Abstracting Denotational Interpreters

A Pattern for Sound, Compositional and Higher-order Static Program Analysis

SEBASTIAN GRAF, Karlsruhe Institute of Technology, Germany
SIMON PEYTON JONES, Epic Games, UK
SVEN KEIDEL, TU Darmstadt, Germany

We explore *denotational interpreters*: denotational semantics that produce coinductive traces of a corresponding small-step operational semantics. By parameterising our denotational interpreter over the semantic domain and then varying it, we recover *dynamic semantics* with different evaluation strategies as well as *summary-based static analyses* such as type analysis, all from the same generic interpreter. Among our contributions is the first provably adequate denotational semantics for call-by-need. The generated traces lend themselves well to describe *operational properties* such as evaluation cardinality, and hence to static analyses abstracting these operational properties. Since static analysis and dynamic semantics share the same generic interpreter definition, soundness proofs via abstract interpretation decompose into showing small abstraction laws about the abstract domain, thus obviating complicated ad-hoc preservation-style proof frameworks.

CCS Concepts: • **Software and its engineering → Semantics**; **Automated static analysis**; *Compilers*; *Procedures, functions and subroutines*; Functional languages; Software maintenance tools.

Additional Key Words and Phrases: Programming language semantics, Abstract Interpretation, Static Program Analysis

ACM Reference Format:
Sebastian Graf, Simon Peyton Jones, and Sven Keidel. 2024. Abstracting Denotational Interpreters: A Pattern for Sound, Compositional and Higher-order Static Program Analysis. *Proc. ACM Program. Lang.* 1, ICFP, Article 1 (January 2024), 73 pages. https://doi.org/10.1145/1111111

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 $\bar{\lambda}x.x + 1$" or "this program never evaluates x". In a functional-language setting, such static analyses are often defined *compositionally* on the input term. For example, consider the claim "(even 42) has type Bool". Type analysis asserts that $\mathit{even} :: \mathsf{Int} \to \mathsf{Bool}$, $42 :: \mathsf{Int}$, and then applies the function type to the argument type to produce the result type $\mathit{even}\ 42 :: \mathsf{Bool}$. The function type $\mathsf{Int} \to \mathsf{Bool}$ is a *summary* of the definition of even: Whenever the argument has type Int, the result has type Bool. Function summaries enable efficient modular higher-order analyses, because it is much faster to apply the summary of a function instead of reanalysing its definition at use sites in other modules.

If the analysis is used in a compiler to inform optimisations, it is important to prove it sound, because lacking soundness can lead to miscompilation of safety-critical applications [Sun et al. 2016]. In order to prove the analysis sound, it is helpful to pick a language semantics that is also

Authors' addresses: Sebastian Graf, Karlsruhe Institute of Technology, Karlsruhe, Germany, sgraf1337@gmail.com; Simon Peyton Jones, Epic Games, Cambridge, UK, simon.peytonjones@gmail.com; Sven Keidel, TU Darmstadt, Darmstadt, Germany, sven.keidel@tu-darmstadt.de.

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

compositional, such as a *denotational semantics* [Scott and Strachey 1971]; then the semantics and the analysis "line up" and the soundness proof is relatively straightforward. Indeed, one can often break up the proof into manageable sub goals by regarding the analysis as an *abstract interpretation* of the compositional semantics [Cousot 2021].

Alas, traditional denotational semantics does not model operational details – and yet 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 and heap, and sees program execution as a sequence of machine states. Now we have two unappealing alternatives:

- Develop a difficult, ad-hoc soundness proof, one that links a non-compositional operational semantics with a compositional analysis.
- Reimagine and reimplement the analysis as an abstraction of the reachable states of an operational semantics. This is the essence of the *Abstracting Abstract Machines* (AAM) [Van Horn and Might 2010] recipe, a very fruitful framework, but one that follows the *call strings* approach [Sharir et al. 1978], reanalysing function bodies at call sites. Hence the new analysis becomes non-modular, leading to scalability problems for a compiler.

In this paper, we resolve the tension by exploring *denotational interpreters*: total, mathematical objects that live at the intersection of structurally-defined *definitional interpreters* [Reynolds 1972] and denotational semantics. Our denotational interpreters generate small-step traces embellished with arbitrary operational detail and enjoy a straightforward encoding in typical higher-order programming languages. Static analyses arise as instantiations of the same generic interpreter, enabling succinct, shared soundness proofs just like for AAM or big-step definitional interpreters [Darais et al. 2017; Keidel et al. 2018]. However, the shared, compositional structure enables a wide range of summary mechanisms in static analyses that we think are beyond the reach of non-compositional reachable-states abstractions like AAM.

We make the following contributions:

- We use a concrete example (absence analysis) to argue for the usefulness of compositional, summary-based analysis in Section 2 and we demonstrate the difficulty of conducting an ad-hoc soundness proof wrt. a non-compositional small-step operational semantics.
- Section 4 walks through the definition of our generic denotational interpreter and its type class algebra in Haskell. We demonstrate the ease with which different instances of our interpreter endow our object language with call-by-name, call-by-need and call-by-value evaluation strategies, each producing (abstractions of) small-step abstract machine traces.
- A concrete instantiation of a denotational interpreter is *total* if it coinductively yields a (possibly-infinite) trace for every input program, including ones that diverge. Section 5.2 proves that the by-name and by-need instantiations are total by embedding the generic interpreter and its instances in Guarded Cubical Agda.
- Section 5.1 proves that the by-need instantiation of our denotational interpreter adequately generates an abstraction of a trace in the lazy Krivine machine [Sestoft 1997], preserving its length as well as arbitrary operational information about each transition taken.
- By instantiating the generic interpreter with a finite, abstract semantic domain in Section 6, we recover summary-based usage analysis, a generalisation of absence analysis in Section 2. Further examples in the Appendix comprise Type Analysis and 0CFA control-flow analysis, demonstrating the wide range of applicability of our framework.
- In Section 7, we apply abstract interpretation to characterise a set of abstraction laws that the type class instances of an abstract domain must satisfy in order to soundly approximate

$$\boxed{\mathcal{A}[\![\_]\!]_{\_} : \mathsf{Exp} \to (\mathsf{Var} \rightharpoonup \mathsf{AbsTy}) \to \mathsf{AbsTy}}$$

$$\begin{aligned}
\mathcal{A}[\![\mathsf{x}]\!]_\rho &= \rho(\mathsf{x}) \\
\mathcal{A}[\![\bar\lambda \mathsf{x}.\mathsf{e}]\!]_\rho &= \mathit{fun}_{\mathsf{x}}(\lambda\theta.\ \mathcal{A}[\![\mathsf{e}]\!]_{\rho[\mathsf{x}\mapsto\theta]}) \\
\mathcal{A}[\![\mathsf{e}\ \mathsf{x}]\!]_\rho &= \mathit{app}(\mathcal{A}[\![\mathsf{e}]\!]_\rho)(\rho(\mathsf{x})) \\
\mathcal{A}[\![\textbf{let}\ \mathsf{x} = \mathsf{e}_1\ \textbf{in}\ \mathsf{e}_2]\!]_\rho &= \mathcal{A}[\![\mathsf{e}_2]\!]_{\rho[\mathsf{x}\mapsto \mathsf{x} \& \mathcal{A}[\![\mathsf{e}_1]\!]_\rho]}
\end{aligned}
\qquad
\begin{array}{llcl}
a \in & \mathsf{Absence} & ::= & \mathsf{A} \mid \mathsf{U} \\
\varphi \in & \mathsf{Uses} & = & \mathsf{Var} \to \mathsf{Absence} \\
\varsigma \in & \mathsf{Summary} & ::= & a \mathbin{⨾} \varsigma \mid \mathsf{Rep}\ a \\
\theta \in & \mathsf{AbsTy} & ::= & \langle \varphi, \varsigma \rangle
\end{array}$$

$$\begin{aligned}
\mathit{fun}_{\mathsf{x}}(f) &= \langle \varphi[\mathsf{x}\mapsto\mathsf{A}], \varphi(\mathsf{x}) \mathbin{⨾} \varsigma \rangle \\
&\quad \text{where } \langle \varphi, \varsigma \rangle = f(\langle [\mathsf{x}\mapsto\mathsf{U}], \mathsf{Rep}\ \mathsf{U} \rangle) \\
\mathit{app}(\langle \varphi_f, a \mathbin{⨾} \varsigma \rangle)(\langle \varphi_a, _ \rangle) &= \langle \varphi_f \sqcup (a * \varphi_a), \varsigma \rangle
\end{aligned}
\qquad
\begin{gathered}
\mathsf{Rep}\ a \equiv a \mathbin{⨾} \mathsf{Rep}\ a \\
\mathsf{A} * \varphi = [\,] \quad \mathsf{U} * \varphi = \varphi \\
\mathsf{x} \,\&\, \langle \varphi, \varsigma \rangle = \langle \varphi[\mathsf{x}\mapsto\mathsf{U}], \varsigma \rangle
\end{gathered}$$

Fig. 1. Absence analysis

by-name and by-need interpretation. None of the proof obligations mention the generic interpreter, and, more remarkably, none of the laws mention the concrete semantics or the Galois connection either! This enables to prove usage analysis sound wrt. the by-name and by-need semantics in half a page, building on reusable semantics-specific theorems.

- We compare to the enormous body of related approaches in Section 8.

2 THE PROBLEM WE SOLVE

What is so difficult about proving a compositional, summary-based analysis sound wrt. a non-compositional small-step operational semantics? We will demonstrate the challenges in this section, by way of a simplified *absence analysis* [Peyton Jones and Partain 1994], a higher-order form of neededness analysis to inform removal of dead bindings in a compiler.

2.1 Object Language

To set the stage, we start by defining the object language of this work, a lambda calculus with *recursive* let bindings and algebraic data types:

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

This language is very similar to that of Launchbury [1993] and Sestoft [1997]. It is factored into *A-normal form*, that is, the arguments of applications are restricted to be variables, so the difference between lazy and eager semantics is manifest in the semantics of **let**. Note that $\bar\lambda x.x$ (with an overbar) denotes syntax, whereas $\lambda x.\ x + 1$ denotes an anonymous mathematical function. In this section, only the highlighted parts are relevant, but the interpreter definition in Section 4 supports data types as well. Throughout the paper we assume that all bound program variables are distinct.

2.2 Absence Analysis

In order to define and explore absence analysis in this subsection, we must clarify what absence means, semantically. A variable x is *absent* in an expression e when e never evaluates x, regardless of the context in which e appears. Otherwise, the variable x is *used* in e.

Figure 1 defines an absence analysis $\mathcal{A}[\![\mathsf{e}]\!]_\rho$ for lazy program semantics that conservatively approximates semantic absence.[1] It takes an environment $\rho \in \mathsf{Var} \rightharpoonup \mathsf{Absence}$ containing absence

[1]For illustrative purposes, our analysis definition only works for the special case of non-recursive let. The generalised definition for recursive as well as non-recursive let is $\mathcal{A}[\![\textbf{let}\ \mathsf{x} = \mathsf{e}_1\ \textbf{in}\ \mathsf{e}_2]\!]_\rho = \mathcal{A}[\![\mathsf{e}_2]\!]_{\rho[\mathsf{x}\mapsto \mathsf{lfp}(\lambda\theta.\ \mathsf{x}\&\mathcal{A}[\![\mathsf{e}_1]\!]_{\rho[\mathsf{x}\mapsto\theta]})]}$.

information about the free variables of e and returns an *absence type* $\langle \varphi, \varsigma \rangle \in \mathsf{AbsTy}$; an abstract representation of e. The first component $\varphi \in \mathsf{Uses}$ of the absence type captures how e uses its free variables by associating an Absence flag with each variable. When $\varphi(\mathrm{x}) = \mathsf{A}$, then x is absent in e; otherwise, $\varphi(\mathrm{x}) = \mathsf{U}$ and x might be used in e. The second component $\varsigma \in \mathsf{Summary}$ of the absence type summarises how e uses actual arguments supplied at application sites. For example, function $f \triangleq \bar{\lambda} x.y$ has absence type $\langle [y \mapsto \mathsf{U}], \mathsf{A} \mathbin{⨾} \mathsf{Rep}\ \mathsf{U} \rangle$. Mapping $[y \mapsto \mathsf{U}]$ indicates that f may use its free variable y. The literal notation $[y \mapsto \mathsf{U}]$ maps any variable other than y to A. Furthermore, summary $\mathsf{A} \mathbin{⨾} \mathsf{Rep}\ \mathsf{U}$ indicates that f's first argument is absent and all further arguments are potentially used. The summary $\mathsf{Rep}\ \mathsf{U}$ denotes an infinite repetition of U, as expressed by the non-syntactic equality $\mathsf{Rep}\ \mathsf{U} \equiv \mathsf{U} \mathbin{⨾} \mathsf{Rep}\ \mathsf{U}$.

We illustrate the analysis at the example expression $\mathrm{e} \triangleq \mathbf{let}\ k = \bar{\lambda} y.\bar{\lambda} z.y\ \mathbf{in}\ k\ x_1\ x_2$, where the initial environment for e, $\rho_{\mathrm{e}}(\mathrm{x}) \triangleq \langle [\mathrm{x} \mapsto \mathsf{U}], \mathsf{Rep}\ \mathsf{U} \rangle$, declares the free variables of e with a pessimistic summary $\mathsf{Rep}\ \mathsf{U}$.

$$
\begin{aligned}
& \mathcal{A}[\![\mathbf{let}\ k = \bar{\lambda} y.\bar{\lambda} z.y\ \mathbf{in}\ k\ x_1\ x_2]\!]_{\rho_{\mathrm{e}}} && \text{\textit{Unfold} } \mathcal{A}[\![\mathbf{let}\ \mathrm{x} = \mathrm{e}_1\ \mathbf{in}\ \mathrm{e}_2]\!]\text{. \textit{NB: Lazy Let!}} & (1)\\
={}& \mathcal{A}[\![k\ x_1\ x_2]\!]_{\rho_{\mathrm{e}}[k \mapsto k \& \mathcal{A}[\![\bar{\lambda} y.\bar{\lambda} z.y]\!]_{\rho_{\mathrm{e}}}]} && \text{\textit{Unf.} } \mathcal{A}[\![\_]\!],\ \rho_1 \triangleq \rho_{\mathrm{e}}[k \mapsto k \& \mathcal{A}[\![\bar{\lambda} y.\bar{\lambda} z.y]\!]_{\rho_{\mathrm{e}}}] & (2)\\
={}& \mathit{app}(\mathit{app}(\rho_1(k))(\rho_1(x_1)))(\rho_1(x_2)) && \text{\textit{Unfold} } \rho_1(k) & (3)\\
={}& \mathit{app}(\mathit{app}(k \ \&\ \mathcal{A}[\![\bar{\lambda} y.\bar{\lambda} z.y]\!]_{\rho_1})(\rho_1(x_1)))(\rho_1(x_2)) && \text{\textit{Unfold} } \mathcal{A}[\![\bar{\lambda}\mathrm{x.e}]\!] \text{ \textit{twice}, } \mathcal{A}[\![\mathrm{x}]\!] & (4)\\
={}& \mathit{app}(\mathit{app}(k \ \&\ \mathit{fun}_y(\lambda \theta_y.\ \mathit{fun}_z(\lambda \theta_z.\ \theta_y)))(\ldots))(\ldots) && \text{\textit{Unfold fun twice, simplify}} & (5)\\
={}& \mathit{app}(\mathit{app}(\langle [k \mapsto \mathsf{U}], \boxed{\mathsf{U}} \mathbin{⨾} \mathsf{A} \mathbin{⨾} \mathsf{Rep}\ \mathsf{U} \rangle)(\boxed{\rho_1(x_1)}))(\ldots) && \text{\textit{Unfold app}, } \rho_1(x_1) = \rho_{\mathrm{e}}(x_1)\text{\textit{, simplify}} & (6)\\
={}& \mathit{app}(\langle [k \mapsto \mathsf{U}, x_1 \mapsto \mathsf{U}], \boxed{\mathsf{A}} \mathbin{⨾} \mathsf{Rep}\ \mathsf{U} \rangle)(\boxed{\rho_1(x_2)}) && \text{\textit{Unfold app, simplify}} & (7)\\
={}& \langle [k \mapsto \mathsf{U}, x_1 \mapsto \mathsf{U}], \mathsf{Rep}\ \mathsf{U} \rangle && & (8)
\end{aligned}
$$

Let us look at the steps in a bit more detail. Step (1) extends the environment with an absence type for the let right-hand side of k. The steps up until (5) successively expose applications of the *app* and *fun* helper functions applied to environment entries for the involved variables. Step (5) then computes the summary as part of the absence type $\mathit{fun}_y(\lambda \theta_y.\ \mathit{fun}_z(\lambda \theta_z.\ \theta_y)) = \langle [], \mathsf{U} \mathbin{⨾} \mathsf{A} \mathbin{⨾} \mathsf{Rep}\ \mathsf{U} \rangle$. The Uses component is empty because $\bar{\lambda} y.\bar{\lambda} z.y$ has no free variables, and $k \,\&\, \ldots$ will add $[k \mapsto \mathsf{U}]$ as the single use. The *app* steps (6) and (7) simply zip up the uses of arguments $\rho_1(x_1)$ and $\rho_1(x_2)$ with the Absence flags in the summary $\mathsf{U} \mathbin{⨾} \mathsf{A} \mathbin{⨾} \mathsf{Rep}\ \mathsf{U}$ as highlighted, adding the Uses from $\rho_1(x_1) = \langle [x_1 \mapsto \mathsf{U}], \mathsf{Rep}\ \mathsf{U} \rangle$ but *not* from $\rho_1(x_2)$, because the first actual argument (x_1) is used whereas the second (x_2) is absent. The join on Uses follows pointwise from the order $\mathsf{A} \sqsubset \mathsf{U}$, i.e., $(\varphi_1 \sqcup \varphi_2)(\mathrm{x}) \triangleq \varphi_1(\mathrm{x}) \sqcup \varphi_2(\mathrm{x})$.

The analysis result $[k \mapsto \mathsf{U}, x_1 \mapsto \mathsf{U}]$ infers k and x_1 as potentially used and x_2 as absent, despite it occurring in argument position, thanks to the summary mechanism.

2.3 Function Summaries, Compositionality and Modularity

Instead of coming up with a summary mechanism, we could simply have "inlined" k during analysis of the example above to see that x_2 is absent in a simple first-order sense. The *call strings* approach to interprocedural program analysis [Sharir et al. 1978] turns this idea into a static analysis, and the AAM recipe could be used to derive a call strings-based absence analysis that is sound by construction. In this subsection, we argue that following this paths gives up on modularity, and thus leads to scalability problems in a compiler.

Let us clarify that by a *summary mechanism*, we mean a mechanism for approximating the semantics of a function call in terms of the domain of a static analysis, often yielding a symbolic, finite representation. In the definition of $\mathcal{A}[\![\_]\!]$, we took care to explicate the mechanism via *fun* and *app*. The former approximates a functional $(\lambda\theta.\ \ldots) : \mathsf{AbsTy} \to \mathsf{AbsTy}$ into a finite AbsTy, and *app* encodes the adjoint ("reverse") operation.[2]

To support efficient separate compilation, a compiler analysis must be *modular*, and summaries are indispensable in achieving that. Let us say that our example function $k = (\bar{\lambda} y.\bar{\lambda} z.y)$ is defined in module A and there is a use site $(k\ x_1\ x_2)$ in module B. Then a *modular analysis* must not reanalyse A.k at its use site in B. Our analysis $\mathcal{A}[\![\_]\!]$ facilitates that easily, because it can serialise the summarised AbsTy for k into module A's signature file. Do note that this would not have been possible for the functional $(\lambda\theta_y.\ \lambda\theta_z.\ \theta_y) : \mathsf{AbsTy} \to \mathsf{AbsTy} \to \mathsf{AbsTy}$ that describes the inline expansion of k, which a call strings-based analysis would need to invoke at every use site.

The same way summaries enable efficient *inter*-module compilation, they enable efficient *intra*-module compilation for *compositional* static analyses such as $\mathcal{A}[\![\_]\!]$.[3] Compositionality implies that $\mathcal{A}[\![\mathbf{let}\ f = \bar{\lambda} x.\mathsf{e}_{big}\ \mathbf{in}\ f\ f\ f\ f]\!]$ is a function of $\mathcal{A}[\![\bar{\lambda} x.\mathsf{e}_{big}]\!]$, itself a function of $\mathcal{A}[\![\mathsf{e}_{big}]\!]$. In order to satisfy the scalability requirements of a compiler and guarantee termination of the analysis in the first place, it is important not to repeat the work of analysing $\mathcal{A}[\![\mathsf{e}_{big}]\!]$ at every use site of f. Thus, it is necessary to summarise $\mathcal{A}[\![\bar{\lambda} x.\mathsf{e}_{big}]\!]$ into a finite AbsTy, rather than to call the inline expansion of type $\mathsf{AbsTy} \to \mathsf{AbsTy}$ multiple times, ruling out an analysis that is purely based on call strings.

2.4 Problem: Proving Soundness of Summary-Based Analyses

In this subsection, we demonstrate the difficulty of proving summary-based analyses sound.

Theorem 1 ($\mathcal{A}[\![\_]\!]$ infers absence). *If* $\mathcal{A}[\![\mathsf{e}]\!]_{\rho_{\mathsf{e}}} = \langle \varphi, \varsigma \rangle$ *and* $\varphi(\mathsf{x}) = \mathsf{A}$*, then* x *is absent in* e.

What are the main obstacles to prove it? As the first step, we must define what absence *means*, in a formal sense. There are many ways to do so, and it is not at all clear which is best. One plausible definition is in terms of the standard operational semantics in Section 3:

Definition 2 (Absence). *A variable* x *is* used *in an expression* e *if and only if there exists a trace* $(\mathbf{let}\ \mathsf{x} = \mathsf{e}'\ \mathbf{in}\ \mathsf{e}, \rho, \mu, \kappa) \hookrightarrow^* \ \ldots \xhookrightarrow{\textsc{Look}(\mathsf{x})} \ldots$ *that looks up the heap entry of* x*, i.e., it evaluates* x*. Otherwise,* x *is* absent *in* e.

Note that absence is a property of many different traces, each embedding the expression e in different machine contexts so as to justify rewrites via contextual improvement [Moran and Sands 1999]. Furthermore, we must prove sound the summary mechanism, captured in the following *substitution lemma* [Pierce 2002]:[4]

Lemma 3 (Substitution). $\mathcal{A}[\![\mathsf{e}]\!]_{\rho[\mathsf{x} \mapsto \rho(\mathsf{y})]} \sqsubseteq \mathcal{A}[\![(\bar{\lambda}\mathsf{x}.\mathsf{e})\ \mathsf{y}]\!]_{\rho}$.

Definition 2 and the substitution Lemma 3 will make a reappearance in Section 7. They are necessary components in a soundness proof, and substitution is not too difficult to prove for a simple summary mechanism. Building on these definitions, we may finally attempt the proof for Theorem 1. We suggest for the reader to have a cursory look by clicking on the theorem number, linking to the Appendix. The proof is exemplary of far more ambitious proofs such as in Sergey et al. [2017] and Breitner [2016, Section 4]. Though seemingly disparate, these proofs all follow an established preservation-style proof technique at heart.[5] The proof of Sergey et al. [2017] for a

[2] Proving that *fun* and *app* form a Galois connection is indeed important for a soundness proof and corresponds to a substitution Lemma 3.

[3] Cousot and Cousot [2002] understand modularity as degrees of compositionality.

[4] This statement amounts to $id \sqsubseteq app \circ fun_x$, one half of a Galois connection. The other half $fun_x \circ app \sqsubseteq id$ is eta-expansion $\mathcal{A}[\![\bar{\lambda}\mathsf{x}.\mathsf{e}\ \mathsf{x}]\!]_{\rho} \sqsubseteq \mathcal{A}[\![\mathsf{e}]\!]_{\rho}$.

[5] A "mundane approach" according to Nielson et al. [1999, Section 4.1], applicable to *trace properties*, but not to *hyperproperties* [Clarkson and Schneider 2010].

generalisation of $\mathcal{A}[\![\_]\!]$ is roughly structured as follows (non-clickable references to Figures and Lemmas below reference Sergey et al. [2017]):

(1) Instrument a standard call-by-need semantics (a variant of our reference in Section 3) such that heap lookups decrement a per-address counter; when heap lookup is attempted and the counter is 0, the machine is stuck. For absence, the instrumentation is simpler: the Look transition in Figure 2 carries the let-bound variable that is looked up.
(2) Give a declarative type system that characterises the results of the analysis (i.e., $\mathcal{A}[\![\_]\!]$) in a lenient (upwards closed) way. In case of Theorem 1, we define an analysis function on machine configurations for the proof.
(3) Prove that evaluation of well-typed terms in the instrumented semantics is bisimilar to evaluation of the term in the standard semantics, i.e., does not get stuck when the standard semantics would not. A classic *logical relation* [Nielson et al. 1999]. In our case, we prove that evaluation preserves the analysis result.

Alas, the effort in comprehending such a proof in detail, let alone formulating it, is enormous.

- The instrumentation (1) can be semantically non-trivial; for example the semantics in Sergey et al. [2017] becomes non-deterministic. Does this instrumentation still express the desired semantic property?
- Step (2) all but duplicates a complicated analysis definition (i.e., $\mathcal{A}[\![\_]\!]$) into a type system (in Figure 7) with subtle adjustments expressing invariants for the preservation proof.
- Furthermore, step (2) extends this type system to small-step machine configurations (in Figure 13), i.e., stacks and heaps, the scoping of which is mutually recursive.[6] Another page worth of Figures; the amount of duplicated proof artifacts is staggering. In our case, the analysis function on machine configurations is about as long as on expressions.
- This is all setup before step (3) proves interesting properties about the semantic domain of the analysis. Among the more interesting properties is the *substitution lemma* A.8 to be applied during beta reduction; exactly as in our proof.
- While proving that a single step $\sigma_1 \hookrightarrow \sigma_2$ preserves analysis information in step (3), we noticed that we actually got stuck in the Upd case, and would need to redo the proof using step-indexing [Appel and McAllester 2001]. In our experience this case hides the thorniest of surprises; that was our experience while proving Theorem 56 which gives a proper account. Although the proof in Sergey et al. [2017] is perceived as detailed and rigorous, it is quite terse in the corresponding EUpd case of the single-step safety proof in lemma A.6.

The main takeaway: Although analysis and semantics might be reasonably simple, the soundness proof that relates both is *not*; it necessitates an explosion in formal artefacts and the parts of the proof that concern the domain of the analysis are drowned in coping with semantic subtleties that ultimately could be shared with similar analyses. Furthermore, the inevitable hand-waving in proofs of this size around said semantic subtleties diminishes confidence in the soundness of the proof to the point where trust can only be recovered by full mechanisation.

It would be preferable to find a framework to *prove these distractions rigorously and separately*, once and for all, and then instantiate this framework for absence analysis or cardinality analysis, so that only the highlights of the preservation proof such as the substitution lemma need to be shown.

Abstract interpretation provides such a framework. Alas, the book of Cousot [2021] starts from a *compositional* semantics to derive compositional analyses, but small-step operational semantics are non-compositional! This begs the question if we could have started from a compositional denotational semantics. While we could have done so for absence or strictness analysis, denotational

[6] We believe that this extension can always be derived systematically from a context lemma [Moran and Sands 1999, Lemma 3.2] and imitating what the type system does on the closed expression derivable from a configuration via the context lemma.

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

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

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

semantics is insufficient to express *operational properties* such as *usage cardinality*, i.e., "e evaluates x at most u times", but usage cardinality is the entire point of the analysis in Sergey et al. [2017].[7]

For these reasons, we set out to find a ***compositional semantics that exhibits operational detail*** just like the trace-generating semantics of Cousot [2021], and were successful. The example of usage analysis in Section 6 (generalising $\mathcal{A}[\![\_]\!]$, as suggested above) demonstrates that we can ***derive summary-based analyses as an abstract interpretation*** from our semantics. Since both semantics and analysis are derived from the same compositional generic interpreter, the equivalent of the preservation proof for usage analysis in Lemma 9 takes no more than a substitution lemma and a bit of plumbing. Hence our *denotational interpreter* does not only enjoy useful compositional semantics and analyses as instances, the soundness proofs become compositional in the semantic domain as well.

3 REFERENCE SEMANTICS: LAZY KRIVINE MACHINE

Before we get to introduce our novel denotational interpreters, let us recall the semantic ground truth of this work and others [Breitner 2016; Sergey et al. 2017]: The Mark II machine of Sestoft [1997] given in Figure 2, a small-step operational semantics. It is a Lazy Krivine (LK) machine implementing call-by-need. (A close sibling for call-by-value would be a CESK machine [Felleisen and Friedman 1987].) A reasonable call-by-name semantics can be recovered by removing the Upd rule and the pushing of update frames in Look. Furthermore, we will ignore $\textsc{Case}_1$ and $\textsc{Case}_2$ in this section because we do not consider data types for now.

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. There is one harmless non-standard extension: For Look transitions, we take note of the let-bound variable y which allocated the heap binding that the machine is about to look up. The association from address to let-bound variable is maintained in the first component of a heap entry triple and requires slight adjustments of the $\textsc{Let}_1$, Look and Upd rules.

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 $\mathsf{dom}(f)$ is finite and $\mathsf{rng}(f)$ denotes its range. The literal

[7]Useful applications of the "at most once" cardinality are given in Sergey et al. [2017]; Turner et al. [1995], motivating inlining into function bodies that are called at most once, for example.

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 $\mathit{init}(\mathrm{e}) = (\mathrm{e}, [], [], \mathbf{stop})$ and the final machine states are of the form $(\mathrm{v}, \_, \_, \mathbf{stop})$. We bake into $\sigma \in \mathbb{S}$ the simplifying invariant of *well-addressedness*: Any address a occurring 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.

We conclude with two example traces. The first one evaluates $\mathbf{let}\ i = \bar{\lambda}x.x\ \mathbf{in}\ i\ i$:

$$\begin{array}{llllll}
(\mathbf{let}\ i = \bar{\lambda}x.x\ \mathbf{in}\ i\ i, [], [], \mathbf{stop}) & \xhookrightarrow{\textsc{Let}_1} & (i\ i, \rho_1, \mu, \mathbf{stop}) & \xhookrightarrow{\textsc{App}_1} & (i, \rho_1, \mu, \kappa) & \xhookrightarrow{\textsc{Look}(i)} \\
\boxed{(\bar{\lambda}x.x, \rho_1, \mu, \mathbf{upd}(\mathrm{a}_1) \cdot \kappa)} & \boxed{\xhookrightarrow{\textsc{Upd}}} & (\bar{\lambda}x.x, \rho_1, \mu, \kappa) & \xhookrightarrow{\textsc{App}_2} & (x, \rho_2, \mu, \mathbf{stop}) & \xhookrightarrow{\textsc{Look}(i)} \\
\boxed{(\bar{\lambda}x.x, \rho_1, \mu, \mathbf{upd}(\mathrm{a}_1) \cdot \mathbf{stop})} & \boxed{\xhookrightarrow{\textsc{Upd}}} & (\bar{\lambda}x.x, \rho_1, \mu, \mathbf{stop}) & & &
\end{array} \tag{1}$$

$$\text{where}\ \ \kappa = \mathbf{ap}(\mathrm{a}_1) \cdot \mathbf{stop}, \quad \rho_1 = [i \mapsto \mathrm{a}_1], \quad \rho_2 = [i \mapsto \mathrm{a}_1, x \mapsto \mathrm{a}_1], \quad \mu = [\mathrm{a}_1 \mapsto (i, \rho_1, \bar{\lambda}x.x)]$$

The corresponding by-name trace simply omits the highlighted update steps. The second example evaluates $\mathrm{e} \triangleq \mathbf{let}\ i = (\bar{\lambda}y.\bar{\lambda}x.x)\ i\ \mathbf{in}\ i\ i$, demonstrating memoisation of i:

$$\begin{array}{l}
(\mathrm{e}, [], [], \mathbf{stop}) \xhookrightarrow{\textsc{Let}_1} (i\ i, \rho_1, \mu_1, \mathbf{stop}) \xhookrightarrow{\textsc{App}_1} (i, \rho_1, \mu_1, \kappa_1) \xhookrightarrow{\textsc{Look}(i)} ((\bar{\lambda}y.\bar{\lambda}x.x)\ i, \rho_1, \mu_1, \kappa_2) \\
\xhookrightarrow{\textsc{App}_1} (\bar{\lambda}y.\bar{\lambda}x.x, \rho_1, \mu_1, \mathbf{ap}(\mathrm{a}_1) \cdot \kappa_2) \xhookrightarrow{\textsc{App}_2} (\bar{\lambda}x.x, \rho_2, \mu_1, \kappa_2) \xhookrightarrow{\textsc{Upd}} (\bar{\lambda}x.x, \rho_2, \mu_2, \kappa_1) \\
\xhookrightarrow{\textsc{App}_2} (x, \rho_3, \mu_2, \mathbf{stop}) \xhookrightarrow{\textsc{Look}(i)} (\bar{\lambda}x.x, \rho_2, \mu_2, \mathbf{upd}(\mathrm{a}_1) \cdot \mathbf{stop}) \xhookrightarrow{\textsc{Upd}} (\bar{\lambda}x.x, \rho_2, \mu_2, \mathbf{stop})
\end{array} \tag{2}$$

$$\text{where}\ \begin{array}{l} \rho_1 = [i \mapsto \mathrm{a}_1], \qquad \rho_2 = [i \mapsto \mathrm{a}_1, y \mapsto \mathrm{a}_1], \quad \rho_3 = [i \mapsto \mathrm{a}_1, y \mapsto \mathrm{a}_1, x \mapsto \mathrm{a}_1], \\ \mu_1 = (\rho_1, (i, \bar{\lambda}y.\bar{\lambda}x.x)\ i), \mu_2 = [\mathrm{a}_1 \mapsto (i, \rho_2, \bar{\lambda}x.x)], \kappa_1 = \mathbf{ap}(\mathrm{a}_1) \cdot \mathbf{stop}, \kappa_2 = \mathbf{upd}(\mathrm{a}_1) \cdot \kappa_1 \end{array}$$

4 A DENOTATIONAL INTERPRETER

In this section, we present the main contribution of this work, namely a generic *denotational interpreter*[8] for a functional language which we can instantiate with different semantic domains. The choice of semantic domain determines the *evaluation strategy* (call-by-name, call-by-value, call-by-need) and the degree to which *operational detail* can be observed. Yet different semantic domains give rise to useful *summary-based* static analyses such as usage analysis in Section 6, all from the same interpreter skeleton. Our generic denotational interpreter enable sharing of soundness proofs, thus drastically simplifying the soundness proof obligation per derived analysis (Section 7).

