A/x] \in$ Value implies $B[V/x] \in$ Value by (15), and that for every $i \in \mathbb{N}_+$, $B[A/x] = \mathbf{error}(i)$ implies $B = \mathbf{error}(i)$, given that $A \neq \mathbf{error}(i)$ due to (13) and the definitions of $\rightarrowtail$ and values. For the induction step ($t \rightarrow t + 1$), we first note that we can assume $A \neq V$, because for $A = V$ the induction claim follows trivially from $(S[A/x], B[A/x]) \rightarrowtail^{t+1} (\overline{S}, \overline{M})$. Then, if $B = x$, it follows from $(S[A/x], A) \rightarrowtail^{t+1} (\overline{S}, \overline{M})$, (13), Observation 2.4, $A \neq V$, and Observation A.1 that $(S[A/x], V) \rightarrowtail^{t'} (\overline{S}, \overline{M})$ for some $t' \leq t$, which together with the induction hypothesis for t' implies the induction claim. If $B \neq x$, then we proceed by case distinction on the first transition in $(S[A/x], B[A/x]) \rightarrowtail (S', B') \rightarrowtail^t (\overline{S}, \overline{M})$ as follows.

[4] The notion of the characteristics of end configurations is defined as follows. An end configuration of the form (Id, V') with $V' \in$ Value is called *of characteristic* 0, one of the form $(S, \mathbf{error}(i))$ for some $i \in \mathbb{N}_+$ is called *of characteristic* i.

[5] The notion of substitution in evaluation frames and evaluation stacks is defined as follows. The result of substituting an untyped term M for all free occurrences of x in an evaluation frame E is denoted by $E[M/x]$. Substitution in an evaluation stack S, denoted by $S[M/x]$, is by corresponding substitution in all evaluation frames constituting S.

Case a: $B[A/x] = E\{B'\}$ and $S' = S[A/x] \circ E$, where E is an evaluation frame and $B' \notin \mathit{Value}$. Then by $B \neq x$ and the definition of evaluation frames, clearly $B = E'\{M\}$ for some evaluation frame E' and $M \in \mathit{Untyped}(\{x\})$ with $E = E'[A/x]$ and $B' = M[A/x]$. If $M = x$, it follows from $(S[A/x] \circ (E'[A/x]), A) \rightarrowtail^t (\overline{S}, \overline{M})$, (13), Observation 2.4, $(S[A/x] \circ (E'[A/x]), V) \rightarrowtail (S[A/x], (E'[A/x])\{V\})$, and Observation A.1 that $(S[A/x], (E'[A/x])\{V\}) \rightarrowtail^{t'} (\overline{S}, \overline{M})$ for some $t' < t$, which together with the induction hypothesis for t' implies the induction claim. If $M \neq x$, then since $B' = M[A/x] \notin \mathit{Value}$, by (14) also $M[V/x] \notin \mathit{Value}$ holds. Thus, we then have that $(S[V/x], B[V/x]) \rightarrowtail (S[V/x] \circ (E'[V/x]), M[V/x])$, which together with $(S', B') \rightarrowtail^t (\overline{S}, \overline{M})$ and the induction hypothesis for t implies the induction claim.

Case b: $S[A/x] = S' \circ E$ and $B' = E\{B[A/x]\}$, where E is an evaluation frame and $B[A/x] \in \mathit{Value}$. Then clearly $S = S'' \circ E'$ for some evaluation stack S'' and evaluation frame E' with $S' = S''[A/x]$ and $E = E'[A/x]$. Since $B[A/x] \in \mathit{Value}$, by (15) also $B[V/x] \in \mathit{Value}$ holds. Thus, we have that $(S[V/x], B[V/x]) \rightarrowtail (S''[V/x], (E'[V/x])\{B[V/x]\})$, which together with $(S', B') \rightarrowtail^t (\overline{S}, \overline{M})$ and the induction hypothesis for t implies the induction claim.

