Compositional Trace Semantics for Lambda Calculus

ANONYMOUS AUTHOR(S)

We present a class of denotational semantics for lambda calculus that generates coinductive traces of a corresponding small-step operational semantics. The appeal of our method is that the generated traces can be elaborated with as much operational detail as needed, and that the structural definition enables favorable induction principles. Use of Guarded Domain Theory enables mutable state and thus the first denotational semantics for call-by-need lambda calculus. We demonstrate the usefulness of our trace semantics by proving correct a static analysis for estimating the number of variable lookups by abstract interpretation.

Additional Key Words and Phrases: Programming language semantics

1 INTRODUCTION

A *static program analysis* infers facts about a program, such as "this program is well-typed", "this higher-order function is always called with argument $\lambda x.x+1$" or "this program never evaluates x". In a functional-language setting, such static analyses are often defined by *structural recursion* on the input term. To prove the analysis correct with respect to the language semantics, it is much easier if the semantics is also defined by structural recursion, as is the case for a traditional *denotational semantics*; then the denotational semantics and the static analysis "line up" and the correctness proof is relatively straightforward. Indeed, one can often regard the analysis as an abstraction, or *abstract interpretation*, of the denotational semantics [Cousot 2021].

Alas, traditional denotational semantics does not model operational details – but those details might be the whole point of the analysis. For example, we might want to ask "How often does e evaluate its free variable x?"; but a standard denotational semantics simply does not express the concept of "evaluating a variable". So we are typically driven to use an *operational semantics* [Plotkin 2004], which directly models operational details like the stack, heap, and sharing, and sees program execution as a sequence of machine states. Now we have two un-appealing alternatives:

- Peform a difficult correctness proof, that links an operational semantics for the language with an analysis defined by structural recursion.
- Re-imagine and re-implement the analysis as an abstraction of the reachable states of an operational semantics. This is the essence of Abstracting Abstract Machines [Van Horn and Might 2010], a very fruitful approach, but one that requires the entire analysis to cleverly reason about heaps.

In this paper we explore a new third path: *to enrich a structurally-defined semantics so that it expresses enough operational detail to support our static analysis.* We make the following contributions:

- We use a concrete example (usage analysis) to explain the problems we sketched above: the lack of operational detail in denotational semantics, and the structural mismatch between the semantics and the analysis (Section 2).
- We describe a new *trace-based denotational semantics for lambda calculus*, that is defined compositionally by structural recursion, but still produces an execution trace that we can later elaborate with as much operational detail as we please (Section 3). Unlike traditional

2475-1421/2024/1-ART1
https://doi.org/10.1145/1111111

formulations of denotational semantics, our semantics is defined by *guarded recursion* [Bizjak et al. 2016] and thus total as a mathematical function, even in the diverging case. We prove it adequate wrt. a standard operational semantics.

- This trace-based semantics encodes the bare minimum of information in traces to be useful. In Section 4 we will see how simple it is to *instrument* our semantics to provide richer traces, carrying *events* describing small-step transitions. The resulting idea of an *eventful* semantics borrows ideas from the *maximal trace semantics* of Cousot [2021].
- In Section 5 we provide several meta-theoretic results about our new class of semantics, such as a partially-ordered heap forcing relation, a characterisation of denotations that allows us to recover a useful semantic equivalance relation and statements of compositionality.
- We finally prove usage analysis correct analysis wrt. the eventful semantics in Section 6, showing that update avoidance [Gustavsson 1998] is an improvement [Moran and Sands 1999] whenever a let binding is evaluated at most once. The definition of "evaluated at most once" is in terms of the *semantic usage abstraction* of our eventful semantics, and our usage analysis is shown to be an abstraction thereof.
- We believe that our notion of trace-based semantics for lambda calculus is the first to model operational detail of call-by-need behavior while enjoying a structural definition. Our pattern naturally generalises to similar definitions generating call-by-value traces, and we believe that its trace-based nature enables unprecedented application of Cousot's intraprocedural framework in interprocedural contexts without the need for a control-flow graph abstraction. We elaborate this claim in Section 7, where we compare to Related Work.

2 PROBLEM STATEMENT

2.1 Usage Analysis and Deadness, Intuitively

Let us begin by defining the object language of this work, a lambda calculus with recursive let bindings and algebraic data types:

$$\begin{array}{rrcll} \text{Variables} & \mathsf{x}, \mathsf{y} & \in & \mathsf{Var} & \quad \text{Constructors } K \in \mathsf{Con} \quad \text{with arity } \alpha_K \in \mathbb{N} \\ \text{Values} & \mathsf{v} & \in & \mathsf{Val} & ::= \quad \bar{\lambda}\mathsf{x}.\mathsf{e} \mid K\,\overline{\mathsf{x}}^{\alpha_K} \\ \text{Expressions} & \mathsf{e} & \in & \mathsf{Exp} & ::= \quad \mathsf{x} \mid \mathsf{v} \mid \mathsf{e}\;\mathsf{x} \mid \mathbf{let}\;\mathsf{x} = \mathsf{e}_1\;\mathbf{in}\;\mathsf{e}_2 \mid \mathbf{case}\;\mathsf{e}\;\mathbf{of}\;\overline{K\,\overline{\mathsf{x}}^{\alpha_K} \to \mathsf{e}} \end{array}$$

The form is reminiscent of Launchbury [1993] and Sestoft [1997] because it is factored into *A-normal form*, that is, the arguments of applications are restricted to be variables, so the difference between call-by-name and call-by-value manifests purely in the semantics of **let**. Note that $\bar{\lambda}x.x$ denotes syntax, whereas $\lambda x.\ x + 1$ denotes a function in math. In this section, only the highlighted parts are relevant; we will discuss data types in Section 3. From hereon throughout, we assume that all labels and bound program variables are distinct.

We give a standard call-by-name denotational semantics $\mathcal{S}_\perp[\![\_]\!]$ in Figure 1b [Scott and Strachey 1971], assigning meaning to our syntax by means of the Scott domain $\mathbb{D}^\perp$ defined in Figure 1a.[1]

Figure 1c defines a static *usage analysis*. The idea is that $\mathcal{S}_u[\![\mathsf{e}]\!]_\rho$ is a function $d \in \mathbb{D}^u$ whose domain is the free variables of e; this function maps each free variable x of e to its *evaluation cardinality* $u \in \mathsf{Usg}$, an upper bound on the number of times that x will be evaluated when e is evaluated (regardless of context). Thus, Usg is equipped with a total order that corresponds to the inclusion relation of the denoted interval of cardinalities: $\perp = 0 \sqsubset 1 \sqsubset \omega = \top$, where ω denotes *any* number of evaluations. This order extends pointwise to a partial ordering on $\mathbb{D}^u$, so $d_1 \sqsubseteq d_2$

[1] The Scott domain $\mathbb{D}^\perp$ is the least fixed-point of the functional $F(X) = [X \to_c X]_\perp$, where $[X \to_c X]_\perp$ is the topology of Scott-continuous endofunctions on X together with a distinct least element $\perp$.

$$\begin{array}{rrcll} \text{Scott Domain} & d & \in & \mathbb{D}^\perp = [\mathbb{D}^\perp \to_c \mathbb{D}^\perp]_\perp & (\rho_1 \sqcup \rho_2)(\mathsf{x}) = \rho_1(\mathsf{x}) \sqcup \rho_2(\mathsf{x}) \\ \text{Usage cardinality} & u & \in & \mathsf{Usg} = \{0, 1, \omega\} & (\rho_1 + \rho_2)(\mathsf{x}) = \rho_1(\mathsf{x}) + \rho_2(\mathsf{x}) \\ \text{Usage Domain} & d & \in & \mathbb{D}^u = \mathsf{Var} \to \mathsf{Usg} & (u * \rho_1)(\mathsf{x}) = u * \rho_1(\mathsf{x}) \end{array}$$

(a) Syntax of semantic domains

$\boxed{\mathcal{S}_\perp[\![\_]\!] : \mathsf{Exp} \to (\mathsf{Var} \to \mathbb{D}^\perp) \to \mathbb{D}^\perp}$

$$\begin{array}{rl} \mathcal{S}_\perp[\![\mathsf{x}]\!]_\rho = & \rho(\mathsf{x}) \\ \mathcal{S}_\perp[\![\bar\lambda \mathsf{x}.\mathsf{e}]\!]_\rho = & \lambda d.\ \mathcal{S}_\perp[\![\mathsf{e}]\!]_{\rho[\mathsf{x}\mapsto d]} \\ \mathcal{S}_\perp[\![\mathsf{e}\ \mathsf{x}]\!]_\rho = & \begin{cases} f(\rho(\mathsf{x})) & \text{if } \mathcal{S}_\perp[\![\mathsf{e}]\!] = f \\ \perp & \text{otherwise} \end{cases} \\ \mathcal{S}_\perp[\![\textbf{let } \mathsf{x}{=}\mathsf{e}_1 \textbf{ in } \mathsf{e}_2]\!]_\rho = & \begin{array}{ll} \text{letrec } \rho'. & \rho' = \rho \sqcup [\mathsf{x}\mapsto d_1] \\ & d_1 = \mathcal{S}_\perp[\![\mathsf{e}_1]\!]_{\rho'} \\ \text{in} & \mathcal{S}_\perp[\![\mathsf{e}_2]\!]_{\rho'} \end{array} \end{array}$$

(b) Denotational semantics after Scott

$\boxed{\mathcal{S}_u[\![\_]\!] : \mathsf{Exp} \to (\mathsf{Var} \to \mathbb{D}^u) \to \mathbb{D}^u}$

$$\begin{array}{rl} \mathcal{S}_u[\![\mathsf{x}]\!]_\rho = & \rho(\mathsf{x}) \\ \mathcal{S}_u[\![\bar\lambda \mathsf{x}.\mathsf{e}]\!]_\rho = & \omega * \mathcal{S}_u[\![\mathsf{e}]\!]_{\rho[\mathsf{x}\mapsto\perp]} \\ \mathcal{S}_u[\![\mathsf{e}\ \mathsf{x}]\!]_\rho = & \mathcal{S}_u[\![\mathsf{e}]\!] + \omega * \rho(\mathsf{x}) \\ \mathcal{S}_u[\![\textbf{let } \mathsf{x}{=}\mathsf{e}_1 \textbf{ in } \mathsf{e}_2]\!]_\rho = & \begin{array}{ll} \text{letrec } \rho'. & \rho' = \rho \sqcup [\mathsf{x}\mapsto d_1] \\ & d_1 = [\mathsf{x}\mapsto 1] + \mathcal{S}_u[\![\mathsf{e}_1]\!]_{\rho'} \\ \text{in} & \mathcal{S}_u[\![\mathsf{e}_2]\!]_{\rho'} \end{array} \end{array}$$

(c) Naïve usage analysis

Fig. 1. Connecting usage analysis to denotational semantics

whenever $d_1(\mathsf{x}) \sqsubseteq d_2(\mathsf{x})$, for all x and $\perp_{\mathbb{D}^u} = \lambda\_.\ \perp_{\mathsf{Usg}}$. We will often leave off the subscript of, e.g., $\perp_{\mathbb{D}^u}$ when it can be inferred from context.

So, for example $\mathcal{S}_u[\![x + 1]\!]_\rho$ is a function mapping x to 1 and all other variables to 0 (for a suitable ρ that we discuss in due course). Any number of uses beyond 1, such as 2 for x in $\mathcal{S}_u[\![x + x]\!]_\rho$, are collapsed to $d(x) = \omega$. Similarly $\mathcal{S}_u[\![\bar\lambda v.v + z]\!]_\rho$ maps z to ω, since the lambda might be called many times.

For open expressions, we need a way to describe the denotations of its free variables with a *usage environment*[2] $\tilde\rho \in \mathsf{Var} \to \mathbb{D}^u$. Given a usage environment $\tilde\rho$, $\tilde\rho(\mathsf{x})(\mathsf{y})$ models an upper bound on how often the expression bound to x evaluates y.

We say that x is *dead* in an element $d \in \mathbb{D}^u$ whenever $d(\mathsf{x}) = 0$, and otherwise that x is *potentially live* in d. Let us call

$$\tilde\rho_\Delta(\mathsf{x}) \triangleq \lambda \mathsf{y}.\ (\!|\, \mathsf{x} = \mathsf{y}\ ?\ 1 \,:\, 0 \,|\!)$$

the "diagonal" usage environment, assigning each free variable x a unique meaning in terms of a denotation that evaluates x once and nothing else. Then we will say that x is dead in e whenever it is dead in $\mathcal{S}_u[\![\mathsf{e}]\!]_{\tilde\rho_\Delta}$, i.e., $\mathcal{S}_u[\![\mathsf{e}]\!]_{\tilde\rho_\Delta}(\mathsf{x}) = 0$. In this way, $\mathcal{S}_u[\![\_]\!]$ can be used to infer facts of the form "e never evaluates x" from the introduction. When $d(\mathsf{x}) \sqsupset 0$ we say that x is *potentially live* in e.

Note that the usage analysis is naïve in its treatment of function application: It assumes that that every function deeply evaluates its argument. Whenever x is potentially live in $\tilde\rho(y)$, the analysis will report that x is potentially live in $(f\ y)$, regardless of whether f evaluates its argument *at all*. In turn, the analysis assumes in the lambda case that liveness of the argument has been accounted for in the application case, hence the lambda-bound variable is dead in any variable and denoted by $\perp$ in the extended $\tilde\rho$. The result is that $\mathcal{S}_u[\![x + \bar\lambda y.y\ z]\!]_{\tilde\rho_\Delta}(x) = 1$, because x is never passed as an argument to the lambda.

You might be convinced now that although the definition of this analysis is rather simple and the results it finds are rather imprecise, it is already quite tricky in detail. A proof of its correctness is in order.

[2]Note that we will occassionally adorn with ~ to disambiguate elements of the analysis domain from the semantic domain.

2.2 Usage Analysis Infers Denotational Deadness

The requirement (in the sense of informal specification) on an assertion such as "x is dead" in a program like **let** $x = \mathsf{e}_1$ **in** e_2 is that we may rewrite to **let** $x = \mathit{panic}$ **in** e_2 and even to e_2 without observing any change in semantics. Doing so reduces code size and heap allocation.

This can be made formal in the following definition of deadness in terms of $\mathcal{S}_\bot[\![\_]\!]$:

Definition 1 (Deadness)**.** *A variable* x *is* dead *in an expression* e *if and only if, for all* $\rho \in \mathsf{Var} \to \mathbb{D}^\bot$ *and* $d_1, d_2 \in \mathbb{D}^\bot$*, we have* $\mathcal{S}_\bot[\![\mathsf{e}]\!]_{\rho[\mathsf{x} \mapsto d_1]} = \mathcal{S}_\bot[\![\mathsf{e}]\!]_{\rho[\mathsf{x} \mapsto d_2]}$*. Otherwise,* x *is* live*.*

Indeed, if we know that x is dead in e_2, then the following equation justifies our rewrite above: $\mathcal{S}_\bot[\![\textbf{let } x = \mathsf{e}_1 \textbf{ in } \mathsf{e}_2]\!]_\rho = \mathcal{S}_\bot[\![\mathsf{e}_2]\!]_{\rho[x \mapsto d]} = \mathcal{S}_\bot[\![\mathsf{e}_2]\!]_{\rho[x \mapsto \rho(x)]} = \mathcal{S}_\bot[\![\mathsf{e}_2]\!]_\rho$ (for all ρ and the suitable d), where we make use of deadness in the second $=$. So our definition of deadness is not only simple to grasp, but also simple to exploit.

We can now prove our usage analysis correct as a deadness analysis in terms of this notion: (All proofs can be found in the Appendix; in case of the extended version via clickable links.)

Theorem 2 ($\mathcal{S}_u[\![\_]\!]$ is a correct deadness analysis)**.** *Let* e *be an expression and* x *a variable. If* $\mathcal{S}_u[\![\mathsf{e}]\!]_{\tilde{\rho}_\Delta}(\mathsf{x}) = 0$ *then* x *is dead in* e*.*

The main takeaway of this Section is the simplicity of this proof! Since $\mathcal{S}_\bot[\![\_]\!]$ and $\mathcal{S}_u[\![\_]\!]$ are so similar in structure, a proof by induction on the program expression is possible. The induction hypothesis needs slight generalisation and that admittedly involves a bit of trial and error, but all in all, it is a simple and direct proof, at just under a page of accessible prose.

Simplicity of proofs is a good property to strive for, so let us formulate an explicit goal:

> **Goal 1**: Give a semantics that makes correctness proofs similarly simple.

This is a rather vague goal and should be seen as the overarching principle of this work. In support of this principle, we will formulate a few more concrete goals that each require more context to make sense.

2.3 Continuity and Divergence

Our notion of deadness has a blind spot wrt. diverging computations: $\mathcal{S}_\bot[\![\_]\!]$ denotes any looping program such as $\mathsf{e}_{\mathit{loop}} \triangleq \textbf{let } \mathit{loop} = \bar{\lambda} x.\mathit{loop}\ x \textbf{ in } \mathit{loop}\ \mathit{loop}$ by $\bot$, the same element that is meant to indicate partiality in the approximation order of the domain (e.g., insufficiently specified input). Hence any looping program is automatically dead in all its free variables, even though any of them might influence which particular endless loop is taken.

This is not only a curiosity of $\mathcal{S}_u[\![\_]\!]$; it also applies to the original control-flow analysis work [Shivers 1991, p. 23] where it is remedied by the introduction of a *non-standard semantic interpretation* that assigns meaning to diverging programs where the denotational semantics only says $\bot$. The bottom element of the non-standard domain is the empty set of reached program labels. So if the non-standard semantics were to assign the bottom element to diverging programs like the denotational semantics does, it would neglect that any part of the program before the loop was ever reached. The non-standard semantic interpretation is far more reasonable, but credibility of this approach solely rests on the structural similarity to the standard denotational semantics.