Denotational interpreters can be implemented in any higher-order language such as OCaml, Scheme or Java with explicit thunks, but we picked Haskell for convenience.[9]

4.1 Semantic Domain

Just as traditional denotational semantics, denotational interpreters assign meaning to programs in some *semantic domain*. Traditionally, the semantic domain $\mathbb{D}$ comprises *semantic values* such as base values (integers, strings, etc.) and functions $\mathbb{D} \to \mathbb{D}$. One of the main features of these semantic domains is that they lack *operational*, or, *intensional detail* that is unnecessary to assigning each

[8]This term was coined by Might [2010]. We find it fitting, because a denotational interpreter is both a *denotational semantics* [Scott and Strachey 1971] as well as a total *definitional interpreter* [Reynolds 1972].

[9]We extract from this document a runnable Haskell file which we add as a Supplement, containing the complete definitions. Furthermore, the (terminating) interpreter outputs are directly generated from this extract.

$$
\begin{array}{l}
\textbf{data}\ \mathsf{Exp} \\
\quad = \mathsf{Var}\ \mathsf{Name} \mid \mathsf{Let}\ \mathsf{Name}\ \mathsf{Exp}\ \mathsf{Exp} \\
\quad \mid \mathsf{Lam}\ \mathsf{Name}\ \mathsf{Exp} \mid \mathsf{App}\ \mathsf{Exp}\ \mathsf{Name} \\
\quad \mid \mathsf{ConApp}\ \mathsf{Tag}\ [\mathsf{Name}] \mid \mathsf{Case}\ \mathsf{Exp}\ \mathsf{Alts} \\
\textbf{type}\ \mathsf{Name} = \mathsf{String} \\
\textbf{type}\ \mathsf{Alts} = \mathsf{Tag} :\rightharpoonup ([\mathsf{Name}], \mathsf{Exp}) \\
\textbf{data}\ \mathsf{Tag} = ...; \mathit{conArity} :: \mathsf{Tag} \rightarrow \mathsf{Int}
\end{array}
$$

Fig. 3. Syntax

$$
\begin{array}{l}
\textbf{type}\ (:\rightharpoonup) = \mathsf{Map}; \varepsilon :: \mathsf{Ord}\ k \Rightarrow k :\rightharpoonup v \\
\_[\_ \mapsto \_] :: \mathsf{Ord}\ k \Rightarrow (k :\rightharpoonup v) \rightarrow k \rightarrow v \rightarrow (k :\rightharpoonup v) \\
\_[\overline{\_ \mapsto \_}] :: \mathsf{Ord}\ k \Rightarrow (k :\rightharpoonup v) \rightarrow [k] \rightarrow [v] \\
\qquad\qquad\qquad\qquad \rightarrow (k :\rightharpoonup v) \\
(!) :: \mathsf{Ord}\ k \Rightarrow (k :\rightharpoonup v) \rightarrow k \rightarrow v \\
\mathit{dom} :: \mathsf{Ord}\ k \Rightarrow (k :\rightharpoonup v) \rightarrow \mathsf{Set}\ k \\
(\in) :: \mathsf{Ord}\ k \Rightarrow k \rightarrow \mathsf{Set}\ k \rightarrow \mathsf{Bool} \\
(\lhd) :: (b \rightarrow c) \rightarrow (a :\rightharpoonup b) \rightarrow (a :\rightharpoonup c) \\
\mathit{assocs} :: (k :\rightharpoonup v) \rightarrow [(k, v)]
\end{array}
$$

Fig. 4. Environments

observationally distinct expression a distinct meaning. For example, it is not possible to observe evaluation cardinality, which is the whole point of analyses such as usage analysis (Section 6).

A distinctive feature of our work is that our semantic domains are instead *traces* that describe the *steps* taken by an abstract machine, and that *end* in semantic values. It is possible to describe usage cardinality as a property of the traces thus generated, as required for a soundness proof of usage analysis. We choose $\mathsf{D}_{\mathbf{na}}$, defined below, as the first example of such a semantic domain, because it is simple and illustrative of the approach. Instantiated at $\mathsf{D}_{\mathbf{na}}$, our generic interpreter will produce precisely the traces of the by-name variant of the Krivine machine in Figure 2.

We can define the semantic domain $\mathsf{D}_{\mathbf{na}}$ for a call-by-***na***me variant of our language as follows:[10]

$$
\begin{array}{l}
\textbf{type}\ \mathsf{D}\ \tau = \tau\ (\mathsf{Value}\ \tau); \textbf{type}\ \mathsf{D}_{\mathbf{na}} = \mathsf{D}\ \mathsf{T} \\
\textbf{data}\ \mathsf{T}\ v = \mathsf{Step}\ \mathsf{Event}\ (\mathsf{T}\ v) \mid \mathsf{Ret}\ v \\
\textbf{data}\ \mathsf{Event} = \mathsf{Lookup}\ \mathsf{Name} \mid \mathsf{Update} \mid \mathsf{App}_1 \mid \mathsf{App}_2 \\
\qquad\qquad\quad \mid \mathsf{Let}_0 \mid \mathsf{Let}_1 \mid \mathsf{Case}_1 \mid \mathsf{Case}_2 \\
\textbf{data}\ \mathsf{Value}\ \tau = \mathsf{Stuck} \mid \mathsf{Fun}\ (\mathsf{D}\ \tau \rightarrow \mathsf{D}\ \tau) \mid \mathsf{Con}\ \mathsf{Tag}\ [\mathsf{D}\ \tau]
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Monad}\ \mathsf{T}\ \textbf{where} \\
\quad \mathit{return}\ v = \mathsf{Ret}\ v \\
\quad \mathsf{Ret}\ v \gg\!= k = k\ v \\
\quad \mathsf{Step}\ e\ \tau \gg\!= k = \mathsf{Step}\ e\ (\tau \gg\!= k)
\end{array}
$$

A trace T either returns a value (Ret) or makes a small-step transition (Step). Each step Step *ev rest* is decorated with an event *ev*, which describes what happens in that step. For example, event Lookup x describes the lookup of variable x :: Name in the environment. Note that the choice of Event is use-case (i.e. analysis) specific and suggests a spectrum of intensionality, with **data** Event = Unit on the more abstract end of the spectrum and arbitrary syntactic detail attached to each of Event's constructors at the intensional end of the spectrum.[11]

A trace in $\mathsf{D}_{\mathbf{na}} = \mathsf{T}\ (\mathsf{Value}\ \mathsf{T})$ eventually terminates with a Value that is either stuck (Stuck), a function waiting to be applied to a domain value (Fun), or a constructor constructor application giving the denotations of its fields (Con). We postpone worries about well-definedness and totality of this encoding to Section 5.2.

4.2 The Interpreter

Traditionally, a denotational semantics is expressed as a mathematical function, often written $[\![e]\!]_\rho$, to give an expression e :: Exp a meaning, or *denotation*, in terms of some semantic domain

[10] For a realistic implementation, we would define D as a **newtype** to keep type class resolution decidable and non-overlapping. We will however stick to a **type** synonym in this presentation in order to elide noisy wrapping and unwrapping of constructors.

[11] If our language had facilities for input/output and more general side-effects, we could have started from a more elaborate trace construction such as (guarded) interaction trees [Frumin et al. 2023; Xia et al. 2019].

```
𝒮⟦_⟧_ :: (Trace d, Domain d, HasBind d)
       ⇒ Exp → (Name :⇀ d) → d
𝒮⟦e⟧ρ = case e of
  Var x | x ∈ dom ρ → ρ ! x
        | otherwise  → stuck
  Lam x body → fun x $ λd →
    step App₂ (𝒮⟦body⟧(ρ[x↦d]))
  App e x | x ∈ dom ρ → step App₁ $
              apply (𝒮⟦e⟧ρ) (ρ ! x)
          | otherwise  → stuck
  Let x e₁ e₂ → bind
    (λd₁ → 𝒮⟦e₁⟧ρ[x↦step (Lookup x) d₁])
    (λd₁ → step Let₁ (𝒮⟦e₂⟧ρ[x↦step (Lookup x) d₁]))
  ConApp k xs
    | all (∈ dom ρ) xs, length xs ≡ conArity k
    → con k (map (ρ!) xs)
    | otherwise
    → stuck
  Case e alts → step Case₁ $
    select (𝒮⟦e⟧ρ) (cont ◁ alts)
    where
      cont (xs, eᵣ) ds | length xs ≡ length ds
                       = step Case₂ (𝒮⟦eᵣ⟧ρ[xs↦ds])
                       | otherwise
                       = stuck
```

```
class Trace d where
  step :: Event → d → d
class Domain d where
  stuck :: d
  fun :: Name → (d → d) → d
  apply :: d → d → d
  con :: Tag → [d] → d
  select :: d → (Tag :⇀ ([d] → d)) → d
class HasBind d where
  bind :: (d → d) → (d → d) → d
```

(a) Interface of traces and values

```
instance Trace (T v) where
  step = Step
instance Monad τ ⇒ Domain (D τ) where
  stuck = return Stuck
  fun _ f = return (Fun f)
  apply d a = d ≫= λv → case v of
    Fun f → f a; _ → stuck
  con k ds = return (Con k ds)
  select dv alts = dv ≫= λv → case v of
    Con k ds | k ∈ dom alts → (alts ! k) ds
    _                       → stuck
instance HasBind D_na where
  bind rhs body = let d = rhs d in body d
```

(b) Concrete by-name semantics for D_{na}

Fig. 5. Abstract Denotational Interpreter

D. The environment $\rho :: \mathsf{Name} :\rightharpoonup \mathsf{D}$ gives meaning to the free variables of e, by mapping each free variable to its denotation in D. We sketch the Haskell encoding of Exp in Figure 3 and the API of environments and sets in Figure 4. For concise notation, we will use a small number of infix operators: $(:\rightharpoonup)$ as a synonym for finite Maps, with $m \,!\, x$ for looking up x in m, ε for the empty map, $m[x \mapsto d]$ for updates, $\mathit{assocs}\ m$ for a list of key-value pairs in m, $f \lhd m$ for mapping f over every value in m, $\mathit{dom}\ m$ for the set of keys present in the map, and $(\in)$ for membership tests in that set.

Our denotational interpreter $\mathcal{S}[\![\_]\!]_{\_} :: \mathsf{Exp} \to (\mathsf{Name} :\rightharpoonup \mathsf{D}_{\mathbf{na}}) \to \mathsf{D}_{\mathbf{na}}$ can have a similar type as $[\![\_]\!]_{\_}$. However, to derive both dynamic semantics and static analysis as instances of the same generic interpreter $\mathcal{S}[\![\_]\!]_{\_}$, we need to vary the type of its semantic domain, which is naturally expressed using type-class overloading, thus:

$$\mathcal{S}[\![\_]\!]_{\_} :: (\mathsf{Trace}\ d, \mathsf{Domain}\ d, \mathsf{HasBind}\ d) \Rightarrow \mathsf{Exp} \to (\mathsf{Name} :\rightharpoonup d) \to d.$$

We have parameterised the semantic domain d over three type classes Trace, Domain and HasBind, whose signatures are given in Figure 5a.[12] Each of the three type classes offer knobs that we will tweak to derive different evaluation strategies as well as static analyses.

Figure 5 gives the complete definition of $\mathcal{S}[\![\_]\!]_\_$ together with instances for domain $\mathsf{D}_{\mathbf{na}}$ that we introduced in Section 4.1. Together this is enough to actually run the denotational interpreter to produce traces. We use $read :: \mathsf{String} \to \mathsf{Exp}$ as a parsing function, and a Show instance for $\mathsf{D}\ \tau$ that displays traces. For example, we can evaluate the expression **let** $i = \bar{\lambda} x.x$ **in** $i\ i$ like this:

$$\lambda\!\!> \mathcal{S}[\![read\ \texttt{"let i = λx.x in i i"}]\!]_\varepsilon :: \mathsf{D}_{\mathbf{na}}$$

$$\textsc{Let}_1 \hookrightarrow \textsc{App}_1 \hookrightarrow \textsc{Look}(i) \hookrightarrow \textsc{App}_2 \hookrightarrow \textsc{Look}(i) \hookrightarrow \langle\lambda\rangle,$$

where $\langle\lambda\rangle$ means that the trace ends in a Fun value. We cannot print $\mathsf{D}_{\mathbf{na}}$s or Functions thereof, but in this case the result would be the value $\bar{\lambda} x.x$. This is in direct correspondence to the earlier call-by-name small-step trace (1) in Section 3.

The definition of $\mathcal{S}[\![\_]\!]_\_$, given in Figure 5, is by structural recursion over the input expression. For example, to get the denotation of $\mathsf{Lam}\ x\ body$, we must recursively invoke $\mathcal{S}[\![\_]\!]_\_$ on $body$, extending the environment to bind x to its denotation. We wrap that body denotation in $step$ App_2, to prefix the trace of $body$ with an App_2 event whenever the function is invoked, where $step$ is a method of class Trace. Finally, we use fun to build the returned denotation; the details necessarily depend on the Domain, so fun is a method of class Domain. While the lambda-bound $x :: \mathsf{Name}$ passed to fun is ignored in in the Domain $\mathsf{D}_{\mathbf{na}}$ instance of the concrete by-name semantics, it is useful for abstract domains such as that of usage analysis (Section 6). The other cases follow a similar pattern; they each do some work, before handing off to type class methods to do the domain-specific work.

The HasBind type class defines a particular *evaluation strategy*, as we shall see in Section 4.3. The $bind$ method of HasBind is used to give meaning to recursive let bindings: it takes two functionals for building the denotation of the right-hand side and that of the let body, given a denotation for the right-hand side. The concrete implementation for $bind$ given in Figure 5b computes a d such that $d = rhs\ d$ and passes the recursively-defined d to $body$.[13] Doing so yields a call-by-name evaluation strategy, because the trace d will be unfolded at every occurrence of x in the right-hand side e_1. We will shortly see examples of eager evaluation strategies that will yield from d inside $bind$ instead of calling $body$ immediately.

We conclude this subsection with a few examples. First we demonstrate that our interpreter is *productive*: we can observe prefixes of diverging traces without risking a looping interpreter. To observe prefixes, we use a function $takeT :: \mathsf{Int} \to \mathsf{T}\ v \to \mathsf{T}\ (\mathsf{Maybe}\ v)$: $takeT\ n\ \tau$ returns the first n steps of τ and replaces the final value with Nothing (printed as ...) if it goes on for longer.

$$\lambda\!\!> takeT\ 5\ \$\ \mathcal{S}[\![read\ \texttt{"let x = x in x"}]\!]_\varepsilon :: \mathsf{T}\ (\mathsf{Maybe}\ (\mathsf{Value}\ \mathsf{T}))$$

$$\textsc{Let}_1 \hookrightarrow \textsc{Look}(x) \hookrightarrow \textsc{Look}(x) \hookrightarrow \textsc{Look}(x) \hookrightarrow \textsc{Look}(x) \hookrightarrow \ldots$$

$$\lambda\!\!> takeT\ 9\ \$\ \mathcal{S}[\![read\ \texttt{"let w = λy. y y in w w"}]\!]_\varepsilon :: \mathsf{T}\ (\mathsf{Maybe}\ (\mathsf{Value}\ \mathsf{T}))$$

$$\textsc{Let}_1 \hookrightarrow \textsc{App}_1 \hookrightarrow \textsc{Look}(w) \hookrightarrow \textsc{App}_2 \hookrightarrow \textsc{App}_1 \hookrightarrow \textsc{Look}(w) \hookrightarrow \textsc{App}_2 \hookrightarrow \textsc{App}_1 \hookrightarrow \textsc{Look}(w) \hookrightarrow \ldots$$

[12] One can think of these type classes as a fold-like final encoding [Carette et al. 2007] of a domain. However, the significance is in the *decomposition* of the domain, not the choice of encoding.

[13] Such a d corresponds to the *guarded fixpoint* of rhs. Strict languages can define this fixpoint as $d\ () = rhs\ (d\ ())$.

$$
\begin{array}{l}
\mathcal{S}_{\mathbf{name}}[\![e]\!]_\rho = \mathcal{S}[\![e]\!]_\rho :: \mathsf{D}\ (\mathsf{ByName}\ \mathsf{T}) \\
\mathbf{newtype}\ \mathsf{ByName}\ \tau\ v = \mathsf{ByName}\ \{\mathit{unByName} :: \tau\ v\} \\
\mathbf{instance}\ \mathsf{Monad}\ \tau \Rightarrow \mathsf{Monad}\ (\mathsf{ByName}\ \tau)\ \mathbf{where}\ \ldots \\
\mathbf{instance}\ \mathsf{Trace}\ (\tau\ v) \Rightarrow \mathsf{Trace}\ (\mathsf{ByName}\ \tau\ v)\ \mathbf{where}\ \ldots \\
\mathbf{instance}\ \mathsf{HasBind}\ (\mathsf{D}\ (\mathsf{ByName}\ \tau))\ \mathbf{where}\ \ldots
\end{array}
$$

Fig. 6. Redefinition of call-by-name semantics from Figure 5b

The reason $\mathcal{S}[\![\_]\!]_\_$ is productive is due to the coinductive nature of T's definition in Haskell.[14] Productivity requires that the monadic bind operator ($\gg\!=$) for T guards the recursion, as in the delay monad of Capretta [2005].

Data constructor values are printed as $Con(K)$, where K indicates the Tag. Data types allow for interesting ways (type errors) to get Stuck (i.e., the **wrong** value of Milner [1978]), printed as $\lightning$:

$\lambda\!>\ \mathcal{S}[\![\mathit{read}$ `"let zro = Z() in let one = S(zro) in case one of { S(z) -> z }"` $]\!]_\varepsilon :: \mathsf{D}_{\mathbf{na}}$

$\textsc{Let}_1 \hookrightarrow \textsc{Let}_1 \hookrightarrow \textsc{Case}_1 \hookrightarrow \textsc{Look}(\mathit{one}) \hookrightarrow \textsc{Case}_2 \hookrightarrow \textsc{Look}(\mathit{zro}) \hookrightarrow \langle Con(Z)\rangle$

$\lambda\!>\ \mathcal{S}[\![\mathit{read}$ `"let zro = Z() in zro zro"` $]\!]_\varepsilon :: \mathsf{D}_{\mathbf{na}}$

$\textsc{Let}_1 \hookrightarrow \textsc{App}_1 \hookrightarrow \textsc{Look}(\mathit{zro}) \hookrightarrow \langle \lightning \rangle$

4.3 More Evaluation Strategies

By varying the HasBind instance of our type D, we can endow our language Exp with different evaluation strategies. The appeal of that is, firstly, that it is possible to do so! Furthermore, we thus introduce the — to our knowledge — first provably adequate denotational semantics for call-by-need. We will go on to prove usage analysis sound wrt. by-need evaluation in Section 7. The different by-value semantics demonstrate versatility, in that our approach is applicable to strict languages as well and thus can be used to study the differences between by-need and by-value evaluation.

Following a similar approach as Darais et al. [2017], we maximise reuse by instantiating the same D at different wrappers of T, rather than reinventing Value and T.

4.3.1 Call-by-name. We redefine by-name semantics via the ByName *trace transformer* in Figure 6, so called because ByName τ inherits its Monad and Trace instance from τ and in reminiscence of Darais et al. [2015]. The old $\mathsf{D}_{\mathbf{na}}$ can be recovered as D (ByName T) and we refer to its interpreter instance as $\mathcal{S}_{\mathbf{name}}[\![e]\!]_\rho$.

4.3.2 Call-by-need. The use of a stateful heap is essential to the call-by-need evaluation strategy in order to enable memoisation. So how do we vary θ such that D θ accommodates state? We certainly cannot perform the heap update by updating entries in ρ, because those entries are immutable once inserted, and we do not want to change the generic interpreter. That rules out $\theta \cong \mathsf{T}$ (as for ByName T), because then repeated occurrences of the variable x must yield the same trace $\rho\,!\,x$. However, the whole point of memoisation is that every evaluation of x after the first one leads to a potentially different, shorter trace. This implies we have to *paramaterise* every occurrence of x over the current heap μ at the time of evaluation, and every evaluation of x must subsequently update this heap with its value, so that the next evaluation of x returns the value directly. In other words, we need a representation $\mathsf{D}\ \theta \cong \mathsf{Heap} \to \mathsf{T}\ (\mathsf{Value}\ \theta, \mathsf{Heap})$.

[14]In a strict language, we need to introduce a thunk in the definition of Step, e.g., `Step of event * (unit -> 'a t)`.

```
S_need⟦e⟧_ρ(μ) = unByNeed (S⟦e⟧_ρ :: D (ByNeed T)) μ
type Addr = Int; type Heap τ = Addr :⇀ D τ; nextFree :: Heap τ → Addr
newtype ByNeed τ v = ByNeed {unByNeed :: Heap (ByNeed τ) → τ (v, Heap (ByNeed τ))}
get :: Monad τ ⇒ ByNeed τ (Heap (ByNeed τ));        get   = ByNeed (λμ → return (μ, μ))
put :: Monad τ ⇒ Heap (ByNeed τ) → ByNeed τ ();  put μ = ByNeed (λ_ → return ((), μ))
instance Monad τ ⇒ Monad (ByNeed τ) where ...
instance (∀v. Trace (τ v)) ⇒ Trace (ByNeed τ v) where step e m = ByNeed (step e ∘ unByNeed m)
fetch :: Monad τ ⇒ Addr → D (ByNeed τ); fetch a = get >>= λμ → μ ! a
memo :: ∀τ. (Monad τ, ∀v. Trace (τ v)) ⇒ Addr → D (ByNeed τ) → D (ByNeed τ)
memo a d = d >>= λv → ByNeed (upd v)
  where upd Stuck μ = return (Stuck :: Value (ByNeed τ), μ)
        upd v     μ = step Update (return (v, μ[a ↦ memo a (return v)]))
instance (Monad τ, ∀v. Trace (τ v)) ⇒ HasBind (D (ByNeed τ)) where
  bind rhs body = do μ ← get
                     let a = nextFree μ
                     put μ[a ↦ memo a (rhs (fetch a))]
                     body (fetch a)
```

Fig. 7. Call-by-need

Our trace transformer ByNeed in Figure 7 solves this type equation via $\theta \triangleq$ ByNeed T. It embeds a standard state transformer monad,[15] whose key operations *get* and *put* are given in Figure 7.

So the denotation of an expression is no longer a trace, but rather a *stateful function returning a trace* with state Heap (ByNeed τ) in which to allocate call-by-need thunks. The Trace instance of ByNeed τ simply forwards to that of τ (i.e., often T), pointwise over heaps. Doing so needs a Trace instance for τ (Value (ByNeed τ), Heap (ByNeed τ)), but we found it more succinct to use a quantified constraint ($\forall v$. Trace (τ v)), that is, we require a Trace (τ v) instance for every choice of v. Given that τ must also be a Monad, that is not an onerous requirement.

The key part is again the implementation of HasBind for D (ByNeed τ), because that is the only place where thunks are allocated. The implementation of *bind* designates a fresh heap address *a* to hold the denotation of the right-hand side. Both *rhs* and *body* are called with *fetch a*, a denotation that looks up *a* in the heap and runs it. If we were to omit the *memo a* action explained next, we would thus have recovered another form of call-by-name semantics based on mutable state instead of guarded fixpoints such as in ByName and ByValue. The whole purpose of the *memo a d* combinator then is to *memoise* the computation of *d* the first time we run the computation, via *fetch a* in the Var case of $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_(\_)$. So *memo a d* yields from *d* until it has reached a value, and then *upd*ates the heap after an additional Update step. Repeated access to the same variable will run the replacement *memo a* (*return v*), which immediately yields *v* after performing a *step* Update that does nothing.[16]

Although the code is carefully written, it is worth stressing how compact and expressive it is. We were able to move from traces to stateful traces just by wrapping traces T in a state transformer

[15] Indeed, we derive its monad instance via StateT (Heap (ByNeed τ)) τ [Blöndal et al. 2018].

[16] More serious semantics would omit updates after the first evaluation as an *optimisation*, i.e., update with $\mu[a \mapsto return\ v]$, but doing so complicates relating the semantics to Figure 2, where omission of update frames for values behaves differently. For now, our goal is not to formalise this optimisation, but rather to show adequacy wrt. an established semantics.

$$\mathcal{S}_{\mathbf{value}}[\![e]\!]_\rho = \mathcal{S}[\![e]\!]_\rho :: \mathsf{D}\ (\mathsf{ByValue}\ \mathsf{T})$$
$$\begin{array}{l}
\mathbf{newtype}\ \mathsf{ByValue}\ \tau\ v = \mathsf{ByValue}\ \{ \mathit{unByValue} :: \tau\ v \} \\
\mathbf{instance}\ \mathsf{Monad}\ \tau \Rightarrow \mathsf{Monad}\ (\mathsf{ByValue}\ \tau)\ \mathbf{where}\ \ldots \\
\mathbf{instance}\ \mathsf{Trace}\ (\tau\ v) \Rightarrow \mathsf{Trace}\ (\mathsf{ByValue}\ \tau\ v)\ \mathbf{where}\ \ldots
\end{array}$$
$$\begin{array}{l}
\mathbf{class}\ \mathsf{Extract}\ \tau\ \mathbf{where}\ \mathit{getValue} :: \tau\ v \to v \\
\mathbf{instance}\ \mathsf{Extract}\ \mathsf{T}\ \mathbf{where}\ \mathit{getValue}\ (\mathsf{Ret}\ v) = v; \mathit{getValue}\ (\mathsf{Step}\ _\ \tau) = \mathit{getValue}\ \tau
\end{array}$$
$$\begin{array}{l}
\mathbf{instance}\ (\mathsf{Trace}\ (\mathsf{D}\ (\mathsf{ByValue}\ \tau)), \mathsf{Monad}\ \tau, \mathsf{Extract}\ \tau) \Rightarrow \mathsf{HasBind}\ (\mathsf{D}\ (\mathsf{ByValue}\ \tau))\ \mathbf{where} \\
\quad \mathit{bind}\ \mathit{rhs}\ \mathit{body} = \mathit{step}\ \mathsf{Let}_0\ (\mathbf{do}\ v_1 \leftarrow d; \mathit{body}\ (\mathit{return}\ v_1)) \\
\qquad\qquad\qquad\qquad \mathbf{where}\ d = \mathit{rhs}\ (\mathit{return}\ v) \qquad\qquad :: \mathsf{D}\ (\mathsf{ByValue}\ \tau) \\
\qquad\qquad\qquad\qquad\qquad\quad v = \mathit{getValue}\ (\mathit{unByValue}\ d) :: \mathsf{Value}\ (\mathsf{ByValue}\ \tau)
\end{array}$$

Fig. 8. Call-by-value

ByNeed, without modifying the main $\mathcal{S}[\![\_]\!]_\_$ function at all. In doing so, we provide the simplest encoding of a denotational by-need semantics that we know of.[17]

Here is an example evaluating **let** $i = (\bar{\lambda} y.\bar{\lambda} x.x)\ i$ **in** $i\ i$, starting in an empty heap:

$\lambda\!>\ \mathcal{S}_{\mathbf{need}}[\![\mathit{read}$ `"let i = (λy.λx.x) i in i i"`$]\!]_\varepsilon(\varepsilon) :: \mathsf{T}\ (\mathsf{Value}\ \_, \mathsf{Heap}\ \_)$

$\textsc{Let}_1 \hookrightarrow \textsc{App}_1 \hookrightarrow \textsc{Look}(i) \hookrightarrow \textsc{App}_1 \hookrightarrow \textsc{App}_2 \hookrightarrow \textsc{Upd} \hookrightarrow \textsc{App}_2 \hookrightarrow \textsc{Look}(i) \hookrightarrow \textsc{Upd} \hookrightarrow \langle(\lambda, [0{\mapsto}\_])\rangle$

This trace is in clear correspondence to the earlier by-need LK trace (2). We can observe memoisation at play: Between the first bracket of Look and Upd events, the heap entry for i goes through a beta reduction before producing a value. This work is cached, so that the second Look bracket does not do any beta reduction.

4.3.3 Call-by-value. Call-by-value eagerly evaluates a let-bound RHS and then substitutes its *value*, rather than the reduction trace that led to the value, into every use site.

The call-by-value evaluation strategy is implemented with the ByValue trace transformer shown in Figure 8. Function *bind* defines a denotation $d :: \mathsf{D}\ (\mathsf{ByValue}\ \tau)$ of the right-hand side by mutual recursion with $v :: \mathsf{Value}\ (\mathsf{ByValue}\ \tau)$ that we will discuss shortly.

As its first action, *bind* yields a Let_0 event, announcing in the trace that the right-hand side of a **let** is to be evaluated. Then monadic bind $v_1 \leftarrow d; \mathit{body}\ (\mathit{return}\ v_1)$ yields steps from the right-hand side d until its value $v_1 :: \mathsf{Value}\ (\mathsf{ByValue}\ \tau)$ is reached, which is then passed *return*ed (i.e., wrapped in Ret) to the let *body*. Note that the steps in d are yielded *eagerly*, and only once, rather than duplicating the trace at every use site in *body*, as the by-name form *body d* would.

To understand the recursive definition of the denotation of the right-hand side d and its value v, consider the case $\tau = \mathsf{T}$. Then $\mathit{return} = \mathsf{Ret}$ and we get $d = \mathit{rhs}\ (\mathsf{Ret}\ v)$ for the value v at the end of the trace d, as computed by the type class instance method $\mathit{getValue} :: \mathsf{T}\ v \to v$.[18] The effect of $\mathsf{Ret}\ (\mathit{getValue}\ (\mathit{unByValue}\ d))$ is that of stripping all Steps from d.[19]

Since nothing about *getValue* is particularly special to T, it lives in its own type class Extract so that we get a HasBind instance for different types of Traces, such as more abstract ones in Section 6.

Let us trace **let** $i = (\bar{\lambda} y.\bar{\lambda} x.x)\ i$ **in** $i\ i$ for call-by-value:

[17] It is worth noting that nothing in our approach is particularly specific to Exp or Value! We have built similar interpreters for PCF, where the `rec`, `let` and non-atomic argument constructs can simply reuse *bind* to recover a call-by-need semantics. The Event type needs semantics- and use-case-specific adjustment, though.

[18] The keen reader may have noted that we could use Extract to define a MonadFix instance for deterministic τ.