Case c: $S' = S[A/x]$ and $B[A/x] \rightsquigarrow B'$. Note that $A \notin \mathit{Value}$, because for $A \in \mathit{Value}$ we get $A = V$ by (13) and the definition of $\rightarrowtail$, in contradiction to our assumption $A \neq V$. From $B[A/x] \rightsquigarrow B'$, $B \neq x$, and $A \notin \mathit{Value}$ follows, by the definitions of $\rightsquigarrow$ and values, the existence of an $M \in \mathit{Untyped}(\{x\})$ such that $B' = M[A/x]$ and $B[V/x] \rightsquigarrow M[V/x]$. Then we have that $(S[V/x], B[V/x]) \rightarrowtail (S[V/x], M[V/x])$, which together with $(S', B') \rightarrowtail^t (\overline{S}, \overline{M})$ and the induction hypothesis for t implies the induction claim. This completes the case distinction, and thus the proof. □

Lemma A.3. *Let x, A, V, B, and S be as in Lemma A.2. If $(S, B[V/x])$ leads to an end configuration, then so does $(S, B[A/x])$.*

Proof: We prove the slightly more general statement that for every $B \in \mathit{Untyped}(\{x\})$ and evaluation stack S whose free variables are in $\{x\}$, if $(S[V/x], B[V/x])$ leads to an end configuration, then so does $(S[A/x], B[A/x])$.[6] We prove it by induction on the number t of steps required for $(S[V/x], B[V/x])$ to lead to an end configuration. For the induction base ($t = 0$), we use that $(S[V/x], B[V/x])$ being an end configuration implies either $S = \mathit{Id}$ and $B[V/x] \in \mathit{Value}$ or $B[V/x] = \mathbf{error}(i)$ for some $i \in \mathbb{N}_+$. In the former case, the claim follows from (13) if $B = x$, and from the fact that $B[V/x] \in \mathit{Value}$ implies $B[A/x] \in \mathit{Value}$ by (14) from the proof of Lemma A.2 if $B \neq x$. In the case $B[V/x] = \mathbf{error}(i)$ for some $i \in \mathbb{N}_+$, the claim follows from the fact that for every $i \in \mathbb{N}_+$, $B[V/x] = \mathbf{error}(i)$ implies $B = \mathbf{error}(i)$ by

[6] We use the notion of substitution in evaluation frames and evaluation stacks as in Footnote 5.

the definition of values. For the induction step ($t \to t+1$), assume that $(S[V/x], B[V/x]) \rightarrowtail (S', B')$ and (S', B') leads to an end configuration in t steps. If $A \in \mathit{Value}$, then $A = V$ by (13) and the definition of $\rightarrowtail$, in which case the induction claim follows trivially. Otherwise, a case distinction on the transition $(S[V/x], B[V/x]) \rightarrowtail (S', B')$ as detailed below yields that there exist $M \in \mathit{Untyped}(\{x\})$ and an evaluation stack S'' whose free variables are in $\{x\}$ with $(S[A/x], B[A/x]) \rightarrowtail^* (S''[A/x], M[A/x])$, $S' = S''[V/x]$, and $B' = M[V/x]$. Together with the induction hypothesis for $(S', B') = (S''[V/x], M[V/x])$ this implies the induction claim.

Case a: $B[V/x] = E\{B'\}$ and $S' = S[V/x] \circ E$, where E is an evaluation frame and $B' \notin \mathit{Value}$. Then by the definitions of values and evaluation frames, clearly $B = E'\{M\}$ for some evaluation frame E' and $M \in \mathit{Untyped}(\{x\})$ with $E = E'[V/x]$ and $B' = M[V/x]$. Consequently, $B[A/x] = (E'[A/x])\{M[A/x]\}$. Since $B' = M[V/x] \notin \mathit{Value}$, by (15) from the proof of Lemma A.2 also $M[A/x] \notin \mathit{Value}$ holds. Thus, we have that $(S[A/x], B[A/x]) \rightarrowtail (S[A/x] \circ (E'[A/x]), M[A/x])$, which satisfies the requirements with $S'' = S \circ E'$.

Case b: $S[V/x] = S' \circ E$ and $B' = E\{B[V/x]\}$, where E is an evaluation frame and $B[V/x] \in \mathit{Value}$. Then clearly $S = S'' \circ E'$ for some evaluation stack S'' and evaluation frame E' with $S' = S''[V/x]$ and $E = E'[V/x]$. If $B = x$, then by (13) and Observation 2.4 we have $(S[A/x], B[A/x]) \rightarrowtail^* (S''[A/x] \circ (E'[A/x]), V) \rightarrowtail (S''[A/x], (E'[A/x])\{V\})$, which satisfies the requirements with $M = E'\{V\}$. If $B \neq x$, then by (14) from the proof of Lemma A.2, $B[V/x] \in \mathit{Value}$ implies $B[A/x] \in \mathit{Value}$, so that $(S[A/x], B[A/x]) \rightarrowtail (S''[A/x], (E'[A/x])\{B[A/x]\})$, which satisfies the requirements with $M = E'\{B\}$.