Furthermore, as is often done, $\mathcal{S}_\bot[\![\_]\!]$ abuses $\bot$ as a collecting pool for error cases. This shows in the following example: x is dead in $(\bar{\lambda} y.\bar{\lambda} z.z)\ x$, but dropping the **let** binding in **let** $x = \mathsf{e}_1$ **in** $(\bar{\lambda} y.\bar{\lambda} z.z)\ x$ introduces a scoping error which is not observable under $\mathcal{S}_\bot[\![\_]\!]$. We could take inspiration in the work of Milner [1978] and navigate around the issue by introducing a **wrong** denotation for errors which is propagated strictly; then we would notice when we optimise a looping program such as $\mathsf{e}_{\mathit{loop}}$ into one that has a scoping error.

We can try to apply that same trick to diverging programs and assign them a total domain element div distinct from all other elements. But then when would we go from $\bot$ to div in the least fixed-point in $\mathcal{S}_\bot[\![\textbf{let } \_ = \_ \textbf{ in } \_]\!]$? There is no way to introduce div unless we make it a partial element that approximates every total element; but then we could have just chosen $\bot$ as the denotation to begin with. Hence denoting diverging programs by a partial element $\bot$ or div is a fundamental phenomenon of traditional denotational semantics and brings with it annoying pitfalls. To see one, consider the predicate "Denotation d will get stuck and not diverge". This predicate is not *admissable* [Abramsky et al. 1994], because an admissable predicate that is at all satisfiable would need to be true for the partial elements $d = \bot$ and $d = \text{div}$. A practical consequence is that such a predicate could not be proven about a recursive let binding because that would need admissability to apply fixpoint induction.

We would rather not need to reason about continuity and admissability. Hence we formulate the following goal:

> **Goal 2**: Find a semantic domain in which diverging programs can be denoted by a total element.

2.4 Operational Detail and Structural Mismatch

Blind spots and annoyances notwithstanding, the denotational notion of deadness above is quite reasonable and Theorem 2 is convincing. But our usage analysis infers more detailed cardinality information; for example, it can infer whether a binding is evaluated at most once.[3] This information can be useful to inline a binding that is not a value or to avoid update frames under call-by-need [Gustavsson 1998; Sergey et al. 2017].[4] Thus, our usage analysis should satisfy the following generalisation of Theorem 2:

Theorem 3 (Correctness of $\mathcal{S}_u[\![\_]\!]$)**.** *Let* e *be an expression,* x *a variable. If* $\mathcal{S}_u[\![\text{e}]\!]_{\tilde{\rho}_\Delta}(\text{x}) \sqsubseteq u$ *then* e *evaluates* x *at most* u *times.*

Unfortunately, our denotational semantics does not allow us to express the operational property "e evaluates x at most u times", so this theorem cannot be proven correct.

> **Goal 3**: Find a semantics that can observe operational detail such as arbitrary evaluation cardinality.

2.5 Structural Mismatch

Let us try a different approach then and define a stronger notion of deadness in terms of a small-step operational semantics such as the Mark II machine of Sestoft [1997] given in Figure 2, the semantic ground truth for this work. (A close sibling for call-by-value would be a CESK machine [Felleisen and Friedman 1987] or a simpler derivative thereof.) It is a Lazy Krivine (LK) machine implementing call-by-need, so for a meaningful comparison to the call-by-name semantics $\mathcal{S}_\bot[\![\_]\!]$, we ignore rules Case$_1$, Case$_2$, Upd and the pushing of update frames in Look for now to recover a call-by-name Krivine machine with explicit heap addresses.[5]

[3] Thus our usage analysis is a combination of deadness analysis and *sharing analysis* [Gustavsson 1998].

[4] A more useful application of the "at most once" cardinality is the identification of *one-shot* lambdas [Sergey et al. 2017] — functions which are called at most once for every activation — because it allows floating of heap allocations from a hot code path into cold function bodies. Simplicity prohibits $\mathcal{S}_u[\![\_]\!]$ from inferring such properties.

[5] Note that discarding update frames makes the heap entries immutable, which makes the explicit heap unnecessary. Of course, for call-by-name we would not need a heap to begin with, but the point is to get a glimpse at the effort necessary for call-by-need.

$$
\begin{array}{rrcll}
\text{Addresses} & \mathsf{a} & \in & \mathsf{Addr} \simeq \mathbb{N} & \text{States } \sigma \in \mathbb{S} = \mathsf{Exp} \times \mathbb{E} \times \mathbb{H} \times \mathbb{K} \\
\text{Environments} & \rho & \in & \mathbb{E} = \mathsf{Var} \rightharpoonup \mathsf{Addr} & \text{Heaps } \mu \in \mathbb{H} = \mathsf{Addr} \rightharpoonup \mathbb{E} \times \mathsf{Exp} \\
\text{Continuations} & \kappa & \in & \mathbb{K} ::= \mathbf{stop} \mid \mathbf{ap}(\mathsf{a}) \cdot \kappa \mid \mathbf{sel}(\rho, \overline{K\ \bar{\mathsf{x}}^{\alpha_K} \to \mathsf{e}}) \cdot \kappa \mid \mathbf{upd}(\mathsf{a}) \cdot \kappa &
\end{array}
$$

Rule	$\sigma_1 \hookrightarrow \sigma_2$	where
Bind	$(\mathbf{let}\ \mathsf{x} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2, \rho, \mu, \kappa) \hookrightarrow (\mathsf{e}_2, \rho', \mu[\mathsf{a} \mapsto (\rho', \mathsf{e}_1)], \kappa)$	$\mathsf{a} \notin \mathrm{dom}(\mu),\ \rho' = \rho[\mathsf{x} \mapsto \mathsf{a}]$
App$_1$	$(\mathsf{e}\ \mathsf{x}, \rho, \mu, \kappa) \hookrightarrow (\mathsf{e}, \rho, \mu, \mathbf{ap}(\mathsf{a}) \cdot \kappa)$	$\mathsf{a} = \rho(\mathsf{x})$
Case$_1$	$(\mathbf{case}\ \mathsf{e}_s\ \mathbf{of}\ \overline{K\ \bar{\mathsf{x}} \to \mathsf{e}_r}, \rho, \mu, \kappa) \hookrightarrow (\mathsf{e}_s, \rho, \mu, \mathbf{sel}(\rho, \overline{K\ \bar{\mathsf{x}} \to \mathsf{e}_r}) \cdot \kappa)$	
Look	$(\mathsf{x}, \rho, \mu, \kappa) \hookrightarrow (\mathsf{e}, \rho', \mu, \mathbf{upd}(\mathsf{a}) \cdot \kappa)$	$\mathsf{a} = \rho(\mathsf{x}),\ (\rho', \mathsf{e}) = \mu(\mathsf{a})$
App$_2$	$(\bar{\lambda}\mathsf{x}.\mathsf{e}, \rho, \mu, \mathbf{ap}(\mathsf{a}) \cdot \kappa) \hookrightarrow (\mathsf{e}, \rho[\mathsf{x} \mapsto \mathsf{a}], \mu, \kappa)$	
Case$_2$	$(K'\ \overline{y}, \rho, \mu, \mathbf{sel}(\rho', \overline{K\ \bar{\mathsf{x}} \to \mathsf{e}}) \cdot \kappa) \hookrightarrow (\mathsf{e}_i, \rho'[\overline{\mathsf{x}_i \mapsto \mathsf{a}}], \mu, \kappa)$	$K_i = K',\ \overline{\mathsf{a} = \rho(\mathsf{y})}$
Upd	$(\mathsf{v}, \rho, \mu, \mathbf{upd}(\mathsf{a}) \cdot \kappa) \hookrightarrow (\mathsf{v}, \rho, \mu[\mathsf{a} \mapsto (\rho, \mathsf{v})], \kappa)$	

Fig. 2. Lazy Krivine transition semantics $\hookrightarrow$

The configurations σ in this transition system resemble abstract machine states, consisting of a control expression e, an environment ρ mapping lexically-scoped variables to their current heap address, a heap μ listing a closure for each address, and a stack of continuation frames κ.

The notation $f \in A \rightharpoonup B$ used in the definition of ρ and μ denotes a finite map from A to B, a partial function where the domain $\mathrm{dom}(f)$ is finite and $\mathrm{rng}(f)$ denotes its range. The literal notation $[a_1 \mapsto b_1, ..., a_n \mapsto b_n]$ denotes a finite map with domain $\{a_1, ..., a_n\}$ that maps a_i to b_i. Function update $f[a \mapsto b]$ maps a to b and is otherwise equal to f.

The initial machine state for a closed expression e is given by the injection function $inj(\mathsf{e}) = (\mathsf{e}, [], [], \mathbf{stop})$ and the final machine states are of the form $(\mathsf{v}, \_, \_, \mathbf{stop})$. We bake into $\sigma \in \mathbb{S}$ the simplifying invariant of *well-addressedness*: Any address a occuring in ρ, κ or the range of μ must be an element of $\mathrm{dom}(\mu)$. It is easy to see that the transition system maintains this invariant and that it is still possible to observe scoping errors which are thus confined to lookup in ρ.

This machine allows us to observe variable lookups as Look transitions. Unfortunately, we still suffer from immense proof complexity caused by the *structural mismatch* between transition semantics and compositional analysis, as we shall see in due course. Let us first try to define a notion of deadness that is similar to Definition 1, in terms of *contextual equivalence* as in [Moran and Sands 1999]:

Definition 4 (Deadness, operationally)**.** *Let* e *be an expression and* x *a variable.* x *is* dead *in* e *if and only if for any evaluation context* (ρ, μ, κ) *and expressions* $\mathsf{e}_1, \mathsf{e}_2$ *(where* x *does not occur in the context) the sequences of transitions* $(\mathbf{let}\ \mathsf{x} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}, \rho, \mu, \kappa) \hookrightarrow^*$ *and* $(\mathbf{let}\ \mathsf{x} = \mathsf{e}_2\ \mathbf{in}\ \mathsf{e}, \rho, \mu, \kappa) \hookrightarrow^*$ *operate in lockstep. Otherwise,* x *is* live*.*

This definition captures diverging behaviors correctly and straightforwardly legitimises the transformation we want to perform, without any mention of addresses. It is however unwieldy in a correctness proof due to its use of bisimulation, so a bit of rejigging is in order:

Lemma 5 (Without proof)**.** x *is dead in* e *if and only if for any evaluation context* (ρ, μ, κ) *and* $\mathsf{a} \notin \mathrm{dom}(\mu)$ *there exists no sequence of transitions* $(\mathsf{e}, \rho[\mathsf{x} \mapsto \mathsf{a}], \mu[\mathsf{a} \mapsto ([], \bar{\lambda}z.z)], \kappa) \hookrightarrow^* (\mathsf{y}, \rho', \mu', \kappa')$ *such that* $\rho'(\mathsf{y}) = \mathsf{a}$*.*

This property is a bit easier to handle in a proof. However, note that it is not compositional in e: To see that, consider a variable occurrence y; is x dead in y? That depends on which expression y is bound to in the heap, but our deadness predicate does not make assumptions about free variables.

Consequently, it is impossible to prove that $\mathcal{S}_u[\![\mathrm{e}]\!]$ satisfies Theorem 2 by direct structural induction on e (in a way that would be useful to the proof).

Instead, such proofs are often conducted by induction over the reflexive transitive closure of the transition relation. For that it is necessary to give an inductive hypothesis that considers environments, stacks and heaps. One way is to extend the analysis function $\mathcal{S}_u[\![\_]\!]$ to entire configurations and then prove that if $\sigma_1 \hookrightarrow \sigma_2$ we have $\mathcal{S}_u[\![\sigma_2]\!] \sqsubseteq F(\mathcal{S}_u[\![\sigma_1]\!])$, where F is the abstraction of the particular transition rule taken and is often left implicit. This is a daunting task, for multiple reasons: First off, $\mathcal{S}_u[\![\_]\!]$ might be quite complicated in practice and extending it to entire configurations multiplies this complexity. Secondly, $\mathcal{S}_u[\![\_]\!]$ makes use of fixpoints in the let case, so $\mathcal{S}_u[\![\sigma_2]\!] \sqsubseteq F(\mathcal{S}_u[\![\sigma_1]\!])$ relates fixpoints that are "out of sync", implying a need for fixpoint induction.

In call-by-need, there will be a fixpoint between the heap and stack due to update frames acting like heap bindings whose right-hand side is under evaluation (a point that is very apparent in the semantics of Ariola et al. [1995]), so fixpoint induction needs to be applied *at every case of the proof*, diminishing confidence in correctness unless the proof is fully mechanised.

For an analogy with type systems: What we just tried is like attempting a proof of preservation by referencing the result of an inference algorithm rather than the declarative type system. So what is often done instead is to define a declarative and more permissive *correctness relation* $C(\sigma)$ to prove preservation $C(\sigma_1) \implies C(\sigma_2)$ (e.g., that C is *logical* wrt. $\hookrightarrow$). C is chosen such that (1) it is strong enough to imply the property of interest (deadness) (2) it is weak enough that it is implied by the analysis result for an initial state ($\tilde{\rho}(\mathrm{x}) \not\sqsubseteq \mathcal{S}_u[\![\mathrm{e}]\!]_{\tilde{\rho}}$). Examples of this approach are the "well-annotatedness" relation in [Sergey et al. 2017, Lemma 4.3] or $\sim_V$ in [Nielson et al. 1999, Theorem 2.21]) and in fact the correctness relations we give in Sections 3 and 4. We found it quite hard to come up with a suitable ad-hoc correctness relation for usage analysis, though, and postpone further discussion to Section 6, where the full correctness relation in Theorem 3 and its proof is implicitly derived by abstract interpretation [Cousot 2021].

We have just descended into a mire of complexity to appreciate the following goal:

Goal 4: Avoid structural mismatch between semantics and analysis.

Goal 4 can be seen as a necessary condition for Goal 1: Structural mismatch makes a simple proof impossible. Since our analysis is compositional, we strive for a compositional semantics in Sections 3 and 4. An alternative to avoid structural mismatch is to follow Abstracting Abstract Machines [Van Horn and Might 2010]. We will compare our approach to theirs in Section 7.

3 A DENOTATIONAL SEMANTICS FOR CALL-BY-NEED

3.1 Transition System

Figure 2 gave an operational semantics in terms of a small-step transition system closest to the lazy Krivine machine [Ager et al. 2004] for Launchbury's language as presented in Sestoft [1997]. It is worth having a second look at the workings of our gold standard.

When the control expression of a state σ (selected via $ctrl(\sigma)$) is a value v, we call σ a *return* state and say that the continuation (selected via $cont(\sigma)$) drives evaluation. Otherwise, σ is an *evaluation* state and $ctrl(\sigma)$ drives evaluation. The entries in the heap μ are *closures* of the form (ρ, e), where the environment ρ closes over the expression e. Finally, $cont(\sigma)$ lists actions to be taken in a return state, such as applying the result to an argument address or updating a heap entry with its value.

Heap entries are introduced via Bind transitions under a *fresh* address $a \notin \mathrm{dom}(\mu)$ that we call an *activation* of the let-bound variable x. The lexical activation of every variable in scope is maintained in ρ. The App$_1$ rule pushes an *application frame* with the address of the argument variable onto the stack, while the rule Look pushes an *update frame* with the address of the variable the heap entry

of which is accessed. When a return state is reached, the original heap entry is overwritten with the value in the control. Similarly to application, evaluating a **case** expression pushes a *selector frame* onto the stack via Case$_1$. When subsequent evaluation reaches a data constructor value, a Case$_2$ transition looks up the corresponding case alternative in the selector frame. For our examples, we will assume that we have defined the data type `bool ::= tt | ff`, where `tt` and `ff` are two nullary data constructors.

Let us conclude with an example trace in this transition system, evaluating $\mathrm{e} \triangleq \textbf{let}\ i = \bar{\lambda} x.x\ \textbf{in}\ i\ i$:

$$\begin{array}{rcl} (\mathrm{e}, [], [], \textbf{stop}) & \hookrightarrow & (i\ i, \rho_1, \mu, \textbf{stop}) \hookrightarrow (i, \rho_1, \mu, \kappa_1) \hookrightarrow (\bar{\lambda} x.x, \rho_1, \mu, \kappa_2) \\ \hookrightarrow (\bar{\lambda} x.x, \rho_1, \mu, \kappa_1) & \hookrightarrow & (x, \rho_2, \mu, \textbf{stop}) \hookrightarrow (\bar{\lambda} x.x, \rho_1, \mu, \kappa_3) \hookrightarrow (\bar{\lambda} x.x, \rho_1, \mu, \textbf{stop}) \end{array}$$

$$\text{where} \quad \begin{array}{lll} \rho_1 = [i \mapsto \mathrm{a}_1] & \rho_2 = [i \mapsto \mathrm{a}_1, x \mapsto \mathrm{a}_1] & \mu = [\mathrm{a}_1 \mapsto (\rho_1, \bar{\lambda} x.x)] \\ \kappa_1 = \textbf{ap}(\mathrm{a}_1) \cdot \textbf{stop} & \kappa_2 = \textbf{upd}(\mathrm{a}_1) \cdot \kappa_1 & \kappa_3 = \textbf{upd}(\mathrm{a}_1) \cdot \textbf{stop} \end{array}$$

3.2 (Algebraic) Domain Theory

The challenge that domain theory sets out to solve is that the "inductive datatype"

$$D ::= \mathsf{Fun}(f \in D \to D) \mid \bot$$

is ill-defined: The usual interpretation of such a declaration as the least fixed-point of the implied set-valued functional $F(X) \triangleq \{f \mid \forall a \in X.\ \exists b \in X.\ f(a) = b\} \cup \{\bot\}$ does not exist.