[19] We could have defined d as one big guarded fixpoint $\mathit{fix}\ (\mathit{rhs} \circ \mathit{return} \circ \mathit{getValue} \circ \mathit{unByValue})$, but some co-authors prefer to see the expanded form.

$$\mathcal{S}_{\mathbf{vinit}}[\![e]\!]_\rho(\mu) = \mathit{unByVInit}\ (\mathcal{S}[\![e]\!]_\rho :: \mathsf{D}\ (\mathsf{ByVInit\ T}))\ \mu$$

$$\begin{array}{l}
\mathbf{newtype}\ \mathsf{ByVInit}\ \tau\ v = \mathsf{ByVInit}\ \{\mathit{unByVInit} :: \mathsf{Heap}\ (\mathsf{ByVInit}\ \tau) \to \tau\ (v, \mathsf{Heap}\ (\mathsf{ByVInit}\ \tau))\} \\
\mathbf{instance}\ (\mathsf{Monad}\ \tau, \forall v.\ \mathsf{Trace}\ (\tau\ v)) \Rightarrow \mathsf{HasBind}\ (\mathsf{D}\ (\mathsf{ByVInit}\ \tau))\ \mathbf{where} \\
\quad \mathit{bind}\ \mathit{rhs}\ \mathit{body} = \mathbf{do}\ \mu \leftarrow \mathit{get} \\
\qquad\qquad\qquad\qquad\quad \mathbf{let}\ a = \mathit{nextFree}\ \mu \\
\qquad\qquad\qquad\qquad\quad \mathit{put}\ \mu[a \mapsto \mathit{stuck}] \\
\qquad\qquad\qquad\qquad\quad \mathit{step}\ \mathsf{Let}_0\ (\mathit{memo}\ a\ (\mathit{rhs}\ (\mathit{fetch}\ a))) \ggg \mathit{body} \circ \mathit{return}
\end{array}$$

Fig. 9. Call-by-value with lazy initialisation

$$\begin{array}{l}
\mathcal{S}_{\mathbf{clair}}[\![e]\!]_\rho = \mathit{runClair}\ \$\ \mathcal{S}[\![e]\!]_\rho :: \mathsf{T}\ (\mathsf{Value}\ (\mathsf{Clairvoyant\ T})) \\
\mathbf{data}\ \mathsf{Fork}\ f\ a = \mathsf{Empty} \mid \mathsf{Single}\ a \mid \mathsf{Fork}\ (f\ a)\ (f\ a); \mathbf{data}\ \mathsf{ParT}\ m\ a = \mathsf{ParT}\ (m\ (\mathsf{Fork}\ (\mathsf{ParT}\ m)\ a)) \\
\mathbf{instance}\ \mathsf{Monad}\ \tau \Rightarrow \mathsf{Alternative}\ (\mathsf{ParT}\ \tau)\ \mathbf{where} \\
\quad \mathit{empty} = \mathsf{ParT}\ (\mathit{pure}\ \mathsf{Empty}); l \mathbin{<|>} r = \mathsf{ParT}\ (\mathit{pure}\ (\mathsf{Fork}\ l\ r)) \\
\mathbf{newtype}\ \mathsf{Clairvoyant}\ \tau\ a = \mathsf{Clairvoyant}\ (\mathsf{ParT}\ \tau\ a) \\
\mathit{runClair} :: \mathsf{D}\ (\mathsf{Clairvoyant\ T}) \to \mathsf{T}\ (\mathsf{Value}\ (\mathsf{Clairvoyant\ T})) \\
\mathbf{instance}\ (\mathsf{Extract}\ \tau, \mathsf{Monad}\ \tau, \forall v.\ \mathsf{Trace}\ (\tau\ v)) \Rightarrow \mathsf{HasBind}\ (\mathsf{D}\ (\mathsf{Clairvoyant}\ \tau))\ \mathbf{where} \\
\quad \mathit{bind}\ \mathit{rhs}\ \mathit{body} = \mathsf{Clairvoyant}\ (\mathit{skip} \mathbin{<|>} \mathit{let}') \ggg \mathit{body} \\
\qquad \mathbf{where}\ \mathit{skip} = \mathit{return}\ (\mathsf{Clairvoyant}\ \mathit{empty}) \\
\qquad\qquad\quad \mathit{let}' = \mathit{fmap}\ \mathit{return}\ \$\ \mathit{step}\ \mathsf{Let}_0\ \$\ \ldots\ \mathit{fix}\ \ldots\ \mathit{rhs}\ \ldots\ \mathit{getValue}\ \ldots
\end{array}$$

Fig. 10. Clairvoyant Call-by-value

$\lambda\!> \mathcal{S}_{\mathbf{value}}[\![\mathit{read}$ `"let i = (λy.λx.x) i in i i"` $]\!]_\varepsilon$

$\textsc{Let}_0 \hookrightarrow \textsc{App}_1 \hookrightarrow \textsc{App}_2 \hookrightarrow \textsc{Let}_1 \hookrightarrow \textsc{App}_1 \hookrightarrow \textsc{Look}(i) \hookrightarrow \textsc{App}_2 \hookrightarrow \textsc{Look}(i) \hookrightarrow \langle\lambda\rangle$

The beta reduction of $(\bar{\lambda}y.\bar{\lambda}x.x)\ i$ now happens once within the $\textsc{Let}_0/\textsc{Let}_1$ bracket; the two subsequent $\textsc{Look}$ events immediately halt with a value.

Alas, this model of call-by-value does not yield a total interpreter! Consider the case when the right-hand side accesses its value before yielding one, e.g.,

$\lambda\!> \mathit{takeT}\ 5\ \$\ \mathcal{S}_{\mathbf{value}}[\![\mathit{read}$ `"let x = x in x x"` $]\!]_\varepsilon$

$\textsc{Let}_0 \hookrightarrow \textsc{Look}(x) \hookrightarrow \textsc{Let}_1 \hookrightarrow \textsc{App}_1 \hookrightarrow \textsc{Look}(x) \hookrightarrow$ `^CInterrupted`

This loops forever unproductively, rendering the interpreter unfit as a denotational semantics.

4.3.4 Lazy Initialisation and Black-holing. Recall that our simple ByValue transformer above yields a potentially looping interpreter. Typical strict languages work around this issue in either of two ways: They enforce termination of the RHS statically (OCaml, ML), or they use *lazy initialisation* techniques [Nakata 2010; Nakata and Garrigue 2006] (Scheme, recursive modules in OCaml). We recover a total interpreter using the semantics in Nakata [2010], building on the same encoding as ByNeed and initialising the heap with a *black hole* [Peyton Jones 1992] *stuck* in *bind* as in Figure 9.

$\lambda\!> \mathcal{S}_{\mathbf{vinit}}[\![\mathit{read}$ `"let x = x in x x"` $]\!]_\varepsilon(\varepsilon) :: \mathsf{T}\ (\mathsf{Value}\ \_, \mathsf{Heap}\ \_)$

$\textsc{Let}_0 \hookrightarrow \textsc{Look}(x) \hookrightarrow \textsc{Let}_1 \hookrightarrow \textsc{App}_1 \hookrightarrow \textsc{Look}(x) \hookrightarrow \langle(\lightning, [0 \mapsto \_])\rangle$

4.3.5 *Clairvoyant Call-by-value.*

Clairvoyant call-by-value [Hackett and Hutton 2019] is an approach to call-by-need semantics that exploits non-determinism and a cost model to absolve of the heap. We can instantiate our interpreter to generate the shortest clairvoyant call-by-value trace as well, as sketched out in Figure 10. Doing so yields an evaluation strategy that either skips or speculates let bindings, depending on whether or not the binding is needed:

$\lambda\!>\ \mathcal{S}_{\mathbf{clair}}[\![\mathit{read}$ `"let f = λx.x in let g = λy.f in g"` $]\!]_{\varepsilon} :: \mathsf{T}\ (\mathsf{Value}\ (\mathsf{Clairvoyant}\ \mathsf{T}))$

$\textsc{Let}_1 \hookrightarrow \textsc{Let}_0 \hookrightarrow \textsc{Let}_1 \hookrightarrow \textsc{Look}(g) \hookrightarrow \langle\lambda\rangle$

$\lambda\!>\ \mathcal{S}_{\mathbf{clair}}[\![\mathit{read}$ `"let f = λx.x in let g = λy.f in g g"` $]\!]_{\varepsilon} :: \mathsf{T}\ (\mathsf{Value}\ (\mathsf{Clairvoyant}\ \mathsf{T}))$

$\textsc{Let}_0 \hookrightarrow \textsc{Let}_1 \hookrightarrow \textsc{Let}_0 \hookrightarrow \textsc{Let}_1 \hookrightarrow \textsc{App}_1 \hookrightarrow \textsc{Look}(g) \hookrightarrow \textsc{App}_2 \hookrightarrow \textsc{Look}(f) \hookrightarrow \langle\lambda\rangle$

The first example discards f, but the second needs it, so the trace starts with an additional $\textsc{Let}_0$ event. Similar to ByValue, the interpreter is not total so it is unfit as a denotational semantics without a complicated domain theoretic judgment. Furthermore, the decision whether or not a $\textsc{Let}_0$ is needed can be delayed for an infinite amount of time, as exemplified by

$\lambda\!>\ \mathcal{S}_{\mathbf{clair}}[\![\mathit{read}$ `"let i = Z() in let w = λy.y y in w w"` $]\!]_{\varepsilon} :: \mathsf{T}\ (\mathsf{Value}\ (\mathsf{Clairvoyant}\ \mathsf{T}))$

```
^CInterrupted
```

The program diverges without producing even a prefix of a trace because the binding for i might be needed at an unknown point in the future (a *liveness property* and hence impossible to verify at runtime). This renders Clairvoyant call-by-value inadequate for verifying properties of infinite executions.

5 TOTALITY AND SEMANTIC ADEQUACY

In this section, we prove that $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_{\_}$ produces small-step traces of the lazy Krivine machine and is indeed a *denotational semantics.*[20] Excitingly, to our knowledge, $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_{\_}$ is the first denotational call-by-need semantics that was proven so! Specifically, denotational semantics must be total and adequate. *Totality* says that the interpreter is well-defined for every input expression and *adequacy* says that the interpreter produces similar traces as the reference semantics. This is an important result because it allows us to switch between operational reference semantics and denotational interpreter as needed, thus guaranteeing compatibility of definitions such as absence in Definition 2. As before, all the proofs can be found in the Appendix.

5.1 Adequacy of $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_{\_}$

For proving adequacy of $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_{\_}$, we give an abstraction function α from small-step traces in the lazy Krivine machine (Figure 2) to denotational traces T, with Events and all, such that

$$\alpha(\mathit{init}(\mathrm{e}) \hookrightarrow \ldots) = \mathcal{S}_{\mathbf{need}}[\![e]\!]_{\varepsilon}(\varepsilon),$$

where $\mathit{init}(\mathrm{e}) \hookrightarrow \ldots$ denotes the *maximal* (i.e. longest possible) LK trace evaluating the closed expression e. For example, for the LK trace (2), α produces the trace at the end of Section 4.3.2.

It turns out that function α preserves a number of important observable properties, such as termination behavior (i.e. stuck, diverging, or balanced execution [Sestoft 1997]), length of the trace and transition events, as expressed in the following Theorem:

Theorem 4 (Strong Adequacy). *Let e be a closed expression, $\tau \triangleq \mathcal{S}_{\mathbf{need}}[\![e]\!]_{\varepsilon}(\varepsilon)$ the denotational by-need trace and $\mathit{init}(\mathrm{e}) \hookrightarrow \ldots$ the maximal lazy Krivine trace. Then*

- *τ preserves the observable termination properties of $\mathit{init}(\mathrm{e}) \hookrightarrow \ldots$ in the above sense.*

[20] Similar results for $\mathcal{S}_{\mathbf{name}}[\![\_]\!]_{\_}$ and $\mathcal{S}_{\mathbf{vinit}}[\![\_]\!]_{\_}(\_)$ should be derivative.

- *τ preserves the length (i.e., number of* Step*s) of $init(\mathrm{e}) \hookrightarrow \ldots$ (i.e., number of transitions).*
- *every ev :: Event in τ = Step ev ... corresponds to the transition rule taken in $init(\mathrm{e}) \hookrightarrow \ldots$.*

Proof sketch. Define α by coinduction and prove $\alpha(init(\mathrm{e}) \hookrightarrow \ldots) = \mathcal{S}_{\mathbf{need}}[\![e]\!]_{\varepsilon}(\varepsilon)$ by Löb induction. Then it suffices to prove that α preserves the observable properties of interest. The full proof for a rigorous reformulation of this result can be found in the Appendix. □

5.2 Totality of $\mathcal{S}_{\mathbf{name}}[\![\_]\!]_{\_}$ and $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_{\_}$

Theorem 5 (Totality). *The interpreters $\mathcal{S}_{\mathbf{name}}[\![e]\!]_{\rho}$ and $\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho}(\mu)$ are defined for every e, ρ, μ.*

Proof sketch. In the Supplement, we provide an implementation of the generic interpreter $\mathcal{S}[\![\_]\!]_{\_}$ and its instances at ByName and ByNeed in Guarded Cubical Agda, which offers a total type theory with *guarded recursive types* Møgelberg and Veltri [2019]. Agda enforces that all encodable functions are total, therefore $\mathcal{S}_{\mathbf{name}}[\![\_]\!]_{\_}$ and $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_{\_}$ must be total as well.

The essential idea of the totality proof is that *there is only a finite number of transitions between every Look transition*. In other words, if every environment lookup produces a Step constructor, then our semantics is total by coinduction. Such an argument is quite natural to encode in guarded recursive types, hence our use of Guarded Cubical Agda is appealing. See Appendix B.1 for the details of the encoding in Agda. □

6 STATIC ANALYSIS

So far, our semantic domains have all been *infinite*, simply because the dynamic traces they express are potentially infinite as well. However, by instantiating the *same* generic denotational interpreter with a *finite* semantic domain, we can run the interpreter on the program statically, at compile time, to yield a *finite* abstraction of the dynamic behavior. This gives us a *static program analysis.*

We can get a wide range of static analyses, simply by choosing an appropriate semantic domain. For example, we have successfully realised the following analyses as denotational interpreters:

- Appendix C.1 defines a Hindley-Milner-style *type analysis* with let generalisation, inferring types such as $\forall \alpha_3.$ `option` $(\alpha_3 \to \alpha_3)$. Polymorphic types act as summaries in the sense of the Introduction, and fixpoints are solved via unification.
- Appendix C.2 defines 0CFA *control-flow analysis* [Shivers 1991] as an instance of our generic interpreter. The summaries are sets of labelled expressions that evaluation might return. These labels are given meaning in an abstract store. For a function label, the abstract store maintains a single point approximation of the function's abstract transformer.
- We have refactored relevant parts of *Demand Analysis* in the Glasgow Haskell Compiler into an abstract denotational interpreter as an artefact. The resulting compiler bootstraps and passes the testsuite.[21] Demand Analysis is the real-world implementation of the cardinality analysis work of [Sergey et al. 2017], implementing strictness analysis as well. This is to demonstrate that our framework scales to real-world compilers.

In this section, we demonstrate this idea in detail, using a much simpler version of GHC's Demand Analysis: a summary-based *usage analysis*, the code of which is given in Figure 11.

6.1 Trace Abstraction in Trace T_{U}

In order to recover usage analysis as an instance of our generic interpreter, we must define its finite semantic domain D_{U}. Often, the first step in doing so is to replace the potentially infinite traces T

[21] There is a small caveat: we did not try to optimise for compiler performance in our proof of concept and hence it regresses in a few compiler performance test cases. None of the runtime performance test cases regress and the inferred demand signatures stay unchanged.

```
data U = U₀ | U₁ | U_ω
type Uses = Name :⇀ U
class UVec a where
  (+) :: a → a → a
  (∗) :: U → a → a
instance UVec U where ...
instance UVec Uses where ...

data T_U v = ⟨Uses, v⟩
instance Trace (T_U v) where
  step (Lookup x) ⟨φ, v⟩ = ⟨[x ↦ U₁] + φ, v⟩
  step _          τ      = τ
instance Monad T_U where
  return a = ⟨ε, a⟩
  ⟨φ₁, a⟩ >>= k = let ⟨φ₂, b⟩ = k a in ⟨φ₁ + φ₂, b⟩

S_usage⟦e⟧_ρ = S⟦e⟧_ρ :: D_U
instance Domain D_U where
  stuck                   = ⊥
  fun x f                 = case f ⟨[x ↦ U₁], Rep U_ω⟩ of
    ⟨φ, v⟩ → ⟨φ[x ↦ U₀], φ !? x ⸪ v⟩
  apply ⟨φ₁, v₁⟩ ⟨φ₂, _⟩ = case peel v₁ of
    (u, v₂) → ⟨φ₁ + u ∗ φ₂, v₂⟩
  con _ ds                = foldl apply ⟨ε, Rep U_ω⟩ ds
  select d fs             =
    d >> lub [ f (replicate (conArity k) ⟨ε, Rep U_ω⟩)
             | (k, f) ← assocs fs]
instance HasBind D_U where
  bind rhs body = body (kleeneFix rhs)

data Value_U = U ⸪ Value_U | Rep U
type D_U = T_U Value_U
instance Lat U where ...
instance Lat Uses where ...
instance Lat Value_U where ...
instance Lat D_U where ...
peel :: Value_U → (U, Value_U)
peel (Rep u) = (u, (Rep u))
peel (u ⸪ v)  = (u, v)
(!?) :: Uses → Name → U
m !? x | x ∈ dom m = m ! x
       | otherwise  = U₀
```

Fig. 11. Summary-based usage analysis

in dynamic semantic domains such as $\mathsf{D}_{\mathbf{na}}$ with a finite type such as T_{U} in Figure 11. A *usage trace* $\langle\varphi, \mathit{val}\rangle :: \mathsf{T}_{\mathsf{U}}\ v$ is a pair of a value $\mathit{val} :: v$ and a finite map $\varphi :: \mathsf{Uses}$, mapping variables to a *usage* U. The usage $\varphi\ !?\ x$ assigned to x is meant to approximate the number of $\mathsf{Lookup}\ x$ events; U_0 means "at most 0 times", U_1 means "at most 1 times", and U_ω means "an unknown number of times". In this way, T_{U} is an *abstraction* of T: it squashes all $\mathsf{Lookup}\ x$ events into a single entry $\varphi\ !?\ x :: \mathsf{U}$ and discards all other events.

Consider as an example the by-name trace evaluating e $\triangleq$ **let** $i = \bar{\lambda} x.x$ **in let** $j = \bar{\lambda} y.y$ **in** $i\ j\ j$:

$$\textsc{Let}_1 \hookrightarrow \textsc{Let}_1 \hookrightarrow \textsc{App}_1 \hookrightarrow \textsc{App}_1 \hookrightarrow \textsc{Look}(i) \hookrightarrow \textsc{App}_2 \hookrightarrow \textsc{Look}(j) \hookrightarrow \textsc{App}_2 \hookrightarrow \textsc{Look}(j) \hookrightarrow \langle\lambda\rangle$$

We would like to abstract this trace into $\langle[\,i \mapsto \mathsf{U}_1, j \mapsto \mathsf{U}_\omega], \ldots\rangle$. One plausible way to achieve this is to replace every $\mathsf{Step}\ (\mathsf{Lookup}\ x)\ \ldots$ in the by-name trace with a call to $\mathit{step}\ (\mathsf{Lookup}\ x)\ \ldots$ from the $\mathsf{Trace}\ \mathsf{T}_{\mathsf{U}}$ instance in Figure 11, quite similar to *foldr step* on lists. The *step* implementation increments the usage of x whenever a $\mathsf{Lookup}\ x$ event occurs. The addition operation used to carry out incrementation is defined in type class instances $\mathsf{UVec}\ \mathsf{U}$ and $\mathsf{UVec}\ \mathsf{Uses}$, together with scalar multiplication.[22] For example, $\mathsf{U}_0 + u = u$ and $\mathsf{U}_1 + \mathsf{U}_1 = \mathsf{U}_\omega$ in U, as well as $\mathsf{U}_0 * u = \mathsf{U}_0$, $\mathsf{U}_\omega * \mathsf{U}_1 = \mathsf{U}_\omega$. These operations lift to Uses pointwise, e.g., $[\,i \mapsto \mathsf{U}_1] + (\mathsf{U}_\omega * [\,j \mapsto \mathsf{U}_1]) = [\,i \mapsto \mathsf{U}_1, j \mapsto \mathsf{U}_\omega]$.

Following through on the *foldr step* idea to abstract a T into T_{U} amounts to what Darais et al. [2017] call a *collecting semantics* of the interpreter. Such semantics-specific collecting variants are easily achievable for us as well. It is as simple as defining a Monad instance on T_{U} mirroring trace concatenation and then running our interpreter at, e.g., $\mathsf{D}\ (\mathsf{ByName}\ \mathsf{T}_{\mathsf{U}}) \cong \mathsf{T}_{\mathsf{U}}\ (\mathsf{Value}\ \mathsf{T}_{\mathsf{U}})$ on

[22]We think that UVec models U-modules. It is not a vector space because U lacks inverses, but the intuition is close enough.

expression e from earlier:

$$\mathcal{S}[\![(\textbf{let } i = \bar{\lambda}x.x \textbf{ in let } j = \bar{\lambda}y.y \textbf{ in } i\ j\ j)]\!]_{\varepsilon} = \langle [i \mapsto \mathsf{U}_1, j \mapsto \mathsf{U}_\omega], \lambda \rangle :: \mathsf{D}\ (\mathsf{ByName}\ \mathsf{T}_\mathsf{U}).$$

It is nice to explore whether the Trace instance encodes the desired operational property in this way, but of little practical relevance because this interpreter instance will diverge whenever the input expression diverges. We fix this in the next subsection by introducing a finite $\mathsf{Value}_\mathsf{U}$ to replace $\mathsf{Value}\ \mathsf{T}_\mathsf{U}$.

6.2 Value Abstraction $\mathsf{Value}_\mathsf{U}$ and Summarisation in Domain D_U

In this subsection, we complement the finite trace type T_U from the previous subsection with a corresponding finite semantic value type $\mathsf{Value}_\mathsf{U}$ to get the finite semantic domain $\mathsf{D}_\mathsf{U} = \mathsf{T}_\mathsf{U}\ \mathsf{Value}_\mathsf{U}$ in Figure 11, and thus a *static usage analysis* $\mathcal{S}_{\textbf{usage}}[\![\_]\!]_\_$ when we instantiate $\mathcal{S}[\![\_]\!]_\_$ at D_U.

The definition of $\mathsf{Value}_\mathsf{U}$ is just a copy of $\varsigma \in \mathsf{Summary}$ in Figure 1 that lists argument usage U instead of Absence flags; the entire intuition transfers. For example, the $\mathsf{Value}_\mathsf{U}$ summarising $\bar{\lambda}y.\bar{\lambda}z.y$ is $\mathsf{U}_1 \mathbin{⨾} \mathsf{U}_0 \mathbin{⨾} \mathsf{Rep}\ \mathsf{U}_\omega$, because the first argument is used once while the second is used 0 times. What we previously called absence types $\theta \in \mathsf{AbsTy}$ in Figure 1 is now the abstract semantic domain D_U. It is now evident that usage analysis is a modest generalisation of absence analysis in Figure 1: a variable is absent (A) when it has usage U_0, otherwise it is used (U).

Consider $\mathcal{S}_{\textbf{usage}}[\![(\textbf{let } k = \bar{\lambda}y.\bar{\lambda}z.y \textbf{ in } k\ x_1\ x_2)]\!]_{\rho_e} = \langle [k \mapsto \mathsf{U}_1, x_1 \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle$, analysing the example expression from Section 2. Usage analysis successfully infers that x_1 is used at most once and that x_2 is absent, because it does not occur in the reported Uses.

On the other hand, $\mathcal{S}_{\textbf{usage}}[\![(\textbf{let } i = \bar{\lambda}x.x \textbf{ in let } j = \bar{\lambda}y.y \textbf{ in } i\ i\ j)]\!]_{\varepsilon} = \langle [i \mapsto \mathsf{U}_\omega, j \mapsto \mathsf{U}_\omega], \mathsf{Rep}\ \mathsf{U}_\omega \rangle$ demonstrates the limitations of the first-order summary mechanism. While the program trace would only have one lookup for j, the analysis is unable to reason through the indirect call and conservatively reports that j may be used many times.

The Domain instance is responsible for implementing the summary mechanism. While *stuck* expressions do not evaluate anything and hence are denoted by $\bot = \langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_0 \rangle$, the *fun* and *apply* functions play exactly the same roles as fun_x and *app* in Figure 1. Let us briefly review how the summary for the right-hand side $\bar{\lambda}x.x$ of i in the previous example is computed:

$$\begin{aligned}
&\mathcal{S}[\![\mathsf{Lam}\ x\ (\mathsf{Var}\ x)]\!]_\rho = \mathit{fun}\ x\ (\lambda d \to \mathit{step}\ \mathsf{App}_2\ (\mathcal{S}[\![\mathsf{Var}\ x]\!]_{\rho[x \mapsto d]}))\\
&= \textbf{case}\ \mathit{step}\ \mathsf{App}_2\ (\mathcal{S}[\![\mathsf{Var}\ x]\!]_{\rho[x \mapsto \langle [x \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle]})\ \textbf{of}\ \langle \varphi, v \rangle \to \langle \varphi[x \mapsto \mathsf{U}_0], \varphi\ !?\ x \mathbin{⨾} \mathsf{Rep}\ \mathsf{U}_\omega \rangle\\
&= \textbf{case}\ \langle [x \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle\ \textbf{of}\ \langle \varphi, v \rangle \to \langle \varphi[x \mapsto \mathsf{U}_0], \varphi\ !?\ x \mathbin{⨾} \mathsf{Rep}\ \mathsf{U}_\omega \rangle\\
&= \langle \varepsilon, \mathsf{U}_1 \mathbin{⨾} \mathsf{Rep}\ \mathsf{U}_\omega \rangle
\end{aligned}$$

The definition of *fun x* applies the lambda body to a *proxy* $\langle [x \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle$ to summarise how the body uses its argument by way of looking at how it uses x.[23] Every use of x's proxy will contribute a usage of U_1 on x, and multiple uses in the lambda body would accumulate to a usage of U_ω. In this case there is only a single use of x and the final usage $\varphi\ !?\ x = \mathsf{U}_1$ from the lambda body will be prepended to the summarised value. Occurrences of x must make do with the top value ($\mathsf{Rep}\ \mathsf{U}_\omega$) from x's proxy for lack of knowing the actual argument at call sites.

The definition of *apply* to apply such summaries to an argument is nearly the same as in Figure 1, except for the use of $+$ instead of $\sqcup$ to carry over $\mathsf{U}_1 + \mathsf{U}_1 = \mathsf{U}_\omega$, and an explicit *peel* to view a $\mathsf{Value}_\mathsf{U}$ in terms of $⨾$ (it is $\mathsf{Rep}\ u \equiv u \mathbin{⨾} \mathsf{Rep}\ u$). The usage u thus pelt from the value determines how often the actual argument was evaluated, and multiplying the uses of the argument φ_2 with u accounts for that.

[23] As before, the exact identity of x is exchangeable; we use it as a De Bruijn level.

$$\textbf{class } \mathsf{Eq}\ a \Rightarrow \mathsf{Lat}\ a \textbf{ where } \bot :: a; (\sqcup) :: a \to a \to a;$$
$$\mathit{kleeneFix} :: \mathsf{Lat}\ a \Rightarrow (a \to a) \to a; \qquad \mathit{lub} :: \mathsf{Lat}\ a \Rightarrow [a] \to a$$
$$\mathit{kleeneFix}\ f = \mathit{go}\ \bot \textbf{ where } \mathit{go}\ x = \textbf{let } x' = f\ x \textbf{ in if } x' \sqsubseteq x \textbf{ then } x' \textbf{ else } \mathit{go}\ x'$$

Fig. 12. Order theory and Kleene iteration

The example $\mathcal{S}_{\mathbf{usage}}[\![(\mathbf{let}\ z = Z()\ \mathbf{in\ case}\ S(z)\ \mathbf{of}\ S(n) \to n)]\!]_{\varepsilon} = \langle [z \mapsto \mathsf{U}_{\omega}], \mathsf{Rep}\ \mathsf{U}_{\omega} \rangle$ illustrates the summary mechanism for data types. Our analysis imprecisely infers that z might be used many times when it is only used once. That is because we tried to keep $\mathsf{Value}_{\mathsf{U}}$ intentionally simple, so our analysis assumes that every data constructor uses its fields many times.[24] This is achieved in *con* by repeatedly *apply*ing to the top value ($\mathsf{Rep}\ \mathsf{U}_{\omega}$), as if a data constructor was a lambda-bound variable. Dually, *select* does not need to track how fields are used and can pass $\langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_{\omega} \rangle$ as proxies for field denotations. The result uses anything the scrutinee expression used, plus the upper bound of uses in case alternatives, one of which will be taken.

Much more could be said about the way in which finiteness of D_{U} rules out injective implementations of $\mathit{fun}\ x :: (\mathsf{D}_{\mathsf{U}} \to \mathsf{D}_{\mathsf{U}}) \to \mathsf{D}_{\mathsf{U}}$ and thus requires the aforementioned *approximate* summary mechanism, but it is easy to get sidetracked in doing so. There is another potential source of approximation: the HasBind instance discussed next.

6.3 Finite Fixpoint Strategy in HasBind D_{U} and Totality

The third and last ingredient to recover a static analysis is the fixpoint strategy in HasBind D_{U}, to be used for recursive let bindings.

For the dynamic semantics in Section 4 we made liberal use of *guarded fixpoints*, that is, recursively defined values such as $\mathbf{let}\ d = \mathit{rhs}\ d\ \mathbf{in}\ \mathit{body}\ d$ in HasBind $\mathsf{D}_{\mathbf{na}}$ (Figure 5). At least for $\mathcal{S}_{\mathbf{name}}[\![\_]\!]_{\_}$ and $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_{\_}$, we have proved in Section 5.1 that these fixpoints always exist by a coinductive argument. Alas, among other things this argument relies on the Step constructor — and thus the *step* method — of the trace type T being *lazy* in the tail of the trace!

When we replaced T in favor of the finite, inductive type T_{U} in Section 6.1 to get a collecting semantics D (ByName T_{U}), we got a partial interpreter. That was because the *step* implementation of T_{U} is *not* lazy, and hence the guarded fixpoint $\mathbf{let}\ d = \mathit{rhs}\ d\ \mathbf{in}\ \mathit{body}\ d$ is not guaranteed to exist.

In general, finite trace types cannot have a lazy *step* implementation, so finite domains such as D_{U} require a different fixpoint strategy to ensure termination. Depending on the abstract domain, different fixpoint strategies can be employed. For an unusual example, in our type analysis Appendix C.1, we generate and solve a constraint system via unification to define fixpoints. In case of D_{U}, we compute least fixpoints by Kleene iteration *kleeneFix* in Figure 12. *kleeneFix* requires us to define an order on D_{U}, which is induced by $\mathsf{U}_0 \sqsubset \mathsf{U}_1 \sqsubset \mathsf{U}_{\omega}$ in the same way that the order on AbsTy in Section 2.2 was induced from the order $\mathsf{A} \sqsubset \mathsf{U}$ on Absence flags. The iteration procedure terminates whenever the type class instances of D_{U} are monotone and there are no infinite ascending chains in D_{U}.

The keen reader may feel indignant because our $\mathsf{Value}_{\mathsf{U}}$ indeed contains such infinite chains, for example, $\mathsf{U}_1 \mathbin{⨾} \mathsf{U}_1 \mathbin{⨾} \ldots \mathbin{⨾} \mathsf{Rep}\ \mathsf{U}_0$! This is easily worked around in practice by employing appropriate widening measures such as bounding the depth of $\mathsf{Value}_{\mathsf{U}}$. The resulting definition of HasBind is safe for by-name and by-need semantics.[25]

[24]It is clear how to do a better job at least for products; see Sergey et al. [2017].

[25]Never mind totality; why is the use of *least* fixpoints even correct? The fact that we are approximating a safety property [Lamport 1977] is important. We discuss this topic in Appendix D.2.

$$\textsc{Mono}\quad \dfrac{d_1 \sqsubseteq d_2 \qquad f_1 \sqsubseteq f_2}{apply\ f_1\ d_1 \sqsubseteq apply\ f_2\ d_2 \textit{ and so on, for all methods of } \mathsf{Trace}, \mathsf{Domain}, \mathsf{HasBind}}$$

$$\textsc{Step-App}\quad step\ ev\ (apply\ d\ a) \sqsubseteq apply\ (step\ ev\ d)\ a \qquad \textsc{Step-Sel}\quad step\ ev\ (select\ d\ alts) \sqsubseteq select\ (step\ ev\ d)\ alts$$

$$\textsc{Unwind-Stuck}\quad stuck \sqsubseteq \bigsqcup\{apply\ stuck\ a, select\ stuck\ alts\} \qquad \textsc{Intro-Stuck}\quad stuck \sqsubseteq \bigsqcup\{apply\ (con\ k\ ds)\ a, select\ (fun\ x\ f)\ alts\}$$

$$\textsc{Beta-App}\quad \dfrac{f\ d = step\ \mathsf{App}_2\ (\mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_{\rho[x\mapsto d]})}{f\ a \sqsubseteq apply\ (fun\ x\ f)\ a} \qquad \textsc{Beta-Sel}\quad \dfrac{\begin{array}{lll}(alts\,!\,k)\ ds & \mid len\ ds \not\equiv len\ xs & = stuck \\ & \mid otherwise & = step\ \mathsf{Case}_2\ (\mathcal{S}_{\widehat{\mathsf{D}}}[\![e_r]\!]_{\rho[\overline{xs\mapsto ds}]})\end{array}}{(alts\,!\,k)\ (map\ (\rho_1\,!)\ ys) \sqsubseteq select\ (con\ k\ (map\ (\rho_1\,!)\ ys))\ alts}$$

$$\textsc{Bind-ByName}\quad \dfrac{rhs\ d_1 = \mathcal{S}_{\widehat{\mathsf{D}}}[\![e_1]\!]_{\rho[x\mapsto step\ (\mathsf{Lookup}\ x)\ d_1]} \qquad body\ d_1 = step\ \mathsf{Let}_1\ (\mathcal{S}_{\widehat{\mathsf{D}}}[\![e_2]\!]_{\rho[x\mapsto d_1]})}{body\ (lfp\ rhs) \sqsubseteq bind\ rhs\ body}$$

$$\boxed{\textsc{Step-Inc}\quad d \sqsubseteq step\ ev\ d \qquad \textsc{Update}\quad step\ \mathsf{Update}\ d = d}$$

Fig. 13. By-name and $\boxed{\text{by-need}}$ abstraction laws for type class instances of abstract domain $\widehat{\mathsf{D}}$

It is nice to define dynamic semantics and static analyses in the same framework, but another important benefit is that correctness proofs become simpler, as we will see next.