Case c: $S' = S[V/x]$ and $B[V/x] \rightsquigarrow B'$. Then by the definitions of values and $\rightsquigarrow$, there are two cases to consider. If $B' = M[V/x]$ for some $M \in \mathit{Untyped}(\{x\})$ with $B[A/x] \rightsquigarrow M[A/x]$, then $(S[A/x], B[A/x]) \rightarrowtail (S[A/x], M[A/x])$, which satisfies the requirements with $S'' = S$. If $B = E\{x\}$ for some evaluation frame E, then from $B[V/x] \rightsquigarrow B'$ follows, by the definitions of evaluation frames and $\rightsquigarrow$, the existence of an $M \in \mathit{Untyped}(\{x\})$ such that $B' = M[V/x]$ and $(E[A/x])\{V\} \rightsquigarrow M[A/x]$. Since $A \notin \mathit{Value}$, we have that $(S[A/x], B[A/x]) \rightarrowtail (S[A/x] \circ (E[A/x]), A) \rightarrowtail^* (S[A/x] \circ (E[A/x]), V) \rightarrowtail (S[A/x], (E[A/x])\{V\}) \rightarrowtail (S[A/x], M[A/x])$, where the "$\rightarrowtail^*$"-part follows from (13) by Observation 2.4. This satisfies the requirements with $S'' = S$. It also completes the case distinction, and thus the proof. □

Proof of Lemma 3.8: We need the notions of the characteristics of end configurations and of substitution in evaluation frames and evaluation stacks, as defined in Footnotes 4 and 5. Further, we use the notation $(F^n\ A)$ also for untyped terms. Recall $\Omega = \mathbf{fix}(\lambda x.x) \in \mathit{Untyped}$ from Observation 2.2.

Now, let $F \in \mathit{Untyped}$. We will use the fact that for every $n \in \mathbb{N}$ and $N \in$

Untyped($\{x\}$):

$$N[(F^n\ \Omega)/x] \in \mathit{Value} \Leftrightarrow N[\mathbf{fix}(F)/x] \in \mathit{Value}. \tag{16}$$

First, we prove for every end configuration $(\overline{S}, \overline{M})$ the following statement by induction on $n \in \mathbb{N}$:

(I) For every $t \in \mathbb{N}$, $M \in \mathit{Untyped}(\{x\})$, and evaluation stack S whose free variables are in $\{x\}$, if $(S[(F^n\ \Omega)/x], M[(F^n\ \Omega)/x]) \rightarrowtail^t (\overline{S}, \overline{M})$, then $(S[(F^{n+1}\ \Omega)/x], M[(F^{n+1}\ \Omega)/x])$ leads to an end configuration of the same characteristic as $(\overline{S}, \overline{M})$ in t steps.