To see that, suppose μF was that set. Then there exists an injection ("data constructor") Fun from $\mu F \to \mu F = \mu F^{\mu F}$ into μF. We can see that $\{\bot, (\lambda \_.\ \bot)\} \subseteq \mu F$, so there are at least two elements in μF. Then to accomodate $\mu F^{\mu F}$, μF must be at least as large as $2^{\mu F}$, the set of two-valued functions on μF. But this latter set is one-to-one with $\wp(\mu F)$ and it is a known result by Cantor that $\wp(\mu F)$ has greater cardinality than μF, in contradiction to the existence of the injection Fun.

Domain theory, on the other hand, interprets the implied recursion equation in terms of topology and continuous functions, where the fixed-point exists when restricted to *algebraic domains* [Scott 1970]. At the same time, algebraic domains are expressive enough to encode any computable function as a continuous function.

3.3 (Synthetic) Guarded Domain Theory

As we have discussed in Section 2.3, there are a few strings attached to working with continuity and partiality in the context of denotational semantics.

The key to getting rid of partiality and thus denoting infinite computations with total elements is to avoid working with algebraic domains altogether and instead work in a total type theory with *guarded recursive types*, such as Guarded Dependent Type Theory (GDTT) [Bizjak et al. 2016] or Ticked Cubical Type Theory [Møgelberg and Veltri 2019].[6] The fundamental innovation of these theories is the integration of the "later" modality $\triangleright$ which allows to define coinductive data types with negative recursive occurrences such in our "data type" D from Section 3.2, as first realised by Nakano [2000].

Whereas previous theories of coinduction require syntactic productivity checks [Coquand 1994], requiring tiresome constraints on the form of guarded recursive functions, the appeal of GDTT is that productivity is instead proven semantically, in the type system.

The way that GDTT achieves this is roughly as follows: The type $\triangleright T$ represents data of type T that will become available after a finite amount of computation, such as unrolling one layer of

[6] Of course, in reality we are just using GDTT as a meta language [Moggi 2007] with a known domain-theoretic model in terms of the topos of trees [Bizjak et al. 2016]. This meta language is sufficiently expressive as a logic to express proofs, though, justifying the view that we are extending "math" with the ability to conveniently reason about computable functions on infinite data without needing to think about topology and approximation directly.

a fixpoint definition. It comes with a general fixpoint combinator $\mathsf{fix} : \forall A.\ (\triangleright A \to A) \to A$ that can be used to define both coinductive *types* (via guarded recursive functions on the universe of types [Birkedal and Mogelberg 2013]) as well as guarded recursive *terms* inhabiting said types. The classic example is that of coinductive streams:

$$Str = \mathbb{N} \times \triangleright Str \qquad ones = \mathsf{fix}(r : \triangleright Str).\ (1, r),$$

where $ones : Str$ is the constant stream of 1. In particular, Str is the fixpoint of a locally contractive functor $F(X) = \mathbb{N} \times \triangleright X$. According to Birkedal and Mogelberg [2013], any type expression in simply typed lambda calculus defines a locally contractive functor as long as any occurrence of X is under a $\triangleright$, so we take that as the well-formedness criterion of coinductive types in this work. The most exciting consequence is that $D ::= \mathsf{Fun}(f \in \triangleright D \to \triangleright D) \mid \bot$ (where $\bot$ is interpreted as a plain nullary data constructor rather than as the least element of some partial order) is a sound coinductive encoding of the data type in Section 3.2. Even unguarded positive occurrences as in $D ::= \mathsf{Fun}(f \in \triangleright D \to D) \mid \bot$ are permissible as

As a type constructor, $\triangleright$ is an applicative functor [McBride and Paterson 2008] via functions

$$\mathsf{next} : \forall A.\ A \to \triangleright A \qquad \_ \circledast \_ : \forall A, B.\ \triangleright(A \to B) \to \triangleright A \to \triangleright B,$$

allowing us to apply a familiar framework of reasoning around $\triangleright$. In order not to obscure our work with pointless symbol pushing in, e.g., Figure 3, we will often omit the idiom brackets [McBride and Paterson 2008] $\{\!|\_|\!\}$ to indicate where the $\triangleright$ "effects" happen. Rest assured, all $\triangleright$ are present in the Guarded Cubical Agda development for Figures 3 and 5 in the Supplement.

3.4 Definition

Figure 3 defines $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$ by structural recursion on an input expression e. Given a free variable environment ρ, $\mathcal{S}_{\mathsf{L}}[\![\mathsf{e}]\!]_\rho$ assigns e a denotation d in terms of the semantic domain $\mathbb{D}^{\mathsf{L}}$ of stateful call-by-need trace functions. If such a trace function is supplied the heap μ just before e takes control, then $d(\mu)$ is a trace τ starting at e in that heap μ. As in the LK transition system ($\hookrightarrow$), the job of the environment ρ is to assign meaning to free variables of e via address tokens a bound in the heap. If evaluation of e terminates, then τ will be a finite list of Ls ending with $\langle v, \mu' \rangle_{\checkmark}$ for some return semantic value v and heap μ'. Otherwise, it might be finite but stuck ($\lightning$), or diverge without ever leaving e, in which case τ will be an infinite layering of Ls.

One can think of L (read as "later" or "delayed") as introducing a finite portion of latency into the trace – an atomic computation step. It makes crucial use of the later modality $\triangleright$ to naturally encode divergence as an infinite sequence of productive steps.

Compared to the LK transition semantics, the most striking difference is that the returned trace does not contain any intermediate information such as an environment or control stack component. The entire state is internal to the definition of $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$: The stack in particular is implicitly encoded in call structure while the environment follows lexical structure. We will see that for every LK transition, the trace will have one L step.

The second difference is that the heap μ does not map to syntactic closures but to delayed semantic denotations $\triangleright\mathbb{D}^{\mathsf{L}}$. This is so that the transitive negative occurrence arising through $\mathbb{D}^{\mathsf{L}} \to \mathbb{H} \to \mathbb{D}^{\mathsf{L}}$ is broken and the definition "type-checks". With algebraic domain theory, we would instead have to justify that this domain equation has a solution and that $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$ is in fact continuous.

The choice to have environments map to addresses leads to function values $\mathsf{Fun}(f)$ that take said addresses as parameter, rather than a (necessarily guarded) $\mathbb{D}^{\mathsf{L}}$ as in $\mathcal{S}_{\bot}[\![\_]\!]$.This allows the embedding of function values $\mathsf{Fun}(f)$ in the lambda case $\mathcal{S}_{\mathsf{L}}[\![\tilde{\lambda}\mathsf{x}.\mathsf{e}]\!]$, enabling a compositional definition of the application case $\mathcal{S}_{\mathsf{L}}[\![\mathsf{e}\ \mathsf{x}]\!]$, just as in $\mathcal{S}_{\bot}[\![\_]\!]$.

$$\begin{array}{rrcl} \text{Environment} & \rho \in \mathbb{E} & = & \mathrm{Var} \rightharpoonup \mathrm{Addr} \\ \text{Trace} & \tau \in \mathbb{T}^{\mathsf{L}} & ::= & \langle v, \mu \rangle_{\checkmark} \mid \lightning \mid \mathsf{L}\ \tau^{\blacktriangleright} \\ \text{Domain} & d \in \mathbb{D}^{\mathsf{L}} & = & \mathbb{H} \to \mathbb{T}^{\mathsf{L}} \end{array} \qquad \begin{array}{rrcl} \text{Heap} & \mu \in \mathbb{H} & = & \mathrm{Addr} \rightharpoonup \blacktriangleright\mathbb{D}^{\mathsf{L}} \\ \text{Delayed trace} & \tau^{\blacktriangleright} \in \blacktriangleright\mathbb{T}^{\mathsf{L}} & & \\ \text{Delayed element} & d^{\blacktriangleright} \in \blacktriangleright\mathbb{D}^{\mathsf{L}} & & \end{array}$$

$$\text{Value} \quad v \in \mathbb{V}^{\mathsf{L}} ::= \mathrm{Fun}(f \in (\mathrm{Addr} \to \blacktriangleright\mathbb{D}^{\mathsf{L}})) \mid \mathrm{Con}(K, \overline{\mathsf{a}}^{\alpha_K})$$

$$(\gg\!\beta\!=) : \mathbb{D}^{\mathsf{L}} \to (\mathbb{V}^{\mathsf{L}} \rightharpoonup \mathbb{D}^{\mathsf{L}}) \to \mathbb{D}^{\mathsf{L}} \quad \mathit{ret} : \mathbb{V}^{\mathsf{L}} \to \mathbb{D}^{\mathsf{L}} \quad \mathit{apply} : \mathbb{D}^{\mathsf{L}} \times \mathrm{Addr} \to \mathbb{D}^{\mathsf{L}}$$
$$\mathit{memo} : \mathrm{Addr} \times \mathbb{D}^{\mathsf{L}} \to \mathbb{D}^{\mathsf{L}} \quad \mathit{select} : \mathbb{D}^{\mathsf{L}} \times ((K : \mathrm{Con}) \times \mathsf{a}^{\alpha_K} \rightharpoonup \mathbb{D}^{\mathsf{L}}) \to \mathbb{D}^{\mathsf{L}}$$

$$(d \gg\!\beta\!= f)(\mu) = \begin{cases} \mathsf{L}^n\ f(v, \mu') & d(\mu) = \mathsf{L}^n\ \langle v, \mu' \rangle_{\checkmark} \text{ and } (v, \mu') \in \mathrm{dom}(f) \\ \mathsf{L}^n\ \lightning & d(\mu) = \mathsf{L}^n\ \langle v, \mu' \rangle_{\checkmark} \text{ and } (v, \mu') \notin \mathrm{dom}(f) \\ d(\mu) & \text{otherwise} \end{cases}$$

$$\begin{array}{rcl} \mathit{ret}(v)(\mu) & = & \langle v, \mu \rangle_{\checkmark} \\ \mathit{apply}(d, \mathsf{a}) & = & d \gg\!\beta\!= \lambda(\mathrm{Fun}(f)).\ \mathsf{L}\ f(\mathsf{a}) \\ \mathit{select}(d, \mathit{alts}) & = & d \gg\!\beta\!= \lambda(\mathrm{Con}(K_s, \overline{\mathsf{a}})).\ \mathit{alts}(K_s, \overline{\mathsf{a}}) \quad \text{where } (K_s, \overline{\mathsf{a}}) \in \mathrm{dom}(\mathit{alts}) \\ \mathit{memo}(\mathsf{a}, d) & = & d \gg\!\beta\!= \lambda v.\ (\lambda \mu.\ \mathsf{L}\ \langle v, \mu[\mathsf{a} \mapsto \mathit{memo}(\mathsf{a}, \mathit{ret}(v))] \rangle_{\checkmark}) \end{array}$$

$$\mathcal{S}_{\mathsf{L}}[\![\_]\!] : \mathrm{Exp} \to (\mathrm{Var} \rightharpoonup \mathrm{Addr}) \to \mathbb{D}^{\mathsf{L}}$$

$$\mathcal{S}_{\mathsf{L}}[\![\mathsf{x}]\!]_\rho(\mu) = \begin{cases} \mathsf{L}\ \mu(\rho(\mathsf{x}))(\mu) & \mathsf{x} \in \mathrm{dom}(\rho) \\ \lightning & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{\mathsf{L}}[\![\bar{\lambda}\mathsf{x}.\mathsf{e}]\!]_\rho = \mathit{ret}(\mathrm{Fun}(\lambda \mathsf{a}.\ \mathcal{S}_{\mathsf{L}}[\![\mathsf{e}]\!]_{\rho[\mathsf{x} \mapsto \mathsf{a}]}))$$

$$\mathcal{S}_{\mathsf{L}}[\![\mathsf{e}\ \mathsf{x}]\!]_\rho(\mu) = \begin{cases} \mathsf{L}\ \mathit{apply}(\mathcal{S}_{\mathsf{L}}[\![\mathsf{e}]\!]_\rho, \mathsf{a})(\mu) & \mathsf{x} \in \mathrm{dom}(\rho) \\ \lightning & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{\mathsf{L}}[\![\textbf{let } \mathsf{x} = \mathsf{e}_1 \textbf{ in } \mathsf{e}_2]\!]_\rho(\mu) = \begin{array}{l} \text{let } \rho' = \rho[\mathsf{x} \mapsto \mathsf{a}] \quad \text{where } \mathsf{a} \notin \mathrm{dom}(\mu) \\ \text{in } \mathsf{L}\ \mathcal{S}_{\mathsf{L}}[\![\mathsf{e}_2]\!]_{\rho'}(\mu[\mathsf{a} \mapsto \mathit{memo}(\mathsf{a}, \mathcal{S}_{\mathsf{L}}[\![\mathsf{e}_1]\!]_{\rho'})]) \end{array}$$

$$\mathcal{S}_{\mathsf{L}}[\![K\ \overline{\mathsf{x}}]\!]_\rho(\mu) = \begin{cases} \mathit{ret}(\mathrm{Con}(K, \overline{\rho(\mathsf{x})}))(\mu) & \overline{\mathsf{x} \in \mathrm{dom}(\rho)} \\ \lightning & \text{otherwise} \end{cases}$$

$$\mathcal{S}_{\mathsf{L}}[\![\textbf{case } \mathsf{e}_s \textbf{ of } \overline{K\ \overline{\mathsf{x}} \to \mathsf{e}_r}]\!]_\rho(\mu) = \begin{array}{l} \text{let } \mathit{alts} = \lambda(K_i, \overline{\mathsf{a}}).\ \mathsf{L}\ \mathcal{S}_{\varepsilon}[\![\mathsf{e}_{r_i}]\!]_{\rho[\overline{\mathsf{x}_i \mapsto \mathsf{a}}]} \\ \text{in } \mathsf{L}\ \mathit{select}(\mathcal{S}_{\varepsilon}[\![\mathsf{e}_s]\!]_\rho, \mathit{alts})(\mu) \end{array}$$

Fig. 3. Call-by-need Denotational Semantics $\mathcal{S}_{\mathsf{L}}[\![-]\!]$

Let us now understand $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$ by way of evaluating the example program from earlier, $\mathsf{e} \triangleq \textbf{let } i = \bar{\lambda}x.x \textbf{ in } i\ i$:

where
$\rho_1 = [i \mapsto \mathsf{a}_1]$
$\rho_2 = \rho_1[x \mapsto \mathsf{a}_1]$
$\mu = [\mathsf{a}_1 \mapsto \mathit{memo}(\mathsf{a}_1, \mathcal{S}_{\mathsf{L}}[\![\bar{\lambda}x.x]\!]_{\rho_1})]$
$f = \mathsf{a} \mapsto \mathcal{S}_{\mathsf{L}}[\![\mathsf{x}]\!]_{\rho_1[\mathsf{x} \mapsto \mathsf{a}]}$

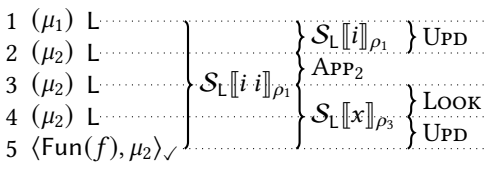
The annotations to the right of the trace can be understood as denoting the "call graph" of $\mathcal{S}_{\mathsf{L}}[\![\mathsf{e}]\!]_{[]}$, with the corresponding LK transitions as leaves. Evaluation begins at timestamp 1 with a Bind transition to timestamp 2, which $\mathcal{S}_{\mathsf{L}}[\![\mathsf{e}]\!]_{[]}$ acknowledges by emitting an L. A fresh address a_1 is allocated for variable i and the heap is extended with $memo(\mathsf{a}, \mathcal{S}_{\mathsf{L}}[\![\bar{\lambda}x.x]\!]_{\rho_1})$. It is interesting to realise that this process does not involve a fixpoint despite the recursive semantics of **let**. Of course, the self-application in $\mathcal{S}_{\mathsf{L}}[\![\mathsf{x}]\!]$ does the job just as well, as we will see in a moment.

Evaluation recurses into the body $\mathcal{S}_{\mathsf{L}}[\![i\ i]\!]_{\rho_1}$, yielding another App$_1$ transition (corresponding to a bland L) into $\mathcal{S}_{\mathsf{L}}[\![i]\!]_{\rho_1}$. Note that the target state $\mathcal{S}_{\mathsf{L}}[\![i]\!]_{\rho_1}$ will later be fed (via $\gg\beta=$) into the anonymous function in *apply*. This scheme is quite common: Continuation items of the transition semantics ("data") are reflected into the call stack of the trace semantics ("code"). The reverse process can be recognised as defunctionalisation [Reynolds 1972].

$\mathcal{S}_{\mathsf{L}}[\![i]\!]_{\rho_1}$ guides the trace through a heap lookup: After yielding an L corresponding to the Look transition, μ is dereferenced at $\rho(i)$ where we find $memo(\mathsf{a}, \mathcal{S}_{\mathsf{L}}[\![\bar{\lambda}x.x]\!]_{\rho_1})$. This heap entry is "self-applied" to μ, effectively "tying the knot". The memoisation action will run $\mathcal{S}_{\mathsf{L}}[\![\bar{\lambda}x.x]\!]_{\rho_1}$ to completeness through $(\gg\beta=)$ on heap μ at timestamp 4. Since that is already a value, $(\gg\beta=)$ calls its second argument with $\langle \mathsf{Fun}(f), \mu\rangle_{\checkmark}$ and we witness for the first time a reduction operation, making an Upd transition to update $\mu(\mathsf{a})$. Note that this update has no effect on the heap μ because $ret(\mathsf{Fun}(f))$ is precisely the same as $\mathcal{S}_{\mathsf{L}}[\![\bar{\lambda}x.x]\!]_{\rho_1}$.