7 GENERIC BY-NAME AND BY-NEED SOUNDNESS

In this section we prove and apply a generic abstract interpretation theorem of the form

$$abstract\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\varepsilon}) \sqsubseteq \mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_{\varepsilon}.$$

This statement reads as follows: for a closed expression e, the *static analysis* result $\mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_{\varepsilon}$ on the right-hand side *overapproximates* ($\sqsupseteq$) a property of the by-need *semantics* $\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\varepsilon}$ on the left-hand side. The abstraction function $abstract :: \mathsf{D}\ (\mathsf{ByNeed}\ \mathsf{T}) \to \widehat{\mathsf{D}}$ describes what semantic property we are interested in, in terms of the abstract semantic domain $\widehat{\mathsf{D}}$ of $\mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_{\rho}$, which is short for $\mathcal{S}[\![e]\!]_{\rho} :: \widehat{\mathsf{D}}$. In our framework, *abstract* is entirely derived from type class instances on $\widehat{\mathsf{D}}$.

We will instantiate the theorem at D_{U} in order to prove that usage analysis $\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho} = \mathcal{S}_{\mathsf{D}_{\mathsf{U}}}[\![e]\!]_{\rho}$ infers absence, just as absence analysis in Section 2. This proof will be much simpler than the proof for Theorem 1.

This section will only discuss abstraction of closed terms in a high-level, top-down way, but of course the underlying Theorem 56 in the Appendix considers open terms and is best approached bottom-up.

7.1 Sound By-name and By-need Interpretation

This subsection is dedicated to the following proof rule for sound by-need interpretation, referring to the *abstraction laws* in Figure 13 by name:

$$\frac{\begin{array}{c}\textsc{Mono} \quad \textsc{Step-App} \quad \textsc{Step-Sel} \quad \textsc{Unwind-Stuck} \\ \textsc{Intro-Stuck} \quad \textsc{Beta-App} \quad \textsc{Beta-Sel} \quad \textsc{Bind-ByName} \quad \textsc{Step-Inc} \quad \textsc{Update}\end{array}}{\mathit{abstract}\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\varepsilon}) \sqsubseteq \mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_{\varepsilon}}$$

In other words: prove the abstraction laws for an abstract domain $\widehat{\mathsf{D}}$ of your choosing and we give you for free a proof of sound abstract by-need interpretation for the static analysis $\mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_{\varepsilon}$!

This proof rule is *opinionated*, in so far as *we* get to determine the abstraction function *abstract* based on the Trace, Domain and Lat instance on your $\widehat{\mathsf{D}}$. The gist is as follows: *abstract* eliminates every Step *evt* in the by-need trace with a call to *step evt*, and eliminates every concrete Value at the end of the trace with a call to the corresponding Domain method. That is, Fun turns into *fun*, Con into *con*, and Stuck into *stuck*, considering the final heap for nested abstraction (the subtle details are best left to the Appendix). Thanks to fixing *abstract*, the abstraction laws can be simplified drastically, as discussed at the end of this subsection. The precise definition of *abstract* can be found in the proof of the following theorem, embodying the proof rule above:

Theorem 6 (Sound By-need Interpretation). *Let* $\widehat{\mathsf{D}}$ *be a domain with instances for* Trace, Domain, HasBind *and* Lat, *and let abstract be the abstraction function described above. If the abstraction laws in Figure 13 hold, then* $\mathcal{S}_{\widehat{\mathsf{D}}}[\![\_]\!]_{\_}$ *is an abstract interpreter that is sound wrt. abstract, that is,*

$$\mathit{abstract}\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\varepsilon}) \sqsubseteq \mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_{\varepsilon}.$$

Let us unpack law Beta-App to see how the abstraction laws in Figure 13 are to be understood. For a preliminary reading, it is best to ignore the syntactic premises above inference lines. To prove Beta-App, one has to show that $\forall f\ a\ x.\ f\ a \sqsubseteq \mathit{apply}\ (\mathit{fun}\ x\ f)\ a$ in the abstract domain $\widehat{\mathsf{D}}$.[26] This states that summarising f through *fun*, then *apply*ing the summary to a must approximate a direct call to f; it amounts to proving correct the summary mechanism.[27] In Section 2, we have proved a substitution Lemma 3, which is a syntactic form of this statement. We will need a similar lemma for usage analysis below, and it is useful to illustrate the next point, so we prove it here:

Lemma 7 (Substitution). $\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho[x\mapsto\rho\,!\,y]} \sqsubseteq \mathcal{S}_{\mathbf{usage}}[\![\mathsf{Lam}\ x\ e\ `\mathsf{App}`\ y]\!]_{\rho}$.

In order to apply this lemma in step $\sqsubseteq$ below, it is important that the premise provides us with the syntactic definition of $f\ d \triangleq \mathit{step}\ \mathsf{App}_2\ (\mathcal{S}_{\mathsf{D_U}}[\![e]\!]_{\rho[x\mapsto d]})$. Then we get, for $a \triangleq \rho\,!\,y :: \mathsf{D_U}$,

$$f\ a = \mathit{step}\ \mathsf{App}_2\ (\mathcal{S}_{\mathsf{D_U}}[\![e]\!]_{\rho[x\mapsto a]}) = \mathcal{S}_{\mathsf{D_U}}[\![e]\!]_{\rho[x\mapsto a]} \sqsubseteq \mathcal{S}_{\mathsf{D_U}}[\![\mathsf{Lam}\ x\ e\ `\mathsf{App}`\ y]\!]_{\rho} = \mathit{apply}\ (\mathit{fun}\ x\ f)\ a. \tag{1}$$

Without the syntactic premise of Beta-App to rule out undefinable entities in $\mathsf{D_U} \to \mathsf{D_U}$, the rule cannot be proved for usage analysis; we give a counterexample in the Appendix (Example 46).[28]

Rule Beta-Sel states a similar substitution property for data constructor redexes, which is why it needs to duplicate much of the *cont* function in Figure 5 into its premise. Rule Bind-ByName expresses that the abstract *bind* implementation must be sound for by-name evaluation, that is, it must approximate passing the least fixpoint *lfp* of the *rhs* functional to *body*.[29] The remaining rules

[26] Again, the exact identity of x is irrelevant. We only use it as a De Bruijn level; it suffices that x is chosen fresh.
[27] To illustrate this point: if we were to pick dynamic Values as the summary as in the “collecting semantics” $\mathsf{D}\ (\mathsf{ByNeed}\ \mathsf{T_U})$, we would not need to show anything! Then $\mathit{apply}\ (\mathit{return}\ (\mathsf{Fun}\ f))\ a = f\ a$.
[28] Finding domains where all entities d are definable is the classic full abstraction problem [Plotkin 1977].
[29] We expect that for sound by-value abstraction it suffices to replace Bind-ByName with a law Bind-ByValue mirroring the *bind* instance of ByValue, but have not attempted a formal proof.

are congruence rules involving *step* and *stuck* as well as the obvious monotonicity requirement for all involved operations. In the Appendix, we show a result similar to Theorem 6 for by-name evaluation which does not require the by-need specific rules Step-Inc and Update.

Note that none of the laws mention the concrete semantics or α. This is how our opinionated approach pays off: because both concrete semantics and α are known, the usual abstraction laws such as $\alpha\ (\mathit{apply}\ d\ a) \sqsubseteq \widehat{\mathit{apply}}\ (\alpha\ d)\ (\alpha\ a)$ further decompose into Beta-App. We think this is an important advantage to our approach, because the author of the analysis does not need to reason about the concrete semantics in order to soundly approximate a semantic trace property expressed via Trace instance!

7.2 A Much Simpler Proof That Usage Analysis Infers Absence

Equipped with the generic soundness Theorem 6, we will prove in this subsection that usage analysis from Section 6 infers absence in the same sense as absence analysis from Section 2. The reason we do so is to evaluate the proof complexity of our approach against the preservation-style proof framework in Section 2.

The first step is to leave behind the definition of absence in terms of the LK machine in favor of one using $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_$. That is a welcome simplification because it leaves us with a single semantic artefact — the denotational interpreter — instead of an operational semantics and a separate static analysis as in Section 2. Thanks to adequacy (Theorem 4), this new notion is not a redefinition but provably equivalent to Definition 2:

Lemma 8 (Denotational absence)**.** *Variable x is used in e if and only if there exists a by-need evaluation context E and expression e' such that the trace $\mathcal{S}_{\mathbf{need}}[\![E[\mathsf{Let}\ x\ e'\ e]]\!]_\varepsilon(\varepsilon)$ contains a $\mathsf{Lookup}\ x$ event. (Otherwise, x is absent in e.)*

We define the by-need evaluation contexts for our language in the Appendix. Thus insulated from the LK machine, we may restate and prove Theorem 1 for usage analysis.

Lemma 9 ($\mathcal{S}_{\mathbf{usage}}[\![\_]\!]_\_$ abstracts $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_$)**.** *Let e be a closed expression and abstract the abstraction function above. Then $\mathit{abstract}\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_\varepsilon) \sqsubseteq \mathcal{S}_{\mathbf{usage}}[\![e]\!]_\varepsilon$.*

Theorem 10 ($\mathcal{S}_{\mathbf{usage}}[\![\_]\!]_\_$ infers absence)**.** *Let $\rho_e \triangleq [\overline{y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega\rangle}]$ be the initial environment with an entry for every free variable y of an expression e. If $\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_e} = \langle \varphi, v\rangle$ and $\varphi\ !?\ x = \mathsf{U}_0$, then x is absent in e.*

Proof sketch. If x is used in e, there is a trace $\mathcal{S}_{\mathbf{need}}[\![E[\mathsf{Let}\ x\ e'\ e]]\!]_\varepsilon(\varepsilon)$ containing a $\mathsf{Lookup}\ x$ event. The abstraction function *abstract* induced by D_U aggregates lookups in the trace into a $\varphi' :: \mathsf{Uses}$, e.g., $\mathit{abstract}\ (\textsc{Look}(i) \hookrightarrow \textsc{Look}(x) \hookrightarrow \textsc{Look}(i) \hookrightarrow \langle ...\rangle) = \langle [i\ \mapsto \mathsf{U}_\omega, x\ \mapsto \mathsf{U}_1], ...\rangle$. Clearly, it is $\varphi'\ !?\ x \sqsupseteq \mathsf{U}_1$, because there is at least one $\mathsf{Lookup}\ x$. Lemma 9 and a context invariance Lemma 38 prove that the computed φ approximates φ', so $\varphi\ !?\ x \sqsupseteq \varphi'\ !?\ x \sqsupseteq \mathsf{U}_1 \neq \mathsf{U}_0$. □

Let us compare to the preservation-style proof framework in Section 2.

- Where there were multiple separate *semantic artefacts* such as a separate small-step semantics and an extension of the absence analysis function to machine configurations σ in order to state a preservation lemma, our proof only has a single semantic artefact that needs to be defined and understood: the denotational interpreter, albeit with different instantiations.
- What is more important is that a simple proof for Lemma 9 in half a page (we encourage the reader to take a look) replaces a tedious, error-prone and incomplete (for a lack of step indexing) *proof for the preservation lemma.* Of course, we lean on Theorem 6 to prove what amounts to a preservation lemma; the difference is that our proof properly accounts

for heap update and can be shared with other analyses that are sound wrt. by-name and by-need such as type analysis and 0CFA.

Thus, we achieve our goal of proving semantic distractions "once and for all".

8 RELATED WORK

Call-by-need, Semantics. Arguably, Josephs [1989] described the first denotational by-need semantics, predating the work of Launchbury [1993] and Sestoft [1997], but not the more machine-centric (rather than transition system centric) work on the G-machine [Johnsson 1984]. We improve on Josephs's work in that our encoding is simpler, rigorously defined (Section 5.2) and proven adequate wrt. Sestoft's by-need semantics (Section 5.1). Sestoft [1997] related the derivations of Launchbury's big-step natural semantics for our language to the subset of *balanced* small-step LK traces. Balanced traces are a proper subset of our maximal LK traces that — by nature of big-step semantics — excludes stuck and diverging traces.

Our denotational interpreter bears strong resemblance to a denotational semantics [Scott and Strachey 1971], or to a definitional interpreter [Reynolds 1972] featuring a finally encoded domain [Carette et al. 2007] using higher-order abstract syntax [Pfenning and Elliott 1988]. The key distinction to these approaches is that we generate small-step traces, totally and adequately, observable by abstract interpreters.

Definitional Interpreters. Reynolds [1972] introduced "definitional interpreter" as an umbrella term to classify prevalent styles of interpreters for higher-order languages at the time. Chiefly, it differentiates compositional interpreters that necessarily use higher-order functions of the meta language from those that do not, and are therefore non-compositional. The former correspond to (partial) denotational interpreters, whereas the latter correspond to big-step interpreters.

Ager et al. [2004] pick up on Reynold's idea and successively transform a partial denotational interpreter into a variant of the LK machine, going the reverse route of Section 5.1.

Coinduction and Fuel. 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, much like we did for a denotational semantics. The work of Atkey and McBride [2013]; Møgelberg and Veltri [2019] had big influence on our use of the later modality and Löb induction.

Our trace type T is appropriate for tracking "pure" transition events, but it is not up to the task of modelling user input, for example. We expect that guarded interaction trees [Frumin et al. 2023; Xia et al. 2019] would be very simple to integrate into our framework to help with that.

Contextual Improvement. Abstract interpretation is useful to prove that an analysis approximates the right trace property, but it does not make any claim on whether a transformation conditional on some trace property is actually sound, yet alone an *improvement* [Moran and Sands 1999]. If we were to prove dead code elimination correct based on our notion of absence, would we use our denotational interpreter to do so? Probably not; we would try to conduct as much of the proof as possible in the equational theory, i.e., on syntax. If need be, we could always switch to denotational interpreters via Theorem 4, just as in Lemma 8. Hackett and Hutton [2019] have done so as well.

Abstract Interpretation and Relational Analysis. Cousot [2021] recently condensed his seminal work rooted in Cousot and Cousot [1977]. The book advocates a compositional, trace-generating semantics and then derives compositional analyses by calculational design, inspiring us to attempt the same. However, while Cousot and Cousot [1994, 2002] work with denotational semantics for higher-order language, it was unclear to us how to derive a compositional, *trace-generating* semantics for a higher-order language. The required changes to the domain definitions seemed daunting, to say the least. Our solution delegates this complexity to the underlying theory of guarded recursive type theory [Møgelberg and Veltri 2019].

We deliberately tried to provide a simple framework and thus stuck to cartesian (i.e., pointwise) abstraction of environments as in Cousot [2021, Chapter 27], but we expect relational abstractions to work just as well. Our generic denotational interpreter is a higher-order generalisation of the generic abstract interpreter in Cousot [2021, Chapter 21]. Our abstraction laws in Figure 13 correspond to Definition 27.1 and Theorem 6 to Theorem 27.4.

Control-Flow Analysis. CFA [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 in practice it is often $k \leqslant 1$.

The Abstracting Abstract Machines [Van Horn and Might 2010] derives a computable *reachable states semantics* [Cousot 2021] from any small-step semantics, by bounding the size of the heap. Many analyses such as control-flow analysis arise as abstractions of reachable states. In fact, we think that CFA can be used to turn any finite Trace instance such as T_{U} into a static analysis, without the need to define a custom summary mechanism.

Darais et al. [2017] and others apply the AAM recipe to big-step interpreters in the style of Reynolds. Backhouse and Backhouse [2004] and Keidel et al. [2018] show that in doing so, correctness of shared code follows by parametricity [Wadler 1989]. We found it quite elegant to utilise parametricity in this way, but unfortunately the free theorem for our interpreter is too weak because it excludes the syntactic premises in Figure 13.

Whenever AAM is involved, abstraction follows some monadic structure inherent to dynamic semantics [Darais et al. 2017; Sergey et al. 2013]. In our work, this is apparent in the Domain $(\mathsf{D}\ \tau)$ instance depending on Monad τ. Decomposing such structure into a layer of reusable monad transformers has been the subject of Darais et al. [2015] and Keidel and Erdweg [2019]. The *trace transformers* in Section 4 enable a similar reuse. Likewise, Keidel et al. [2023] discusses a sound, declarative approach to reuse fixpoint combinators which we hope to apply in implementations of our framework as well.

Summaries of Functionals vs. Call Strings. Lomet [1977] used procedure summaries to capture aliasing effects, crediting the approach to untraceable reports by Allen [1974] and Rosen [1975]. Sharir et al. [1978] were aware of both [Cousot and Cousot 1977] and [Allen 1974], and generalised aliasing summaries into the "functional approach" to interprocedural data flow analysis, distinguishing it from the "call strings approach" (i.e. k-CFA).

That is not to say that the approaches cannot be combined; inter-modular analysis led Shivers [1991, Section 3.8.2] to implement the *xproc* summary mechanism. He also acknowledged the need for accurate intra-modular summary mechanisms for scalability reasons in Section 11.3.2. We are however doubtful that the powerset-centric AAM approach could integrate summary mechanisms; the whole recipe rests on the fact that the set of expressions and thus evaluation contexts is finite.

Mangal et al. [2014] have shown that a summary-based analysis can be equivalent to ∞-CFA for arbitrary complete lattices and outperform 2-CFA in both precision and speed.

Cardinality Analysis. More interesting cardinality analyses involve the inference of summaries called *demand transformers* [Sergey et al. 2017], such as implemented in the Demand Analysis of the Glasgow Haskell Compiler. The inner workings of the analysis are most similar to Clairvoyant call-by-value [Hackett and Hutton 2019], so it is a shame that the Clairvoyant instantiation leads to partiality.

REFERENCES

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

Frances E. Allen. 1974. Interprocedural Data Flow Analysis. In *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, Jack L. Rosenfeld (Ed.). North-Holland, 398–402.

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-Carrying Code. *ACM Trans. Program. Lang. Syst.* 23, 5 (sep 2001), 657–683. https://doi.org/10.1145/504709.504712

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

Robert Atkey and Conor McBride. 2013. Productive coprogramming with guarded recursion. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 197–208. https://doi.org/10.1145/2500365.2500597

Kevin Backhouse and Roland Backhouse. 2004. Safety of abstract interpretations for free, via logical relations and Galois connections. *Science of Computer Programming* 51, 1 (2004), 153–196. https://doi.org/10.1016/j.scico.2003.06.002 Mathematics of Program Construction (MPC 2002).

Lars Birkedal and Aleš Bizjak. 2023. *Lecture Notes on Iris: Higher-Order Concurrent Separation Logic.* Aarhus University, Aarhus, Denmark. https://iris-project.org/tutorial-pdfs/iris-lecture-notes.pdf.

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

Baldur Blöndal, Andres Löh, and Ryan Scott. 2018. Deriving via: or, how to turn hand-written instances into an anti-pattern. *SIGPLAN Not.* 53, 7 (sep 2018), 55–67. https://doi.org/10.1145/3299711.3242746

Joachim Breitner. 2016. *Lazy Evaluation: From natural semantics to a machine-checked compiler transformation.* Ph. D. Dissertation. Karlsruher Institut für Technologie, Fakultät für Informatik. https://doi.org/10.5445/IR/1000054251

Venanzio Capretta. 2005. General Recursion via Coinductive Types. *Logical Methods in Computer Science* Volume 1, Issue 2 (July 2005). https://doi.org/10.2168/LMCS-1(2:1)2005

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2007. Finally Tagless, Partially Evaluated. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 222–238.

Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (sep 2010), 1157–1210.

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/

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

P. Cousot and R. Cousot. 1994. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)*. 95–112. https://doi.org/10.1109/ICCL.1994.288389

Patrick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *Compiler Construction*, R. Nigel Horspool (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 159–179.

Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons. 2006. Fast and Loose Reasoning is Morally Correct. *SIGPLAN Not.* 41, 1 (jan 2006), 206–217. https://doi.org/10.1145/1111320.1111056

David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. 2017. Abstracting Definitional Interpreters (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 12 (aug 2017), 25 pages. https://doi.org/10.1145/3110256

David Darais, Matthew Might, and David Van Horn. 2015. Galois transformers and modular abstract interpreters: reusable metatheory for program analysis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. Association for Computing Machinery, New York, NY, USA, 552–571. https://doi.org/10.1145/2814270.2814308

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

Dan Frumin, Amin Timany, and Lars Birkedal. 2023. Modular Denotational Semantics for Effects with Guarded Interaction Trees. arXiv:2307.08514 [cs.PL]

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 Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 410–423. https://doi.org/10.1145/237721.240882

Thomas Johnsson. 1984. Efficient Compilation of Lazy Evaluation. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction* (Montreal, Canada) *(SIGPLAN '84)*. Association for Computing Machinery, New York, NY, USA, 58–69. https://doi.org/10.1145/502874.502880

Mark B. Josephs. 1989. The semantics of lazy functional languages. *Theoretical Computer Science* 68, 1 (1989), 105–111. https://doi.org/10.1016/0304-3975(89)90122-9

Sven Keidel and Sebastian Erdweg. 2019. Sound and reusable components for abstract interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 176 (oct 2019), 28 pages. https://doi.org/10.1145/3360602

Sven Keidel, Sebastian Erdweg, and Tobias Hombücher. 2023. Combinator-Based Fixpoint Algorithms for Big-Step Abstract Interpreters. *Proc. ACM Program. Lang.* 7, ICFP, Article 221 (aug 2023), 27 pages. https://doi.org/10.1145/3607863

Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. 2018. Compositional Soundness Proofs of Abstract Interpreters. *Proc. ACM Program. Lang.* 2, ICFP, Article 72 (jul 2018), 26 pages. https://doi.org/10.1145/3236767

L. Lamport. 1977. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering* SE-3, 2 (1977), 125–143. https://doi.org/10.1109/TSE.1977.229904

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).

D. B. Lomet. 1977. Data Flow Analysis in the Presence of Procedure Calls. *IBM Journal of Research and Development* 21, 6 (1977), 559–571. https://doi.org/10.1147/rd.216.0559

Ravi Mangal, Mayur Naik, and Hongseok Yang. 2014. A Correspondence between Two Approaches to Interprocedural Analysis in the Presence of Join. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 513–533.

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

Matthew Might. 2010. Architectures for interpreters: Substitutional, denotational, big-step and small-step. https://web.archive.org/web/20100216131108/https://matt.might.net/articles/writing-an-interpreter-substitution-denotational-big-step-small-step/. Accessed: 2010-02-16.

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

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

Alan Mycroft. 1980. The Theory and Practice of Transforming Call-by-need into Call-by-value. In *International Symposium on Programming, Proceedings of the Fourth 'Colloque International sur la Programmation', Paris, France, 22-24 April 1980 (Lecture Notes in Computer Science, Vol. 83)*, Bernard J. Robinet (Ed.). Springer, 269–281. https://doi.org/10.1007/3-540-09981-6_19

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.

Keiko Nakata. 2010. Denotational Semantics for Lazy Initialization of letrec. In *7th Workshop on Fixed Points in Computer Science, FICS 2010, Brno, Czech Republic, August 21-22, 2010*, Luigi Santocanale (Ed.). Laboratoire d'Informatique Fondamentale de Marseille, 61–67. https://hal.archives-ouvertes.fr/hal-00512377/document#page=62

Keiko Nakata and Jacques Garrigue. 2006. Recursive Modules for Programming. *SIGPLAN Not.* 41, 9 (sep 2006), 74–86. https://doi.org/10.1145/1160074.1159813

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

Simon Peyton Jones and Will Partain. 1994. *Measuring the effectiveness of a simple strictness analyser.* Springer London, London, 201–221. https://doi.org/10.1007/978-1-4471-3236-3_17

Simon L. Peyton Jones. 1992. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *Journal of Functional Programming* (1992). https://doi.org/10.1017/S0956796800000319

F. Pfenning and C. Elliott. 1988. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '88)*. Association for Computing Machinery, New York, NY, USA, 199–208. https://doi.org/10.1145/53990.54010

Benjamin C. Pierce. 2002. *Types and Programming Languages* (1st ed.). The MIT Press.

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

Barry K Rosen. 1975. *Data flow analysis for recursive PL/I programs.* IBM Thomas J. Watson Research Center.

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

Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. 2013. Monadic abstract interpreters. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 399–410. https://doi.org/10.1145/2491956.2491979

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

Micha Sharir, Amir Pnueli, et al. 1978. *Two approaches to interprocedural data flow analysis.* New York University. Courant Institute of Mathematical Sciences ....

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

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. Transfinite Iris: Resolving an Existential Dilemma of Step-Indexed Separation Logic. 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, 80–95. https://doi.org/10.1145/3453483.3454031

Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 294–305. https://doi.org/10.1145/2931037.2931074

David N. Turner, Philip Wadler, and Christian Mossin. 1995. Once upon a type. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (La Jolla, California, USA) *(FPCA '95)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/224164.224168

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

Philip Wadler. 1989. Theorems for Free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, Joseph E. Stoy (Ed.). ACM, 347–359. https://doi.org/10.1145/99370.99404

Philip Wadler and R. J. M. Hughes. 1987. Projections for strictness analysis. In *Proc. of a Conference on Functional Programming Languages and Computer Architecture* (Portland, Oregon, USA). Springer-Verlag, Berlin, Heidelberg, 385–407.

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

START OF APPENDIX

A PROOFS FOR SECTION 2 (THE PROBLEM WE SOLVE)

Theorem 1 (**$\mathcal{A}[\![\_]\!]$ infers absence**). *If $\mathcal{A}[\![\mathrm{e}]\!]_{\rho_{\mathrm{e}}} = \langle \varphi, \varsigma \rangle$ and $\varphi(\mathrm{x}) = \mathsf{A}$, then* x *is absent in* e.

Proof. See the proof at the end of this section. □

Definition 2 (**Absence**). *A variable* x *is* used *in an expression* e *if and only if there exists a trace* $(\mathbf{let}\ \mathrm{x} = \mathrm{e}'\ \mathbf{in}\ \mathrm{e}, \rho, \mu, \kappa) \hookrightarrow^* \ldots \xrightarrow{\textsc{Look}(\mathrm{x})} \ldots$ *that looks up the heap entry of* x*, i.e., it evaluates* x*. Otherwise,* x *is* absent *in* e.

Note that for the proofs we assume the recursive let definition

$$\mathcal{A}[\![\mathbf{let}\ \mathrm{x} = \mathrm{e}_1\ \mathbf{in}\ \mathrm{e}_2]\!]_\rho = \mathcal{A}[\![\mathrm{e}_2]\!]_{\rho[\mathrm{x} \mapsto \mathsf{lfp}(\lambda\theta.\ \mathrm{x} \& \mathcal{A}[\![\mathrm{e}_1]\!]_{\rho[\mathrm{x}\mapsto\theta]})]}.$$

The partial order on AbsTy necessary for computing the least fixpoint lfp follows structurally from $\mathsf{A} \sqsubset \mathsf{U}$ (i.e., product order, pointwise order).

Abbreviation 11. *The syntax $\theta.\varphi$ for an* AbsTy *$\theta = \langle \varphi, \varsigma \rangle$ returns the φ component of θ. The syntax $\theta.\varsigma$ returns the ς component of θ.*

Definition 12 (Abstract substitution). *We call $\varphi[\mathrm{x} \Mapsto \varphi'] \triangleq \varphi[\mathrm{x} \mapsto \mathsf{A}] \sqcup (\varphi(\mathrm{x}) * \varphi')$ the* abstract substitution *operation on* Uses *and overload this notation for* AbsTy*, so that $(\langle \varphi, \varsigma \rangle)[\mathrm{x} \Mapsto \varphi_{\mathrm{y}}] \triangleq \langle \varphi[\mathrm{x} \Mapsto \varphi_{\mathrm{y}}], \varsigma \rangle$.*

Abstract substitution is useful to give a concise description of the effect of syntactic substitution:

Lemma 13. $\mathcal{A}[\![(\bar\lambda \mathrm{x}.\mathrm{e})\ \mathrm{y}]\!]_\rho = (\mathcal{A}[\![\mathrm{e}]\!]_{\rho[\mathrm{x}\mapsto\langle[\mathrm{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]})[\mathrm{x} \Mapsto \rho(\mathrm{y}).\varphi]$.

Proof. Follows by unfolding the application and lambda case and then refolding abstract substitution. □

Lemma 14. *Lambda-bound uses do not escape their scope. That is, when* x *is lambda-bound in* e*, it is*

$$(\mathcal{A}[\![\mathrm{e}]\!]_\rho).\varphi(\mathrm{x}) = \mathsf{A}.$$

Proof. By induction on e. In the lambda case, any use of x is cleared to A when returning. □

Lemma 15. $\mathcal{A}[\![(\bar\lambda \mathrm{x}.\bar\lambda \mathrm{y}.\mathrm{e})\ \mathrm{z}]\!]_\rho = \mathcal{A}[\![\bar\lambda \mathrm{y}.((\bar\lambda \mathrm{x}.\mathrm{e})\ \mathrm{z})]\!]_\rho$.

Proof.

$$\begin{aligned}
&\mathcal{A}[\![(\bar\lambda \mathrm{x}.\bar\lambda \mathrm{y}.\mathrm{e})\ \mathrm{z}]\!]_\rho \\
&= (\mathit{fun}_{\mathrm{y}}(\lambda\theta_{\mathrm{y}}.\ \mathcal{A}[\![\mathrm{e}]\!]_{\rho[\mathrm{x}\mapsto\langle[\mathrm{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle,\mathrm{y}\mapsto\theta_{\mathrm{y}}]}))[\mathrm{x} \Mapsto \rho(\mathrm{z}).\varphi] && \text{Unfold } \mathcal{A}[\![_]\!], \text{Lemma 13} \\
&= \mathit{fun}_{\mathrm{y}}(\lambda\theta_{\mathrm{y}}.\ (\mathcal{A}[\![\mathrm{e}]\!]_{\rho[\mathrm{x}\mapsto\langle[\mathrm{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle,\mathrm{y}\mapsto\theta_{\mathrm{y}}]})[\mathrm{x} \Mapsto \rho(\mathrm{z}).\varphi]) && \rho(\mathrm{z})(\mathrm{y}) = \mathsf{A} \text{ by Lemma 14}, \mathrm{x} \neq \mathrm{y} \neq \mathrm{z} \\
&= \mathcal{A}[\![\bar\lambda \mathrm{y}.((\bar\lambda \mathrm{x}.\mathrm{e})\ \mathrm{z})]\!]_\rho && \text{Refold } \mathcal{A}[\![_]\!]
\end{aligned}$$

□

Lemma 16. $\mathcal{A}[\![(\bar\lambda \mathrm{x}.\mathrm{e})\ \mathrm{y}\ \mathrm{z}]\!]_\rho = \mathcal{A}[\![(\bar\lambda \mathrm{x}.\mathrm{e}\ \mathrm{z})\ \mathrm{y}]\!]_\rho$.

Proof.

$$\begin{aligned}
&\mathcal{A}[\![(\bar\lambda \mathrm{x}.\mathrm{e})\ \mathrm{y}\ \mathrm{z}]\!]_\rho \\
&= \mathit{app}((\mathcal{A}[\![\mathrm{e}]\!]_{\rho[\langle[\mathrm{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]})[\mathrm{x} \Mapsto \rho(\mathrm{y}).\varphi])(\rho(\mathrm{z})) && \text{Unfold } \mathcal{A}[\![_]\!], \text{Lemma 13} \\
&= \mathit{app}(\mathcal{A}[\![\mathrm{e}]\!]_{\rho[\langle[\mathrm{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]})(\rho(\mathrm{z}))[\mathrm{x} \Mapsto \rho(\mathrm{y}).\varphi] && \rho(\mathrm{z})(\mathrm{x}) = \mathsf{A} \text{ by Lemma 14}, \mathrm{y} \neq \mathrm{x} \neq \mathrm{z} \\
&= \mathcal{A}[\![(\bar\lambda \mathrm{x}.\mathrm{e}\ \mathrm{z})\ \mathrm{y}]\!]_\rho && \text{Refold } \mathcal{A}[\![_]\!]
\end{aligned}$$

□

Lemma 17. $\mathcal{A}[\![\mathbf{let}\ \mathrm{z} = (\bar\lambda \mathrm{x}.\mathrm{e}_1)\ \mathrm{y}\ \mathbf{in}\ (\bar\lambda \mathrm{x}.\mathrm{e}_2)\ \mathrm{y}]\!]_\rho = \mathcal{A}[\![(\bar\lambda \mathrm{x}.\mathbf{let}\ \mathrm{z} = \mathrm{e}_1\ \mathbf{in}\ \mathrm{e}_2)\ \mathrm{y}]\!]_\rho$.