The induction step ($n \to n+1$) follows from $S[(F^{n+1}\ \Omega)/x] = S'[(F^n\ \Omega)/x]$, $M[(F^{n+1}\ \Omega)/x] = M'[(F^n\ \Omega)/x]$, $S[(F^{n+2}\ \Omega)/x] = S'[(F^{n+1}\ \Omega)/x]$, and $M[(F^{n+2}\ \Omega)/x] = M'[(F^{n+1}\ \Omega)/x]$, where $S' = S[(F\ x)/x]$ and $M' = M[(F\ x)/x]$. For the induction base ($n = 0$), we prove by induction on $t \in \mathbb{N}$ that for every $M \in \mathit{Untyped}(\{x\})$ and evaluation stack S whose free variables are in $\{x\}$, if $(S[\Omega/x], M[\Omega/x]) \rightarrowtail^t (\overline{S}, \overline{M})$, then $(S[(F\ \Omega)/x], M[(F\ \Omega)/x])$ leads to an end configuration of the same characteristic as $(\overline{S}, \overline{M})$ in t steps. The induction base ($t = 0$) is straightforward, using the facts that $(S[\Omega/x], M[\Omega/x]) = (\overline{S}, \overline{M})$ implies either $S = \overline{S} = \mathit{Id}$ and $M[\Omega/x] = \overline{M} \in \mathit{Value}$ or $M[\Omega/x] = \overline{M} = \mathbf{error}(i)$ for some $i \in \mathbb{N}_+$, that $M[\Omega/x] \in \mathit{Value}$ implies $M[(F\ \Omega)/x] \in \mathit{Value}$ by (16), and that $M[\Omega/x] = \mathbf{error}(i)$ implies $M = \mathbf{error}(i)$. For the induction step ($t \to t+1$), assume that $(S[\Omega/x], M[\Omega/x]) \rightarrowtail (S', M') \rightarrowtail^t (\overline{S}, \overline{M})$. Note that $M \neq x$, because otherwise $(S[\Omega/x], \Omega) \rightarrowtail^{t+1} (\overline{S}, \overline{M})$, which would be in contradiction to Observation 2.2. A case distinction on the transition $(S[\Omega/x], M[\Omega/x]) \rightarrowtail (S', M')$ as detailed below then yields that there exist $N \in \mathit{Untyped}(\{x\})$ and an evaluation stack S'' whose free variables are in $\{x\}$ with $(S[(F\ \Omega)/x], M[(F\ \Omega)/x]) \rightarrowtail (S''[(F\ \Omega)/x], N[(F\ \Omega)/x])$, $S' = S''[\Omega/x]$, and $M' = N[\Omega/x]$. Together with $(S', M') \rightarrowtail^t (\overline{S}, \overline{M})$ and the induction hypothesis for t, this implies the induction claim.

Case a: $M[\Omega/x] = E\{M'\}$ and $S' = S[\Omega/x] \circ E$, where E is an evaluation frame and $M' \notin \mathit{Value}$. Then by the definitions of Ω and evaluation frames, clearly $M = E'\{N\}$ for some evaluation frame E' and $N \in \mathit{Untyped}(\{x\})$ with $E = E'[\Omega/x]$ and $M' = N[\Omega/x]$. Thus, $M[(F\ \Omega)/x] = (E'[(F\ \Omega)/x])\{N[(F\ \Omega)/x]\}$. Since $M' = N[\Omega/x] \notin \mathit{Value}$, by (16) we have that $N[(F\ \Omega)/x] \notin \mathit{Value}$ also holds. Thus, $(S[(F\ \Omega)/x], M[(F\ \Omega)/x]) \rightarrowtail (S[(F\ \Omega)/x] \circ (E'[(F\ \Omega)/x]), N[(F\ \Omega)/x])$, which satisfies the requirements with $S'' = S \circ E'$.

Case b: $S[\Omega/x] = S' \circ E$ and $M' = E\{M[\Omega/x]\}$, where E is an evaluation frame and $M[\Omega/x] \in \mathit{Value}$. Then clearly $S = S'' \circ E'$ for some evaluation stack S'' and evaluation frame E' with $S' = S''[\Omega/x]$ and $E = E'[\Omega/x]$. Since $M[\Omega/x] \in \mathit{Value}$, by (16) also $M[(F\ \Omega)/x] \in \mathit{Value}$ holds. Thus, we have

$(S[(F\ \Omega)/x], M[(F\ \Omega)/x]) \rightarrowtail (S''[(F\ \Omega)/x], (E'[(F\ \Omega)/x])\{M[(F\ \Omega)/x]\})$, which satisfies the requirements with $N = E'\{M\}$.

Case c: $S' = S[\Omega/x]$ and $M[\Omega/x] \rightsquigarrow M'$. From $M \neq x$ and $M[\Omega/x] \rightsquigarrow M'$ follows, by the definitions of Ω and $\rightsquigarrow$, the existence of an $N \in \mathit{Untyped}(\{x\})$ such that $M' = N[\Omega/x]$ and $M[(F\ \Omega)/x] \rightsquigarrow N[(F\ \Omega)/x]$, and consequently $(S[(F\ \Omega)/x], M[(F\ \Omega)/x]) \rightarrowtail (S[(F\ \Omega)/x], N[(F\ \Omega)/x])$, which satisfies the requirements with $S'' = S$. This completes the case distinction.

Now, using statement (I), we prove for every end configuration $(\overline{S}, \overline{M})$ the following statement by induction on $t \in \mathbb{N}$:

(II) For every $M \in \mathit{Untyped}(\{x\})$ and evaluation stack S whose free variables are in $\{x\}$, if $(S[\mathbf{fix}(F)/x], M[\mathbf{fix}(F)/x]) \rightarrowtail^t (\overline{S}, \overline{M})$, then there exists an $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, $(S[(F^n\ \Omega)/x], M[(F^n\ \Omega)/x])$ leads to an end configuration of the same characteristic as $(\overline{S}, \overline{M})$.