After the heap update, we leave $\mathcal{S}_{\mathsf{L}}[\![i]\!]$ in timestamp 5, where $\gg\beta=$ yields to the anonymous function in *apply* (from the earlier call to $\mathcal{S}_{\mathsf{L}}[\![i\ i]\!]_{\rho_1}$), yielding another App$_2$ reduction and giving control to $f(\rho_1(i)) = \mathcal{S}_{\mathsf{L}}[\![x]\!]_{\rho_1[x\mapsto\rho_1(i)]}$. Since x is an alias for i, steps 6 to 8 just repeat the same same heap update sequence we observed in steps 3 to 5, concluding the example.

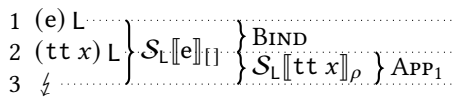
It is useful to review another example involving an observable heap update. The following trace begins right before the heap update occurs in **let** $i = (\bar{\lambda}y.\bar{\lambda}x.x)\ i$ **in** $i\ i$, that is, after reaching the value in $\mathcal{S}_{\mathsf{L}}[\![(\bar{\lambda}y.\bar{\lambda}x.x)\ i]\!]_{\rho_1}$. We will indicate the heap before a transition takes place by writing it before L:



where
$\rho_1 = [i \mapsto \mathsf{a}_1]$
$\rho_2 = [i \mapsto \mathsf{a}_1, y \mapsto \mathsf{a}_1]$
$\rho_3 = [i \mapsto \mathsf{a}_1, y \mapsto \mathsf{a}_1, x \mapsto \mathsf{a}_1]$
$\mu_1 = [\mathsf{a}_1 \mapsto memo(\mathsf{a}_1, \mathcal{S}_{\mathsf{L}}[\![(\bar{\lambda}y.\bar{\lambda}x.x)\ i]\!]_{\rho_1})]$
$\mu_2 = [\mathsf{a}_1 \mapsto memo(\mathsf{a}_1, \mathcal{S}_{\mathsf{L}}[\![\bar{\lambda}x.x]\!]_{\rho_2})]$
$f = \lambda d.\ \mathcal{S}_{\mathsf{L}}[\![\mathsf{x}]\!]_{\rho_2[\mathsf{x}\mapsto d]}$

Note that both the denotation in the heap μ_1 *and* its environment are updated in timestamp 2, and that the new denotation is immediately visible on the next heap lookup in timestamp 3, so that $\mathcal{S}_{\mathsf{L}}[\![\bar{\lambda}x.x]\!]_{\rho_2}$ takes control rather than $\mathcal{S}_{\mathsf{L}}[\![(\bar{\lambda}y.\bar{\lambda}x.x)\ i]\!]_{\rho_1}$, just as the transition system requires.

The handling of data types and case expressions is routine (if a bit syntactically heavy) and not so different to denotational semantics in call-by-name or call-by-value. It is a useful inclusion because it allows us to observe type errors other than scoping errors, as well as enable more interesting examples. Let us consider evaluation of the closed expression $\mathsf{e} \triangleq$ **let** $x = \mathtt{tt}$ **in** $\mathtt{tt}\ x$. $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$ makes it is easy to observe that the trace gets stuck (indicating the control expression in front of every L):



where
$\rho = [x \mapsto \mathsf{a}_1]$
$\mu = [\mathsf{a}_1 \mapsto \mathcal{S}_{\mathsf{L}}[\![\mathtt{tt}]\!]_{\rho}]$

Crucially, $\gg\beta=$ is equipped to propagate $\lightning$ up the call stack (through potential Upd transitions, in particular), similar to Milner's **wrong**.

Diverging traces hold no new surprises, other than they are observably different to stuck traces.

3.5 Maximal LK Traces

It turns out that the traces $\mathcal{S}_{\mathsf{L}}[\![\mathsf{e}]\!]$ generates correspond to *maximal* traces in the LK transition system. Let us make precise what that means.

A transition system is characterised by the set of *traces* it generates. An *LK trace* is a trace in $(\hookrightarrow)$, i.e., a non-empty and potentially infinite sequence of LK states $(\sigma_i)_{i\in\overline{n}}$ (where $\overline{n} = \{m \in \mathbb{N}_+ \mid m \leq n\}$ when $n \in \mathbb{N}$, $\overline{\omega} = \mathbb{N}$), such that $\sigma_i \hookrightarrow \sigma_{i+1}$ for $i, (i+1) \in \overline{n}$.

The *source* state σ_0 exists for finite and infinite traces, while the *target* state σ_n is only defined when $n \neq \omega$ is finite.

For proofs, we will often regard $(\sigma_i)_{i\in\overline{n}}$ as an object of type $\mathbb{S}^\infty \triangleq \exists n \in \mathbb{N}_\omega.\ \overline{n} \to \mathbb{S}$, where $\mathbb{N}_\omega$ is defined by guarded recursion as $\mathbb{N}_\omega = \{Z\} + \blacktriangleright\mathbb{N}_\omega$. The constructor for the right sum alternative is written *succ*. Now $\mathbb{N}_\omega$ contains all natural numbers (where n is encoded as $(succ)^n(Z)$) and the transfinite limit ordinal $\omega = succ(succ(\ldots))$. We will write $1 + m$ to denote $succ(m)$ (a different kind of $+$ than in the recursive equation for $\mathbb{N}_\omega$), just as we are used to from $\mathbb{N}$. Hence, when $(\sigma_i)_{i\in\overline{n}} \in \mathbb{S}^\infty$ is an LK trace and $n > 0$, then $(\sigma_{i+1})_{i\in\overline{n-1}} \in \blacktriangleright\mathbb{S}^\infty$ is the guarded tail of the trace with an associated induction principle.

An important kind of trace is one that never leaves the evaluation context of its source state:

Definition 6 (Deep, interior and balanced traces)**.** *An LK trace $(\sigma_i)_{i\in\overline{n}}$ is* κ-deep *if every intermediate continuation $\kappa_i = cont(\sigma_i)$ extends κ (so $\kappa_i = \kappa$ or $\kappa_i = \ldots \cdot \kappa$, abbreviated $\kappa_i = \ldots\kappa$).*

A trace $(\sigma_i)_{i\in\overline{n}}$ is called interior *if it is $cont(\sigma_0)$-deep. Furthermore, an interior trace $(\sigma_i)_{i\in\overline{n}}$ is* balanced *[Sestoft 1997] if the target state exists and is a return state with continuation $cont(\sigma_0)$.*

We notate κ-deep, interior and balanced traces as κ deep $(\sigma_i)_{i\in\overline{n}}$*,* $(\sigma_i)_{i\in\overline{n}}$ inter *and* $(\sigma_i)_{i\in\overline{n}}$ bal*, respectively.*

Example 7. *Let $\rho = [x \mapsto \mathsf{a}_1], \mu = [\mathsf{a}_1 \mapsto ([], \bar{\lambda}y.y)]$ and κ an arbitrary continuation. The trace*

$$(x, \rho, \mu, \kappa) \hookrightarrow (\bar{\lambda}y.y, \rho, \mu, \mathbf{upd}(\mathsf{a}_1) \cdot \kappa) \hookrightarrow (\bar{\lambda}y.y, \rho, \mu, \mathbf{upd}(\mathsf{a}_1) \cdot \kappa) \hookrightarrow (\bar{\lambda}y.y, \rho, \mu, \kappa)$$

is interior and balanced. Its prefixes are interior but not balanced. The trace suffix

$$(\bar{\lambda}y.y, \rho, \mu, \mathbf{upd}(\mathsf{a}_1) \cdot \kappa) \hookrightarrow (\bar{\lambda}y.y, \rho, \mu, \kappa)$$

is neither interior nor balanced.

We will say that the transition rules Look, App$_1$, Case$_1$ and Bind are interior, because the lifting into a trace is, whereas the returning transitions Upd, App$_2$ and Case$_2$ are not.

A balanced trace starting at a focus expression e and ending with v loosely corresponds to a derivation of $\mathsf{e} \Downarrow \mathsf{v}$ in a natural big-step semantics [Sestoft 1997] or a non-$\bot$ result in a denotational semantics.

It is when a derivation in a natural semantics does not exist that a small-step semantics shows finesse, in that it differentiates two different kinds of *maximally interior* (or, just *maximal*) traces:

Definition 8 (Maximal trace)**.** *An LK trace $(\sigma_i)_{i\in\overline{n}}$ is* maximal *if and only if it is interior and there is no σ_{n+1} such that $(\sigma_i)_{i\in\overline{n+1}}$ is interior. More formally (and without a negative occurrence of "interior"),*

$$(\sigma_i)_{i\in\overline{n}}\ \mathsf{max} \triangleq (\sigma_i)_{i\in\overline{n}}\ \mathsf{inter} \wedge (\nexists \sigma_{n+1}.\ \sigma_n \hookrightarrow \sigma_{n+1} \wedge cont(\sigma_{n+1}) = \ldots cont(\sigma_0))$$

We notate maximal traces as $(\sigma_i)_{i\in\overline{n}}$ max*.*

We call infinite and interior traces *diverging*. A maximally finite, but unbalanced trace is called *stuck*. Note that usually stuckness is associated with a state of a transition system rather than a trace. That is not possible in our framework; the following example clarifies.

Example 9 (Stuck and diverging traces). *Consider the interior trace*

$$(\mathtt{tt}\ x, [x \mapsto \mathsf{a}_1], [\mathsf{a}_1 \mapsto \ldots], \kappa) \hookrightarrow (\mathtt{tt}, [x \mapsto \mathsf{a}_1], [\mathsf{a}_1 \mapsto \ldots], \mathbf{ap}(\mathsf{a}_1) \cdot \kappa)$$

It is stuck, but its singleton suffix is balanced. An example for a diverging trace where $\rho = [x \mapsto \mathsf{a}_1]$ *and* $\mu = [\mathsf{a}_1 \mapsto (\rho, x)]$ *is*

$$(\mathbf{let}\ x = x\ \mathbf{in}\ x, [], [], \kappa) \hookrightarrow (x, \rho, \mu, \kappa) \hookrightarrow (x, \rho, \mu, \mathbf{upd}(\mathsf{a}_1) \cdot \kappa) \hookrightarrow \ldots$$

Lemma 10 (Characterisation of maximal traces). *An LK trace* $(\sigma_i)_{i \in \overline{n}}$ *is maximal if and only if it is balanced, diverging or stuck.*

Interiority guarantees that the particular initial stack κ of a maximal trace is irrelevant to execution, so maximal traces that differ only in the initial stack are bisimilar.

One class of maximal traces is of particular interest: The maximal trace starting in $inj(\mathsf{e})$! Whether it is infinite, stuck or balanced is the defining operational characteristic of e. If we can show that $\mathcal{S}_{\mathsf{L}}[\![\mathsf{e}]\!]$ distinguishes these behaviors of e, we have proven it an adequate replacement for the LK transition system.

3.6 Adequacy

Figure 4 shows the correctness predicate C in our endeavour to prove $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$ adequate. It builds on the framework of maximal LK traces established in the last section; specifically it encodes that an *abstraction* of every maximal LK trace can be recovered by running $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$ starting from the abstraction of an initial state.

The family of abstraction functions makes precise the intuitive connection between the semantic objects in $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$ and the syntactic objects in the transition system.

We will sometimes need to disambiguate the clashing definitions from Section 3 and Section 2. We do so by adorning semantic objects with a tilde, so $\tilde{\mu} \triangleq \alpha_{\mathbb{H}}(\mu)$ denotes a semantic heap which in this instance is defined to be the abstraction of a syntactic heap μ.

Note first that $\alpha^{\infty}_{\mathbb{S}}$ is defined by guarded recursion over the LK trace, in the sense defined in Section 3.5. As such, the expression $\{\!|\alpha_{\mathbb{S}^\infty}((\sigma_{i+1})_{i \in \overline{n-1}}, \kappa)|\!\}$ has type $\blacktriangleright \mathbb{T}^{\mathsf{L}}$ (the $\blacktriangleright$ in the type of $(\sigma_{i+1})_{i \in \overline{n-1}}$ maps through $\alpha^{\infty}_{\mathbb{S}}$ via the idiom brackets). Likewise, $=$ on $\mathbb{T}^{\mathsf{L}}$ is defined in the obvious structural way by guarded recursion (as it would be if it was a finite, inductive type).

Our first goal is to establish a few auxiliary lemmas showing what kind of properties of LK traces are preserved by $\alpha^{\infty}_{\mathbb{S}}$ and in which way. Let us warm up by defining a length function on traces:

Definition 11 (Length of a trace). *The* length $len : \mathbb{T}^{\mathsf{L}} \to \mathbb{N}_\omega$ *of a trace is defined by guarded recursion*

$$len(\tau) = \begin{cases} 1 + \mathsf{next}\ len \circledast \tau^{\blacktriangleright} & \tau = \mathsf{L}\ \tau^{\blacktriangleright} \\ 0 & otherwise \end{cases}$$

Lemma 12 (Preservation of length). *Let* $(\sigma_i)_{i \in \overline{n}}$ *be an arbitrary trace. Then* $len(\alpha^{\infty}_{\mathbb{S}}((\sigma_i)_{i \in \overline{n}}, cont(\sigma_0))) = n$.

Lemma 13 (Preservation of characteristic). *Let* $(\sigma_i)_{i \in \overline{n}}$ *be a maximal trace. Then* $\alpha^{\infty}_{\mathbb{S}}((\sigma_i)_{i \in \overline{n}}, cont(\sigma_0))$ *is ...*

- *... infinite if and only if* $(\sigma_i)_{i \in \overline{n}}$ *is diverging*
- *... ending with* $\langle \_, \_ \rangle_{\checkmark}$ *if and only if* $(\sigma_i)_{i \in \overline{n}}$ *is balanced*
- *... ending with* $\lightning$ *if and only if* $(\sigma_i)_{i \in \overline{n}}$ *is stuck*

$$
\begin{array}{rcl}
\alpha_{\mathbb{H}}([\overline{\mathsf{a} \mapsto (\rho, \mathsf{e})}]) & = & [\overline{\mathsf{a} \mapsto \mathit{memo}(\mathsf{a}, \mathcal{S}_{\mathsf{L}}[\![\mathsf{e}]\!]_{\rho})}] \\
\alpha_{\mathbb{V}^{\mathsf{L}}}(\bar{\lambda}\mathsf{x}.\mathsf{e}, \rho, \mu, \kappa) & = & \langle \mathsf{Fun}(\lambda \mathsf{a}.\, \mathcal{S}_{\mathsf{L}}[\![\mathsf{e}]\!]_{\rho[\mathsf{x} \mapsto \mathsf{a}]}), \alpha_{\mathbb{H}}(\mu) \rangle_{\checkmark} \\
\alpha_{\mathbb{V}^{\mathsf{L}}}(K\ \overline{\mathsf{x}}, \rho, \mu, \kappa) & = & \langle \mathsf{Con}(K, \overline{\rho(\mathsf{a})}), \alpha_{\mathbb{H}}(\mu) \rangle_{\checkmark} \\
\alpha_{\mathbb{S}^{\infty}}((\sigma_i)_{i \in \overline{n}}, \kappa) & = & \begin{cases} \mathsf{L}\ \{\!|\alpha_{\mathbb{S}^{\infty}}((\sigma_{i+1})_{i \in \overline{n-1}}, \kappa)|\!\} & n > 0 \\ \alpha_{\mathbb{V}^{\mathsf{L}}}(\sigma_0) & \mathit{ctrl}(\sigma_0)\ \text{value}\ \wedge\ \mathit{cont}(\sigma_0) = \kappa \\ \lightning & \text{otherwise} \end{cases} \\
C((\sigma_i)_{i \in \overline{n}}) & = & (\sigma_i)_{i \in \overline{n}}\ \mathsf{max} \Longrightarrow \forall((\mathsf{e}, \rho, \mu, \kappa) = \sigma_0).\ \alpha_{\mathbb{S}^{\infty}}((\sigma_i)_{i \in \overline{n}}, \kappa) = \mathcal{S}_{\mathsf{L}}[\![\mathsf{e}]\!]_{\rho}(\alpha_{\mathbb{H}}(\mu))
\end{array}
$$

Fig. 4. Correctness predicate for $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$

The previous lemma is interesting as it allows us to apply the classifying terminology of interior traces to a τ that is an abstraction of a *maximal* LK trace. For such a maximal τ we will say that it is balanced when it ends with $\langle \_, \_ \rangle_{\checkmark}$, stuck if ending in $\lightning$ and diverging if infinite.

We are now ready to prove the main correctness predicate:

Theorem 14 (Correctness of $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$)**.** *$C$ from Figure 4 holds. That is, whenever $(\sigma_i)_{i \in \overline{n}}$ is a maximal LK trace with source state $(\mathsf{e}, \rho, \mu, \kappa)$, we have $\alpha_{\mathbb{S}^{\infty}}((\sigma_i)_{i \in \overline{n}}, \kappa) = \mathcal{S}_{\mathsf{L}}[\![\mathsf{e}]\!]_{\rho}(\alpha_{\mathbb{H}}(\mu))$.*

Theorem 16 and Lemma 13 are the key to proving a strong version of adequacy for $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$, where σ is defined to be a *final* state if $\mathit{ctrl}(\sigma)$ is a value and $\mathit{cont}(\sigma) = \mathbf{stop}$.