Proof. The key of this lemma is that it is equivalent to postpone the abstract substitution from the let RHS e_1 to the let body e_2. This can easily be proved by induction on e_2, which we omit here, but indicate the respective step below as “hand-waving”. Note that we assume the (more general) recursive let semantics as defined at the begin of this section.

$$
\begin{array}{ll}
\quad \mathcal{A}\llbracket \textbf{let}\ \mathsf{z} = (\bar{\lambda}\mathsf{x}.\mathsf{e}_1)\ \mathsf{y}\ \textbf{in}\ (\bar{\lambda}\mathsf{x}.\mathsf{e}_2)\ \mathsf{y} \rrbracket_{\rho} & \\
& \quad \text{\textit{Unfold} } \mathcal{A}\llbracket\_\rrbracket \\
= \mathcal{A}\llbracket (\bar{\lambda}\mathsf{x}.\mathsf{e}_2)\ \mathsf{y} \rrbracket_{\rho[\mathsf{z}\mapsto \mathsf{lfp}(\lambda\theta.\ \mathsf{z}\&\mathcal{A}\llbracket(\bar{\lambda}\mathsf{x}.\mathsf{e}_1)\ \mathsf{y}\rrbracket_{\rho[\mathsf{z}\mapsto\theta]})]} & \\
& \quad \textit{Lemma 13} \\
= (\mathcal{A}\llbracket \mathsf{e}_2 \rrbracket_{\rho[\mathsf{x}\mapsto\langle[\mathsf{x}\mapsto \mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle,\mathsf{z}\mapsto \mathsf{lfp}(\lambda\theta.\ \mathsf{z}\&(\mathcal{A}\llbracket \mathsf{e}_1\rrbracket_{\rho[\mathsf{x}\mapsto\langle[\mathsf{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle,\mathsf{z}\mapsto\theta]})[\mathsf{x}\Mapsto\rho(\mathsf{y}).\varphi])]})[\mathsf{x} \Mapsto \rho(\mathsf{y}).\varphi] & \\
& \quad \textit{Hand-waving abov} \\
= (\mathcal{A}\llbracket \mathsf{e}_2 \rrbracket_{\rho[\mathsf{x}\mapsto\langle[\mathsf{x}\mapsto \mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle,\mathsf{z}\mapsto \mathsf{lfp}(\lambda\theta.\ \mathsf{z}\&\mathcal{A}\llbracket \mathsf{e}_1\rrbracket_{\rho[\mathsf{x}\mapsto\langle[\mathsf{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle,\mathsf{z}\mapsto\theta]})]})[\mathsf{x} \Mapsto \rho(\mathsf{y}).\varphi] & \\
& \quad \text{\textit{Refold} } \mathcal{A}\llbracket\_\rrbracket \\
= (\mathcal{A}\llbracket \textbf{let}\ \mathsf{z} = \mathsf{e}_1\ \textbf{in}\ \mathsf{e}_2 \rrbracket_{\rho[\mathsf{x}\mapsto\langle[\mathsf{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]})[\mathsf{x} \Mapsto \rho(\mathsf{y}).\varphi] & \\
& \quad \textit{Lemma 13} \\
= \mathcal{A}\llbracket (\bar{\lambda}\mathsf{x}.\textbf{let}\ \mathsf{z} = \mathsf{e}_1\ \textbf{in}\ \mathsf{e}_2)\ \mathsf{y} \rrbracket_{\rho} &
\end{array}
$$

□

Lemma 3 (Substitution). $\mathcal{A}\llbracket \mathsf{e} \rrbracket_{\rho[\mathsf{x}\mapsto\rho(\mathsf{y})]} \sqsubseteq \mathcal{A}\llbracket (\bar{\lambda}\mathsf{x}.\mathsf{e})\ \mathsf{y} \rrbracket_{\rho}$.

Proof. By induction on e.

- **Case** z: When $\mathsf{x} \neq \mathsf{z}$, then z is bound outside the lambda and can’t possibly use x, so $\rho(\mathsf{z}).\varphi(\mathsf{x}) = \mathsf{A}$. We have

$$
\begin{array}{ll}
\quad \mathcal{A}\llbracket \mathsf{z} \rrbracket_{\rho[\mathsf{x}\mapsto\rho(\mathsf{y})]} & \\
& \quad \mathsf{x} \neq \mathsf{z} \\
= \rho(\mathsf{z}) & \\
& \quad \text{\textit{Refold} } \mathcal{A}\llbracket\_\rrbracket \\
= \mathcal{A}\llbracket \mathsf{z} \rrbracket_{\rho[\mathsf{x}\mapsto\langle[\mathsf{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]} & \\
& \quad \rho(\mathsf{z}).\varphi(\mathsf{x}) = \mathsf{A} \\
= (\mathcal{A}\llbracket \mathsf{z} \rrbracket_{\rho[\mathsf{x}\mapsto\langle[\mathsf{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]})[\mathsf{x} \Mapsto \rho(\mathsf{y}).\varphi] & \\
& \quad \textit{Lemma 13} \\
= \mathcal{A}\llbracket (\bar{\lambda}\mathsf{x}.\mathsf{z})\ \mathsf{y} \rrbracket_{\rho} &
\end{array}
$$

  Otherwise, we have $\mathsf{x} = \mathsf{z}$, thus $\rho(\mathsf{x}) = \langle[\mathsf{x} \mapsto \mathsf{U}], \varsigma = \mathsf{Rep}\ \mathsf{U}\rangle$, and thus

$$
\begin{array}{ll}
\quad \mathcal{A}\llbracket \mathsf{z} \rrbracket_{\rho[\mathsf{x}\mapsto\rho(\mathsf{y})]} & \\
& \quad \mathsf{x} = \mathsf{z} \\
= \rho(\mathsf{y}) & \\
& \quad \varsigma \sqsubseteq \mathsf{Rep}\ \mathsf{U} \\
\sqsubseteq \langle \rho(\mathsf{y}).\varphi, \mathsf{Rep}\ \mathsf{U} \rangle & \\
& \quad \text{\textit{Definition of} } \_[\_ \Mapsto \_] \\
= (\langle[\mathsf{x} \mapsto \mathsf{U}], \mathsf{Rep}\ \mathsf{U}\rangle)[\mathsf{x} \mapsto \rho(\mathsf{y}).\varphi] & \\
& \quad \text{\textit{Refold} } \mathcal{A}\llbracket\_\rrbracket \\
= (\mathcal{A}\llbracket \mathsf{z} \rrbracket_{\rho[\mathsf{x}\mapsto\langle[\mathsf{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]})[\mathsf{x} \Mapsto \rho(\mathsf{y}).\varphi] & \\
& \quad \textit{Lemma 13} \\
= \mathcal{A}\llbracket (\bar{\lambda}\mathsf{x}.\mathsf{z})\ \mathsf{y} \rrbracket_{\rho} &
\end{array}
$$

- **Case** $\bar{\lambda}\mathsf{z}.\mathsf{e}'$:

$$
\begin{array}{ll}
\quad \mathcal{A}\llbracket \bar{\lambda}\mathsf{z}.\mathsf{e}' \rrbracket_{\rho[\mathsf{x}\mapsto\rho(\mathsf{y})]} & \\
& \quad \text{\textit{Unfold} } \mathcal{A}\llbracket\_\rrbracket \\
= \mathit{fun}_{\mathsf{z}}(\lambda\theta_{\mathsf{z}}.\ \mathcal{A}\llbracket \mathsf{e}' \rrbracket_{\rho[\mathsf{z}\mapsto\theta_{\mathsf{z}},\mathsf{x}\mapsto\rho(\mathsf{y})]}) & \\
& \quad \textit{Induction hypothesis, monotonicity} \\
\sqsubseteq \mathit{fun}_{\mathsf{z}}(\lambda\theta_{\mathsf{z}}.\ \mathcal{A}\llbracket (\bar{\lambda}\mathsf{x}.\mathsf{e}')\ \mathsf{y} \rrbracket_{\rho[\mathsf{z}\mapsto\theta_{\mathsf{z}}]}) & \\
& \quad \text{\textit{Refold} } \mathcal{A}\llbracket\_\rrbracket \\
= \mathcal{A}\llbracket \bar{\lambda}\mathsf{z}.((\bar{\lambda}\mathsf{x}.\mathsf{e}')\ \mathsf{y}) \rrbracket_{\rho} & \\
& \quad \textit{Lemma 15} \\
= \mathcal{A}\llbracket (\bar{\lambda}\mathsf{x}.\bar{\lambda}\mathsf{z}.\mathsf{e}')\ \mathsf{y} \rrbracket_{\rho} &
\end{array}
$$

- **Case** $\mathsf{e}'~\mathsf{z}$: When $\mathsf{x} = \mathsf{z}$:

$$\begin{aligned}
&\mathcal{A}[\![\mathsf{e}'~\mathsf{z}]\!]_{\rho[\mathsf{x}\mapsto\rho(\mathsf{y})]} \\
={}& \mathit{app}(\mathcal{A}[\![\mathsf{e}']\!]_{\rho[\mathsf{x}\mapsto\rho(\mathsf{y})]})(\rho(\mathsf{y})) && \langle\, \textit{Unfold } \mathcal{A}[\![_]\!] \,\rangle \\
\sqsubseteq{}& \mathit{app}(\mathcal{A}[\![(\bar\lambda\mathsf{x}.\mathsf{e}')~\mathsf{y}]\!]_{\rho})(\rho(\mathsf{y})) && \langle\, \textit{Induction hypothesis, monotonicity} \,\rangle \\
={}& \mathcal{A}[\![(\bar\lambda\mathsf{x}.\mathsf{e}')~\mathsf{y}~\mathsf{y}]\!]_{\rho} && \langle\, \textit{Refold } \mathcal{A}[\![_]\!] \,\rangle \\
={}& \mathcal{A}[\![(\bar\lambda\mathsf{x}.\mathsf{e}'~\mathsf{y})~\mathsf{y}]\!]_{\rho} && \langle\, \textit{Lemma 16} \,\rangle \\
={}& \mathcal{A}[\![(\bar\lambda\mathsf{x}.\mathsf{e}'~\mathsf{x})~\mathsf{y}]\!]_{\rho} && \langle\, \textit{Compositionality in } (\bar\lambda\mathsf{x}.\mathsf{e}'~\square)~\mathsf{y} \,\rangle \\
={}& \mathcal{A}[\![(\bar\lambda\mathsf{x}.\mathsf{e}'~\mathsf{z})~\mathsf{y}]\!]_{\rho} && \langle\, \mathsf{x} = \mathsf{z} \,\rangle
\end{aligned}$$

When $\mathsf{x} \neq \mathsf{z}$:

$$\begin{aligned}
&\mathcal{A}[\![\mathsf{e}'~\mathsf{z}]\!]_{\rho[\mathsf{x}\mapsto\rho(\mathsf{y})]} \\
={}& \mathit{app}(\mathcal{A}[\![\mathsf{e}']\!]_{\rho[\mathsf{x}\mapsto\rho(\mathsf{y})]})(\rho(\mathsf{z})) && \langle\, \textit{Unfold } \mathcal{A}[\![_]\!] \,\rangle \\
\sqsubseteq{}& \mathit{app}(\mathcal{A}[\![(\bar\lambda\mathsf{x}.\mathsf{e}')~\mathsf{y}]\!]_{\rho})(\rho(\mathsf{z})) && \langle\, \textit{Induction hypothesis, monotonicity} \,\rangle \\
={}& \mathcal{A}[\![(\bar\lambda\mathsf{x}.\mathsf{e}')~\mathsf{y}~\mathsf{z}]\!]_{\rho} && \langle\, \textit{Refold } \mathcal{A}[\![_]\!] \,\rangle \\
={}& \mathcal{A}[\![(\bar\lambda\mathsf{x}.\mathsf{e}'~\mathsf{z})~\mathsf{y}]\!]_{\rho} && \langle\, \textit{Lemma 16} \,\rangle
\end{aligned}$$

- **Case let** $\mathsf{z} = \mathsf{e}_1$ **in** e_2:

$$\begin{aligned}
&\mathcal{A}[\![\mathbf{let}~\mathsf{z} = \mathsf{e}_1~\mathbf{in}~\mathsf{e}_2]\!]_{\rho[\mathsf{x}\mapsto\rho(\mathsf{y})]} \\
={}& \mathcal{A}[\![\mathsf{e}_2]\!]_{\rho[\mathsf{x}\mapsto\rho(\mathsf{y}),\mathsf{z}\mapsto\mathsf{lfp}(\lambda\theta.~\mathsf{z}\&\mathcal{A}[\![\mathsf{e}_1]\!]_{\rho[\mathsf{x}\mapsto\rho(\mathsf{y}),\mathsf{z}\mapsto\theta]})]} && \langle\, \textit{Unfold } \mathcal{A}[\![_]\!] \,\rangle \\
\sqsubseteq{}& \mathcal{A}[\![(\bar\lambda\mathsf{x}.\mathsf{e}_2)~\mathsf{y}]\!]_{\rho[\mathsf{z}\mapsto\mathsf{lfp}(\lambda\theta.~\mathsf{z}\&\mathcal{A}[\![(\bar\lambda\mathsf{x}.\mathsf{e}_1)~\mathsf{y}]\!]_{\rho[\mathsf{z}\mapsto\theta]})]} && \langle\, \textit{Induction hypothesis, monotonicity} \,\rangle \\
={}& \mathcal{A}[\![\mathbf{let}~\mathsf{z} = (\bar\lambda\mathsf{x}.\mathsf{e}_1)~\mathsf{y}~\mathbf{in}~(\bar\lambda\mathsf{x}.\mathsf{e}_2)~\mathsf{y}]\!]_{\rho} && \langle\, \textit{Refold } \mathcal{A}[\![_]\!] \,\rangle \\
={}& \mathcal{A}[\![(\bar\lambda\mathsf{x}.\mathbf{let}~\mathsf{z} = \mathsf{e}_1~\mathbf{in}~\mathsf{e}_2)~\mathsf{y}]\!]_{\rho} && \langle\, \textit{Lemma 17} \,\rangle
\end{aligned}$$

□

Whenever there exists ρ such that $\rho(\mathsf{x}).\varphi \not\sqsubseteq (\mathcal{A}[\![\mathsf{e}]\!]_{\rho}).\varphi$ (recall that $\theta.\varphi$ selects the Uses in the first field of the pair θ), then also $\rho_{\mathsf{e}}(\mathsf{x}).\varphi \not\sqsubseteq \mathcal{A}[\![\mathsf{e}]\!]_{\rho_{\mathsf{e}}}$. The following lemma captures this intuition:

Lemma 18 (Diagonal factoring). *Let ρ and ρ_Δ be two environments such that $\forall \mathsf{x}.~\rho(\mathsf{x}).\varsigma = \rho_\Delta(\mathsf{x}).\varsigma$. If $\rho_\Delta.\varphi(\mathsf{x}) \sqsubseteq \rho_\Delta.\varphi(\mathsf{y})$ if and only if $\mathsf{x} = \mathsf{y}$, then every instantiation of $\mathcal{A}[\![\mathsf{e}]\!]$ factors through $\mathcal{A}[\![\mathsf{e}]\!]_{\rho_\Delta}$, that is,*

$$\mathcal{A}[\![\mathsf{e}]\!]_{\rho} = (\mathcal{A}[\![\mathsf{e}]\!]_{\rho_\Delta})[\overline{\mathsf{x} \Mapsto \rho(\mathsf{x}).\varphi}]$$

Proof. By induction on e.

- **Case** $\mathsf{e} = \mathsf{y}$: We assert $\mathcal{A}[\![\mathsf{y}]\!]_{\rho} = \rho(\mathsf{y}) = \rho_\Delta(\mathsf{y})[\mathsf{y} \Mapsto \rho(\mathsf{y}).\varphi]$ by simple unfolding.
- **Case** $\mathsf{e} = \mathsf{e}'~\mathsf{y}$:

$$\begin{aligned}
&\mathcal{A}[\![\mathsf{e}'~\mathsf{y}]\!]_{\rho} \\
={}& \mathit{app}(\mathcal{A}[\![\mathsf{e}']\!]_{\rho}, \rho(\mathsf{y})) && \langle\, \textit{Unfold } \mathcal{A}[\![_]\!] \,\rangle \\
={}& \mathit{app}((\mathcal{A}[\![\mathsf{e}']\!]_{\rho_\Delta})[\overline{\mathsf{x} \Mapsto \rho(\mathsf{x}).\varphi}], \rho_\Delta(\mathsf{y})[\overline{\mathsf{x} \Mapsto \rho(\mathsf{x}).\varphi}]). && \langle\, \textit{Induction hypothesis, variable case} \,\rangle \\
={}& \mathit{app}(\mathcal{A}[\![\mathsf{e}']\!]_{\rho_\Delta}, \rho_\Delta(\mathsf{y}))[\overline{\mathsf{x} \Mapsto \rho(\mathsf{x}).\varphi}] && \langle\, \sqcup \textit{ and } * \textit{ commute with } _[_ \Mapsto _] \,\rangle \\
={}& (\mathcal{A}[\![\mathsf{e}'~\mathsf{y}]\!]_{\rho_\Delta})[\overline{\mathsf{x} \Mapsto \rho(\mathsf{x}).\varphi}] && \langle\, \textit{Refold } \mathcal{A}[\![_]\!] \,\rangle
\end{aligned}$$

$$\boxed{C\llbracket\_\rrbracket : \mathbb{S} \to \mathsf{AbsTy}}$$

$$\begin{aligned}
C\llbracket(\mathsf{e}, \rho, \mu, \kappa)\rrbracket &= \mathit{apps}_\mu(\kappa, \mathcal{A}\llbracket\mathsf{e}\rrbracket_{\alpha(\mu)\circ\rho}) \\
\alpha(\mu) &= \mathsf{lfp}(\lambda\tilde{\mu}.\ [\mathsf{a} \mapsto \mathsf{x} \,\&\, \mathcal{A}\llbracket\mathsf{e}'\rrbracket_{\tilde{\mu}\circ\rho'} \mid \mu(\mathsf{a}) = (\mathsf{x}, \rho', \mathsf{e}')]) \\
\mathit{apps}_\mu(\mathbf{stop}, \theta) &= \theta \\
\mathit{apps}_\mu(\mathbf{ap}(\mathsf{a}) \cdot \kappa, \theta) &= \mathit{apps}_\mu(\kappa, \mathit{app}(\theta, \alpha(\mu)(\mathsf{a}))) \\
\mathit{apps}_\mu(\mathbf{upd}(\mathsf{a}) \cdot \kappa, \theta) &= \mathit{apps}_\mu(\kappa, \theta)
\end{aligned}$$

Fig. 14. Absence analysis extended to small-step configurations

- **Case** $\mathsf{e} = \bar{\lambda}\mathsf{y}.\mathsf{e}'$: Note that $\mathsf{x} \neq \mathsf{y}$ because y is not free in e.

$$\begin{aligned}
& \mathcal{A}\llbracket\bar{\lambda}\mathsf{y}.\mathsf{e}'\rrbracket_\rho && \\
&= \mathit{lam}_\mathsf{y}(\lambda\theta.\ \mathcal{A}\llbracket\mathsf{e}'\rrbracket_{\rho[\mathsf{y}\mapsto\theta]}) && \textit{Unfold } \mathcal{A}\llbracket_\rrbracket \\
&= \mathit{lam}_\mathsf{y}(\lambda\theta.\ (\mathcal{A}\llbracket\mathsf{e}'\rrbracket_{\rho[\mathsf{y}\mapsto\langle[\mathsf{y}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]})) && \textit{Property of } \mathit{lam}_\mathsf{y} \\
&= \mathit{lam}_\mathsf{y}(\lambda\theta.\ (\mathcal{A}\llbracket\mathsf{e}'\rrbracket_{\rho_\Delta[\mathsf{y}\mapsto\langle[\mathsf{y}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]})[\overline{\mathsf{x} \Mapsto \rho(\mathsf{x}).\varphi}, \mathsf{y} \Mapsto [\mathsf{y} \mapsto \mathsf{U}]]) && \textit{Induction hypothesis} \\
&= \mathit{lam}_\mathsf{y}(\lambda\theta.\ (\mathcal{A}\llbracket\mathsf{e}'\rrbracket_{\rho_\Delta[\mathsf{y}\mapsto\langle[\mathsf{y}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]})[\overline{\mathsf{x} \Mapsto \rho(\mathsf{x}).\varphi}]) && \theta[\mathsf{y} \Mapsto [\mathsf{y} \mapsto \mathsf{U}]] = \theta \\
&= \mathit{lam}_\mathsf{y}(\lambda\theta.\ (\mathcal{A}\llbracket\mathsf{e}'\rrbracket_{\rho_\Delta[\mathsf{y}\mapsto\theta]})[\overline{\mathsf{x} \Mapsto \rho(\mathsf{x}).\varphi}]) && \theta[\mathsf{y} \Mapsto [\mathsf{y} \mapsto \mathsf{U}]] = \theta \\
&= \mathit{lam}_\mathsf{y}(\lambda\theta.\ \mathcal{A}\llbracket\mathsf{e}'\rrbracket_{\rho_\Delta[\mathsf{y}\mapsto\theta]})[\overline{\mathsf{x} \Mapsto \rho(\mathsf{x}).\varphi}] && \textit{Property of } \mathit{lam}_\mathsf{y} \\
&= (\mathcal{A}\llbracket\bar{\lambda}\mathsf{y}.\mathsf{e}'\rrbracket_{\rho_\Delta})[\overline{\mathsf{x} \Mapsto \rho(\mathsf{x}).\varphi}] && \textit{Refold } \mathcal{A}\llbracket_\rrbracket
\end{aligned}$$

- **Case let** $\mathsf{y} = \mathsf{e}_1$ **in** e_2: Note that $\mathsf{x} \neq \mathsf{y}$ because y is not free in e.

$$\begin{aligned}
& \mathcal{A}\llbracket\mathbf{let}\ \mathsf{y} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2\rrbracket_\rho && \\
&= \mathcal{A}\llbracket\mathsf{e}_2\rrbracket_{\rho[\mathsf{y}\mapsto\mathsf{lfp}(\lambda\theta.\ \mathsf{y}\&\mathcal{A}\llbracket\mathsf{e}_1\rrbracket_{\rho[\mathsf{y}\mapsto\theta]})]} && \textit{Unfold } \mathcal{A}\llbracket_\rrbracket \\
&= \mathcal{A}\llbracket\mathsf{e}_2\rrbracket_{\rho[\mathsf{y}\mapsto\mathsf{lfp}(\lambda\theta.\ \mathsf{y}\&(\mathcal{A}\llbracket\mathsf{e}_1\rrbracket_{\rho_\Delta[\mathsf{y}\mapsto\langle[\mathsf{y}\mapsto\mathsf{U}],\theta.\varsigma\rangle]})[\overline{\mathsf{x}\Mapsto\rho(\mathsf{x}).\varphi},\mathsf{y}\Mapsto\theta.\varphi])]} && \textit{Induction hypothesis} \\
&= \mathcal{A}\llbracket\mathsf{e}_2\rrbracket_{\rho[\mathsf{y}\mapsto\mathsf{lfp}(\lambda\theta.\ \mathsf{y}\&(\mathcal{A}\llbracket\mathsf{e}_1\rrbracket_{\rho_\Delta[\mathsf{y}\mapsto\theta]})[\overline{\mathsf{x}\Mapsto\rho(\mathsf{x}).\varphi}])]} && \textit{Again, backwards} \\
& \textit{Similarly for } \mathsf{e}_2\textit{, hand-waving to push out the subst as in Lemma 17} && \\
&= (\mathcal{A}\llbracket\mathsf{e}_2\rrbracket_{\rho_\Delta[\mathsf{y}\mapsto\mathsf{lfp}(\lambda\theta.\ \mathsf{y}\&\mathcal{A}\llbracket\mathsf{e}_1\rrbracket_{\rho_\Delta[\mathsf{y}\mapsto\theta]})]})[\overline{\mathsf{x} \Mapsto \rho(\mathsf{x}).\varphi}] && \textit{Refold } \mathcal{A}\llbracket_\rrbracket \\
&= (\mathcal{A}\llbracket\mathbf{let}\ \mathsf{y} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2\rrbracket_{\rho_\Delta})[\overline{\mathsf{x} \Mapsto \rho(\mathsf{x}).\varphi}] &&
\end{aligned}$$

□

For the purposes of the preservation proof, we will write $\tilde{\rho}$ with a tilde to denote that abstract environment of type $\mathsf{Var} \to \mathsf{AbsTy}$, to disambiguate it from a concrete environment ρ from the LK machine.

In Figure 14, we give the extension of $C\llbracket\_\rrbracket$ to whole machine configurations σ. Although $C\llbracket\_\rrbracket$ looks like an entirely new definition, it is actually derivative of $\mathcal{A}\llbracket\_\rrbracket$ via a context lemma à la Moran and Sands [1999, Lemma 3.2]: The environments ρ simply govern the transition from syntax to operational representation in the heap. The bindings in the heap are to be treated as mutually recursive let bindings, hence a fixpoint is needed. For safety properties such as absence, a least fixpoint is appropriate. Apply frames on the stack correspond to the application case of $\mathcal{A}\llbracket\_\rrbracket$ and invoke the summary mechanism. Update frames are ignored because our analysis is not heap-sensitive.

Now we can prove that $\mathcal{C}[\![\_]\!]$ is preserved/improves during reduction:

Lemma 19 (Preservation of $\mathcal{C}[\![\_]\!]$). *If $\sigma_1 \hookrightarrow \sigma_2$, then $\mathcal{C}[\![\sigma_1]\!] \sqsupseteq \mathcal{C}[\![\sigma_2]\!]$.*

Proof. By cases on the transition.

- **Case** Let$_1$: Then $\mathsf{e} = \mathbf{let}\ \mathsf{y} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2$ and

$$(\mathbf{let}\ \mathsf{y} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2, \rho, \mu, \kappa) \hookrightarrow (\mathsf{e}_2, \rho[\mathsf{y} \mapsto \mathsf{a}], \mu[\mathsf{a} \mapsto (\mathsf{y}, \rho[\mathsf{y} \mapsto \mathsf{a}], \mathsf{e}_1)], \kappa).$$

  Abbreviating $\rho_1 \triangleq \rho[\mathsf{y} \mapsto \mathsf{a}]$, $\mu_1 \triangleq \mu[\mathsf{a} \mapsto (\mathsf{y}, \rho_1, \mathsf{e}_1)$, we have

$$\begin{aligned}
& \mathcal{C}[\![\sigma_1]\!] && \text{Unfold } \mathcal{C}[\![\sigma_1]\!] \\
&= \mathit{apps}_\mu(\kappa)(\mathcal{A}[\![\mathbf{let}\ \mathsf{y} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2]\!]_{\alpha(\mu)\circ\rho}) && \text{Unfold } \mathcal{A}[\![\mathbf{let}\ \mathsf{y} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2]\!] \\
&= \mathit{apps}_\mu(\kappa)(\mathcal{A}[\![\mathsf{e}_2]\!]_{(\alpha(\mu)\circ\rho)[\mathsf{y}\mapsto \mathsf{y} \,\&\, \mathsf{lfp}(\lambda\theta.\ \mathcal{A}[\![\mathsf{e}_1]\!]_{(\alpha(\mu)\circ\rho)[\mathsf{y}\mapsto\theta]})]}) && \text{Move fixpoint outwards, refold } \alpha \\
&= \mathit{apps}_{\mu_1}(\kappa)(\mathcal{A}[\![\mathsf{e}_2]\!]_{\alpha(\mu_1)\circ\rho_1}) && \text{Refold } \mathcal{C}[\![\sigma_2]\!] \\
&= \mathcal{C}[\![\sigma_2]\!]
\end{aligned}$$

- **Case** App$_1$: Then $(\mathsf{e}'\ \mathsf{y}, \rho, \mu, \kappa) \hookrightarrow (\mathsf{e}', \rho, \mu, \mathbf{ap}(\rho(\mathsf{y})) \cdot \kappa)$.

$$\begin{aligned}
& \mathcal{C}[\![\sigma_1]\!] && \text{Unfold } \mathcal{C}[\![\sigma_1]\!] \\
&= \mathit{apps}_\mu(\kappa)(\mathcal{A}[\![\mathsf{e}'\ \mathsf{y}]\!]_{\alpha(\mu)\circ\rho}) && \text{Unfold } \mathcal{A}[\![\mathsf{e}'\ \mathsf{y}]\!]_{(\alpha(\mu)\circ\rho)} \\
&= \mathit{apps}_\mu(\kappa)(\mathit{app}(\mathcal{A}[\![\mathsf{e}']\!]_{\alpha(\mu)\circ\rho}, \alpha(\mu)(\rho(\mathsf{y})))) && \text{Rearrange} \\
&= \mathit{apps}_\mu(\mathbf{ap}(\rho(\mathsf{y})) \cdot \kappa)(\mathcal{A}[\![\mathsf{e}']\!]_{\alpha(\mu)\circ\rho}) && \text{Refold } \mathcal{C}[\![\sigma_2]\!] \\
&= \mathcal{C}[\![\sigma_2]\!]
\end{aligned}$$

- **Case** App$_2$: Then $(\bar{\lambda}\mathsf{y}.\mathsf{e}', \rho, \mu, \mathbf{ap}(\mathsf{a}) \cdot \kappa) \hookrightarrow (\mathsf{e}', \rho[\mathsf{y} \mapsto \mathsf{a}], \mu, \kappa)$.

$$\begin{aligned}
& \mathcal{C}[\![\sigma_1]\!] && \text{Unfold } \mathcal{C}[\![\sigma_1]\!] \\
&= \mathit{apps}_\mu(\mathbf{ap}(\mathsf{a}) \cdot \kappa)(\mathcal{A}[\![\bar{\lambda}\mathsf{y}.\mathsf{e}']\!]_{\alpha(\mu)\circ\rho}) && \text{Unfold apps} \\
&= \mathit{apps}_\mu(\kappa)(\mathit{app}(\mathcal{A}[\![\bar{\lambda}\mathsf{y}.\mathsf{e}']\!]_{\alpha(\mu)\circ\rho}, \alpha(\mu)(\mathsf{a}))) && \text{Unfold RHS of Lemma 3} \\
&\sqsupseteq \mathit{apps}_\mu(\kappa)(\mathcal{A}[\![\mathsf{e}']\!]_{(\alpha(\mu)\circ\rho)[\mathsf{y}\mapsto\alpha(\mu)(\mathsf{a})]}) && \text{Rearrange} \\
&= \mathit{apps}_\mu(\kappa)(\mathcal{A}[\![\mathsf{e}']\!]_{(\alpha(\mu)\circ\rho[\mathsf{y}\mapsto\mathsf{a}])}) && \text{Refold } \mathcal{C}[\![\sigma_2]\!] \\
&= \mathcal{C}[\![\sigma_2]\!]
\end{aligned}$$

- **Case** Look: Then $\mathsf{e} = \mathsf{y}$, $\mathsf{a} \triangleq \rho(\mathsf{y})$, $(\mathsf{z}, \rho', \mathsf{e}') \triangleq \mu(\mathsf{a})$ and $(\mathsf{y}, \rho, \mu, \kappa) \hookrightarrow (\mathsf{e}', \rho', \mu, \mathbf{upd}(\mathsf{a}) \cdot \kappa)$.

$$\begin{aligned}
& \mathcal{C}[\![\sigma_1]\!] && \text{Unfold } \mathcal{C}[\![\sigma_1]\!] \\
&= \mathit{apps}_\mu(\kappa)(\mathcal{A}[\![\mathsf{y}]\!]_{\alpha(\mu)\circ\rho}) && \text{Unfold } \mathcal{A}[\![\mathsf{y}]\!] \\
&= \mathit{apps}_\mu(\kappa)((\alpha(\mu) \circ \rho)(\mathsf{y})) && \text{Unfold } \alpha \\
&= \mathit{apps}_\mu(\kappa)(\mathsf{z} \,\&\, \mathcal{A}[\![\mathsf{e}']\!]_{\alpha(\mu)\circ\rho'}) && \text{Drop } [\mathsf{z} \mapsto \mathsf{U}] \\
&\sqsupseteq \mathit{apps}_\mu(\kappa)(\mathcal{A}[\![\mathsf{e}']\!]_{\alpha(\mu)\circ\rho'}) && \text{Definition of } \mathit{apps}_\mu \\
&= \mathit{apps}_\mu(\mathbf{upd}(\mathsf{a}) \cdot \kappa)(\mathcal{A}[\![\mathsf{e}']\!]_{\alpha(\mu)\circ\rho'}) && \text{Refold } \mathcal{C}[\![\sigma_2]\!] \\
&= \mathcal{C}[\![\sigma_2]\!]
\end{aligned}$$

- **Case** Upd: Then $(\mathsf{v}, \rho, \mu[\mathsf{a} \mapsto (\mathsf{y}, \rho', \mathsf{e}')], \mathbf{upd}(\mathsf{a}) \cdot \kappa) \hookrightarrow (\mathsf{v}, \rho, \mu[\mathsf{a} \mapsto (\mathsf{y}, \rho, \mathsf{v})], \kappa)$.