The induction base $(t = 0)$ is straightforward, using the facts that $(S[\mathbf{fix}(F)/x], M[\mathbf{fix}(F)/x]) = (\overline{S}, \overline{M})$ implies either $S = \overline{S} = \mathit{Id}$ and $M[\mathbf{fix}(F)/x] = \overline{M} \in \mathit{Value}$ or $M[\mathbf{fix}(F)/x] = \overline{M} = \mathbf{error}(i)$ for some $i \in \mathbb{N}_+$, that $M[\mathbf{fix}(F)/x] \in \mathit{Value}$ implies $M[(F^n\ \Omega)/x] \in \mathit{Value}$ for every $n \in \mathbb{N}$ by (16), and that for every $i \in \mathbb{N}_+$, $M[\mathbf{fix}(F)/x] = \mathbf{error}(i)$ implies $M = \mathbf{error}(i)$. For the induction step $(t \to t+1)$, we proceed by case distinction on the first transition in $(S[\mathbf{fix}(F)/x], M[\mathbf{fix}(F)/x]) \rightarrowtail (S', M') \rightarrowtail^t (\overline{S}, \overline{M})$ as follows.

Case a: $M[\mathbf{fix}(F)/x] = E\{M'\}$ and $S' = S[\mathbf{fix}(F)/x] \circ E$, where E is an evaluation frame and $M' \notin \mathit{Value}$. Then by the definition of evaluation frames, clearly $M = E'\{N\}$ for some evaluation frame E' and $N \in \mathit{Untyped}(\{x\})$ with $E = E'[\mathbf{fix}(F)/x]$ and $M' = N[\mathbf{fix}(F)/x]$. By $(S', M') \rightarrowtail^t (\overline{S}, \overline{M})$ and the induction hypothesis for t, we then know that there exists an $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, $(S[(F^n\ \Omega)/x] \circ (E'[(F^n\ \Omega)/x]), N[(F^n\ \Omega)/x])$ leads to an end configuration of the same characteristic as $(\overline{S}, \overline{M})$. To establish the induction claim, it thus suffices to show that for every $n \in \mathbb{N}$, $(S[(F^n\ \Omega)/x], M[(F^n\ \Omega)/x]) \rightarrowtail (S[(F^n\ \Omega)/x] \circ (E'[(F^n\ \Omega)/x]), N[(F^n\ \Omega)/x])$. But this follows from $M[(F^n\ \Omega)/x] = (E'[(F^n\ \Omega)/x])\{N[(F^n\ \Omega)/x]\}$ and $N[(F^n\ \Omega)/x] \notin \mathit{Value}$, where the latter is established by (16) from $M' = N[\mathbf{fix}(F)/x] \notin \mathit{Value}$.

Case b: $S[\mathbf{fix}(F)/x] = S' \circ E$ and $M' = E\{M[\mathbf{fix}(F)/x]\}$, where E is an evaluation frame and $M[\mathbf{fix}(F)/x] \in \mathit{Value}$. Then clearly $S = S'' \circ E'$ for some evaluation stack S'' and evaluation frame E' with $S' = S''[\mathbf{fix}(F)/x]$ and $E = E'[\mathbf{fix}(F)/x]$. By $(S', M') \rightarrowtail^t (\overline{S}, \overline{M})$ and the induction hypothesis for t, we then know that there exists an $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, $(S''[(F^n\ \Omega)/x], (E'[(F^n\ \Omega)/x])\{M[(F^n\ \Omega)/x]\})$ leads to an end configuration of the same characteristic as $(\overline{S}, \overline{M})$. To establish the induction claim, it thus suffices to show that for every $n \in \mathbb{N}$, $(S[(F^n\ \Omega)/x], M[(F^n\ \Omega)/x]) \rightarrowtail$

$(S''[(F^n\ \Omega)/x], (E'[(F^n\ \Omega)/x])\{M[(F^n\ \Omega)/x]\})$. But this follows from the facts that $S[(F^n\ \Omega)/x] = S''[(F^n\ \Omega)/x] \circ (E'[(F^n\ \Omega)/x])$ and $M[(F^n\ \Omega)/x] \in \mathit{Value}$, where the latter is established by (16) from $M[\mathbf{fix}(F)/x] \in \mathit{Value}$.