Theorem 15 (Total adequacy of $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$)**.** *Let $\tau = \mathcal{S}_{\mathsf{L}}[\![\mathsf{e}]\!]_{[]}([])$.*

- *τ ends with $\langle \_, \_ \rangle_{\checkmark}$ (is balanced) iff there exists a final state σ such that $\mathit{inj}(\mathsf{e}) \hookrightarrow^* \sigma$.*
- *τ ends with $\lightning$ (is stuck) iff there exists a non-final state σ such that $\mathit{inj}(\mathsf{e}) \hookrightarrow^* \sigma$ and there exists no σ' such that $\sigma \hookrightarrow \sigma'$.*
- *τ is infinite (is diverging) iff for all σ with $\mathit{inj}(\mathsf{e}) \hookrightarrow^* \sigma$ there exists σ' with $\sigma \hookrightarrow \sigma'$.*

3.7 Discussion

We can already give perspective on two of the goals we set ourselves in Section 2: As the domain $\mathbb{D}^{\mathsf{L}}$ of our semantics $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$ is defined by guarded recursion, the approximation order between elements of the domain is discrete and all elements are total. $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$ is formulated entirely within this internal language and as such, looping programs are denoted by total elements as well, fullfilling Goal 2.

4 EVENTFUL SEMANTICS

In the previous section, we gave a trace-based call-by-need semantics and proved it adequate wrt. the LK transition semantics. With compositionality and structural induction we recover strong advantages of denotational semantics.

It seems that Theorem 15 is at odds with the fact that the information encoded in the generated traces is by far not enough to recover the LK transition system in the sense of Cousot [2021, Chapter 43]. It is however easily possible to "elaborate" $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$ to include the proper intermediate state. We think it is rather uninteresting to give the closed, elaborated definition of $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$. For one, such a definition would necessarily give up on simplicity. Furthermore, for our usage analysis in Section 2, we do not care so much about the *state* in which variable lookup happens, but rather about at which *address* the Look transition happened.

In this section, we will embellish the generated traces to track the instantiations of the transition rules taken, calling those instantiations *events*. The resulting *eventful trace semantics* $\mathcal{S}_{\varepsilon}[\![\_]\!]$ is (closer to) the preferred framework of Cousot [2021].[7]

[7] Cousot's *stateless* semantics even goes one step further and rematerialises the heap as needed from the history of the trace.

$$
\begin{array}{rcl}
\text{Event } \varepsilon \in \mathbb{E}_\mathbb{V} & ::= & \mathsf{bind}(\mathsf{x}, \mathsf{a} \mapsto d) \mid \mathsf{look}(\mathsf{a}) \mid \mathsf{upd}(\mathsf{a} \mapsto v) \\
& \mid & \mathsf{app}_1(\mathsf{a}) \mid \mathsf{app}_2(\mathsf{x} \mapsto \mathsf{a}) \mid \mathsf{case}_1(d) \mid \mathsf{case}_2(K, \overline{\mathsf{x} \mapsto \mathsf{a}}) \\
\text{Heap } \mu \in \mathbb{H} & = & \mathsf{Addr} \rightharpoonup \blacktriangleright\mathbb{D}^\varepsilon \qquad \text{Delayed event } \varepsilon^\blacktriangleright \in \blacktriangleright\mathbb{E}_\mathbb{V} \\
\text{Program trace } \tau \in \mathbb{T}^\varepsilon & ::= & \langle v, \mu \rangle_\checkmark \mid \lightning \mid \varepsilon^\blacktriangleright :: \tau^\blacktriangleright \qquad \text{Delayed trace } \tau^\blacktriangleright \in \blacktriangleright\mathbb{T}^\varepsilon \\
\text{Eventful domain } d \in \mathbb{D}^\varepsilon & = & \mathbb{H} \to \mathbb{T}^\varepsilon \qquad \text{Delayed element } d^\blacktriangleright \in \blacktriangleright\mathbb{D}^\varepsilon \\
\text{Eventful value } v \in \mathbb{V}^\varepsilon & ::= & \mathsf{Fun}(f \in \mathsf{Addr} \to \mathbb{D}^\varepsilon) \mid \mathsf{Con}(K, \overline{\mathsf{a}}^{\alpha_K})
\end{array}
$$

$$
\boxed{(\gg\!\beta\!=) : \mathbb{D}^\varepsilon \to (\mathbb{V}^\varepsilon \rightharpoonup \mathbb{D}^\varepsilon) \to \mathbb{D}^\varepsilon \quad \mathit{ret} : \mathbb{V}^\varepsilon \to \mathbb{D}^\varepsilon \quad \mathit{apply} : \mathbb{D}^\varepsilon \times \mathsf{Addr} \to \mathbb{D}^\varepsilon \quad \mathit{memo} : \mathsf{Addr} \times \mathbb{D}^\varepsilon \to \mathbb{D}^\varepsilon \quad \mathit{select} : \mathbb{D}^\varepsilon \times ((K : \mathsf{Con}) \times \mathsf{a}^{\alpha_K} \rightharpoonup \mathbb{D}^\varepsilon) \to \mathbb{D}^\varepsilon}
$$

$$
(d \gg\!\beta\!= f)(\mu) = \begin{cases} \overline{\varepsilon ::} f(v)(\mu') & d(\mu) = \overline{\varepsilon ::} \langle v, \mu' \rangle_\checkmark \text{ and } v \in \mathsf{dom}(f) \\ \overline{\varepsilon ::} \lightning & d(\mu) = \overline{\varepsilon ::} \langle v, \mu' \rangle_\checkmark \text{ and } v \notin \mathsf{dom}(f) \\ d(\mu) & \text{otherwise} \end{cases}
$$

$$
\begin{array}{rcl}
\mathit{ret}(v)(\mu) & = & \langle v, \mu \rangle_\checkmark \\
\mathit{apply}(d, \mathsf{a}) & = & d \gg\!\beta\!= \lambda(\mathsf{Fun}(f)).\ f(\mathsf{a}) \\
\mathit{select}(d, \mathit{alts}) & = & d \gg\!\beta\!= \lambda(\mathsf{Con}(K_s, \overline{d_s^\blacktriangleright})).\ \mathit{alts}(K_s, \overline{d_s^\blacktriangleright}) \quad \text{where } (K_s, \overline{d_s^\blacktriangleright}) \in \mathsf{dom}(\mathit{alts}) \\
\mathit{memo}(\mathsf{a}, d) & = & d \gg\!\beta\!= \lambda v.\ (\lambda \mu'.\ \mathsf{upd}(\mathsf{a} \mapsto v) :: \langle v, \mu'[\mathsf{a} \mapsto \mathit{memo}(\mathsf{a}, \mathit{ret}(v))] \rangle_\checkmark)
\end{array}
$$

$$
\boxed{\mathcal{S}_\varepsilon[\![\_]\!] : \mathsf{Exp} \to (\mathsf{Var} \rightharpoonup \mathsf{Addr}) \to \mathbb{D}^\varepsilon}
$$

$$
\begin{array}{rcl}
\mathcal{S}_\varepsilon[\![\mathsf{x}]\!]_\rho(\mu) & = & \begin{cases} \mathsf{look}(\mathsf{a}) :: \mu(\rho(\mathsf{x}))(\mu) & \mathsf{x} \in \mathsf{dom}(\rho) \\ \lightning & \text{otherwise} \end{cases} \\
\mathcal{S}_\varepsilon[\![\bar{\lambda}\mathsf{x}.\mathsf{e}]\!]_\rho & = & \mathit{ret}(\mathsf{Fun}(\lambda \mathsf{a}.\ \mathsf{app}_2(\mathsf{x} \mapsto \mathsf{a}) :: \mathcal{S}_\varepsilon[\![\mathsf{e}]\!]_{\rho[\mathsf{x} \mapsto \mathsf{a}]})) \\
\mathcal{S}_\varepsilon[\![\mathsf{e}\ \mathsf{x}]\!]_\rho(\mu) & = & \begin{cases} \mathsf{app}_1(\rho(x)) :: \mathit{apply}(\mathcal{S}_\varepsilon[\![\mathsf{e}]\!]_\rho, \rho(\mathsf{x}))(\mu) & \mathsf{x} \in \mathsf{dom}(\rho) \\ \lightning & \text{otherwise} \end{cases} \\
\mathcal{S}_\varepsilon[\![\mathbf{let}\ \mathsf{x} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2]\!]_\rho(\mu) & = & \begin{array}{l} \mathbf{let}\ \ \rho' = \rho[\mathsf{x} \mapsto \mathsf{a}] \quad \text{where } \mathsf{a} \notin \mathsf{dom}(\mu) \\ \quad\ \ d_1^\blacktriangleright = \mathcal{S}_\varepsilon[\![\mathsf{e}_1]\!]_{\rho'} \\ \mathbf{in}\ \ \mathsf{bind}(\mathsf{x}, \mathsf{a} \mapsto d_1^\blacktriangleright) :: \mathcal{S}_\varepsilon[\![\mathsf{e}_2]\!]_{\rho'}(\mu[\mathsf{a} \mapsto \mathit{memo}(\mathsf{a}, d_1^\blacktriangleright)]) \end{array} \\
\mathcal{S}_\varepsilon[\![K\ \overline{\mathsf{x}}]\!]_\rho(\mu) & = & \begin{cases} \mathit{ret}(\mathsf{Con}(K, \overline{\rho(\mathsf{x})}))(\mu) & \overline{\mathsf{x} \in \mathsf{dom}(\rho)} \\ \lightning & \text{otherwise} \end{cases} \\
\mathcal{S}_\varepsilon[\![\mathbf{case}\ \mathsf{e}_s\ \mathbf{of}\ \overline{K\ \overline{\mathsf{x}} \to \mathsf{e}_r}]\!]_\rho(\mu) & = & \begin{array}{l} \mathbf{let}\ \ \mathit{alts} = \lambda(K_i, \overline{\mathsf{a}}).\ \mathsf{case}_2(K_i, \overline{\mathsf{x}_i \mapsto \mathsf{a}}) :: \mathcal{S}_\varepsilon[\![\mathsf{e}_{r_i}]\!]_{\rho[\overline{\mathsf{x}_i \mapsto \mathsf{a}}]} \\ \mathbf{in}\ \ \mathsf{case}_1(\mathcal{S}_\varepsilon[\![\mathsf{e}_s]\!]_\rho) :: \mathit{select}(\mathcal{S}_\varepsilon[\![\mathsf{e}_s]\!]_\rho, \mathit{alts})(\mu) \end{array}
\end{array}
$$

Fig. 5. Eventful Trace Semantics

4.1 Favouring Events over State

Figure 5 gives the definition for the eventful trace semantics $\mathcal{S}_\varepsilon[\![\_]\!]$. The domain of eventful traces $\mathbb{D}^\varepsilon$ maps a heap to a possibly infinite or stuck program trace τ, just as the vanilla semantics $\mathcal{S}_\mathsf{L}[\![\_]\!]$.

Upon a transition of the internal machine state, our new semantics emits an *event* ε with the cons :: constructor, rather than just delaying with an uninformative L. In fact, each event carries with it part of the "closure" of the corresponding LK transition rule, which is readily available for analysis. For example, the look event carries the heap address which is accessed, bind events carry the let-bound variable, its fresh address and the denotation of the right-hand side. We thus

open up our semantics for instrumentation. Certainly, we could have omitted some or added more information than what the events carry; in this work we mostly care about look events.

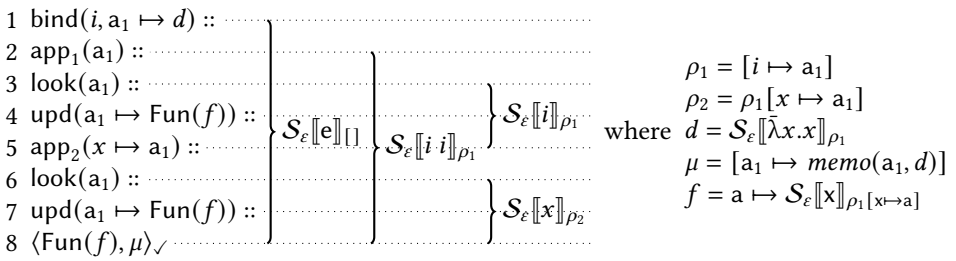
There is one minor structural difference to $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$: Fun values now return an unguarded $\mathbb{D}^{\varepsilon}$ so that the app_2 event can carry the lambda-bound variable of the lambda expression from whence it came. This unguarded (positive) occurrence is not prohibited by the type construction rules of guarded domain theory, but Møgelberg and Veltri [2019, Section 5.2] suggests that we could work around this issue by defining $\mathbb{D}^{\varepsilon}$ as a simultaneous guarded recursive and inductive type, i.e., $\mathbb{D}^{\varepsilon} = \mathsf{fix}\, X.\ \mathsf{lfp}\, Y.\ \ldots \mathsf{Fun}(f \in X \to Y)\ldots$, where fix is the guarded fixpoint operator and lfp is the least fixed-point operator.

All results connecting $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$ to the LK transition semantics also hold for $\mathcal{S}_{\varepsilon}[\![\_]\!]$, since $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$ can be recovered by throwing away all events and replacing :: with L.

Thus the only thing worth verifying is that the events actually correspond to meaningful LK concepts. One can read off this correpondence in the abstraction function for the extended correctness predicate for $\mathcal{S}_{\varepsilon}[\![\_]\!]$ which we give in **??** in the Appendix.

Theorem 16 (Correctness of $\mathcal{S}_{\varepsilon}[\![\_]\!]$). *C from* **??** *holds. That is, whenever* $(\sigma_i)_{i\in\overline{n}}$ *is a maximal LK trace with source state* $(\mathsf{e}, \rho, \mu, \kappa)$*, we have* $\alpha_{\mathbb{T}^{\varepsilon}}((\sigma_i)_{i\in\overline{n}}, \kappa) = \mathcal{S}_{\varepsilon}[\![\mathsf{e}]\!]_{\rho}(\alpha_{\mathbb{H}}(\mu))$.

It is clear that this style of semantics allows to observe arbitrary operational detail on an as-needed basis, thus we fulfill our Goal 3. Let us conclude this section by giving the embellished version of the earlier example from Section 3, **let** $i = \bar{\lambda}x.x$ **in** $i\ i$:



5 THE IMPERATIVE ESSENCE OF MEMOISATION

Before we can talk about abstraction, we have to record a few meta-theoretic properties. We will prove these properties in terms of the eventful semantics $\mathcal{S}_{\varepsilon}[\![\_]\!]$ but they apply to the $\mathcal{S}_{\mathsf{L}}[\![\_]\!]$ as well.

5.1 Heap Forcing and Laziness

As we have observed in Section 2, the semantics of call-by-need is more complicated than the semantics for call-by-name and call-by-value because it relies on a heap to implement thunk memoisation. This leads to a tricky domain model that passes around heaps as state and exposes tiresome operational detail such as heap addresses.

On the other hand, memoisation is a very "benign" use of state and often we can reason rather naturally about call-by-need programs without thinking too much about heaps, in contrast to a call-by-name calculus with arbitrary assignments to ref cells.

For example, we can observe that heaps in our semantics evolve in a very specific way: When $\mathcal{S}_{\varepsilon}[\![\mathsf{e}]\!]_{\rho}(\mu) = \ldots :: \langle v, \mu'\rangle_{\checkmark}$, we intuit that μ' must be "at least as evaluated" as μ, e.g., for all $\mathsf{a} \in \mathrm{dom}(\mu)$ either $\mu(\mathsf{a}) = \mu'(\mathsf{a})$ or $\mu'(\mathsf{a})$ is "the value of" $\mu(\mathsf{a})$. We make this observation precise via the following definitions:

Definition 17 (Big-step). *For all* $d \in \mathbb{D}^{\varepsilon}$, $v \in \mathbb{V}^{\varepsilon}$, $\mu, \mu' \in \mathbb{H}$ *we say that* d evaluates *in* μ *to* (v, μ') *(written* $\langle d, \mu\rangle \Downarrow \langle v, \mu'\rangle$*) if and only if* $d(\mu) = \ldots :: \langle v, \mu'\rangle_{\checkmark}$.

Big-step notation deliberately throws away much information about the trace, retaining only whether it was balanced and what the final value and heap is, hence the name "big-step" is fitting.

Definition 18 (Heap forcing). *For all $\mu_1, \mu_2 \in \mathbb{H}$ we say that μ_1* forces *to μ_2 (written $\mu_1 \rightsquigarrow \mu_2$) if and only if*

(1) $\mathrm{dom}(\mu_1) \subseteq \mathrm{dom}(\mu_2)$, *and*
(2) *For all* $\mathrm{a} \in \mathrm{dom}(\mu_1)$, *either*
   - $\mu_1(\mathrm{a}) = \mu_2(\mathrm{a})$, *or*
   - *there exists* v, μ_1' *such that* $\mu_2(\mathrm{a}) = \mathit{memo}(\mathrm{a}, \mathit{ret}(v))$ *and* $\langle \mu_1(\mathrm{a}), \mu_1 \rangle \Downarrow \langle v, \mu_1' \rangle$. *Additionally, if* $\mu_1 \neq \mu_1'$ *then* $\mu_1' \rightsquigarrow \mu_2$.

Our earlier intuition can now be formulated as follows: If $\langle \mathcal{S}_\varepsilon[\![\mathrm{e}]\!]_\rho, \mu \rangle \Downarrow \langle v, \mu' \rangle$, then $\mu \rightsquigarrow \mu'$.

Just as lambda terms are often identified modulo consistent renaming (α-equivalence), we have so far treated LK machine states σ equivalent modulo consistent *readdressing*, i.e., consistent substitution of all address values. Likewise, for the remainder of this work, we will identify trace-based *heaps* modulo consistent readdressing. Of course, a rigorous treatment would have to carry around readdressing substitutions and apply them to adjust the domain of a heap, its entries, as well as the heaps, domain elements and values in the returned traces.