This case is a bit hand-wavy and shows how heap update during by-need evaluation is dreadfully complicated to handle, even though $\mathcal{A}[\![\_]\!]$ is heap-less and otherwise correct wrt. by-name evaluation. The culprit is that in order to show $\mathcal{C}[\![\sigma_2]\!] \sqsubseteq \mathcal{C}[\![\sigma_1]\!]$, we have to show

$$\mathcal{A}[\![\mathrm{v}]\!]_{\alpha(\mu)\circ\rho} \sqsubseteq \mathcal{A}[\![\mathrm{e}']\!]_{\alpha(\mu')\circ\rho'}. \tag{1}$$

Intuitively, this is somewhat clear, because μ "evaluates to" μ' and v is the value of e′, in the sense that there exists $\sigma' = (\mathrm{e}', \rho', \mu', \kappa)$ such that $\sigma' \hookrightarrow^* \sigma_1 \hookrightarrow \sigma_2$.
Alas, who guarantees that such a σ' actually exists? We would need to rearrange the lemma for that and argue by step indexing (a.k.a. coinduction) over prefixes of *maximal traces* (to be rigorously defined later). That is, we presume that the statement

$$\forall n.\ \sigma_0 \hookrightarrow^n \sigma_2 \Longrightarrow \mathcal{C}[\![\sigma_2]\!] \sqsubseteq \mathcal{C}[\![\sigma_0]\!]$$

has been proved for all $n < k$ and proceed to prove it for $n = k$. So we presume $\sigma_0 \hookrightarrow^{k-1} \sigma_1 \hookrightarrow \sigma_2$ and $\mathcal{C}[\![\sigma_1]\!] \sqsubseteq \mathcal{C}[\![\sigma_0]\!]$ to arrive at a similar setup as before, only with a stronger assumption about σ_1. Specifically, due to the balanced stack discipline we know that $\sigma_0 \hookrightarrow^{k-1} \sigma_1$ factors over σ' above. We may proceed by induction over the balanced stack discipline (we will see in Section 5.1 that this amounts to induction over the big-step derivation) of the trace $\sigma' \hookrightarrow^* \sigma_1$ to show Equation (1).
This reasoning was not specific to $\mathcal{A}[\![\_]\!]$ at all. We will show a more general result in Lemma 53.(a) that can be reused across many more analyses.
Assuming Equation (1) has been proved, we proceed

$$\begin{aligned}
& \mathcal{C}[\![\sigma_1]\!] && \\
={}& \mathit{apps}_\mu(\mathbf{upd}(\mathrm{a}) \cdot \kappa)(\mathcal{A}[\![\mathrm{v}]\!]_{\alpha(\mu)\circ\rho}) && \text{\textit{Unfold} } \mathcal{C}[\![\sigma_1]\!] \\
={}& \mathit{apps}_\mu(\kappa)(\mathcal{A}[\![\mathrm{v}]\!]_{\alpha(\mu)\circ\rho}) && \text{\textit{Definition of} } \mathit{apps}_\mu \\
\sqsupseteq{}& \mathit{apps}_{\mu[\mathrm{a}\mapsto(\mathrm{y},\rho,\mathrm{v})]}(\kappa)(\mathcal{A}[\![\mathrm{v}]\!]_{\alpha(\mu[\mathrm{a}\mapsto(\mathrm{y},\rho,\mathrm{v})])\circ\rho}) && \text{\textit{Above argument that} } \mathcal{A}[\![\mathrm{v}]\!]_{\alpha(\mu)\circ\rho} \sqsubseteq \mathcal{A}[\![\mathrm{e}']\!]_{\alpha(\mu')\circ\rho'} \\
={}& \mathcal{C}[\![\sigma_2]\!] && \text{\textit{Refold} } \mathcal{C}[\![\sigma_2]\!]
\end{aligned}$$

□

We conclude with the proof for Theorem 1:

Proof. We show the contraposition, that is, if x is used in e, then $\varphi(\mathrm{x}) = \mathrm{U}$.
Since x is used in e, there exists a trace

$$(\mathbf{let}\ \mathrm{x} = \mathrm{e}'\ \mathbf{in}\ \mathrm{e}, \rho, \mu, \kappa) \hookrightarrow (\mathrm{e}, \rho_1, \mu_1, \kappa) \hookrightarrow^* (\mathrm{y}, \rho'[\mathrm{y} \mapsto \mathrm{a}], \mu', \kappa') \xrightarrow{\textsc{Look}(\mathrm{x})} \ldots,$$

where $\rho_1 \triangleq \rho[\mathrm{x} \mapsto \mathrm{a}]$, $\mu_1 \triangleq \mu[\mathrm{a} \mapsto (\mathrm{x}, \rho[\mathrm{x} \mapsto \mathrm{a}], \mathrm{e}')]$. Without loss of generality, we assume the trace prefix ends at the first lookup at a, so $\mu'(\mathrm{a}) = \mu_1(\mathrm{a}) = (\mathrm{x}, \rho_1, \mathrm{e}')$. If that was not the case, we could just find a smaller prefix with this property.

Let us abbreviate $\tilde{\rho} \triangleq (\alpha(\mu_1) \circ \rho_1)$. Under the above assumptions, $\tilde{\rho}(\mathrm{y}).\varphi(\mathrm{x}) = \mathrm{U}$ implies $\mathrm{x} = \mathrm{y}$ for all y, because $\mu_1(\mathrm{a})$ is the only heap entry in which x occurs by our shadowing assumptions on syntax. By unfolding $\mathcal{C}[\![\_]\!]$ and $\mathcal{A}[\![\mathrm{y}]\!]$ we can see that

$$[\mathrm{x} \mapsto \mathrm{U}] \sqsubseteq \alpha(\mu_1)(\mathrm{a}).\varphi = \alpha(\mu')(\mathrm{a}).\varphi = \mathcal{A}[\![\mathrm{y}]\!]_{\alpha(\mu')\circ\rho'[\mathrm{y}\mapsto\mathrm{a}]}.\varphi \sqsubseteq (\mathcal{C}[\![(\mathrm{y}, \rho'[\mathrm{y} \mapsto \mathrm{a}], \mu', \kappa')]\!]).\varphi.$$

By Lemma 19, we also have

$$(\mathcal{C}[\![(\mathrm{y}, \rho'[\mathrm{y} \mapsto \mathrm{a}], \mu', \kappa')]\!]).\varphi \sqsubseteq (\mathcal{C}[\![(\mathrm{e}, \rho_1, \mu_1, \kappa)]\!]).\varphi.$$

And with transitivity, we get $[\mathsf{x} \mapsto \mathsf{U}] \sqsubseteq (\mathcal{C}[\![(\mathsf{e}, \rho_1, \mu_1, \kappa)]\!]).\varphi$. Since there was no other heap entry for x and a cannot occur in κ or ρ due to well-addressedness, we have $[\mathsf{x} \mapsto \mathsf{U}] \sqsubseteq (\mathcal{C}[\![(\mathsf{e}, \rho_1, \mu_1, \kappa)]\!]).\varphi$ if and only if $[\mathsf{x} \mapsto \mathsf{U}] \sqsubseteq (\mathcal{A}[\![\mathsf{e}]\!]_{\tilde{\rho}}).\varphi$. With Lemma 18, we can decompose

$$
\begin{array}{ll}
\quad [\mathsf{x} \mapsto \mathsf{U}] & \\
\sqsubseteq (\mathcal{A}[\![\mathsf{e}]\!]_{\tilde{\rho}}).\varphi & \rangle\ \textit{Above result} \\
= ((\mathcal{A}[\![\mathsf{e}]\!]_{\tilde{\rho}_\Delta})[\overline{\mathsf{y} \Mapsto \tilde{\rho}(\mathsf{y}).\varphi}]).\varphi & \rangle\ \tilde{\rho}_\Delta(\mathsf{x}) \triangleq \langle [\mathsf{x} \mapsto \mathsf{U}], \tilde{\rho}(\mathsf{x}).\varsigma \rangle, \textit{Lemma 18} \\
\sqsubseteq ((\mathcal{A}[\![\mathsf{e}]\!]_{\tilde{\rho}_\mathsf{e}})[\overline{\mathsf{y} \Mapsto \tilde{\rho}(\mathsf{y}).\varphi}]).\varphi & \rangle\ \varsigma \sqsubseteq \mathsf{Rep}\ \mathsf{U}, \textit{hence } \tilde{\rho}_\Delta \sqsubseteq \tilde{\rho}_\mathsf{e} \\
= \bigsqcup \{ \tilde{\rho}(\mathsf{y}).\varphi \mid \mathcal{A}[\![\mathsf{e}]\!]_{\tilde{\rho}_\mathsf{e}}.\varphi(\mathsf{y}) = \mathsf{U} \} & \rangle\ \textit{Definition of } \_[\_ \Mapsto \_]
\end{array}
$$

But since $\tilde{\rho}(\mathsf{y}).\varphi(\mathsf{x}) = \mathsf{U}$ implies $\mathsf{x} = \mathsf{y}$ (refer to definition of $\tilde{\rho}$), we must have $(\mathcal{A}[\![\mathsf{e}]\!]_{\tilde{\rho}_\mathsf{e}}).\varphi(\mathsf{x}) = \mathsf{U}$, as required. ◻

B PROOFS FOR SECTION 5 (TOTALITY AND SEMANTIC ADEQUACY)

Theorem 4 (Strong Adequacy). *Let* e *be a closed expression,* $\tau \triangleq \mathcal{S}_{\mathbf{need}}[\![e]\!]_{\varepsilon}(\varepsilon)$ *the denotational by-need trace and* $\mathit{init}(\mathsf{e}) \hookrightarrow \ldots$ *the maximal lazy Krivine trace. Then*

- τ *preserves the observable termination properties of* $\mathit{init}(\mathsf{e}) \hookrightarrow \ldots$ *in the above sense.*
- τ *preserves the length (i.e., number of* Step*s) of* $\mathit{init}(\mathsf{e}) \hookrightarrow \ldots$ *(i.e., number of transitions).*
- *every* ev :: Event *in* τ = Step ev ... *corresponds to the transition rule taken in* $\mathit{init}(\mathsf{e}) \hookrightarrow \ldots$*.*

Proof. We formally define as $\alpha(\mathit{init}(\mathsf{e}) \hookrightarrow \ldots) \triangleq \alpha_{\mathbb{S}^\infty}(\mathit{init}(\mathsf{e}) \hookrightarrow \ldots, \mathbf{stop})$, where $\alpha_{\mathbb{S}^\infty}$ is defined in Figure 15.

Then $\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\varepsilon}(\varepsilon) = \alpha(\mathit{init}(\mathsf{e}) \hookrightarrow \ldots)$ follows directly from Theorem 27. The preservation results in are a consequence of Lemma 25 and theorem 28; function $\alpha_{\mathbb{E}_\mathsf{v}}$ in Figure 15 encodes the intuition in which LK transitions abstract into Events. ◻

We proceed from the bottom up, beginning with a definition of traces as mathematical sequences, then defining maximal traces, and then relating those maximal traces via Figure 15 to $\mathcal{S}[\![\_]\!]\_$.

Formally, an LK trace is a trace in $(\hookrightarrow)$ from Figure 2, i.e., a non-empty and potentially infinite sequence of LK states $(\sigma_i)_{i \in \overline{n}}$ (where $\overline{n} = \{m \in \mathbb{N} \mid m < n\}$ 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. When the control expression of a state σ (selected via $\mathit{ctrl}(\sigma)$) is a value v, we call σ a *return* state and say that the continuation (selected via $\mathit{cont}(\sigma)$) drives evaluation. Otherwise, σ is an *evaluation* state and $\mathit{ctrl}(\sigma)$ drives evaluation.

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

Definition 20 (Deep, interior and balanced traces)**.** *An LK trace* $(\sigma_i)_{i \in \overline{n}}$ *is* κ-deep *if every intermediate continuation* $\kappa_i \triangleq \mathit{cont}(\sigma_i)$ *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* $\mathit{cont}(\sigma_0)$*-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* $\mathit{cont}(\sigma_0)$*.*
We notate κ*-deep and interior traces as* $\kappa\ \mathsf{deep}\ (\sigma_i)_{i \in \overline{n}}$ *and* $(\sigma_i)_{i \in \overline{n}}\ \mathsf{inter}$*, respectively.*

Here is an example for each of the three cases. We will omit the first component of heap entries in our examples because they bear no semantic significance apart from instrumenting Look transitions, and it is confusing when the heap-bound expression is a variable x, e.g., (y, ρ, x).

Example 21. *Let* $\rho = [x \mapsto \mathsf{a}_1]$, $\mu = [\mathsf{a}_1 \mapsto (\_, [], \bar{\lambda} y.y)]$ *and* κ *an arbitrary continuation. The trace*

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

is interior and balanced. Its proper 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.

As shown by Sestoft [1997], a balanced trace starting at a control expression e and ending with v loosely corresponds to a derivation of $\mathsf{e} \Downarrow \mathsf{v}$ in a natural big-step semantics or a non-$\bot$ result in a Scott-style 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 22 (Maximal, diverging and stuck traces). *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,*

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

We notate maximal traces as $(\sigma_i)_{i\in\overline{n}}\ \mathsf{max}$. Infinite and interior traces are called 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 23 (Stuck and diverging traces). *Consider the interior trace*

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

where `tt` *is a data constructor. 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*

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

Lemma 24 (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.*

Proof. $\Rightarrow$: Let $(\sigma_i)_{i\in\overline{n}}$ be maximal. If $n = \omega$ is infinite, then it is diverging due to interiority, and if $(\sigma_i)_{i\in\overline{n}}$ is stuck, the goal follows immediately. So we assume that $(\sigma_i)_{i\in\overline{n}}$ is maximal, finite and not stuck, so it must be balanced by the definition of stuckness.

$\Leftarrow$: Both balanced and stuck traces are maximal. A diverging trace $(\sigma_i)_{i\in\overline{n}}$ is interior and infinite, hence $n = \omega$. Indeed $(\sigma_i)_{i\in\overline{\omega}}$ is maximal, because the expression σ_ω is undefined and hence does not exist. □

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. This is very much like the semantics of a called function (i.e., big-step evaluator) may not depend on the contents of the call stack.

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

Figure 15 shows the correctness predicate C in our endeavour to prove $\mathcal{S}[\![\_]\!]_\_$ adequate at D (ByNeed T). It encodes that an *abstraction* of every maximal LK trace can be recovered by running $\mathcal{S}[\![\_]\!]_\_$ starting from the abstraction of an initial state.

The family of abstraction functions (they are really *representation functions*, in the sense of Section 7) makes precise the intuitive connection between the definable entities in $\mathcal{S}[\![\_]\!]_\_$ and the syntactic objects in the transition system.

$$
\begin{array}{rcl}
\alpha_{\mathbb{E}}(\mu, [\overline{\mathsf{x} \mapsto \mathsf{a}}]) & = & [\overline{x \mapsto \mathsf{Step}\ (\mathsf{Lookup}\ y)\ (\mathit{fetch}\ a) \mid \mu(\mathsf{a}) = (\mathsf{y}, \_, \_)}] \\
\alpha_{\mathbb{H}}([\overline{\mathsf{a} \mapsto (\_, \rho, \mathsf{e})}]) & = & [\overline{a \mapsto \mathit{memo}\ a\ (\mathcal{S}[\![e]\!]_{\alpha_{\mathbb{E}}(\mu,\rho)})}] \\
\alpha_{\mathbb{S}}(\bar{\lambda}\mathsf{x}.\mathsf{e}, \rho, \mu, \kappa) & = & (\mathsf{Fun}\ (\lambda d \rightarrow \mathsf{Step}\ \mathsf{App}_2\ (\mathcal{S}[\![e]\!]_{(\alpha_{\mathbb{E}}(\mu,\rho))[x \mapsto d]})), \alpha_{\mathbb{H}}(\mu)) \\
\alpha_{\mathbb{S}}(K\ \overline{\mathsf{x}}, \rho, \mu, \kappa) & = & (\mathsf{Con}\ k\ (\mathit{map}\ (\alpha_{\mathbb{E}}(\mu, \rho)\ !)\ \mathit{xs}), \alpha_{\mathbb{H}}(\mu)) \\
\alpha_{\mathbb{E}_\mathbb{v}}(\sigma) & = & \begin{cases}
\mathsf{Let}_1 & \text{when } \sigma = (\mathbf{let}\ \mathsf{x} = \_\ \mathbf{in}\ \_, \_, \mu, \_), \mathsf{a}_{\mathsf{x},i} \notin \mathsf{dom}(\mu) \\
\mathsf{App}_1 & \text{when } \sigma = (\_\ \mathsf{x}, \_, \_, \_) \\
\mathsf{Case}_1 & \text{when } \sigma = (\mathbf{case}\ \_\ \mathbf{of}\ \_, \_, \_, \_) \\
\mathsf{Lookup}\ y & \text{when } \sigma = (\mathsf{x}, \rho, \mu, \_), \mu(\rho(\mathsf{x})) = (\mathsf{y}, \_, \_) \\
\mathsf{App}_2 & \text{when } \sigma = (\bar{\lambda}\_.\_, \_, \_, \mathbf{ap}(\_) \cdot \_) \\
\mathsf{Case}_2 & \text{when } \sigma = (K\ \_, \_, \_, \mathbf{sel}(\_, \_) \cdot \_) \\
\mathsf{Update} & \text{when } \sigma = (\mathsf{v}, \_, \_, \mathbf{upd}(\_) \cdot \_)
\end{cases} \\
\alpha_{\mathbb{S}^\infty}((\sigma_i)_{i \in \overline{n}}, \kappa) & = & \begin{cases}
\mathsf{Step}\ (\alpha_{\mathbb{E}_\mathbb{v}}(\sigma_0))\ \{\!|\alpha_{\mathbb{S}^\infty}((\sigma_{i+1})_{i \in \overline{n-1}}, \kappa)|\!\} & \text{when } n > 0 \\
\mathsf{Ret}\ (\alpha_{\mathbb{S}}(\sigma_0)) & \text{when } \mathit{ctrl}(\sigma_0) \text{ value } \wedge \mathit{cont}(\sigma_0) = \kappa \\
\mathsf{Ret}\ \mathsf{Stuck} & \text{otherwise}
\end{cases} \\
C((\sigma_i)_{i \in \overline{n}}) & = & (\sigma_i)_{i \in \overline{n}} \text{ max} \Longrightarrow \forall ((\mathsf{e}, \rho, \mu, \kappa) = \sigma_0).\ \alpha_{\mathbb{S}^\infty}((\sigma_i)_{i \in \overline{n}}, \kappa) = \mathcal{S}_{\mathbf{need}}[\![e]\!]_{\alpha_{\mathbb{E}}(\mu,\rho)}(\alpha_{\mathbb{H}}(\mu))
\end{array}
$$

Fig. 15. Correctness predicate for $\mathcal{S}[\![\_]\!]_\_$

We will sometimes need to disambiguate the clashing definitions from Section 4 and Section 2. We do so by adorning semantic objects with a tilde, so $\tilde{\mu} \triangleq \alpha_{\mathbb{H}}(\mu) :: \mathsf{Heap}\ (\mathsf{ByNeed}\ \tau)$ denotes a semantic heap which in this instance is defined to be the abstraction of a syntactic heap μ.

Note first that $\alpha_{\mathbb{S}^\infty}$ is defined by guarded recursion over the LK trace, in the following sense: We regard $(\sigma_i)_{i \in \overline{n}}$ as a Sigma type $\mathbb{S}^\infty \triangleq \exists n \in \mathbb{N}_\omega.\ \overline{n} \rightarrow \mathbb{S}$, where $\mathbb{N}_\omega$ is defined by guarded recursion as **data** $\mathbb{N}_\omega = \mathsf{Z} \mid \mathsf{S}\ (\blacktriangleright \mathbb{N}_\omega)$. Now $\mathbb{N}_\omega$ contains all natural numbers (where n is encoded as $(\mathsf{S} \circ \mathit{pure}\ )^n\mathsf{Z})$ and the transfinite limit ordinal $\omega = \mathsf{S}\ (\mathit{pure}\ (\mathsf{S}\ (\mathit{pure}...)))$. We will assume that addition and subtraction are defined as on Peano numbers, and $\omega + \_ = \_ + \omega = \omega$. When $(\sigma_i)_{i \in \overline{n}} \in \mathbb{S}^\infty$ is an LK trace and $n > 1$, then $(\sigma_{i+1})_{i \in \overline{n-1}} \in \blacktriangleright\mathbb{S}^\infty$ is the guarded tail of the trace with an associated coinduction principle.

As such, the expression $\{\!|\alpha_{\mathbb{S}^\infty}((\sigma_{i+1})_{i \in \overline{n-1}}, \kappa)|\!\}$ has type $\blacktriangleright (\mathsf{T}\ (\mathsf{Value}\ (\mathsf{ByNeed}\ \mathsf{T}), \mathsf{Heap}\ (\mathsf{ByNeed}\ \mathsf{T})))$ (the $\blacktriangleright$ in the type of $(\sigma_{i+1})_{i \in \overline{n-1}}$ maps through $\alpha_{\mathbb{S}^\infty}$ via the idiom brackets). Definitional equality $=$ on $\mathsf{T}\ (\mathsf{Value}\ (\mathsf{ByNeed}\ \mathsf{T}), \mathsf{Heap}\ (\mathsf{ByNeed}\ \mathsf{T}))$ is defined in the obvious structural way by guarded recursion (as it would be if it was a finite, inductive type).

The event abstraction function $\alpha_{\mathbb{E}_\mathbb{v}}(\sigma)$ encodes how intensional information from small-step transitions is retained as Events. Its semantics is entirely inconsequential for the adequacy result and we imagine that this function is tweaked on an as-needed basis depending on the particular trace property one is interested in observing. In our example, we focus on $\mathsf{Lookup}\ y$ events that carry with them the $y :: \mathsf{Name}$ of the let binding that allocated the heap entry. This event corresponds precisely to a $\textsc{Look}(\mathsf{y})$ transition, so $\alpha_{\mathbb{E}_\mathbb{v}}(\sigma)$ maps σ to $\mathsf{Lookup}\ y$ when σ is about to make a $\textsc{Look}(\mathsf{y})$ transition. In that case, the focus expression must be x and y is the first component of the heap entry $\mu(\rho(\mathsf{x}))$. The other cases are similar.

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

$$
\begin{array}{l}
\mathit{len} :: \mathsf{T}\ a \to \mathbb{N}_\omega \\
\mathit{len}\ (\mathsf{Ret}\ \_) = \mathsf{Z} \\
\mathit{len}\ (\mathsf{Step}\ \_\ \tau^{\blacktriangleright}) = \mathsf{S}\ \{\!\!|\mathit{len}\ \tau^{\blacktriangleright}|\!\!\}
\end{array}
$$

Lemma 25 (Preservation of length)**.** *Let $(\sigma_i)_{i\in\overline{n}}$ be a trace. Then $\mathit{len}\ (\alpha_{\mathbb{S}^\infty}((\sigma_i)_{i\in\overline{n}}, \mathit{cont}(\sigma_0))) = n$.*

Proof. This is quite simple to see and hence a good opportunity to familiarise ourselves with the concept of *Löb induction*, the induction principle of guarded recursion. Löb induction arises simply from applying the guarded recursive fixpoint combinator to a proposition:

$$\mathsf{l\ddot{o}b} = \mathsf{fix} : \forall P.\ (\blacktriangleright P \Longrightarrow P) \Longrightarrow P$$

That is, we assume that our proposition holds *later*, e.g.

$$IH \in (\blacktriangleright P \triangleq \blacktriangleright(\forall n \in \mathbb{N}_\omega.\ \forall (\sigma_i)_{i\in\overline{n}}.\ \mathit{len}\ (\alpha_{\mathbb{S}^\infty}((\sigma_i)_{i\in\overline{n}}, \mathit{cont}(\sigma_0))) = n))$$

and use IH to prove P. Let us assume n and $(\sigma_i)_{i\in\overline{n}}$ are given, define $\tau \triangleq \alpha_{\mathbb{S}^\infty}((\sigma_i)_{i\in\overline{n}}, \mathit{cont}(\sigma_0))$ and proceed by case analysis over n:

- **Case** Z: Then we have either $\tau = \mathsf{Ret}\ (\alpha_{\mathbb{S}}(\sigma_0))$ or $\tau = \mathsf{Ret}\ \mathsf{Stuck}$, both of which map to Z under len.
- **Case** $\mathsf{S}\ \{\!\!|m|\!\!\}$: Then $\tau = \mathsf{Step}\ \_\ \{\!\!|\alpha^\infty_{\mathbb{S}}((\sigma_{i+1})_{i\in\overline{m}}, \mathit{cont}(\sigma_0))|\!\!\}$, where $(\sigma_{i+1})_{i\in\overline{m}} \in \blacktriangleright\mathbb{S}^\infty$ is the guarded tail of the LK trace $(\sigma_i)_{i\in\overline{n}}$. Now we apply the inductive hypothesis, as follows:

$$(IH \circledast m \circledast (\sigma_{i+1})_{i\in\overline{m}}) \in \blacktriangleright(\mathit{len}\ (\alpha_{\mathbb{S}^\infty}((\sigma_{i+1})_{i\in\overline{m}}, \mathit{cont}(\sigma_0))) = m).$$

  We use this fact and congruence to prove

$$n = \mathsf{S}\ \{\!\!|m|\!\!\} = \mathsf{S}\ (\mathit{len}\ (\alpha_{\mathbb{S}^\infty}((\sigma_{i+1})_{i\in\overline{m}}, \mathit{cont}(\sigma_0)))) = \mathit{len}\ (\alpha_{\mathbb{S}^\infty}((\sigma_i)_{i\in\overline{n}}, \mathit{cont}(\sigma_0))).$$

□

Lemma 26 (Abstraction preserves termination observable)**.** *Let $(\sigma_i)_{i\in\overline{n}}$ be a maximal trace. Then $\alpha_{\mathbb{S}^\infty}((\sigma_i)_{i\in\overline{n}}, \mathit{cont}(\sigma_0))$ is ...*

- *... ending with $\mathsf{Ret}\ (\mathsf{Fun}\ \_)$ or $\mathsf{Ret}\ (\mathsf{Con}\ \_\ \_)$ if and only if $(\sigma_i)_{i\in\overline{n}}$ is balanced.*
- *... infinite if and only if $(\sigma_i)_{i\in\overline{n}}$ is diverging.*
- *... ending with $\mathsf{Ret}\ \mathsf{Stuck}$ if and only if $(\sigma_i)_{i\in\overline{n}}$ is stuck.*

Proof. The second point follows by a similar inductive argument as in Lemma 25.

In the other cases, we may assume that n is finite. If $(\sigma_i)_{i\in\overline{n}}$ is balanced, then σ_n is a return state with continuation $\mathit{cont}(\sigma_0)$, so its control expression is a value. Then $\alpha_{\mathbb{S}^\infty}$ will conclude with $\mathsf{Ret}\ (\alpha_{\mathbb{S}}(\_))$, and the latter is never $\mathsf{Ret}\ \mathsf{Stuck}$. Conversely, if the trace ended with $\mathsf{Ret}\ (\mathsf{Fun}\ \_)$ or $\mathsf{Ret}\ (\mathsf{Con}\ \_\ \_)$, then $\mathit{cont}(\sigma_n) = \mathit{cont}(\sigma_0)$ and $\mathit{ctrl}(\sigma_n)$ is a value, so $(\sigma_i)_{i\in\overline{n}}$ forms a balanced trace. The stuck case is similar. □

The previous lemma is interesting as it allows us to apply the classifying terminology of interior traces to a $\tau :: \mathsf{T}\ a$ 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 $\mathsf{Ret}\ v$ for a $v \not\equiv \mathsf{Stuck}$, stuck if ending in $\mathsf{Ret}\ \mathsf{Stuck}$ and diverging if infinite.

We are now ready to prove the main soundness predicate, proving that $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_$ is an exact abstract interpretation of the LK machine:

Theorem 27 ($\mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_$ abstracts LK machine)**.** *C from Figure 15 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 $\alpha_{\mathbb{S}^\infty}((\sigma_i)_{i\in\overline{n}}, \kappa) = \mathcal{S}_{\mathbf{need}}[\![e]\!]_{\alpha_{\mathbb{E}}(\mu,\rho)}(\alpha_{\mathbb{H}}(\mu))$.*

Proof. By Löb induction, with $IH \in \blacktriangleright C$ as the hypothesis.

We will say that an LK state σ is stuck if there is no applicable rule in the transition system (i.e., the singleton LK trace σ is maximal and stuck).