Case c: $S' = S[\mathbf{fix}(F)/x]$ and $M[\mathbf{fix}(F)/x] \rightsquigarrow M'$. If $M = x$, then we have $M' = F\ \mathbf{fix}(F)$ by the definition of $\rightsquigarrow$. By $(S', M') \rightarrowtail^t (\overline{S}, \overline{M})$ and the induction hypothesis for t, we then know that there exists an $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, $(S[(F^n\ \Omega)/x], F^{n+1}\ \Omega)$ leads to an end configuration of the same characteristic as $(\overline{S}, \overline{M})$, from which the induction claim follows by statement (I). If $M \neq x$, then from $M[\mathbf{fix}(F)/x] \rightsquigarrow M'$ follows, by the definition of $\rightsquigarrow$, the existence of an $N \in \mathit{Untyped}(\{x\})$ such that $M' = N[\mathbf{fix}(F)/x]$ and $M[(F^n\ \Omega)/x] \rightsquigarrow N[(F^n\ \Omega)/x]$ for every $n \in \mathbb{N}$. By $(S', M') \rightarrowtail^t (\overline{S}, \overline{M})$ and the induction hypothesis for t, we also know that given any such N, there exists an $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, $(S[(F^n\ \Omega)/x], N[(F^n\ \Omega)/x])$ leads to an end configuration of the same characteristic as $(\overline{S}, \overline{M})$. To establish the induction claim, it then suffices to show that for every $n \in \mathbb{N}$, $(S[(F^n\ \Omega)/x], M[(F^n\ \Omega)/x]) \rightarrowtail (S[(F^n\ \Omega)/x], N[(F^n\ \Omega)/x])$. But this follows from $M[(F^n\ \Omega)/x] \rightsquigarrow N[(F^n\ \Omega)/x]$. It also completes the case distinction.

Again using statement (I), we now prove for every $n \in \mathbb{N}$ the following statement:

(III) For every $M \in \mathit{Untyped}(\{x\})$ and evaluation stack S whose free variables are in $\{x\}$, if $(S[(F^n\ \Omega)/x], M[(F^n\ \Omega)/x])$ leads to an end configuration, then so does $(S[\mathbf{fix}(F)/x], M[\mathbf{fix}(F)/x])$.

The proof is by induction on the number t of steps required for the former. The induction base ($t = 0$) is straightforward, using the facts that $(S[(F^n\ \Omega)/x], M[(F^n\ \Omega)/x])$ being an end configuration implies either $S = \mathit{Id}$ and $M[(F^n\ \Omega)/x] \in \mathit{Value}$ or $M[(F^n\ \Omega)/x] = \mathbf{error}(i)$ for some $i \in \mathbb{N}_+$, that $M[(F^n\ \Omega)/x] \in \mathit{Value}$ implies $M[\mathbf{fix}(F)/x] \in \mathit{Value}$ by (16), and that for every $i \in \mathbb{N}_+$, $M[(F^n\ \Omega)/x] = \mathbf{error}(i)$ implies $M = \mathbf{error}(i)$. For the induction step ($t \rightarrow t + 1$), assume that $(S[(F^n\ \Omega)/x], M[(F^n\ \Omega)/x]) \rightarrowtail (S', M')$ and (S', M') leads to an end configuration in t steps. We proceed by case distinction on the former transition as follows.

Case a: $M[(F^n\ \Omega)/x] = E\{M'\}$ and $S' = S[(F^n\ \Omega)/x] \circ E$, where E is an evaluation frame and $M' \notin \mathit{Value}$. Then by the definitions of Ω and evaluation frames, there are two cases to consider. If $M = E'\{N\}$ for some evaluation frame E' and $N \in \mathit{Untyped}(\{x\})$ with $E = E'[(F^n\ \Omega)/x]$ and $M' = N[(F^n\ \Omega)/x]$, then $M[\mathbf{fix}(F)/x] = (E'[\mathbf{fix}(F)/x])\{N[\mathbf{fix}(F)/x]\}$. Since by (16), $M' = N[(F^n\ \Omega)/x] \notin \mathit{Value}$ implies $N[\mathbf{fix}(F)/x] \notin \mathit{Value}$, we then have $(S[\mathbf{fix}(F)/x], M[\mathbf{fix}(F)/x]) \rightarrowtail (S[\mathbf{fix}(F)/x] \circ (E'[\mathbf{fix}(F)/x]), N[\mathbf{fix}(F)/x])$, which together with the induction hypothesis for $(S', M') = ((S \circ E')[(F^n\ \Omega)/x], N[(F^n\ \Omega)/x])$ implies the induction claim. If $M = x$, $n > 0$, $E = (-\ (F^{n-1}\ \Omega))$,