Once a heap entry is evaluated, it stays that way during forcing:

Lemma 19. *For all $\mu_1, \mu_2, \mathrm{a}, v$ such that $\mu_1 \rightsquigarrow \mu_2$ and $\mu_1(\mathrm{a}) = \mathit{memo}(\mathrm{a}, \mathit{ret}(v))$, then $\mu_2(\mathrm{a}) = \mathit{memo}(\mathrm{a}, \mathit{ret}(v))$.*

The heap forcing relation is reflexive (it is always $\mu(\mathrm{a}) = \mu(\mathrm{a})$) and also antisymmetric:

Lemma 20 ($(\rightsquigarrow)$ is antisymmetric). *Let μ_1, μ_2 be heaps and $\mu_1 \rightsquigarrow \mu_2, \mu_2 \rightsquigarrow \mu_1$. Then $\mu_1 = \mu_2$.*

(It is worth stressing that antisymmetry only holds on equivalence classes modulo readdressing.) However, $(\rightsquigarrow)$ is not easily proven transitive without further characterisation of the domain elements returned by $\mathcal{S}_\varepsilon[\![\_]\!]$.

Definition 21 (Lazy domain). *An element $d \in \mathbb{D}^\varepsilon$ is* lazy *if and only if*

- (Forces) *For all lazy $\mu, \mu' \in \mathbb{H}, v \in \mathbb{V}^\varepsilon$ such that $\langle d, \mu \rangle \Downarrow \langle v, \mu' \rangle$, v is lazy and $\mu \rightsquigarrow \mu'$.*
- (Postpone) *For all lazy $\mu_1, \mu_2 \in \mathbb{H}$ such that $\mu_1 \rightsquigarrow \mu_2$,*
  - *If $\langle d, \mu_1 \rangle \Downarrow \langle v_1, \mu_1' \rangle$ (for some v_1, μ_1') then $\langle d, \mu_2 \rangle \Downarrow \langle v_2, \mu_2' \rangle$ (for some v_2, μ_2') and $v_1 = v_2$ as well as $\mu_1' \rightsquigarrow \mu_2'$*
  - *If $d(\mu_1)$ diverges then so does $d(\mu_2)$.*
- (Speculate) *For all lazy $\mu_1, \mu_2 \in \mathbb{H}$ such that $\mu_1 \rightsquigarrow \mu_2$ and $d(\mu_1)$ is defined,*
  - *If $\langle d, \mu_2 \rangle \Downarrow \langle v_2, \mu_2' \rangle$ (for some v_2, μ_2') then $\langle d, \mu_1 \rangle \Downarrow \langle v_1, \mu_1' \rangle$ (for some v_1, μ_1') and $v_1 = v_2$ as well as $\mu_1' \rightsquigarrow \mu_2'$*
  - *If $d(\mu_2)$ diverges then so does $d(\mu_1)$.*

A heap μ is lazy *if and only if it is of the form $[\overline{\mathrm{a} \mapsto \mathit{memo}(\mathrm{a}, d)}]$ and all $\overline{d}$ are lazy and defined on μ. A value v is* lazy *if and only if $v = \mathsf{Fun}(f)$ (for some f) implies that $f(\mathrm{a})$ is lazy.*

So when an element d is *lazy*, every input heap forces to the output heap, it evaluates to the same value whether or not the input heap is more or less forced, and $\langle d, \_ \rangle \Downarrow \langle v, \_ \rangle$ is natural (in the category theoretic sense) in that it transfers the forcedness relation from input heaps to output heaps. Diagramatically:

$$\begin{array}{ccc} \mu_1 & \rightsquigarrow & \mu_2 \\ {\scriptstyle \langle d,\_\rangle}\Big\Downarrow{\scriptstyle \langle v,\_\rangle} & & {\scriptstyle \langle d,\_\rangle}\Big\Downarrow{\scriptstyle \langle v,\_\rangle} \\ \mu_1' & \rightsquigarrow & \mu_2' \end{array}$$

$$\boxed{\mathsf{fin}(\tau) \qquad A \vdash d \qquad A \vdash \tau_1 \sim \tau_2 \qquad \mu_1 \approx \mu_2 \qquad d_1 \equiv d_2}$$

$$\frac{\tau = \overline{\ldots :: \tau'} \quad \tau' \neq \_ :: \_}{\mathsf{fin}(\tau)}\ \textsc{Fin} \qquad \frac{\forall \mu.\ A \subseteq \mathsf{dom}(\mu) \Longrightarrow d(\mu) \text{ is defined}}{A \vdash d}\ \textsc{Dom}$$

$$\frac{\blacktriangleright(A \vdash \tau_1 \sim \tau_2)}{A \vdash \_ :: \tau_1 \sim \_ :: \tau_2}\ \textsc{Eq-LR-::} \qquad \frac{A \vdash \tau_1 \sim \tau_2 \quad \mathsf{fin}(\tau_2)}{A \vdash \tau_1 \sim \_ :: \tau_2}\ \textsc{Eq-R-::} \qquad \frac{A \vdash \tau_1 \sim \tau_2 \quad \mathsf{fin}(\tau_1)}{A \vdash \_ :: \tau_1 \sim \tau_2}\ \textsc{Eq-L-::}$$

$$\frac{}{A \vdash \lightning \sim \lightning}\ \textsc{Eq-}\lightning \qquad \frac{\overline{A \vdash \mu_1(\mathsf{a}_1) \sim \mu_2(\mathsf{a}_2)}}{A \vdash \langle \mathsf{Con}(K, \overline{\mathsf{a}_1}), \mu_1 \rangle_\checkmark \sim \langle \mathsf{Con}(K, \overline{\mathsf{a}_2}), \mu_2 \rangle_\checkmark}\ \textsc{Eq-Con}$$

$$\frac{\forall \mathsf{a}.\ \mathsf{a} \in A \Longrightarrow A \vdash f_1(\mathsf{a})(\mu_1) \sim f_2(\mathsf{a})(\mu_2)}{A \vdash \langle \mathsf{Fun}(f_1), \mu_1 \rangle_\checkmark \sim \langle \mathsf{Fun}(f_2), \mu_2 \rangle_\checkmark}\ \textsc{Eq-Fun}$$

$$\frac{\mathsf{dom}(\mu_1) = \mathsf{dom}(\mu_2) \quad \forall \mathsf{a}.\ \blacktriangleright(\mathsf{dom}(\mu_1) \vdash \mu_1(\mathsf{a})(\mu_1) \sim \mu_2(\mathsf{a})(\mu_2))}{\mu_1 \approx \mu_2}\ \textsc{Eq-}\mathbb{H}$$

$$\frac{\forall \mu_1, \mu_2.\ \mu_1 \approx \mu_2 \wedge \mathsf{dom}(\mu_1) \vdash d_1, d_2 \Longrightarrow \mathsf{dom}(\mu_1) \vdash d_1(\mu_1) \sim d_2(\mu_2)}{d_1 \equiv d_2}\ \textsc{Eq-}\mathbb{D}^\varepsilon$$

Fig. 6. Semantic equivalence relation on lazy elements

Well-addressedness (the so far implicit assumption that $d(\mu)$ is defined) in conjunction with the Speculate property means that a lazy element may branch on the contents of the heap, but not on whether or not an entry in the heap is present.

The definitions for lazy heaps, environments and values are necessary by congruence. When we restrict ourselves to lazy heaps, we can prove transitivity of the forcing relation.

Lemma 22 ($(\rightsquigarrow)$ is transitive)**.** *Let μ_1, μ_2, μ_3 be lazy heaps and $\mu_1 \rightsquigarrow \mu_2, \mu_2 \rightsquigarrow \mu_3$. Then $\mu_1 \rightsquigarrow \mu_3$.*

Corollary 23. *$(\rightsquigarrow)$ is a partial order on lazy heaps.*

And now we finally prove that $\mathcal{S}_\varepsilon[\![\_]\!]$ indeed produces a lazy element:

Theorem 24. *For all environments ρ and expressions e, $\mathcal{S}_\varepsilon[\![\mathsf{e}]\!]_\rho$ is a lazy element.*

This result justifies our tacit assumption from now on that all elements d are lazy.

5.2 A Useful Program Equivalence

Consider the two expressions $\mathsf{e}_1 \triangleq \textbf{let}\ x = \mathtt{tt}\ \textbf{in}\ \mathtt{tt}$ and $\mathsf{e}_2 \triangleq \textbf{let}\ x = \mathtt{ff}\ \textbf{in}\ \mathtt{tt}$. For all intents and purposes, these two expressions are equivalent. Indeed, they equate under $\mathcal{S}_\bot[\![\_]\!]$, suggesting that x is dead according to Definition 1.

However, the definitional equality on the lazy domain $\mathbb{D}^\varepsilon$ is *too fine*:

$$\begin{array}{rcl} \mathcal{S}_\varepsilon[\![\mathsf{e}_1]\!]_{[]}([]) & = & \mathsf{bind}(x, \mathsf{a}_1 \mapsto \mathcal{S}_\varepsilon[\![\mathtt{tt}]\!]_\ldots) :: \langle \mathsf{Con}(\mathtt{tt}), [\mathsf{a}_1 \mapsto \mathcal{S}_\varepsilon[\![\mathtt{tt}]\!]_\ldots] \rangle_\checkmark \\ & \neq & \mathsf{bind}(x, \mathsf{a}_1 \mapsto \mathcal{S}_\varepsilon[\![\mathtt{ff}]\!]_\ldots) :: \langle \mathsf{Con}(\mathtt{tt}), [\mathsf{a}_1 \mapsto \mathcal{S}_\varepsilon[\![\mathtt{ff}]\!]_\ldots] \rangle_\checkmark \quad = \quad \mathcal{S}_\varepsilon[\![\mathsf{e}_2]\!]_{[]}([]) \end{array}$$

It would be reasonable to abstract away the contents of the trace to get a coarser equivalence, thus eliminating the difference in bind actions. However, that still leaves the difference in heap entries such as a_1 which will never be looked at! Since the heap entry for a_1 was introduced while

executing $\mathsf{e}_1/\mathsf{e}_2$, we will call them *local* to the trace $\mathcal{S}_\varepsilon[\![\mathsf{e}_i]\!]_{[]}([])$. It is prudent for the observer to ignore local entries.

That is the essence of what the *semantic equivalence relation* ($\equiv$) defined on lazy elements in Figure 6 does. It is coarse enough to equate the example above, but still finer than contextual equivalence, as we shall see.

The key is its observer model for function values, which necessitates the address space parameter A in $\sim$ to track non-local (and thus observable) addresses. Consider a successful evaluation $\langle d_i, \mu_i\rangle \Downarrow \langle \mathsf{Fun}(f_i), \mu_i'\rangle$, as in

$$\begin{array}{rcl}\mathcal{S}_\varepsilon[\![\mathbf{let}\ t = \mathtt{tt}\ \mathbf{in}\ \bar{\lambda} z.t]\!]_{[]}([]) & = & \ldots :: \langle \mathsf{Fun}(\lambda\_.\ \lambda\mu.\ \ldots\mu(\mathsf{a}_1)\ldots), [\mathsf{a}_1 \mapsto d_t]\rangle_\checkmark \\ & \equiv & \ldots :: \langle \mathsf{Fun}(\lambda\_.\ \lambda\mu.\ \ldots\mu(\mathsf{a}_2)\ldots), [\mathsf{a}_1 \mapsto d_f, \mathsf{a}_2 \mapsto d_t]\rangle_\checkmark \\ & = & \mathcal{S}_\varepsilon[\![\mathbf{let}\ f = \mathtt{ff}\ \mathbf{in}\ \mathbf{let}\ t = \mathtt{tt}\ \mathbf{in}\ \bar{\lambda} z.t]\!]_{[]}([])\end{array}$$

The denotations of the two expressions are not definitionally equivalent in the final heap, yet they are semantically equivalent. To see that, we have to relate the heap entries in a_1 and a_2, which only succeeds when the local entries in μ_i' are retained. So $\_ \vdash \_ \sim \_$ may not simply discard final heaps and carry on with the initial one.

On the other hand, some detail must be hidden from the observer who may probe functions for equality by supplying (nearly) any argument denotation they wish via the heap. The following example illustrates how exactly such argument denotations should be restricted:

$$\begin{array}{rcl}\mathcal{S}_\varepsilon[\![\mathbf{let}\ x = \mathtt{tt}\ \mathbf{in}\ \bar{\lambda} y.y]\!]_{[]}([]) & = & \ldots :: \langle \mathsf{Fun}(\lambda\mathsf{a}.\ \lambda\mu.\ \ldots\mu(\mathsf{a})\ldots), [\mathsf{a}_1 \mapsto \mathcal{S}_\varepsilon[\![\mathtt{tt}]\!]_\ldots]\rangle_\checkmark \\ & \equiv & \ldots :: \langle \mathsf{Fun}(\lambda\mathsf{a}.\ \lambda\mu.\ \ldots\mu(\mathsf{a})\ldots), [\mathsf{a}_1 \mapsto \mathcal{S}_\varepsilon[\![\mathtt{ff}]\!]_\ldots]\rangle_\checkmark \\ & = & \mathcal{S}_\varepsilon[\![\mathbf{let}\ x = \mathtt{ff}\ \mathbf{in}\ \bar{\lambda} y.y]\!]_{[]}([]),\end{array}$$

In both cases, the function value $\mathsf{Fun}(\lambda\mathsf{a}.\ \lambda\mu.\ \ldots\mu(\mathsf{a})\ldots)$ is to be probed by the observer with an address of their choosing. If the observer was given free reign, they could choose a_1 and thus be able to observe the different heaps. Since the parameter A tracks non-local addresses (as can be asserted in $\equiv$), we can prohibit such exploits by requiring $\mathsf{a} \in A$. This is not too grave a restriction on the observer, because they may pre-allocate any denotation into the heap they want (respecting well-addressedness, of course) due to laziness.

Lemma 25 (Weakening of address domain)**.** *Let A_1, A_2 be address domains such that $A_1 \subseteq A_2$. For all d, if $A_1 \vdash d$ then $A_2 \vdash d$. For all traces τ_1, τ_2, if $A_2 \vdash \tau_1 \sim \tau_2$ then $A_1 \vdash \tau_1 \sim \tau_2$.*

Lemma 26 (Address domain of $\mathcal{S}_\varepsilon[\![\_]\!]$)**.** *For all address spaces A, environments ρ and expressions* e*, if $\mathsf{rng}(\rho) \subseteq A$ then $A \vdash \mathcal{S}_\varepsilon[\![\mathsf{e}]\!]_\rho$.*

Our implicit assumption on well-addressedness can be encoded as an axiom as follows:

Axiom 27 (Well-addressedness)**.** *For all address spaces A, environments ρ and expressions* e*, if $A \vdash \mathcal{S}_\varepsilon[\![\mathsf{e}]\!]_\rho$ then $\mathsf{rng}(\rho) \subseteq A$.*

One could tweak the definition of $\mathcal{S}_\varepsilon[\![\_]\!]$ such that it is undefined on an input heap μ when $\mathsf{rng}(\rho) \not\subseteq \mathsf{dom}(\mu)$, thus making this axiom a provable lemma.

Evaluating lazy elements on heaps that force to each other is semantically equivalent:

Lemma 28. *Let d be lazy. Then for all μ_1, μ_2 with $\mathsf{dom}(\mu_1) \vdash d$ and $\mu_1 \rightsquigarrow \mu_2$, we have $\mathsf{dom}(\mu_1) \vdash d(\mu_1) \sim d(\mu_2)$.*

Theorem 29. *For any fixed address space A, $A \vdash \_ \sim \_$ is an equivalence relation on lazy traces.*

It follows that ($\approx$) is an equivalence relation on lazy heaps as well.

Unfortunately, the notion of a lazy element we have is too weak to prove reflexivity of ($\equiv$). We fix that by introducing the following extensionality requirement on lazy elements d:

Definition 30 (Extensionality). *A lazy element d is* extensional *if and only if $d \equiv d$.*

Theorem 31. *$(\equiv)$ is an equivalence relation on lazy and extensional elements in $\mathbb{D}^{\varepsilon}$.*

Note that we only make use of extensionality to prove reflexivity; in fact, $(\equiv)$ is still a (perhaps incomparable) partial equivalence relation on lazy elements.

What remains is a proof that $\mathcal{S}_{\varepsilon}[\![\mathsf{e}]\!]_{\rho}$ is indeed extensional.

Theorem 32. *For all environments ρ and expressions e, $\mathcal{S}_{\varepsilon}[\![\mathsf{e}]\!]_{\rho}$ is extensional.*

As before, we will assume that all elements d are lazy and extensional from now on.

5.3 Compositionality

Compositionality is an important property of a semantics (or, rather of a congruence relation on the semantics). Roughly, when a semantics is compositional, the denotation of an expression is a function of the meaning of its subexpressions. The "is a function of" qualifier is most often understood syntactically in terms of a property of *contexts*.