Now let $(\sigma_i)_{i\in\overline{n}}$ be a maximal LK trace with source state $\sigma_0 = (\mathrm{e}, \rho, \mu, \kappa)$ and let $\tau = \mathcal{S}_{\mathbf{need}}[\![e]\!]_{\alpha_{\mathbb{E}}(\mu,\rho)}(\alpha_{\mathbb{H}}(\mu))$. Then the goal is to show $\alpha_{\mathbb{S}^\infty}((\sigma_i)_{i\in\overline{n}}, \kappa) = \tau$. We do so by cases over e, abbreviating $\tilde{\mu} \triangleq \alpha_{\mathbb{H}}(\mu)$ and $\tilde{\rho} \triangleq \alpha_{\mathbb{E}}(\mu, \rho)$:

- **Case** x: Let us assume first that σ_0 is stuck. Then $\mathrm{x} \notin \mathrm{dom}(\rho)$ (because Look is the only transition that could apply), so $\tau = \mathit{Ret}\ \mathit{Stuck}$ and the goal follows from Lemma 26.
  Otherwise, $\sigma_1 \triangleq (\mathrm{e}', \rho_1, \mu, \mathbf{upd}(\mathrm{a}) \cdot \kappa)$, $\sigma_0 \hookrightarrow \sigma_1$ via Look(y), and $\rho(\mathrm{x}) = \mathrm{a}$, $\mu(\mathrm{a}) = (\mathrm{y}, \rho_1, \mathrm{e}')$. This matches the head of the action of $\tilde{\rho}\ x$, which is of the form $\mathit{step}\ (\mathit{Lookup}\ y)\ (\mathit{fetch}\ a)$. To show that the tails equate, it suffices to show that they equate *later*.
  We can infer that $\tilde{\mu}\ a = \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e']\!]_{\tilde{\rho}})$ from the definition of $\alpha_{\mathbb{H}}$, so

  $$\begin{aligned}&\mathit{fetch}\ a\ \tilde{\mu} = \tilde{\mu}\ a\ \tilde{\mu} = \mathcal{S}_{\mathbf{need}}[\![e']\!]_{\tilde{\rho}}(\tilde{\mu}) \ggg \lambda\mathbf{case}\\ &\quad (\mathit{Stuck}, \tilde{\mu}) \to \mathit{Ret}\ (\mathit{Stuck}, \tilde{\mu})\\ &\quad (\mathit{val}, \quad \tilde{\mu}) \to \mathit{Step}\ \mathit{Update}\ (\mathit{Ret}\ (\mathit{val}, \tilde{\mu}[a \mapsto \mathit{memo}\ a\ (\mathit{return}\ \mathit{val})]))\end{aligned}$$

  Let us define $\tau^{\blacktriangleright} \triangleq \{\!\!\{\mathcal{S}_{\mathbf{need}}[\![e']\!]_{\tilde{\rho}}(\tilde{\mu})\}\!\!\}$ and apply the induction hypothesis IH to the maximal trace starting at σ_1. This yields an equality

  $$IH \circledast (\sigma_{i+1})_{i\in\overline{m}} \in \{\!\!\{\alpha_{\mathbb{S}^\infty}((\sigma_{i+1})_{i\in\overline{m}}, \mathbf{upd}(\mathrm{a}) \cdot \kappa) = \tau^{\blacktriangleright}\}\!\!\}$$

  When $\tau^{\blacktriangleright}$ is infinite, we are done. Similarly, if $\tau^{\blacktriangleright}$ ends in $\mathit{Ret}\ \mathit{Stuck}$ then the continuation of $\ggg$ will return $\mathit{Ret}\ \mathit{Stuck}$, indicating by Lemma 25 and Lemma 26 that $(\sigma_{i+1})_{i\in\overline{n-1}}$ is stuck and hence $(\sigma_i)_{i\in\overline{n}}$ is, too.
  Otherwise $\tau^{\blacktriangleright}$ ends after $m-1$ Steps with $\mathit{Ret}\ (\mathit{val}, \tilde{\mu}_m)$ and by Lemma 26 $(\sigma_{i+1})_{i\in\overline{m}}$ is balanced; hence $\mathit{cont}(\sigma_m) = \mathbf{upd}(\mathrm{a})\cdot\kappa$ and $\mathit{ctrl}(\sigma_m)$ is a value. So $\sigma_m = (\mathrm{v}, \rho_m, \mu_m, \mathbf{upd}(\mathrm{a})\cdot\kappa)$ and the Upd transition fires, reaching $(\mathrm{v}, \rho_m, \mu_m[\mathrm{a} \mapsto (\mathrm{y}, \rho_m, \mathrm{v})], \kappa)$ and this must be the target state σ_n (so $m = n-2$), because it remains a return state and has continuation κ, so $(\sigma_i)_{i\in\overline{n}}$ is balanced. Likewise, the continuation argument of $\ggg$ does a $\mathit{Step}\ \mathit{Update}$ on $\mathit{Ret}\ (\mathit{val}, \tilde{\mu}_m)$, updating the heap. By cases on v and the $\mathit{Domain}\ (\mathit{D}\ (\mathit{ByNeed}\ \mathit{T}))$ instance we can see that

  $$\begin{aligned}&\mathit{Ret}\ (\mathit{val}, \tilde{\mu}_m[a \mapsto \mathit{memo}\ a\ (\mathit{return}\ \mathit{val})])\\ =\ &\mathit{Ret}\ (\mathit{val}, \tilde{\mu}_m[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\tilde{\rho}_m})])\\ =\ &\alpha_{\mathbb{S}}(\sigma_n)\end{aligned}$$

  and this equality concludes the proof.
- **Case** e x: The cases where τ gets stuck or diverges before finishing evaluation of e are similar to the variable case. So let us focus on the situation when $\tau^{\blacktriangleright} \triangleq \{\!\!\{\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\tilde{\rho}}(\tilde{\mu})\}\!\!\}$ returns and let σ_m be LK state at the end of the balanced trace $(\sigma_{i+1})_{i\in\overline{m-1}}$ through e starting in stack $\mathbf{ap}(\mathrm{a}) \cdot \kappa$.
  Now, either there exists a transition $\sigma_m \hookrightarrow \sigma_{m+1}$, or it does not. When the transition exists, it must must leave the stack $\mathbf{ap}(\mathrm{a}) \cdot \kappa$ due to maximality, necessarily by an App_2 transition. That in turn means that the value in $\mathit{ctrl}(\sigma_m)$ must be a lambda $\bar{\lambda}\mathrm{y}.\mathrm{e}'$, and $\sigma_{m+1} = (\mathrm{e}', \rho_m[\mathrm{y} \mapsto \rho(\mathrm{x})], \mu_m, \kappa)$.
  Likewise, $\tau^{\blacktriangleright}$ ends in $\alpha_{\mathbb{S}}(\sigma_m) = \mathit{Ret}\ (\mathit{Fun}\ (\lambda d \to \mathit{step}\ \mathit{App}_2\ (\mathcal{S}_{\mathbf{need}}[\![e']\!]_{\tilde{\rho}_m[y\mapsto d]})), \tilde{\mu}_m)$ (where $\tilde{\mu}_m$ corresponds to the heap in σ_m in the usual way). The *fun* implementation of $\mathit{Domain}\ (\mathit{D}\ (\mathit{ByNeed}\ \mathit{T}))$ applies the Fun value to the argument denotation $\tilde{\rho}\ x$, hence it

remains to show that $\tau_2^{\blacktriangleright} \triangleq \mathcal{S}_{\mathbf{need}}[\![e']\!]_{\tilde{\rho}_m[y \mapsto \tilde{\rho}\ x]}(\tilde{\mu}_m)$ is equal to $\alpha_{\mathbb{S}^\infty}((\sigma_{i+m+1})_{i\in\overline{k}}, \kappa)$ *later*, where $(\sigma_{i+m+1})_{i\in\overline{k}}$ is the maximal trace starting at σ_{m+1}.
We can apply the induction hypothesis to this situation. From this and our earlier equalities, we get $\alpha_{\mathbb{S}^\infty}((\sigma_i)_{i\in\overline{n}}, \kappa) = \tau$, concluding the proof of the case where there exists a transition $\sigma_m \hookrightarrow \sigma_{m+1}$.
When $\sigma_m \not\hookrightarrow$, then $\mathit{ctrl}(\sigma_m)$ is not a lambda, otherwise App$_2$ would apply. In this case, *fun* gets to see a Stuck or Con _ _ value, for which it is Stuck as well.

- **Case case** e_s **of** $\overline{K\ \overline{\mathrm{x}} \rightarrow \mathrm{e}_r}$: Similar to the application and lookup case.
- **Cases** $\bar{\lambda}\mathrm{x.e}$, $K\ \overline{\mathrm{x}}$: The length of both traces is $n = 0$ and the goal follows by simple calculation.
- **Case let** $\mathrm{x} = \mathrm{e}_1$ **in** e_2: Let $\sigma_0 = (\mathbf{let}\ \mathrm{x} = \mathrm{e}_1\ \mathbf{in}\ \mathrm{e}_2, \rho, \mu, \kappa)$. Then $\sigma_1 = (\mathrm{e}_2, \rho_1, \mu', \kappa)$ by Let$_1$, where $\rho_1 = \rho[\mathrm{x} \mapsto \mathrm{a}_{\mathrm{x},i}]$, $\mu' = \mu[\mathrm{a}_{\mathrm{x},i} \mapsto (\mathrm{x}, \rho_1, \mathrm{e}_1)]$. Since the stack does not grow, maximality from the tail $(\sigma_{i+1})_{i\in\overline{n-1}}$ transfers to $(\sigma_i)_{i\in\overline{n}}$. Straightforward application of the induction hypothesis to $(\sigma_{i+1})_{i\in\overline{n-1}}$ yields the equality for the tail (after a bit of calculation for the updated environment and heap), which concludes the proof.

□

Theorem 27 and Lemma 26 are the key to proving the following theorem of adequacy, which formalises the intuitive notion of adequacy from before.
(A state σ is *final* when $\mathit{ctrl}(\sigma)$ is a value and $\mathit{cont}(\sigma) = \mathbf{stop}$.)

Theorem 28 (Adequacy of $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$). *Let* $\tau \triangleq \mathcal{S}_{\mathbf{need}}[\![e]\!]_\varepsilon(\varepsilon)$.

- τ *ends with* Ret (Fun _, _) *or* Ret (Con _ _, _) *(is balanced) iff there exists a final state* σ *such that* $\mathit{init}(\mathrm{e}) \hookrightarrow^* \sigma$.
- τ *ends with* Ret (Stuck, _) *(is stuck) iff there exists a non-final state* σ *such that* $\mathit{init}(\mathrm{e}) \hookrightarrow^* \sigma$ *and there exists no* σ' *such that* $\sigma \hookrightarrow \sigma'$.
- τ *is infinite* Step _ (Step _ ...) *(is diverging) iff for all* σ *with* $\mathit{init}(\mathrm{e}) \hookrightarrow^* \sigma$ *there exists* σ' *with* $\sigma \hookrightarrow \sigma'$.
- *The* e::Event *in every* Step e ... *occurrence in* τ *corresponds in the intuitive way to the matching small-step transition rule that was taken.*

Proof. There exists a maximal trace $(\sigma_i)_{i\in\overline{n}}$ starting from $\sigma_0 = \mathit{init}(\mathrm{e})$, and by Theorem 27 we have $\alpha_{\mathbb{S}^\infty}((\sigma_i)_{i\in\overline{n}}, \mathbf{stop}) = \tau$. The correctness of Events emitted follows directly from $\alpha_{\mathbb{E}\mathrm{v}}$.

$\Rightarrow$
- If $(\sigma_i)_{i\in\overline{n}}$ is balanced, its target state σ_n is a return state that must also have the empty continuation, hence it is a final state.
- If $(\sigma_i)_{i\in\overline{n}}$ is stuck, it is finite and maximal, but not balanced, so its target state σ_n cannot be a return state; otherwise maximality implies σ_n has an (initial) empty continuation and the trace would be balanced. On the other hand, the only returning transitions apply to return states, so maximality implies there is no σ' such that $\sigma \hookrightarrow \sigma'$ whatsoever.
- If $(\sigma_i)_{i\in\overline{n}}$ is diverging, $n = \omega$ and for every σ with $\mathit{init}(\mathrm{e}) \hookrightarrow^* \sigma$ there exists an i such that $\sigma = \sigma_i$ by determinism.

$\Leftarrow$
- If σ_n is a final state, it has $\mathit{cont}(\sigma) = \mathit{cont}(\mathit{init}(\mathrm{e})) = []$, so the trace is balanced.
- If σ is not a final state, τ' is not balanced. Since there is no σ' such that $\sigma \hookrightarrow^* \sigma'$, it is still maximal; hence it must be stuck.
- Suppose that $n \in \mathbb{N}_\omega$ was finite. Then, if for every choice of σ there exists σ' such that $\sigma \hookrightarrow \sigma'$, then there must be σ_{n+1} with $\sigma_n \hookrightarrow \sigma_{n+1}$, violating maximality of the trace. Hence it must be infinite. It is also interior, because every stack extends the empty stack, hence it is diverging.

□

B.1 Total Encoding in Guarded Cubical Agda

Whereas traditional theories of coinduction require syntactic productivity checks [Coquand 1994], imposing tiresome constraints on the form of guarded recursive functions, the appeal of guarded type theories is that productivity is instead proven semantically, in the type system. Compared to the alternative of *sized types* [Hughes et al. 1996], guarded types don't require complicated algebraic manipulations of size parameters; however perhaps sized types would work just as well. Any fuel-based (or step-indexed) approach is equivalent to our use of guarded type theory, but we find that the latter is a more direct (and thus preferable) encoding.

The fundamental innovation of guarded recursive type theory is the integration of the "later" modality $\blacktriangleright$ which allows to define coinductive data types with negative recursive occurrences such as in the data constructor $\mathsf{Fun} :: (\mathsf{D}\ \tau \to \mathsf{D}\ \tau) \to \mathsf{Value}\ \tau$ (recall that $\mathsf{D}\ \tau = \tau\ (\mathsf{Value}\ \tau)$), as first realised by Nakano [2000]. The way that is achieved is roughly as follows: The type $\blacktriangleright T$ represents data of type T that will become available after a finite amount of computation, such as unrolling one layer of a fixpoint definition. It comes with a general fixpoint combinator $\mathsf{fix} : \forall A.\ (\blacktriangleright A \to A) \to A$ 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 infinite streams:

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

where $ones : Str$ is the constant stream of 1. In particular, Str is the fixpoint of a locally contractive functor $F(X) = \mathbb{N} \times \blacktriangleright X$. 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 $\blacktriangleright$. The most exciting consequence is that changing the Fun data constructor to $\mathsf{Fun} :: (\blacktriangleright (\mathsf{D}\ \tau) \to \mathsf{D}\ \tau) \to \mathsf{Value}\ \tau$ makes $\mathsf{Value}\ \tau$ a well-defined coinductive data type,[30] whereas syntactic approaches to coinduction reject any negative recursive occurrence.

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

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

allowing us to apply a familiar framework of reasoning around $\blacktriangleright$. In order not to obscure our work with pointless symbol pushing, we will often omit the idiom brackets [McBride and Paterson 2008] $\{\!|\_|\!\}$ to indicate where the $\blacktriangleright$ "effects" happen.

We will now outline the changes necessary to encode $\mathcal{S}[\![\_]\!]_{\_}$ in Guarded Cubical Agda, a system implementing Ticked Cubical Type Theory [Møgelberg and Veltri 2019], as well as the concrete instances D (ByName T) and D (ByNeed T) from Figures 5b and 7. The full, type-checked development is available in the Supplement.

- We need to delay in *step*; thus its definition in Trace changes to $\mathit{step} :: \mathsf{Event} \to \blacktriangleright d \to d$.
- All Ds that will be passed to lambdas, put into the environment or stored in fields need to have the form $\mathit{step}\ (\mathsf{Lookup}\ x)\ d$ for some $x :: \mathsf{Name}$ and a delayed $d :: \blacktriangleright (\mathsf{D}\ \tau)$. This is enforced as follows:
  (1) The Domain type class gains an additional predicate parameter $p :: \mathsf{D} \to \mathsf{Set}$ that will be instantiated by the semantics to a predicate that checks that the D has the required form $\mathit{step}\ (\mathsf{Lookup}\ x)\ d$ for some $x :: \mathsf{Name}$, $d :: \blacktriangleright (\mathsf{D}\ \tau)$.
  (2) Then the method types of Domain use a Sigma type to encode conformance to p. For example, the type of Fun changes to $(\Sigma\ \mathsf{D}\ p \to \mathsf{D}) \to \mathsf{D}$.

[30]The reason why the positive occurrence of $\mathsf{D}\ \tau$ does not need to be guarded is that the type of Fun can more formally be encoded by a mixed inductive-coinductive type, e.g., $\mathsf{Value}\ \tau = \mathsf{fix}\, X.\ \mathsf{lfp}\, Y.\ \ldots \mid \mathsf{Fun}\ (X \to Y) \mid \ldots$

```
data Type = Type :→: Type | TyConApp TyCon [Type] | TyVar Name | Wrong
data PolyType = PT [Name] Type; data TyCon = ...
type Constraint = (Type, Type); type Subst = Name :⇀ Type
data Cts a = Cts (StateT (Set Name, Subst) Maybe a)
emitCt :: Constraint → Cts ();              freshTyVar :: Cts Type
instantiatePolyTy :: PolyType → Cts Type;   generaliseTy :: Cts Type → Cts PolyType
closedType :: Cts PolyType → PolyType
instance Trace (Cts v) where step _ = id
instance Domain (Cts PolyType) where stuck = return (PT [ ] Wrong); ...
instance HasBind (Cts PolyType) where
  bind rhs body = generaliseTy (do
    rhs_ty ← freshTyVar
    rhs_ty' ← rhs (return (PT [ ] rhs_ty)) ≫= instantiatePolyTy
    emitCt (rhs_ty, rhs_ty')
    return rhs_ty) ≫= body ∘ return
```

Fig. 16. Hindley-Milner-style type analysis with Let generalisation

(3) The reason why we need to encode this fact is that the guarded recursive data type Value has a constructor the type of which amounts to $\mathsf{Fun} :: (\mathsf{Name} \times \blacktriangleright (\mathsf{D}\ \tau) \to \mathsf{D}\ \tau) \to \mathsf{Value}\ \tau$, breaking the previously discussed negative recursive cycle by a $\blacktriangleright$, and expecting $x :: \mathsf{Name}$, $d :: \blacktriangleright (\mathsf{D}\ \tau)$ such that the original $\mathsf{D}\ \tau$ can be recovered as $\mathit{step}\ (\mathsf{Lookup}\ x)\ d$. This is in contrast to the original definition $\mathsf{Fun} :: (\mathsf{D}\ \tau \to \mathsf{D}\ \tau) \to \mathsf{Value}\ \tau$ which would *not* type-check. One can understand Fun as carrying the "closure" resulting from *defunctionalising* [Reynolds 1972] a $\Sigma\ \mathsf{D}\ p$, and that this defunctionalisation is presently necessary in Agda to eliminate negative cycles.

- Expectedly, HasBind becomes more complicated because it encodes the fixpoint combinator. We settled on $\mathit{bind} :: \blacktriangleright (\blacktriangleright \mathsf{D} \to \mathsf{D}) \to (\blacktriangleright \mathsf{D} \to \mathsf{D}) \to \mathsf{D}$. We tried rolling up $\mathit{step}\ (\mathsf{Lookup}\ x)\ \_$ in the definition of $\mathcal{S}[\![\_]\!]_{\_}$ to get a simpler type $\mathit{bind} :: (\Sigma\ \mathsf{D}\ p \to \mathsf{D}) \to (\Sigma\ \mathsf{D}\ p \to \mathsf{D}) \to \mathsf{D}$, but then had trouble defining ByNeed heaps independently of the concrete predicate p.
- Higher-order mutable state is among the classic motivating examples for guarded recursive types. As such it is no surprise that the state-passing of the mutable Heap in the implementation of ByNeed requires breaking of a recursive cycle by delaying heap entries, $\mathsf{Heap}\ \tau = \mathsf{Addr} :\rightharpoonup \blacktriangleright (\mathsf{D}\ \tau)$.
- We need to pass around Tick binders in $\mathcal{S}[\![\_]\!]_{\_}$ in a way that the type checker is satisfied; a simple exercise. We find it remarkable how non-invasive these adjustment are!

Thus we have proven that $\mathcal{S}[\![\_]\!]_{\_}$ is a total, mathematical function, and fast and loose equational reasoning about $\mathcal{S}[\![\_]\!]_{\_}$ is not only *morally* correct [Danielsson et al. 2006], but simply *correct*. Furthermore, since evaluation order doesn't matter in Agda and hence for $\mathcal{S}[\![\_]\!]_{\_}$, we could have defined it in a strict language (lowering $\blacktriangleright a$ as $() \to a$) just as well.

C PROOFS FOR SECTION 6 (STATIC ANALYSIS)

C.1 Type Analysis

To demonstrate the flexibility of our approach, we have implemented Hindley-Milner-style type analysis including Let generalisation as an instance of our abstract denotational interpreter. The

Table 1. Examples for type analysis.

#	e	*closedType* $(\mathcal{S}[\![e]\!]_\varepsilon)$
(1)	$\textbf{let } i = \bar{\lambda}x.x \textbf{ in } i\ i\ i\ i\ i\ i$	$\forall\alpha_{11}.\ \alpha_{11} \rightarrow \alpha_{11}$
(2)	$\bar{\lambda}x.\textbf{let } y = x \textbf{ in } y\ x$	**wrong**
(3)	$\textbf{let } i = \bar{\lambda}x.x \textbf{ in let } o = \mathit{Some}(i) \textbf{ in } o$	$\forall\alpha_6.\ \texttt{option}\ (\alpha_6 \rightarrow \alpha_6)$
(4)	$\textbf{let } x = x \textbf{ in } x$	$\forall\alpha_1.\ \alpha_1$

gist is given in Figure 16; we omit large parts of the implementation and the Domain instance for space reasons. While the full implementation can be found in the extract generated from this document, the HasBind instance is a sufficient exemplar of the approach.

The analysis infers most general PolyTypes of the form $\forall\overline{\alpha}.\ \theta$ for an expression, where θ ranges over a Type that can be either a type variable TyVar α, a function type $\theta_1{:}{\rightarrow}{:}\theta_2$, or a type constructor application TyConApp. The Wrong type is used to indicate a type error.

Key to the analysis is maintenance of a consistent set of type constraints as a unifying Substitution. That is why the trace type Cts carries the current unifier as state, with the option of failure indicated by Maybe when the unifier does not exist. Additionally, Cts carries a set of used Names with it to satisfy freshness constraints in *freshTyVar* and *instantiatePolyTy*, as well as to construct a superset of $\mathsf{fv}(\rho)$ in *generaliseTy*.

While the operational detail offered by Trace is ignored by Cts, all the pieces fall together in the implementation of *bind*, where we see yet another domain-specific fixpoint strategy: The knot is tied by calling the iteratee *rhs* with a fresh unification variable type *rhs_ty* of the shape α_1. The result of this call in turn is instantiated to a non-PolyType *rhs_ty′*, perhaps turning a type-scheme $\forall\alpha_2.\ \texttt{option}\ (\alpha_2 \rightarrow \alpha_2)$ into the shape $\texttt{option}\ (\alpha_3 \rightarrow \alpha_3)$ for fresh α_3. Then a constraint is emitted to unify α_1 with $\texttt{option}\ (\alpha_3 \rightarrow \alpha_3)$. Ultimately, the type *rhs_ty* is returned and generalised to $\forall\alpha_3.\ \texttt{option}\ (\alpha_3 \rightarrow \alpha_3)$, because α_3 is not a Name in use before the call to *generaliseTy*, and thus it couldn't have possibly leaked into the range of the ambient type context. The generalised PolyType is then used when analysing the *body*.

Examples. Let us again conclude with some examples in Table 1. Example (1) demonstrates repeated instantiation and generalisation. Example (2) shows that let generalisation does not accidentally generalise the type of y. Example (3) shows an example involving data types and the characteristic approximation to higher-rank types, and example (4) shows that type inference for diverging programs works as expected.

C.2 Control-flow Analysis

In our last example, we will discuss a classic benchmark of abstract higher-order interpreters: Control-flow analysis (CFA). CFA calculates an approximation of which values an expression might evaluate to, so as to narrow down the possible control-flow edges at application sites. The resulting control-flow graph conservatively approximates the control-flow of the whole program and can be used to apply classic intraprocedural analyses such as interval analysis in a higher-order setting.

To facilitate CFA, we have to revise the Domain class to pass down a *label* from allocation sites, which is to serve as the syntactic proxy of the value's control-flow node:

```
type Label = String
class Domain d where
```

```
data Pow a = P (Set a); type Value_C = Pow Label
type ConCache = (Tag, [Value_C]); data FunCache = FC (Maybe (Value_C, Value_C)) (D_C → D_C)
data Cache = Cache (Label :⇀ ConCache) (Label :⇀ FunCache)
data T_C a = T_C (State Cache a); type D_C = T_C Value_C; runCFA :: D_C → Value_C
updFunCache :: Label → (D_C → D_C) → T_C (); cachedCall :: Label → Value_C → D_C
instance HasBind D_C where ...; instance Trace (T_C v) where step _ = id
instance Domain D_C where
  fun _ ℓ f = do updFunCache ℓ f; return (P (Set.singleton ℓ))
  apply dv da = dv >>= λ(P ℓ̄) → da >>= λa → lub <$> traverse (λℓ → cachedCall ℓ a) (Set.toList ℓ̄)
  ...
```

Fig. 17. 0CFA

Table 2. Examples for control-flow analysis.

#	e	$\mathit{runCFA}\ (\mathcal{S}[\![e]\!]_\varepsilon)$
(1)	$\textbf{let}\ i = \bar{\lambda}x.x\ \textbf{in}\ \textbf{let}\ j = \bar{\lambda}y.y\ \textbf{in}\ i\ j$	$\{\lambda y..\}$
(2)	$\textbf{let}\ i = \bar{\lambda}x.x\ \textbf{in}\ \textbf{let}\ j = \bar{\lambda}y.y\ \textbf{in}\ i\ j\ j$	$\{\lambda x.., \lambda y..\}$
(3)	$\textbf{let}\ \omega = \bar{\lambda}x.x\ x\ \textbf{in}\ \omega\ \omega$	$\{\}$
(4)	$\textbf{let}\ x = \textbf{let}\ y = S(x)\ \textbf{in}\ S(y)\ \textbf{in}\ x$	$\{S(y)\}$

```
con :: Label → Tag → [d] → d
fun :: Name → Label → (d → d) → d
```

We omit how to forward labels appropriately in $\mathcal{S}[\![\_]\!]_\_$ and how to adjust Domain instances.

Figure 17 gives a rough outline of how we use this extension to define a 0CFA:[31] An abstract $\mathsf{Value}_\mathsf{C}$ is the usual set of Labels standing in for a syntactic value. The trace abstraction T_C maintains as state a Cache that approximates the shape of values at a particular Label, an abstraction of the heap. For constructor values, the shape is simply a pair of the Tag and $\mathsf{Value}_\mathsf{C}$s for the fields. For a lambda value, the shape is its abstract control-flow transformer, of type $\mathsf{D}_\mathsf{C} \to \mathsf{D}_\mathsf{C}$ (populated by *updFunCache*), plus a single point (v_1, v_2) of its graph (k-CFA would have one point per contour), serving as the transformer's summary.

At call sites in *apply*, we will iterate over each function label and attempt a *cachedCall*. In doing so, we look up the label's transformer and sees if the single point is applicable for the incoming value v, e.g., if $v \sqsubseteq v_1$, and if so return the cached result v_2 straight away. Otherwise, the transformer stored for the label is evaluated at v and the result is cached as the new summary. An allocation site might be re-analysed multiple times with monotonically increasing environment due to fixpoint iteration in *bind*. Whenever that happens, the point that has been cached for that allocation site is cleared, because the function might have increased its result. Then re-evaluating the function at the next *cachedCall* is mandatory.

Note that a D_C transitively (through Cache) recurses into $\mathsf{D}_\mathsf{C} \to \mathsf{D}_\mathsf{C}$, thus introducing vicious cycles in negative position, rendering the encoding non-inductive. This highlights a common challenge with instances of CFA: The obligation to prove that the analysis actually terminates on all inputs; an obligation that we will gloss over in this work.

[31] As before, the extract of this document contains the full, executable definition.

Examples. The first two examples of Table 2 demonstrate a precise and an imprecise result, respectively. The latter is due to the fact that both i and j flow into x. Examples (3) and (4) show that the HasBind instance guarantees termination for diverging programs and cyclic data.

D PROOFS FOR SECTION 7 (GENERIC BY-NAME AND BY-NEED SOUNDNESS)

Theorem 6 (Sound By-need Interpretation). *Let $\widehat{\mathsf{D}}$ be a domain with instances for* Trace, Domain, HasBind *and* Lat, *and let abstract be the abstraction function described above. If the abstraction laws in Figure 13 hold, then $\mathcal{S}_{\widehat{\mathsf{D}}}[\![\_]\!]_{\_}$ is an abstract interpreter that is sound wrt. abstract, that is,*

$$abstract\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_\varepsilon) \sqsubseteq \mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_\varepsilon.$$

Proof. The definition of *abstract* is in terms of the Galois connection *nameNeed* from Figure 18. Let α be the abstraction function from *nameNeed*; then we define

$$abstract\ d = \alpha\ \{d\ \varepsilon\}$$

I.e., we simply run d in the initial empty heap. Do note that *abstract* does not work for open expressions because of this.

When we inline *abstract*, the goal is simply Theorem 56 for the special case where environment and heap are empty. □

Abbreviation 29 (Field access). $\langle \varphi', \nu' \rangle \circ \varphi \triangleq \varphi', \langle \varphi', \nu' \rangle \circ \nu = \nu'$.

For concise notation, we define the following abstract substitution operation:

Definition 30 (Abstract substitution). *We call $\varphi[x \Mapsto \varphi'] \triangleq \varphi[x \mapsto \mathsf{U}_0] + (\varphi\,!\,x) * \varphi'$ the* abstract substitution *operation on* Uses *and overload this notation for* T_{U}, *so that $\langle \varphi, \nu \rangle[x \Mapsto \varphi'] \triangleq \langle \varphi[x \Mapsto \varphi'], \nu \rangle$.*

Lemma 31. $\mathcal{S}[\![\mathsf{Lam}\ x\ e\ \text{`App`}\ y]\!]_\rho = (\mathcal{S}[\![e]\!]_{\rho[x \mapsto \langle [x \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle]})[x \Mapsto (\rho\,!\,y).\varphi]$.

The proof below needs to appeal to a couple of congruence lemmas about abstract substitution, the proofs of which would be tedious and hard to follow, hence they are omitted. These are very similar to lemmas we have proven for absence analysis (cf. Lemma 15).

Lemma 32. $\mathcal{S}_{\mathbf{usage}}[\![\mathsf{Lam}\ y\ (\mathsf{Lam}\ x\ e\ \text{`App`}\ z)]\!]_\rho = \mathcal{S}_{\mathbf{usage}}[\![\mathsf{Lam}\ x\ (\mathsf{Lam}\ y\ e)\ \text{`App`}\ z]\!]_\rho$.

Lemma 33. $\mathcal{S}_{\mathbf{usage}}[\![\mathsf{Lam}\ x\ e\ \text{`App`}\ y\ \text{`App`}\ z]\!]_\rho = \mathcal{S}_{\mathbf{usage}}[\![\mathsf{Lam}\ x\ (e\ \text{`App`}\ z)\ \text{`App`}\ y]\!]_\rho$.

Lemma 34.

$$\begin{array}{ll} & \mathcal{S}_{\mathbf{usage}}[\![\mathsf{Case}\ (\mathsf{Lam}\ x\ e\ \text{`App`}\ y)\ (alts\ (\mathsf{Lam}\ x\ e_r\ \text{`App`}\ y))]\!]_{\rho[x \mapsto \rho\,!\,y]} \\ = & \mathcal{S}_{\mathbf{usage}}[\![\mathsf{Lam}\ x\ (\mathsf{Case}\ e\ (alts\ e_r))\ \text{`App`}\ y]\!]_\rho. \end{array}$$

Lemma 35. $\mathcal{S}_{\mathbf{usage}}[\![\mathsf{Let}\ z\ (\mathsf{Lam}\ x\ e_1\ \text{`App`}\ y)\ (\mathsf{Lam}\ x\ e_2\ \text{`App`}\ y)]\!]_\rho = \mathcal{S}_{\mathbf{usage}}[\![\mathsf{Lam}\ x\ (\mathsf{Let}\ z\ e_1\ e_2)\ \text{`App`}\ y]\!]_\rho$.

Now we can finally prove the substitution lemma:

Lemma 7 (Substitution). $\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho[x \mapsto \rho\,!\,y]} \sqsubseteq \mathcal{S}_{\mathbf{usage}}[\![\mathsf{Lam}\ x\ e\ \text{`App`}\ y]\!]_\rho$.

Proof. We need to assume that x is absent in the range of ρ. This is a "freshness assumption" relating to the identify of x that in practice is always respected by $\mathcal{S}_{\mathbf{usage}}[\![\_]\!]_{\_}$.

Now we proceed by induction on e and only consider non-*stuck* cases.