and $M' = F$, then since $(S', M') = ((S[(F^n\ \Omega)/x] \circ (-\ x))[(F^{n-1}\ \Omega)/x], F[(F^{n-1}\ \Omega)/x])$ leads to an end configuration in t steps and statement (I) holds we know that $((S \circ (-\ x))[(F^n\ \Omega)/x], F[(F^n\ \Omega)/x])$ leads to an end configuration in t steps. The induction claim then follows by the induction hypothesis for this from $(S[\mathbf{fix}(F)/x], \mathbf{fix}(F)) \rightarrowtail (S[\mathbf{fix}(F)/x], F\ \mathbf{fix}(F)) \rightarrowtail (S[\mathbf{fix}(F)/x] \circ (-\ \mathbf{fix}(F)), F)$, where the second transition is valid due to $M' = F \notin \mathit{Value}$.

Case b: $S[(F^n\ \Omega)/x] = S' \circ E$ and $M' = E\{M[(F^n\ \Omega)/x]\}$, where E is an evaluation frame and $M[(F^n\ \Omega)/x] \in \mathit{Value}$. Then clearly $S = S'' \circ E'$ for some evaluation stack S'' and evaluation frame E' with $S' = S''[(F^n\ \Omega)/x]$ and $E = E'[(F^n\ \Omega)/x]$. Since $M[(F^n\ \Omega)/x] \in \mathit{Value}$, by (16) we also have that $M[\mathbf{fix}(F)/x] \in \mathit{Value}$ holds. Thus $(S[\mathbf{fix}(F)/x], M[\mathbf{fix}(F)/x]) \rightarrowtail (S''[\mathbf{fix}(F)/x], (E'[\mathbf{fix}(F)/x])\{M[\mathbf{fix}(F)/x]\})$ holds. Together with the induction hypothesis for $(S', M') = (S''[(F^n\ \Omega)/x], (E'\{M\})[(F^n\ \Omega)/x])$, this implies the induction claim.

Case c: $S' = S[(F^n\ \Omega)/x]$ and $M[(F^n\ \Omega)/x] \rightsquigarrow M'$. If $M = x$, then we have $n > 0$, because otherwise $(S[(F^n\ \Omega)/x], \Omega)$ leads to an end configuration, which would be in contradiction to Observation 2.2. From $M = x$, $n > 0$, and $M[(F^n\ \Omega)/x] \rightsquigarrow M'$ follows, by the definition of $\rightsquigarrow$, that $F = (\lambda x.F')$ and $M' = F'[(F^{n-1}\ \Omega)/x]$ for some $F' \in \mathit{Untyped}(\{x\})$.[7] Then by $(S', M') = ((S[(F^n\ \Omega)/x])[(F^{n-1}\ \Omega)/x], F'[(F^{n-1}\ \Omega)/x])$ leading to an end configuration in t steps and statement (I) we know that $(S[(F^n\ \Omega)/x], F'[(F^n\ \Omega)/x])$ leads to an end configuration in t steps. The induction claim then follows by the induction hypothesis for this from $(S[\mathbf{fix}(F)/x], \mathbf{fix}(F)) \rightarrowtail (S[\mathbf{fix}(F)/x], F\ \mathbf{fix}(F)) \rightarrowtail (S[\mathbf{fix}(F)/x], F'[\mathbf{fix}(F)/x])$. If $M \neq x$, then from $M[(F^n\ \Omega)/x] \rightsquigarrow M'$ follows, by the definitions of Ω and $\rightsquigarrow$, the existence of an $N \in \mathit{Untyped}(\{x\})$ such that $M' = N[(F^n\ \Omega)/x]$ and $M[\mathbf{fix}(F)/x] \rightsquigarrow N[\mathbf{fix}(F)/x]$. Consequently, then $(S[\mathbf{fix}(F)/x], M[\mathbf{fix}(F)/x]) \rightarrowtail (S[\mathbf{fix}(F)/x], N[\mathbf{fix}(F)/x])$. Together with the induction hypothesis for $(S', M') = (S[(F^n\ \Omega)/x], N[(F^n\ \Omega)/x])$, this implies the induction claim. This completes the case distinction.