We follow Moran and Sands [1999] and define *evaluation contexts* E[8] with holes $\Box$ in order to formalise compositionality. Then $\mathsf{E}[\mathsf{e}]$ denotes an expression where the hole in E has been replaced with e, expressing that the bigger expression $\mathsf{E}[\mathsf{e}]$ is a function of e.

$$\mathsf{E} \quad ::= \quad \Box \mid \mathsf{E}\ \mathsf{x} \mid \textbf{let}\ \mathsf{x} = \mathsf{e}\ \textbf{in}\ \mathsf{E} \mid \textbf{let}\ \mathsf{x} = \mathsf{E}_1\ \textbf{in}\ \mathsf{E}_2[\mathsf{x}] \mid \textbf{case}\ \mathsf{E}\ \textbf{of}\ \overline{K\ \overline{\mathsf{x}} \to \mathsf{e}_r}$$

Definition 33 (Compositionality of an equivalence relation). *Let $\cong$ be an equivalence relation on expressions. $\cong$ is* compositional *if and only if for all $\mathsf{e}_1, \mathsf{e}_2$, $\mathsf{e}_1 \cong \mathsf{e}_2$ implies $\mathsf{E}[\mathsf{e}_1] \cong \mathsf{E}[\mathsf{e}_2]$ for all evaluation contexts E.*

It is well-known that *contextual equivalence* is compositional:

Definition 34 (Contextual equivalence, as in [Moran and Sands 1999]). *Two expressions $\mathsf{e}_1, \mathsf{e}_2$ are* contextually equivalent, *written $\mathsf{e}_1 \equiv_{ctx} \mathsf{e}_2$, if their termination behavior coincides in all contexts, e.g., for all closed evaluation contexts E it is*

$$\mathit{len}(\mathcal{S}_{\varepsilon}[\![\mathsf{E}[\mathsf{e}_1]]\!]_{[]}([])) = \omega \iff \mathit{len}(\mathcal{S}_{\varepsilon}[\![\mathsf{E}[\mathsf{e}_2]]\!]_{[]}([])) = \omega$$

where len is the length function from Definition 11.

The definition of Moran and Sands [1999] is in terms of a transition system like $(\hookrightarrow)$, but adequacy (Theorem 15) guarantees that the two notions coincide.

Lemma 35 (Contextual equivalence is compositional). *Let $\mathsf{e}_1, \mathsf{e}_2$ be expressions such that $\mathsf{e}_1 \equiv_{ctx} \mathsf{e}_2$. Then for all evaluation contexts E, we have $\mathsf{E}[\mathsf{e}_1] \equiv_{ctx} \mathsf{E}[\mathsf{e}_2]$.*

$\mathcal{S}_{\varepsilon}[\![\_]\!]$ is compositional wrt. definitional equality $=$ on $\mathbb{D}^{\varepsilon}$:

Lemma 36 (Definitional equivalence is compositional). *Let $\mathsf{e}_1, \mathsf{e}_2$ be expressions such that for all environments ρ, $\mathcal{S}_{\varepsilon}[\![\mathsf{e}_1]\!]_{\rho} = \mathcal{S}_{\varepsilon}[\![\mathsf{e}_2]\!]_{\rho}$. Then for all evaluation contexts E and environments ρ, we have $\mathcal{S}_{\varepsilon}[\![\mathsf{E}[\mathsf{e}_1]]\!]_{\rho} = \mathcal{S}_{\varepsilon}[\![\mathsf{E}[\mathsf{e}_2]]\!]_{\rho}$.*

Most interestingly, $\mathcal{S}_{\varepsilon}[\![\_]\!]$ is also compositional wrt. to $(\equiv)$!

Theorem 37 (Semantic equivalence is compositional). *Let $\mathsf{e}_1, \mathsf{e}_2$ be expressions such that for all environments ρ, $\mathcal{S}_{\varepsilon}[\![\mathsf{e}_1]\!]_{\rho} \equiv \mathcal{S}_{\varepsilon}[\![\mathsf{e}_2]\!]_{\rho}$. Then for all evaluation contexts E and environments ρ, we have $\mathcal{S}_{\varepsilon}[\![\mathsf{E}[\mathsf{e}_1]]\!]_{\rho} \equiv \mathcal{S}_{\varepsilon}[\![\mathsf{E}[\mathsf{e}_2]]\!]_{\rho}$.*

[8] In Definition 4 we already used the term "evaluation context" to denote small-step machine components (ρ, μ, κ) rather than E. Moran and Sands [1999] prove that both notions are equivalent, so this overload is justified.

And finally, we can use this result to prove that $(\equiv)$ is indeed coarser than contextual equivalence:

Corollary 38 (Semantic equivalence implies contextual equivalence)**.** *Let* $\mathsf{e}_1, \mathsf{e}_2$ *be expressions such that* $\mathcal{S}_\varepsilon[\![\mathsf{e}_1]\!]_\rho \equiv \mathcal{S}_\varepsilon[\![\mathsf{e}_2]\!]_\rho$*. Then for all closed evaluation contexts* E*, we have*

$$\mathit{len}(\mathcal{S}_\varepsilon[\![\mathsf{E}[\mathsf{e}_1]]\!]_{[]}([])) = \omega \Leftrightarrow \mathit{len}(\mathcal{S}_\varepsilon[\![\mathsf{E}[\mathsf{e}_1]]\!]_{[]}([])) = \omega.$$

This corollary naturally leads to the converse question: Does contextual equivalence imply semantic equivalence? If that were the case, then $\mathcal{S}_\varepsilon[\![\_]\!]$ would be *fully abstract* via $(\equiv)$ [Plotkin 1977]. Plotkin established that a definition like $\mathcal{S}_\perp[\![\_]\!]$ is not fully abstract via definitional equality on $\mathbb{D}^\perp$. Guarded types prevent probing for diverging computations without executing them, but it is still possible for lazy elements to probe traces for stuckness without propagating it or evaluating multiple traces concurrently, so we have reason to doubt that $\mathcal{S}_\varepsilon[\![\_]\!]$ is fully abstract.

6 ABSTRACT INTERPRETATION

6.1 Lazy Denotational Deadness

Let us now finally try to reformulate semantic deadness in terms of $\mathcal{S}_\varepsilon[\![\_]\!]$ and $\equiv$:

Definition 39 (Denotational deadness, lazily)**.** *An address* a *is* dead *in a denotation* d *if and only if, for all* $\mu_1 \approx \mu_2$ *such that* a *is dead in* $\mathsf{rng}(\mu_i)$*,* $\mathsf{dom}(\mu_1) \cup \{\mathsf{a}\} \vdash d$ *and* d_1, d_2*, we have*

$$\mathsf{dom}(\mu_1) \vdash d(\mu_1[\mathsf{a} \mapsto d_1]) \sim d(\mu_2[\mathsf{a} \mapsto d_2]).$$

A variable x *is* dead *in an expression* e *if and only if any* a *is dead in* $\mathcal{S}_\varepsilon[\![\mathsf{e}]\!]_{\rho[\mathsf{x}\mapsto\mathsf{a}]}$ *for any* ρ *such that* $\mathsf{a} \notin \mathsf{rng}(\rho)$*. Otherwise,* x *is* live*.*

Lemma 40 (Deadness implies irrelevance)**.** *If* x *is dead in* e*, then for all* $\rho, \mathsf{e}_1, \mathsf{e}_2$*,*

$$\mathcal{S}_\varepsilon[\![\mathbf{let}\ \mathsf{x} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}]\!]_\rho \equiv \mathcal{S}_\varepsilon[\![\mathbf{let}\ \mathsf{x} = \mathsf{e}_2\ \mathbf{in}\ \mathsf{e}]\!]_\rho.$$

So if x is dead in e_2, we can justify the following rewrite by irrelevance:

$$\mathbf{let}\ x = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2 \equiv \mathbf{let}\ x = \mathit{panic}\ \mathbf{in}\ \mathsf{e}_2$$

A syntactic *occurrence analysis* could subsequently figure out whether the binding for x can be dropped without introducing scoping errors, a feat that becomes far simpler once huge expressions e_1 are turned into small *panic*s.

We can now prove Theorem 2 in terms of this new characterisation of deadness by induction:

Theorem 41 ($\mathcal{S}_u[\![\_]\!]$ is a correct deadness analysis)**.** *Let* e *be an expression,* x *a variable and* $\tilde{\rho}$ *a usage environment. If* $\tilde{\rho}(\mathsf{x}) \not\sqsubseteq \mathcal{S}_u[\![\mathsf{e}]\!]_{\tilde{\rho}}$ *then* x *is dead in* e*.*

6.2 Discussion

The main proof of Theorem 41 is hardly longer than Theorem 2, but we have to admit that we needed to prove quite a few metatheoretic properties to get there, so we declare only partial victory on Goal 1 from Section 2. For obvious reasons, it seems preferable to stick to a simpler call-by-name semantics without a heap if the property in question (e.g., deadness) can be understood there as well.

Still, with Definition 39 we were at least able to break down the proof into manageable intermediate steps. That is a huge step forward compared to the operational deadness definition of Definition 4 where we weren't even able to come up with a suitable correctness relation.

In hindsight, we could trace back the intermediate steps to come up with at least one proof strategy with Definition 4: The structural induction principle is an important enabling factor and the focus on maximal traces in Figure 4 suggests that we should likely strive for a correctness

$$\boxed{A \vdash_n \tau_1 \lesssim \tau_2 \qquad \mu_1 \lessapprox \mu_2 \qquad d_1 \preccurlyeq d_2}$$

$$\frac{\blacktriangleright(A \vdash_n \tau_1 \lesssim \tau_2)}{A \vdash_n \_ :: \tau_1 \lesssim \_ :: \tau_2}\ \textsc{Imp-LR-::} \qquad \frac{A \vdash_{n+1} \tau_1 \lesssim \tau_2}{A \vdash_n \tau_1 \lesssim \_ :: \tau_2}\ \textsc{Imp-R-::} \qquad \frac{A \vdash_n \tau_1 \lesssim \tau_2}{A \vdash_{n+1} \_ :: \tau_1 \lesssim \tau_2}\ \textsc{Imp-L-::}$$

$$\frac{}{A \vdash_0 \lightning \lesssim \lightning}\ \textsc{Imp-}\lightning \qquad \frac{\overline{A \vdash_n \mu_1(\mathsf{a}_1) \lesssim \mu_2(\mathsf{a}_2))}}{A \vdash_{\Sigma\{\overline{n}\}} \langle \mathsf{Con}(K, \overline{\mathsf{a}_1}), \mu_1 \rangle_\checkmark \lesssim \langle \mathsf{Con}(K, \overline{\mathsf{a}_2}), \mu_2 \rangle_\checkmark}\ \textsc{Imp-Con}$$

$$\frac{\forall \mathsf{a}.\ \mathsf{a} \in A \Longrightarrow A \vdash_n f_1(\mathsf{a})(\mu_1) \lesssim f_2(\mathsf{a})(\mu_2)}{A \vdash_n \langle \mathsf{Fun}(f_1), \mu_1 \rangle_\checkmark \lesssim \langle \mathsf{Fun}(f_2), \mu_2 \rangle_\checkmark}\ \textsc{Imp-Fun}$$

$$\frac{\mathsf{dom}(\mu_1) = \mathsf{dom}(\mu_2) \quad \forall \mathsf{a}.\ \blacktriangleright(\mathsf{dom}(\mu_1) \vdash_0 \mu_1(\mathsf{a})(\mu_1) \lesssim \mu_2(\mathsf{a})(\mu_2))}{\mu_1 \lessapprox \mu_2}\ \textsc{Imp-}\mathbb{H}$$

$$\frac{\forall \mu_1, \mu_2.\ \mu_1 \lessapprox \mu_2 \wedge \mathsf{dom}(\mu_1) \vdash d_1, d_2 \Longrightarrow \mathsf{dom}(\mu_1) \vdash_0 d_1(\mu_1) \lesssim d_2(\mu_2)}{d_1 \preccurlyeq d_2}\ \textsc{Imp-}\mathbb{D}^\varepsilon$$

Fig. 7. Improvement relation

relation characterising a property of maximal traces. This was not obvious to us when we first set out to do the proof; the gap was too large to see how to get to the other side due to the structural mismatch. A nice consequence of successfully avoiding structural mismatch, which we set out to in Goal 4.

6.3 Improvement

Proving that dead bindings can be soundly rewritten is a nice litmus test for the semantics. But does it really make the program faster, or at least not slower?

We can affirm that indirectly: A diverging program *loop* takes more steps than a stuck program *panic*, hence the former runs "slower" than the latter. The stuck and the diverging program are not semantically equivalent, but **let** x = *panic* **in** e is semantically equivalent to **let** x = *loop* **in** e when x is dead in e. Since $(\gg\!\beta\!=)$ runs sub-programs to completion, we would observe execution of *panic* or *loop*, which is the only way in which we could have made the program slower or faster. Since there is no semantic difference, the performance of the program must be unaffected.

This argument is quite vague. In order to put it on firm ground, we define an *improvement relation* $(\preccurlyeq)$ in the style of Moran and Sands [1999] in Figure 7.

Structurally, $(\preccurlyeq)$ is very similar to $(\equiv)$; the main differences are in the rules for $(::)$ and the resulting tracking of *skew credits* $n \in \mathbb{N}$, e.g., we count the applications of Imp-R-:: to spend them on Imp-L-:: or when a value is further scrutinised. Naturally, it is easier to prove $A \vdash_n \tau_1 \lesssim \tau_2$ the more credits we have at our expense and thus the larger n is.

We write $d_1 \eqslantless d_2$ when $d_1 \preccurlyeq d_2$ and $d_2 \preccurlyeq d_1$, in which case both denotations operate in lockstep. We conjecture that $(\preccurlyeq)$ is a sub-relation of the strong contextual improvement relation of Moran and Sands [1999], just as $(\equiv)$ is finer than contextual equivalence.

We could now once again refine our notion of deadness, using lockstep simulation. It is then easy to see that $\mathcal{S}_u[\![\_]\!]$ is correct wrt. to this stronger notion of deadness, because we can in large parts reuse the proof for Theorem 41.

6.4 Evaluation Cardinality

Since our new semantics is able to express evaluation cardinality and thunk update, we may add a new **let1** construct to our language that opts out of memoisation:

$$\varepsilon \in \mathbb{E}_\mathbb{V} \quad ::= \quad \ldots \mid \mathsf{bind1}(\mathsf{x}, \mathsf{a} \mapsto d)$$

$$\mathcal{S}_\varepsilon[\![\textbf{let1 } \mathsf{x} = \mathsf{e}_1 \textbf{ in } \mathsf{e}_2]\!]_\rho(\mu) \quad = \quad \begin{array}{l} \text{let } \rho' = \rho[\mathsf{x} \mapsto \mathsf{a}] \quad \text{where } \mathsf{a} \notin \mathrm{dom}(\mu) \\ \text{in } \ \mathsf{bind1}(\mathsf{x}, \mathsf{a} \mapsto \mathcal{S}_\varepsilon[\![\mathsf{e}_1]\!]_{\rho'}) :: \mathcal{S}_\varepsilon[\![\mathsf{e}_2]\!]_{\rho'}(\mu[\mathsf{a} \mapsto \mathcal{S}_\varepsilon[\![\mathsf{e}_1]\!]_{\rho'}]) \end{array}$$

Any program in which we switch from memoised **let** to **let1** is semantically equivalent after we adjust the definition of lazy heaps accordingly. This is a simple consequence of the fact that $\mathit{memo}(\mathsf{a}, d) \equiv d$ and compositionality.

However, omitting thunk memoisation has measurable effect on performance if the same variable is evaluated repeatedly! We should rather show that whenever x is *evaluated at most once*, it is an improvement to rewrite **let** $x = \mathsf{e}_1$ **in** e_2 to **let1** $x = \mathsf{e}_1$ **in** e_2.

We can sharpen this statement by making use of *tick algebra* [Moran and Sands 1999]. For that, we need to add a notion of ticks to our language (postfix, to mirror *memo*), a routine extension:

$$\varepsilon \in \mathbb{E}_\mathbb{V} \quad ::= \quad \ldots \mid \mathsf{tick}$$

$$\mathcal{S}_\varepsilon[\![\mathsf{e}\checkmark]\!]_\rho(\mu) \quad = \quad \mathcal{S}_\varepsilon[\![\mathsf{e}]\!]_\rho(\mu) \gg\!\beta\!= \lambda v.\, \lambda\mu.\, \mathsf{tick} :: \langle v, \mu\rangle_\checkmark$$

Whenever x is "evaluated at most once", we have $\mathcal{S}_\varepsilon[\![\textbf{let } x = \mathsf{e}_1 \textbf{ in } \mathsf{e}_2]\!]_\rho \mathrel{\not\approx} \mathcal{S}_\varepsilon[\![\textbf{let1 } x = \mathsf{e}_1\checkmark \textbf{ in } \mathsf{e}_2]\!]_\rho$. In Section 2, specifically Theorem 3, we have proclaimed that x is evaluated at most once whenever $\mathcal{S}_u[\![\mathsf{e}_2]\!]_{\tilde\rho_\Delta}(x) \sqsubseteq 1$. We can now make good on that claim:

Theorem 42 (Update avoidance for $\mathcal{S}_u[\![\_]\!]$)**.** *Let* **let** $\mathsf{x} = \mathsf{e}_1$ **in** e_2 *be an expression such that* $\mathcal{S}_u[\![\mathsf{e}_1]\!]_{\tilde\rho_\Delta}(\mathsf{x}) + \mathcal{S}_u[\![\mathsf{e}_2]\!]_{\tilde\rho_\Delta}(\mathsf{x}) \sqsubseteq 1$. *Then* $\mathcal{S}_\varepsilon[\![\textbf{let } \mathsf{x} = \mathsf{e}_1 \textbf{ in } \mathsf{e}_2]\!]_\rho \mathrel{\not\approx} \mathcal{S}_\varepsilon[\![\textbf{let1 } \mathsf{x} = \mathsf{e}_1\checkmark \textbf{ in } \mathsf{e}_2]\!]_\rho$.

But $\mathcal{S}_u[\![\textbf{let } x = \mathsf{e}_1 \textbf{ in } \mathsf{e}_2]\!]_{\tilde\rho_\Delta}(x) \sqsubseteq 1$ is just a sufficient condition for the semantic property of "evaluates x at most once"; it is not a suitable *definition* of that property by far. For example, $((\bar\lambda y.x)\ x)$ evaluates x at most once, but is not recognised as such by $\mathcal{S}_u[\![\_]\!]$.