Finally, the lemma is established by reasoning for every $\tau \in \mathit{Typ}$, $S \in \mathit{Stack}(\tau)$, and $F \in \mathit{Term}(\tau \to \tau)$ as follows. By Definition 3.1, $\top(S, \mathbf{fix}(F))$ is:

- 0 if there is some $V \in \mathit{Value}$ with $([\![S]\!], \mathbf{fix}([\![F]\!])) \rightarrowtail^* (\mathit{Id}, V)$,
- i if there is some evaluation stack S' with $([\![S]\!], \mathbf{fix}([\![F]\!])) \rightarrowtail^* (S', \mathbf{error}(i))$, and
- ∞ otherwise.

For the first two cases, the existence of an $n_0 \in \mathbb{N}$ such that for every $n \geq n_0$, $\top(S, F^n\ \mathbf{fix}(\lambda x :: \tau.x))$ has the same outcome follows from statement (II). For

[7] Note that we are free to assume here that the term variable bound in F is x, because untyped terms are identified up to renaming of bound variables.

the last case, we show that for every $n \in \mathbb{N}$, $\top(S, F^n\ \mathbf{fix}(\lambda x :: \tau.x)) = \infty$ as well, by contradiction. Assume there is some $n \in \mathbb{N}$ with $\top(S, F^n\ \mathbf{fix}(\lambda x :: \tau.x)) \neq \infty$. Then $([\![S]\!], [\![F]\!]^n\ \Omega)$ must lead to an end configuration. Combined with statement (III) this contradicts $\top(S, \mathbf{fix}(F)) = \infty$. Note that for the applications of (II) and (III) above we set $M = x$ and use that $[\![S]\!]$ has no free variables. □

Lemma A.4. *Let $\tau, \tau' \in$ Typ, $L \in$ Term(τ-list), and $M_1 \in$ Term(τ'). Let h and t be term variables and M_2 be a typed term such that $h :: \tau, t :: \tau$-list $\vdash M_2 :: \tau'$. If $\omega(L) = 0$, then $\omega(\mathbf{case}\ L\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1;\ h : t \Rightarrow M_2\}) = \omega(M_1)$ or $\omega(\mathbf{case}\ L\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1;\ h : t \Rightarrow M_2\}) = \omega(M_2[H/h, T/t])$ for some $H \in$ Term(τ) and $T \in$ Term(τ-list).*

Proof: If $\omega(L) = 0$, then by Definitions 2.1 and 2.6 there must be some $V \in$ *Value* with $(Id, [\![L]\!]) \rightarrowtail^* (Id, V)$. By a general property of the type-erasing semantics, there must then exist an $L' \in$ Term(τ-list) with $[\![L']\!] = V$. In other words, by the definitions of $[\![\cdot]\!]$ and values, either $V = \mathbf{nil}$ or $V = [\![H]\!] : [\![T]\!]$ for some $H \in$ Term(τ) and $T \in$ Term(τ-list). Moreover, for $S = (\mathbf{case}\ -\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow [\![M_1]\!];\ h : t \Rightarrow [\![M_2]\!]\})$ we have by Observation 2.4 that $(S, [\![L]\!]) \rightarrowtail^* (S, V)$, and thus $(S, [\![L]\!]) \rightarrowtail^* (Id, [\![M_1]\!])$ or $(S, [\![L]\!]) \rightarrowtail^* (Id, [\![M_2[H/h, T/t]]\!])$. From this, the lemma follows by the determinism of $\rightarrowtail$, since by Observations 3.2 and 3.4(a) and Definition 3.1, $\omega(\mathbf{case}\ L\ \mathbf{of}\ \{\mathbf{nil} \Rightarrow M_1;\ h : t \Rightarrow M_2\})$ is:

- 0 if there is some $V' \in$ *Value* with $(S, [\![L]\!]) \rightarrowtail^* (Id, V')$,
- i if there is some evaluation stack S' with $(S, [\![L]\!]) \rightarrowtail^* (S', \mathbf{error}(i))$, and
- ∞ otherwise. □