Intuitively, we need a function $\mathit{count}_\mathsf{a} : \mathbb{T}^\varepsilon \to \mathbb{N}_\omega$ that counts $\mathsf{look}(\mathsf{a})$ actions at a particular address a over the course of a head-normal form reduction and take the upper bound of this function in arbitrary evaluation contexts.

We distribute these reponsibilities between the two functions *usg* and *ctx* in Figure 8:

Definition 43 (Usage cardinality)**.** *A denotation d evaluates an address* a at most u times *(where $u \in \mathsf{Usg}$) if and only if, for all μ such that* a *is dead in μ,*

$$\mathit{ctx}_{\mathrm{dom}(\mu)}(\mathit{usg}_{\mathbb{T}^\varepsilon}(d(\mu)))(\mathsf{a}) \sqsubseteq u.$$

An expression e *evaluates* x at most u times *if and only if, for* $\mathsf{a} \notin \mathrm{rng}(\rho)$, $\mathcal{S}_\varepsilon[\![\mathsf{e}]\!]_{\rho[\mathsf{x}\mapsto\mathsf{a}]}$ *evaluates* a *at most u times.*

The *semantic usage abstraction* $\mathit{usg}_{\mathbb{T}^\varepsilon} : \mathbb{T}^\varepsilon \to \mathbb{T}^{\lceil\mathsf{look}\rceil}$ is the most precise usage analysis and is induced componentwise on $\mathbb{T}^\varepsilon$, $\mathbb{H}$, $\mathbb{D}^\varepsilon$ and $\mathbb{V}^\varepsilon$ by the event abstraction $\mathit{usg}_{\mathbb{E}_\mathbb{V}}$ in Figure 8.

A $\mathbb{T}^{\lceil\mathsf{look}\rceil}$ trace can then be folded by $\mathit{ctx}_{\mathbb{T}^{\lceil\mathsf{look}\rceil}}$ into an $\mathsf{Addr} \to \mathsf{Usg}$ mapping.[9] Thus, $\mathit{ctx}_{\mathbb{T}^{\lceil\mathsf{look}\rceil}}$ continues a $\mathbb{T}^{\lceil\mathsf{look}\rceil}$ in all possible evaluation contexts, returning the least upper bound of all continued traces.

[9] It is worth noting that for brevity we play fast and loose with guardedness conditions for $\mathbb{T}^{\lceil\mathsf{look}\rceil}$. I.e., **let** *loop* = **let** x = tt **in** *loop* **in** *loop* generates a diverging trace for which $\mathit{usg}_{\mathbb{T}^\varepsilon}$ would only produce an $\mathsf{Addr} \to \mathsf{Usg}$ mapping after an infinite amount of time. A rigorous treatment would have each event emit its own l in a guarded fashion.

$$\begin{array}{llll} \mu \in \mathbb{H}^{\lceil \mathsf{look} \rceil} = \mathsf{Addr} \rightharpoonup \blacktriangleright \mathbb{D}^{\lceil \mathsf{look} \rceil} & d \in \mathbb{D}^{\lceil \mathsf{look} \rceil} = \mathbb{H}^{\lceil \mathsf{look} \rceil} \to \mathbb{T}^{\lceil \mathsf{look} \rceil} \\ u \in \mathsf{Usg} = \{0 \sqsubset 1 \sqsubset \omega\} \subset \mathbb{N}_\omega & \tau \in \mathbb{T}^{\lceil \mathsf{look} \rceil} ::= (l \in \mathsf{Addr} \to \mathsf{Usg}) :> \langle v, \mu \rangle \end{array}$$

$$v \in \mathbb{V}^{\lceil \mathsf{look} \rceil} ::= \mathsf{Fun}(f \in \mathsf{Addr} \to \mathbb{D}^{\lceil \mathsf{look} \rceil})$$

$$\boxed{usg_{\mathbb{E}_\mathbb{v}} : \mathbb{E}_\mathbb{v} \to (\mathsf{Addr} \to \mathsf{Usg}) \quad usg_{\mathbb{T}^\varepsilon} : \mathbb{T}^\varepsilon \to \mathbb{T}^{\lceil \mathsf{look} \rceil}}$$

$$\begin{array}{lcl} usg_{\mathbb{E}_\mathbb{v}}(\varepsilon) & = & \begin{cases} [\mathsf{a} \mapsto 1] & \varepsilon = \textsc{Look}(\mathsf{a}) \\ \lambda\_.\ 0 & \text{otherwise} \end{cases} \\ l_1 +_1 (l_2 :> \langle \tilde{v}, \tilde{\mu} \rangle) & = & (l_1 + l_2) :> \langle \tilde{v}, \tilde{\mu} \rangle \\ usg_{\mathbb{T}^\varepsilon}(\varepsilon :: \tau) & = & usg_{\mathbb{E}_\mathbb{v}}(\varepsilon) +_1 usg_{\mathbb{T}^\varepsilon}(\tau) \\ usg_{\mathbb{T}^\varepsilon}(\langle \mathsf{Fun}(f), \mu \rangle_\checkmark) & = & \lambda\_.\ 0 :> \langle \mathsf{Fun}(usg_{\mathbb{D}^\varepsilon} \circ f), usg_{\mathbb{H}}(\mu) \rangle \\ usg_{\mathbb{T}^\varepsilon}(\lightning) & = & \lambda\_.\ 0 :> \langle \bot_{\mathbb{V}^{\mathbb{D}^u}}, (\lambda\_.\ \bot_{\mathbb{D}^\varepsilon}) \circ \mu \rangle \\ usg_{\mathbb{H}}(\mu) & = & usg_{\mathbb{D}^\varepsilon} \circ \mu \\ usg^{\dashv}_{\mathbb{H}}(\tilde{\mu}) & = & \bigcup \{\mu \mid usg_{\mathbb{H}}(\mu) \sqsubseteq \tilde{\mu}\} \\ usg_{\mathbb{D}^\varepsilon}(d) & = & usg_{\mathbb{T}^\varepsilon} \circ d \circ usg^{\dashv}_{\mathbb{H}} \end{array}$$

$$\boxed{ctx\_ : \wp(\mathsf{Addr}) \to \mathbb{T}^{\lceil \mathsf{look} \rceil} \to \mathsf{Addr} \to \mathsf{Usg}}$$

$$ctx_A(l :> \langle \mathsf{Fun}(\tilde{f}), \tilde{\mu} \rangle) = l + \bigsqcup_{\mathsf{a} \in A} \{\omega * ctx_A(\tilde{f}(\mathsf{a})(\tilde{\mu}))\}$$

Fig. 8. Usage abstraction

Theorem 44 (Update avoidance). *Let* **let** x = e_1 **in** e_2 *be an expression such that* e_2 *evaluates* x *at most once and* x *is dead in* e_1. *Then* $\mathcal{S}_\varepsilon[\![\textbf{let } \mathsf{x} = \mathsf{e}_1 \textbf{ in } \mathsf{e}_2]\!]_\rho \mathrel{\text{⪱}} \mathcal{S}_\varepsilon[\![\textbf{let1 } \mathsf{x} = \mathsf{e}_1\checkmark \textbf{ in } \mathsf{e}_2]\!]_\rho$.

Similarly, one could prove that a binding is dead when it is evaluated at most 0 times. An alternative correctness proof for $\mathcal{S}_u[\![\_]\!]$ can then be formulated by *abstract interpretation*, proving it correct simultaneously as a deadness and a sharing analysis:

Theorem 45 ($\mathcal{S}_u[\![\_]\!]$ approximates semantic usage). *Let* e *be an expression,* x *a variable,* a *an address that is dead in a heap* μ *and* ρ *an environment such that* $\mathsf{a} \notin \mathsf{rng}(\rho)$. *Then*

$$ctx_{\mathsf{dom}(\mu)}(usg_{\mathbb{T}^\varepsilon}(\mathcal{S}_\varepsilon[\![\mathsf{e}]\!]_{\rho[\mathsf{x} \mapsto \mathsf{a}]}(\mu)))(\mathsf{a}) \sqsubseteq \mathcal{S}_u[\![\mathsf{e}]\!]_{\tilde{\rho}_\Delta}(\mathsf{x})$$

A more elegant proof would continually abstract $ctx_A \circ usg_{\mathbb{T}^\varepsilon}$ without relating to the structure of $\mathcal{S}_\varepsilon[\![\_]\!]$ or $\mathcal{S}_u[\![\_]\!]$ at all, encoding the strengthened inductive hypothesis (i.e., the logical relation) of the proof above in a chain of Galois connections. Then the above proof is simply by direct, structural induction on e and equational reasoning – Cousot [2021] calls this process "calculational design". We want to attempt such a proof in the future and are confident that it will yield insightful abstraction functions, for example encoding the transition from heap-based call-by-need to call-by-name or the transition from interprocedural to intraprocedural analysis.

7 RELATED WORK

Coinduction and Traces. Leroy and Grall [2009] show that a coinductive encoding of big-step semantics is able to encode diverging traces by proving it equivalent to a small-step semantics. Their Lemma 10 covers much the same ground as Theorem 15. Our use of the (≫=) operator begs for a Monad instance that has been explored by Xia et al. [2019] for a coinductive trace type where the event and result type are abstracted out as type parameters.

Denotational Semantics. Recent work on *Clairvoyant call-by-value* semantics [Hackett and Hutton 2019] sheds light on a useful, heapless denotational interpretation of call-by-need. Their semantics could be factored in two: A semantics that non-deterministically drops or eagerly evaluates let bindings, and a downstream min function that picks the (non-stuck) trace with the least amount of steps. The continuity restrictions of the algebraic domain on the semantics necessitate fusing both functions. The trace generated by e may not even share a common prefix with the trace generated for e x. We had trouble abstracting such a semantics. It would be interesting to revisit the problem with a guarded domain formulation such as Møgelberg and Vezzosi [2021].

Control-Flow Analysis. Control-flow analysis [Shivers 1991] computes a useful control-flow graph abstraction for higher-order programs. Such an approximation is useful to apply classic data-flow analyses such as constant propagation or dead code elimination to the interprocedural setting. The contour depth parameter k allows to trade precision for performance, although practical applications never seem to go beyond 1. By comparison, our simple, cheerful usage analysis $\mathcal{S}_u[\![\_]\!]$ works without control-flow abstraction. The precision-performance trade-off ranges between the incomputable semantic usage abstraction $usg_{\mathbb{D}^\varepsilon}$ and the computable but naïve usage analysis $\mathcal{S}_u[\![\_]\!]$. Montagu and Jensen [2021] derive control-flow analysis from small-step traces. Their chain of abstractions is inspiring and we think that a variant of our trace-based semantics would be a good fit for their collecting semantics. Specifically, the semantic inclusions of Lemma 2.10 could be replaced by an abstraction similar to $usg_{\mathbb{D}^\varepsilon}$, reusing the correctness predicate in Figure 4.

Reachable States vs. Lexical Reasoning. Abstracting Abstract Machines [Van Horn and Might 2010] is an ingenious recipe to derive a computable *reachable states semantics* [Cousot 2021] from any small-step semantics. By bounding the size of the store, the freely choosably $\widehat{alloc}$ function embodies the precision-performance trade-off. Many analyses such as control-flow analysis (and usage analysis) arise as abstractions of reachable states.

It is unclear how $\mathcal{S}_u[\![\_]\!]$ fits into this framework, because it does not reason about a heap at all. $\mathcal{S}_u[\![\_]\!]$ treats a program variable as an alias-free representative of all its activations, even though there might be multiple activations of the same variable in the heap already. Conservative measures need only be taken when a variable escapes its lexical scope. In particular, for multiple simultaneously live activations of i in, e.g., $\mathbf{let}\ f = \bar{\lambda}x.\mathbf{let}\ i = \bar{\lambda}y.y\ \mathbf{in}\ i\ x\ x\ \mathbf{in}\ f\ f$, $\mathcal{S}_u[\![\_]\!]$ is able to correctly deduce that i is evaluated at most once. More complex analyses may infer interprocedural cardinality information via *demand transformers* [Sergey et al. 2017], likewise without reasoning about heaps during analysis. We intend to connect the notion of demand transformers to transformer abstraction [Cousot 2021] of Fun values in the future (i.e., functions *in the object language*), a connection that has not been possible with prior semantics.

We are curious about the precision-performance trade-offs of lexical analyses such as $\mathcal{S}_u[\![\_]\!]$ viz-à-viz abstractions of reachabe states. Of course, this work has not provided a tunable abstraction recipe like AAM, so such a comparison is presently impossible.

REFERENCES

Samson Abramsky, Dov M. Gabbay, and T. S. E. Maibaum. 1994. *Handbook of logic in computer science. Volume 3. Semantic Structures.* Clarendon Press. https://global.oup.com/academic/product/handbook-of-logic-in-computer-science-9780198537625

Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. 2004. A functional correspondence between call-by-need evaluators and lazy abstract machines. *Inform. Process. Lett.* 90, 5 (2004), 223–232. https://doi.org/10.1016/j.ipl.2004.02.012

Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. 1995. A Call-by-Need Lambda Calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 233–246. https://doi.org/10.1145/199448.199507

Lars Birkedal and Rasmus Ejlers Mogelberg. 2013. Intensional Type Theory with Guarded Recursive Types qua Fixed Points on Universes. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '13)*. IEEE Computer Society, USA, 213–222. https://doi.org/10.1109/LICS.2013.27

Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E. Møgelberg, and Lars Birkedal. 2016. Guarded Dependent Type Theory with Coinductive Types. In *Foundations of Software Science and Computation Structures*, Bart Jacobs and Christof Löding (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 20–35.

Thierry Coquand. 1994. Infinite objects in type theory. In *Types for Proofs and Programs*, Henk Barendregt and Tobias Nipkow (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 62–78.

Patrick Cousot. 2021. *Principles of Abstract Interpretation.* MIT Press. https://mitpress.mit.edu/9780262044905/principles-of-abstract-interpretation/

Mattias Felleisen and D. P. Friedman. 1987. A Calculus for Assignments in Higher-Order Languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Munich, West Germany) *(POPL '87)*. Association for Computing Machinery, New York, NY, USA, 314. https://doi.org/10.1145/41625.41654

Jörgen Gustavsson. 1998. A Type Based Sharing Analysis for Update Avoidance and Optimisation. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '98)*. Association for Computing Machinery, New York, NY, USA, 39–50. https://doi.org/10.1145/289423.289427

Jennifer Hackett and Graham Hutton. 2019. Call-by-Need is Clairvoyant Call-by-Value. *Proc. ACM Program. Lang.* 3, ICFP, Article 114 (jul 2019), 23 pages. https://doi.org/10.1145/3341718

John Launchbury. 1993. A Natural Semantics for Lazy Evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '93)*. Association for Computing Machinery, New York, NY, USA, 144–154. https://doi.org/10.1145/158511.158618

Xavier Leroy and Hervé Grall. 2009. Coinductive big-step operational semantics. *Information and Computation* 207, 2 (2009), 284–304. https://doi.org/10.1016/j.ic.2007.12.004 Special issue on Structural Operational Semantics (SOS).

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of Functional Programming* 18, 1 (2008), 1–13. https://doi.org/10.1017/S0956796807006326

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

Rasmus Ejlers Møgelberg and Niccolò Veltri. 2019. Bisimulation as Path Type for Guarded Recursive Types. *Proc. ACM Program. Lang.* 3, POPL, Article 4 (jan 2019), 29 pages. https://doi.org/10.1145/3290317

Rasmus Ejlers Møgelberg and Andrea Vezzosi. 2021. Two Guarded Recursive Powerdomains for Applicative Simulation. *Electronic Proceedings in Theoretical Computer Science* 351 (dec 2021), 200–217. https://doi.org/10.4204/eptcs.351.13

Eugenio Moggi. 2007. Structuring Operational Semantics: Simplification and Computation. *Electronic Notes in Theoretical Computer Science* 172 (2007), 479–497. https://doi.org/10.1016/j.entcs.2007.02.016 Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin.

Benoît Montagu and Thomas Jensen. 2021. Trace-Based Control-Flow Analysis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 482–496. https://doi.org/10.1145/3453483.3454057

Andrew Moran and David Sands. 1999. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 43–56. https://doi.org/10.1145/292540.292547

Hiroshi Nakano. 2000. A Modality for Recursion. In *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science (LICS '00)*. IEEE Computer Society, USA, 255.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis.* Springer. https://doi.org/10.1007/978-3-662-03811-6

G.D. Plotkin. 1977. LCF considered as a programming language. *Theoretical Computer Science* 5, 3 (1977), 223–255. https://doi.org/10.1016/0304-3975(77)90044-5

Gordon D. Plotkin. 2004. A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming* 60-61 (2004), 17–139. https://doi.org/10.1016/j.jlap.2004.05.001

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, Massachusetts, USA) *(ACM '72)*. Association for Computing Machinery, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

Dana Scott. 1970. *Outline of a Mathematical Theory of Computation.* Technical Report PRG02. OUCL. 30 pages.

Dana Scott and Christopher Strachey. 1971. *Toward a Mathematical Semantics for Computer Languages.* Technical Report PRG06. OUCL. 49 pages.

Ilya Sergey, Dimitrios Vytiniotis, Simon Peyton Jones, and Joachim Breitner. 2017. Modular, higher order cardinality analysis in theory and practice. *Journal of Functional Programming* 27 (2017), e11. https://doi.org/10.1017/S0956796817000016

Peter Sestoft. 1997. Deriving a lazy abstract machine. *Journal of Functional Programming* 7, 3 (1997), 231–264. https://doi.org/10.1017/S0956796897002712

Olin Grigsby Shivers. 1991. Control-Flow Analysis of Higher-Order Languages or Taming Lambda.

David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 51–62. https://doi.org/10.1145/1863543.1863553

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2019. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proc. ACM Program. Lang.* 4, POPL, Article 51 (dec 2019), 32 pages. https://doi.org/10.1145/3371119