- **Case** $\mathsf{Var}\ z$: When $x \neq z$, we have

$$\begin{array}{ll} & \mathcal{S}_{\mathbf{usage}}[\![z]\!]_{\rho[x \mapsto \rho\,!\,y]} \\ = & \langle\!\langle\ x \neq z\ \rangle\!\rangle \\ & \rho\,!\,z \end{array}$$

$$
\begin{array}{ll}
= & \langle\!\langle \text{ Refold } \mathcal{S}_{\textbf{usage}}[\![\_]\!]_{\_} \rangle\!\rangle \\
& \mathcal{S}_{\textbf{usage}}[\![z]\!]_{\rho[x \mapsto prx\ x]} \\
= & \langle\!\langle ((\rho\,!\,z).\varphi)\,!\,x = \mathsf{U}_0 \rangle\!\rangle \\
& (\mathcal{S}_{\textbf{usage}}[\![z]\!]_{\rho[x \mapsto prx\ x]})[x \Mapsto (\rho\,!\,y).\varphi] \\
= & \langle\!\langle \text{ Definition of } \mathcal{S}_{\textbf{usage}}[\![\_]\!]_{\_} \rangle\!\rangle \\
& \mathcal{S}_{\textbf{usage}}[\![\mathsf{Lam}\ x\ (\mathsf{Var}\ z)\ \text{`}\mathsf{App}\text{`}\ y]\!]_{\rho}
\end{array}
$$

Otherwise, we have $x = z$.

$$
\begin{array}{ll}
& \mathcal{S}_{\textbf{usage}}[\![z]\!]_{\rho[x \mapsto \rho\,!\,y]} \\
= & \langle\!\langle x = y, \text{ unfold } \rangle\!\rangle \\
& \rho\,!\,y \\
\sqsubseteq & \langle\!\langle v \sqsubseteq (\mathsf{Rep}\ \mathsf{U}_\omega) \rangle\!\rangle \\
& \langle (\rho\,!\,y).\varphi, \mathsf{Rep}\ \mathsf{U}_\omega \rangle \\
= & \langle\!\langle \text{ Definition of abstract substitution } \rangle\!\rangle \\
& (prx\ x)[x \Mapsto (\rho\,!\,y).\varphi] \\
= & \langle\!\langle \text{ Refold } \mathcal{S}_{\textbf{usage}}[\![\_]\!]_{\_} \rangle\!\rangle \\
& (\mathcal{S}_{\textbf{usage}}[\![z]\!]_{\rho[x \mapsto prx\ x]})[x \Mapsto (\rho\,!\,y).\varphi] \\
= & \langle\!\langle \text{ Definition of } \mathcal{S}_{\textbf{usage}}[\![\_]\!]_{\_} \rangle\!\rangle \\
& \mathcal{S}_{\textbf{usage}}[\![\mathsf{Lam}\ x\ (\mathsf{Var}\ z)\ \text{`}\mathsf{App}\text{`}\ y]\!]_{\rho}
\end{array}
$$

- **Case** $\mathsf{Lam}\ z\ e$:

$$
\begin{array}{ll}
& \mathcal{S}_{\textbf{usage}}[\![\mathsf{Lam}\ z\ e]\!]_{\rho[x \mapsto \rho\,!\,y]} \\
= & \langle\!\langle \text{ Unfold } \mathcal{S}_{\textbf{usage}}[\![\_]\!]_{\_} \rangle\!\rangle \\
& fun\ z\ (\lambda d \rightarrow step\ \mathsf{App}_2\ \$\ \mathcal{S}_{\textbf{usage}}[\![e]\!]_{\rho[x \mapsto \rho\,!\,y][z \mapsto d]}) \\
= & \langle\!\langle \text{ Rearrange, } x \neq z \rangle\!\rangle \\
& fun\ z\ (\lambda d \rightarrow step\ \mathsf{App}_2\ \$\ \mathcal{S}_{\textbf{usage}}[\![e]\!]_{\rho[z \mapsto d][x \mapsto \rho\,!\,y]}) \\
\sqsubseteq & \langle\!\langle \text{ Induction hypothesis, } x \neq z \rangle\!\rangle \\
& fun\ z\ (\lambda d \rightarrow step\ \mathsf{App}_2\ \$\ \mathcal{S}_{\textbf{usage}}[\![\mathsf{Lam}\ x\ e\ \text{`}\mathsf{App}\text{`}\ y]\!]_{\rho[z \mapsto d]}) \\
= & \langle\!\langle \text{ Refold } \mathcal{S}_{\textbf{usage}}[\![\_]\!]_{\_} \rangle\!\rangle \\
& \mathcal{S}_{\textbf{usage}}[\![\mathsf{Lam}\ z\ (\mathsf{Lam}\ x\ e\ \text{`}\mathsf{App}\text{`}\ y)]\!]_{\rho} \\
= & \langle\!\langle x \neq z, \text{ Lemma 32 } \rangle\!\rangle \\
& \mathcal{S}_{\textbf{usage}}[\![\mathsf{Lam}\ x\ (\mathsf{Lam}\ z\ e)\ \text{`}\mathsf{App}\text{`}\ y]\!]_{\rho}
\end{array}
$$

- **Case** $\mathsf{App}\ e\ z$: Consider first the case $x = z$. This case is exemplary of the tedious calculation required to bring the substitution outside. We abbreviate $prx\ x \triangleq \langle [x \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle$.

$$
\begin{array}{ll}
& \mathcal{S}_{\textbf{usage}}[\![\mathsf{App}\ e\ z]\!]_{\rho[x \mapsto \rho\,!\,y]} \\
= & \langle\!\langle \text{ Unfold } \mathcal{S}_{\textbf{usage}}[\![\_]\!]_{\_},\ x = z \rangle\!\rangle \\
& apply\ (\mathcal{S}_{\textbf{usage}}[\![e]\!]_{\rho[x \mapsto \rho\,!\,y]})\ (\rho\,!\,y) \\
\sqsubseteq & \langle\!\langle \text{ Induction hypothesis } \rangle\!\rangle \\
& apply\ (\mathcal{S}_{\textbf{usage}}[\![\mathsf{Lam}\ x\ e\ \text{`}\mathsf{App}\text{`}\ y]\!]_{\rho})\ (\rho\,!\,y) \\
= & \langle\!\langle \text{ Unfold } apply, \mathcal{S}_{\textbf{usage}}[\![\_]\!]_{\_} \rangle\!\rangle \\
& \textbf{let}\ \langle \varphi, v \rangle = (\mathcal{S}_{\textbf{usage}}[\![e]\!]_{\rho[x \mapsto prx\ x]})[x \Mapsto (\rho\,!\,y).\varphi]\ \textbf{in} \\
& \textbf{case}\ peel\ v\ \textbf{of}\ (u, v_2) \rightarrow \langle \varphi + u * ((\rho\,!\,y).\varphi), v_2 \rangle \\
= & \langle\!\langle \text{ Unfold } \_[\_ \Mapsto \_] \rangle\!\rangle \\
& \textbf{let}\ \langle \varphi, v \rangle = \mathcal{S}_{\textbf{usage}}[\![e]\!]_{\rho[x \mapsto prx\ x]}\ \textbf{in}
\end{array}
$$

$$
\begin{array}{ll}
& \textbf{case}\ \mathit{peel}\ v\ \textbf{of}\ (u, v_2) \longrightarrow \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * ((\rho\ !\ y).\varphi) + u * ((\rho\ !\ y).\varphi), v_2 \rangle \\
= & \quad \{\ \text{Refold } \_[\_ \Mapsto \_]\ \} \\
& \textbf{let}\ \langle \varphi, v \rangle = \mathcal{S}_{\textbf{usage}}[\![e]\!]_{\rho[x \mapsto \mathit{prx}\ x]}\ \textbf{in} \\
& \textbf{case}\ \mathit{peel}\ v\ \textbf{of}\ (u, v_2) \longrightarrow \langle \varphi + u * ((\mathit{prx}\ x).\varphi), v_2 \rangle[x \Mapsto (\rho\ !\ y).\varphi] \\
= & \quad \{\ \text{Move out } \_[\_ \Mapsto \_], \text{refold } \mathit{apply}\ \} \\
& (\mathit{apply}\ (\mathcal{S}_{\textbf{usage}}[\![e]\!]_{\rho[x \mapsto \mathit{prx}\ x]})\ (\mathit{prx}\ x))[x \Mapsto (\rho\ !\ y).\varphi] \\
= & \quad \{\ \text{Refold } \mathcal{S}_{\textbf{usage}}[\![\_]\!]_{\_}\ \} \\
& \mathcal{S}_{\textbf{usage}}[\![\mathsf{Lam}\ x\ (\mathsf{App}\ e\ z)\ \text{`}\mathsf{App}\text{`}\ y]\!]_{\rho}
\end{array}
$$

When $x \neq z$:

$$
\begin{array}{ll}
& \mathcal{S}_{\textbf{usage}}[\![\mathsf{App}\ e\ z]\!]_{\rho[x \mapsto \rho\ !\ y]} \\
= & \quad \{\ \text{Unfold } \mathcal{S}_{\textbf{usage}}[\![\_]\!]_{\_}, x \neq z\ \} \\
& \mathit{apply}\ (\mathcal{S}_{\textbf{usage}}[\![e]\!]_{\rho[x \mapsto \rho\ !\ y]})\ (\rho\ !\ z) \\
\sqsubseteq & \quad \{\ \text{Induction hypothesis}\ \} \\
& \mathit{apply}\ (\mathcal{S}_{\textbf{usage}}[\![\mathsf{Lam}\ x\ e\ \text{`}\mathsf{App}\text{`}\ y]\!]_{\rho})\ (\rho\ !\ z) \\
= & \quad \{\ \text{Refold } \mathcal{S}_{\textbf{usage}}[\![\_]\!]_{\_}\ \} \\
& \mathcal{S}_{\textbf{usage}}[\![\mathsf{Lam}\ x\ e\ \text{`}\mathsf{App}\text{`}\ y\ \text{`}\mathsf{App}\text{`}\ z]\!]_{\rho} \\
= & \quad \{\ \text{Lemma 33}\ \} \\
& \mathcal{S}_{\textbf{usage}}[\![\mathsf{Lam}\ x\ (e\ \text{`}\mathsf{App}\text{`}\ z)\ \text{`}\mathsf{App}\text{`}\ y]\!]_{\rho}
\end{array}
$$

- **Case** $\mathsf{ConApp}\ k\ \mathit{xs}$: Let us concentrate on the case of a unary constructor application $\mathit{xs} = [z]$; the multi arity case is not much different.

$$
\begin{array}{ll}
& \mathcal{S}_{\textbf{usage}}[\![\mathsf{ConApp}\ k\ [z]]\!]_{\rho[x \mapsto \rho\ !\ y]} \\
= & \quad \{\ \text{Unfold } \mathcal{S}_{\textbf{usage}}[\![\_]\!]_{\_}\ \} \\
& \mathit{foldl}\ \mathit{apply}\ \langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_{\omega} \rangle\ [\rho[x \mapsto \rho\ !\ y]\ !\ z] \\
\sqsubseteq & \quad \{\ \text{Similar to } \mathsf{Var} \text{ case}\ \} \\
& \mathit{foldl}\ \mathit{apply}\ \langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_{\omega} \rangle\ [(\rho[x \mapsto \mathit{prx}\ x]\ !\ z)[x \Mapsto (\rho\ !\ y).\varphi]] \\
= & \quad \{\ x \text{ dead in } \langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_{\omega} \rangle, \text{push out substitution}\ \} \\
& (\mathit{foldl}\ \mathit{apply}\ \langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_{\omega} \rangle\ [\rho[x \mapsto \mathit{prx}\ x]\ !\ z])[x \Mapsto (\rho\ !\ y).\varphi] \\
= & \quad \{\ \text{Refold } \mathcal{S}_{\textbf{usage}}[\![\_]\!]_{\_}\ \} \\
& \mathcal{S}_{\textbf{usage}}[\![\mathsf{Lam}\ x\ (\mathsf{ConApp}\ k\ [z])\ \text{`}\mathsf{App}\text{`}\ y]\!]_{\rho}
\end{array}
$$

- **Case** $\mathsf{Case}\ e\ \mathit{alts}$: We concentrate on the single alternative e_r, single field binder z case.

$$
\begin{array}{ll}
& \mathcal{S}_{\textbf{usage}}[\![\mathsf{Case}\ e\ [k \mapsto ([z], e_r)]]\!]_{\rho[x \mapsto \rho\ !\ y]} \\
= & \quad \{\ \text{Unfold } \mathcal{S}_{\textbf{usage}}[\![\_]\!]_{\_}, \mathit{step}\ \mathsf{Case}_2 = \mathit{id}\ \} \\
& \mathit{select}\ (\mathcal{S}_{\textbf{usage}}[\![e]\!]_{\rho[x \mapsto \rho\ !\ y]})\ [k \mapsto \lambda[d] \rightarrow \mathcal{S}_{\textbf{usage}}[\![e_r]\!]_{\rho[x \mapsto \rho\ !\ y][z \mapsto d]}] \\
= & \quad \{\ \text{Unfold } \mathit{select}\ \} \\
& \mathcal{S}_{\textbf{usage}}[\![e]\!]_{\rho[x \mapsto \rho\ !\ y]} \gg \mathcal{S}_{\textbf{usage}}[\![e_r]\!]_{\rho[x \mapsto \rho\ !\ y][z \mapsto \langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_{\omega} \rangle]} \\
\sqsubseteq & \quad \{\ \text{Induction hypothesis}\ \} \\
& \mathcal{S}_{\textbf{usage}}[\![\mathsf{Lam}\ x\ e\ \text{`}\mathsf{App}\text{`}\ y]\!]_{\rho} \gg \mathcal{S}_{\textbf{usage}}[\![\mathsf{Lam}\ x\ e_r\ \text{`}\mathsf{App}\text{`}\ y]\!]_{\rho[z \mapsto \langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_{\omega} \rangle]} \\
= & \quad \{\ \text{Refold } \mathit{select}, \mathcal{S}_{\textbf{usage}}[\![\_]\!]_{\_}\ \} \\
& \mathcal{S}_{\textbf{usage}}[\![\mathsf{Case}\ (\mathsf{Lam}\ x\ e\ \text{`}\mathsf{App}\text{`}\ y)\ \mathit{alts}]\!]_{\rho[x \mapsto \rho\ !\ y]} \\
= & \quad \{\ \text{Refold } \mathit{select}, \mathcal{S}_{\textbf{usage}}[\![\_]\!]_{\_}\ \} \\
& \mathcal{S}_{\textbf{usage}}[\![\mathsf{Case}\ (\mathsf{Lam}\ x\ e\ \text{`}\mathsf{App}\text{`}\ y)\ [k \mapsto ([z], \mathsf{Lam}\ x\ e_r\ \text{`}\mathsf{App}\text{`}\ y)]]\!]_{\rho[x \mapsto \rho\ !\ y]}
\end{array}
$$

$$\begin{aligned}
&= \quad \wr\ \text{Lemma 34}\ \int \\
&\quad \mathcal{S}_{\mathbf{usage}}[\![\mathsf{Lam}\ x\ (\mathsf{Case}\ e\ [k \mapsto [z], e_r])\ \text{`}\mathsf{App}\text{`}\ y]\!]_\rho
\end{aligned}$$

- **Case** Let:

$$\begin{aligned}
&\quad \mathcal{S}_{\mathbf{usage}}[\![\mathsf{Let}\ z\ e_1\ e_2]\!]_{\rho[x \mapsto \rho\,!\,y]} \\
&= \quad \wr\ \text{Unfold } \mathcal{S}_{\mathbf{usage}}[\![_]\!]__\ \int \\
&\quad bind\ (\lambda d_1 \to \mathcal{S}_{\mathbf{usage}}[\![e_1]\!]_{\rho[x \mapsto \rho\,!\,y][z \mapsto step\ (\mathsf{Lookup}\ z)\ d_1]}) \\
&\qquad (\lambda d_1 \to step\ \mathsf{Let}_1\ (\mathcal{S}_{\mathbf{usage}}[\![e_2]\!]_{\rho[x \mapsto \rho\,!\,y][z \mapsto step\ (\mathsf{Lookup}\ z)\ d_1]})) \\
&= \quad \wr\ \text{Induction hypothesis; note that } x \text{ is absent in } \rho \text{ and thus the fixpoint}\ \int \\
&\quad bind\ (\lambda d_1 \to \mathcal{S}_{\mathbf{usage}}[\![\mathsf{Lam}\ x\ e_1\ \text{`}\mathsf{App}\text{`}\ y]\!]_{z[step\ (\mathsf{Lookup}\ z)\ d_1 \mapsto _]}) \\
&\qquad (\lambda d_1 \to step\ \mathsf{Let}_1\ (\mathcal{S}_{\mathbf{usage}}[\![\mathsf{Lam}\ x\ e_2\ \text{`}\mathsf{App}\text{`}\ y]\!]_{z[step\ (\mathsf{Lookup}\ z)\ d_1 \mapsto _]})) \\
&= \quad \wr\ \text{Refold } \mathcal{S}_{\mathbf{usage}}[\![_]\!]__\ \int \\
&\quad \mathcal{S}_{\mathbf{usage}}[\![\mathsf{Let}\ z\ (\mathsf{Lam}\ x\ e_1\ \text{`}\mathsf{App}\text{`}\ y)\ (\mathsf{Lam}\ x\ e_1\ \text{`}\mathsf{App}\text{`}\ y)]\!]_\rho \\
&= \quad \wr\ \text{Lemma 35}\ \int \\
&\quad \mathcal{S}_{\mathbf{usage}}[\![\mathsf{Lam}\ x\ (\mathsf{Let}\ z\ e_1\ e_2)\ \text{`}\mathsf{App}\text{`}\ y]\!]_\rho
\end{aligned}$$

□

Lemma 8 (Denotational absence). *Variable x is used in e if and only if there exists a by-need evaluation context E and expression e' such that the trace $\mathcal{S}_{\mathbf{need}}[\![E[\mathsf{Let}\ x\ e'\ e]]\!]_\varepsilon(\varepsilon)$ contains a $\mathsf{Lookup}\ x$ event. (Otherwise, x is absent in e.)*

Proof. Since x is used in e, there exists a trace

$$(\mathbf{let}\ \mathrm{x} = \mathrm{e}'\ \mathbf{in}\ \mathrm{e}, \rho, \mu, \kappa) \hookrightarrow^* \ldots \xrightarrow{\textsc{Look}(\mathrm{x})} \ldots$$

We proceed as follows:

$$\begin{aligned}
& (\mathbf{let}\ \mathrm{x} = \mathrm{e}'\ \mathbf{in}\ \mathrm{e}, \rho, \mu, \kappa) \hookrightarrow^* \ldots \xrightarrow{\textsc{Look}(\mathrm{x})} \ldots && (1) \\
& && \mathrm{E} \triangleq trans(\square, \rho, \mu, \kappa) \\
\Longleftrightarrow\ & init(\mathrm{E}[\mathbf{let}\ \mathrm{x} = \mathrm{e}'\ \mathbf{in}\ \mathrm{e}]) \hookrightarrow^* \ldots \xrightarrow{\textsc{Look}(\mathrm{x})} \ldots && (2) \\
& && \text{Apply } \alpha_{\mathbb{S}^\infty}\ (\text{Figure 15}) \\
\Longleftrightarrow\ & \alpha_{\mathbb{S}^\infty}(init(\mathrm{E}[\mathbf{let}\ \mathrm{x} = \mathrm{e}'\ \mathbf{in}\ \mathrm{e}]) \hookrightarrow^*, []) = \ldots\mathsf{Step}\ (\mathsf{Lookup}\ x)\ldots && (3) \\
& && \text{Theorem 4} \\
\Longleftrightarrow\ & \mathcal{S}_{\mathbf{need}}[\![E[\mathsf{Let}\ x\ e'\ e]]\!]_\varepsilon(\varepsilon) = \ldots\mathsf{Step}\ (\mathsf{Lookup}\ x)\ldots && (4)
\end{aligned}$$

Note that the trace we start with is not necessarily an maximal trace, so step (1) finds a prefix that makes the trace maximal. We do so by reconstructing the syntactic *evaluation context* E with *trans* (cf. Lemma 36) such that

$$init(\mathrm{E}[\mathbf{let}\ \mathrm{x} = \mathrm{e}'\ \mathbf{in}\ \mathrm{e}]) \hookrightarrow^* (\mathbf{let}\ \mathrm{x} = \mathrm{e}'\ \mathbf{in}\ \mathrm{e}, \rho, \mu, \kappa)$$

Then the trace above is contained in the maximal trace starting in $init(\mathrm{E}[\mathbf{let}\ \mathrm{x} = \mathrm{e}'\ \mathbf{in}\ \mathrm{e}])$ and it contains at least one $\textsc{Look}(\mathrm{x})$ transition.

The next two steps apply adequacy of $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_(\_)$ to the trace, making the shift from LK trace to denotational interpreter. □

Lemma 9 ($\mathcal{S}_{\mathbf{usage}}[\![\_]\!]_\_$ abstracts $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_$). *Let e be a closed expression and abstract the abstraction function above. Then abstract $(\mathcal{S}_{\mathbf{need}}[\![e]\!]_\varepsilon) \sqsubseteq \mathcal{S}_{\mathbf{usage}}[\![e]\!]_\varepsilon$.*

Proof. By Theorem 6, it suffices to show the abstraction laws in Figure 13.

- Mono: Always immediate, since $\sqcup$ and $+$ are the only functions matching on U, and these are monotonic.
- Unwind-Stuck, Intro-Stuck: Trivial, since $stuck = \bot$.

- Step-App, Step-Sel, Step-Inc, Update: Follows by unfolding *step*, *apply*, *select* and associativity of +.
- Beta-App: Follows from Lemma 7; see Equation (1).
- Beta-Sel: Follows by unfolding *select* and *con* and applying a lemma very similar to Lemma 7 multiple times.
- Bind-ByName: *kleeneFix* approximates the least fixpoint *lfp* since the iteratee *rhs* is monotone. We have said elsewhere that we omit a widening operator for *rhs* that guarantees that *kleeneFix* terminates.

□

Theorem 10 ($\mathcal{S}_{\mathbf{usage}}[\![\_]\!]_{\_}$ **infers absence).** *Let* $\rho_e \triangleq [\overline{y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle}]$ *be the initial environment with an entry for every free variable* y *of an expression* e*. If* $\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_e} = \langle \varphi, v \rangle$ *and* $\varphi\ !?\ x = \mathsf{U}_0$*, then* x *is absent in* e*.*

Proof. We show the contraposition, that is, if x is used in e, then $\varphi\ !?\ x \neq \mathsf{U}_0$.
By Lemma 8, there exists E, e' such that

$$\mathcal{S}_{\mathbf{need}}[\![E[\mathsf{Let}\ x\ e'\ e]]\!]_{\varepsilon}(\varepsilon) = \ldots\ \mathsf{Step}\ (\mathsf{Lookup}\ x)\ \ldots .$$

This is the big picture of how we prove $\varphi\ !?\ x \neq \mathsf{U}_0$ from this fact:

$$\begin{aligned}
& \mathcal{S}_{\mathbf{need}}[\![E[\mathsf{Let}\ x\ e'\ e]]\!]_{\varepsilon}(\varepsilon) = \ldots \mathsf{Step}\ (\mathsf{Lookup}\ x)\ \ldots && (5)\\
& \qquad \text{\textit{Usage instrumentation}} \\
\implies & (\alpha\ \{\mathcal{S}_{\mathbf{need}}[\![E[\mathsf{Let}\ x\ e'\ e]]\!]_{\varepsilon}(\varepsilon)\}).\varphi \sqsupseteq [x \mapsto \mathsf{U}_1] && (6)\\
& \qquad \text{\textit{Lemma 9}} \\
\implies & (\mathcal{S}_{\mathbf{usage}}[\![E[\mathsf{Let}\ x\ e'\ e]]\!]_{\varepsilon}).\varphi \sqsupseteq [x \mapsto \mathsf{U}_1] && (7)\\
& \qquad \text{\textit{Lemma 38}} \\
\implies & \mathsf{U}_\omega * (\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_e}).\varphi = \mathsf{U}_\omega * \varphi \sqsupseteq [x \mapsto \mathsf{U}_1] && (8)\\
& \qquad \mathsf{U}_\omega * \mathsf{U}_0 = \mathsf{U}_0 \sqsubset \mathsf{U}_1 \\
\implies & \varphi\ !?\ x \neq \mathsf{U}_0 && (9)
\end{aligned}$$

Step (5) instruments the trace by applying the usage abstraction function $\alpha \rightleftharpoons \_ \triangleq \mathit{nameNeed}$. This function will replace every Step constructor with the *step* implementation of T_{U}; The $\mathsf{Lookup}\ x$ event on the right-hand side implies that its image under α is at least $[x \mapsto \mathsf{U}_1]$.
Step (6) applies the central soundness Lemma 9 that is the main topic of this section, abstracting the dynamic trace property in terms of the static semantics.
Finally, step (7) applies Lemma 38, which proves that absence information doesn't change when an expression is put in an arbitrary evaluation context. The final step is just algebra. □

In the proof for Theorem 10 we exploit that usage analysis is somewhat invariant under wrapping of *by-need evaluation contexts*, roughly $\mathsf{U}_\omega * \mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_e} = \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\varepsilon}$. To prove that, we first need to define what the by-need evaluation contexts of our language are.
Moran and Sands [1999, Lemma 4.1] describe a principled way to derive the call-by-need evaluation contexts E from machine contexts $(\square, \mu, \kappa)$ of the Sestoft Mark I machine; a variant of Figure 2 that uses syntactic substitution of variables instead of delayed substitution and addresses, so $\mu \in \mathsf{Var} \rightharpoonup \mathsf{Exp}$ and no closures are needed.
We follow their approach, but inline applicative contexts,[32] thus defining the by-need evaluation contexts with hole $\square$ for our language as

$$\mathsf{E} \in \mathbb{EC} \quad ::= \quad \square \mid \mathsf{E}\ \mathsf{x} \mid \mathbf{case}\ \mathsf{E}\ \mathbf{of}\ \overline{K\ \overline{\mathsf{x}} \rightarrow \mathsf{e}} \mid \mathbf{let}\ \mathsf{x} = \mathsf{e}\ \mathbf{in}\ \mathsf{E} \mid \mathbf{let}\ \mathsf{x} = \mathsf{E}\ \mathbf{in}\ \mathsf{E}[\mathsf{x}]$$

[32] The result is that of Ariola et al. [1995, Figure 3] in A-normal form and extended with data types.

The correspondence to Mark I machine contexts $(\Box, \mu, \kappa)$ is encoded by the following translation function *trans* that translates from mark I machine contexts $(\Box, \mu, \kappa)$ to evaluation contexts E.

$$
\begin{array}{lcl}
\mathit{trans} & : & \mathbb{EC} \times \mathbb{H} \times \mathbb{K} \to \mathbb{EC} \\
\mathit{trans}(\mathsf{E}, [\overline{\mathsf{x} \mapsto \mathsf{e}}], \kappa) & = & \overline{\mathbf{let}\ \mathsf{x} = \mathsf{e}}\ \mathbf{in}\ \mathit{trans}(\mathsf{E}, [], \kappa) \\
\mathit{trans}(\mathsf{E}, [], \mathbf{ap}(\mathsf{x}) \cdot \kappa) & = & \mathit{trans}(\mathsf{E}\ \mathsf{x}, [], \kappa) \\
\mathit{trans}(\mathsf{E}, [], \mathbf{sel}(\overline{K\ \overline{\mathsf{x}} \to \mathsf{e}}) \cdot \kappa) & = & \mathit{trans}(\mathbf{case}\ \mathsf{E}\ \mathbf{of}\ \overline{K\ \overline{\mathsf{x}} \to \mathsf{e}}, [], \kappa) \\
\mathit{trans}(\mathsf{E}, [], \mathbf{upd}(\mathsf{x}) \cdot \kappa) & = & \mathbf{let}\ \mathsf{x} = \mathsf{E}\ \mathbf{in}\ \mathit{trans}(\Box, [], \kappa)[\mathsf{x}] \\
\mathit{trans}(\mathsf{E}, [], \mathbf{stop}) & = & \mathsf{E}
\end{array}
$$

Certainly the most interesting case is that of **upd** frames, encoding by-need memoisation. This translation function has the following property:

Lemma 36 (Translation, without proof). *$\mathit{init}(\mathit{trans}(\Box, \mu, \kappa)[\mathsf{e}]) \hookrightarrow^{*} (\mathsf{e}, \mu, \kappa)$, and all transitions in this trace are search transitions ($\textsc{App}_1$, $\textsc{Case}_1$, $\textsc{Let}_1$, $\textsc{Look}$).*

In other words: every machine configuration σ corresponds to an evaluation context E and a focus expression e such that there exists a trace $\mathit{init}(\mathsf{E}[\mathsf{e}]) \hookrightarrow^{*} \sigma$ consisting purely of search transitions, which is equivalent to all states in the trace except possibly the last being evaluation states.

We encode evaluation contexts in Haskell as follows, overloading hole filling notation $\_[\_]$:

```
data ECtxt = Hole | Apply ECtxt Name | Select ECtxt Alts
           | ExtendHeap Name Expr ECtxt | UpdateHeap Name ECtxt Expr
_[_] :: ECtxt → Expr → Expr
Hole[e]                   = e
(Apply E x)[e]            = App E[e] x
(Select E alts)[e]        = Case E[e] alts
(ExtendHeap x e1 E)[e2]   = Let x e1 E[e2]
(UpdateHeap x E e1)[e2]   = Let x E[e1] e2
```

Lemma 37 (Used variables are free). *If x does not occur in e and in ρ (that is, $\forall y.\ (\rho \mathbin{!} y).\varphi \mathbin{!?} x = \mathsf{U}_0$), then $(\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho}).\varphi \mathbin{!?} x = \mathsf{U}_0$.*

Proof. By induction on e. $\square$

Lemma 38 (Context closure). *Let e be an expression and E be a by-need evaluation context in which x does not occur. Then $(\mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E}).\varphi \mathbin{?!} x \sqsubseteq \mathsf{U}_\omega * ((\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_e}).\varphi \mathbin{!?} x)$, where ρ_E and ρ_e are the initial environments that map free variables z to their proxy $\langle [z \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle$.*

Proof. We will sometimes need that if y does not occur free in e_1, we have By induction on the size of E and cases on E:

- **Case** Hole:

$$
\begin{array}{ll}
 & (\mathcal{S}_{\mathbf{usage}}[\![\mathsf{Hole}[e]]\!]_{\rho_E}).\varphi \mathbin{!?} x \\
= & \quad \langle \text{Definition of } \_[\_] \rangle \\
 & (\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_E}).\varphi \mathbin{!?} x \\
\sqsubseteq & \quad \langle \rho_e = \rho_E \rangle \\
 & \mathsf{U}_\omega * (\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_E}).\varphi \mathbin{!?} x
\end{array}
$$

  By reflexivity.

- **Case** $\mathsf{Apply}\ E\ y$: Since y occurs in E, it must be different to x.

$$
\begin{array}{ll}
 & (\mathcal{S}_{\mathbf{usage}}[\![(\mathsf{Apply}\ E\ y)[e]]\!]_{\rho_E}).\varphi \mathbin{!?} x \\
= & \wr\ \text{Definition of } \_[\_]\ \wr \\
 & (\mathcal{S}_{\mathbf{usage}}[\![\mathsf{App}\ E[e]\ \ y]\!]_{\rho_E}).\varphi \mathbin{!?} x \\
= & \wr\ \text{Definition of } \mathcal{S}_{\mathbf{usage}}[\![\_]\!]_{\_}\ \wr \\
 & (\mathit{apply}\ (\mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E})\ (\rho_E \mathbin{!?} y)).\varphi \mathbin{!?} x \\
= & \wr\ \text{Definition of } \mathit{apply}\ \wr \\
 & \mathbf{let}\ \langle \varphi, v\rangle = \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E}\ \mathbf{in} \\
 & \mathbf{case}\ \mathit{peel}\ v\ \mathbf{of}\ (u, v_2) \rightarrow (\langle \varphi + u * ((\rho_E \mathbin{!?} y).\varphi), v_2\rangle.\varphi \mathbin{!?} x) \\
= & \wr\ \text{Unfold } \langle \varphi, v\rangle.\varphi = \varphi,\ x \text{ absent in } \rho_E \mathbin{!?} y\ \wr \\
 & \mathbf{let}\ \langle \varphi, v\rangle = \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E}\ \mathbf{in} \\
 & \mathbf{case}\ \mathit{peel}\ v\ \mathbf{of}\ (u, v_2) \rightarrow \varphi \mathbin{!?} x \\
= & \wr\ \text{Refold } \langle \varphi, v\rangle.\varphi = \varphi\ \wr \\
 & (\mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E}).\varphi \mathbin{!?} x \\
\sqsubseteq & \wr\ \text{Induction hypothesis}\ \wr \\
 & \mathsf{U}_\omega * (\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_e}).\varphi \mathbin{!?} x
\end{array}
$$

- **Case** $\mathsf{Select}\ E\ \mathit{alts}$: Since x does not occur in alts, it is absent in alts as well by Lemma 37. (Recall that select analyses alts with $\langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_\omega\rangle$ as field proxies.)

$$
\begin{array}{ll}
 & (\mathcal{S}_{\mathbf{usage}}[\![(\mathsf{Select}\ E\ \mathit{alts})[e]]\!]_{\rho_E}).\varphi \mathbin{!?} x \\
= & \wr\ \text{Definition of } \_[\_]\ \wr \\
 & (\mathcal{S}_{\mathbf{usage}}[\![\mathsf{Case}\ E[e]\ \ \mathit{alts}]\!]_{\rho_E}).\varphi \mathbin{!?} x \\
= & \wr\ \text{Definition of } \mathcal{S}_{\mathbf{usage}}[\![\_]\!]_{\_}\ \wr \\
 & (\mathit{select}\ (\mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E})\ (\mathit{cont} \triangleleft \mathit{alts})).\varphi \mathbin{!?} x \\
= & \wr\ \text{Definition of } \mathit{select}\ \wr \\
 & (\mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E} \gg \mathit{lub}\ (...\mathit{alts}...)).\varphi \mathbin{!?} x \\
= & \wr\ x \text{ absent in } \mathit{lub}\ (...\mathit{alts}...)\ \wr \\
 & (\mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E}).\varphi \mathbin{!?} x \\
\sqsubseteq & \wr\ \text{Induction hypothesis}\ \wr \\
 & \mathsf{U}_\omega * (\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_e}).\varphi \mathbin{!?} x
\end{array}
$$

- **Case** $\mathsf{ExtendHeap}\ y\ e_1\ E$: Since x does not occur in e_1, and the initial environment is absent in x as well, we have $(\mathcal{S}_{\mathbf{usage}}[\![e_1]\!]_{\rho_E}).\varphi \mathbin{!?} x = \mathsf{U}_0$ by Lemma 37.

$$
\begin{array}{ll}
 & (\mathcal{S}_{\mathbf{usage}}[\![(\mathsf{ExtendHeap}\ y\ e_1\ E)[e]]\!]_{\rho_E}).\varphi \mathbin{!?} x \\
= & \wr\ \text{Definition of } \_[\_]\ \wr \\
 & (\mathcal{S}_{\mathbf{usage}}[\![\mathsf{Let}\ y\ e_1\ E[e]]\!]_{\rho_E}).\varphi \mathbin{!?} x \\
= & \wr\ \text{Definition of } \mathcal{S}_{\mathbf{usage}}[\![\_]\!]_{\_}\ \wr \\
 & (\mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathit{kleeneFix}\ (\lambda d \rightarrow \mathcal{S}_{\mathbf{usage}}[\![e_1]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Lookup}\ y)\ d]}))]}).\varphi \mathbin{!?} x \\
\sqsubseteq & \wr\ \text{Abstract substitution; Lemma 7}\ \wr \\
 & (\mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega\rangle]})[y \Mapsto \mathit{step} \\
 & \quad (\mathsf{Lookup}\ y)\ (\mathit{kleeneFix}\ (\lambda d \rightarrow \mathcal{S}_{\mathbf{usage}}[\![e_1]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Lookup}\ y)\ d]}))].\varphi \mathbin{!?} x \\
= & \wr\ \text{Unfold } \_[\_ \Mapsto \_],\ \langle \varphi, v\rangle.\varphi = \varphi\ \wr \\
 & \mathbf{let}\ \langle \varphi, \_\rangle = \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega\rangle]}\ \mathbf{in} \\
 & \mathbf{let}\ \langle \varphi_2, \_\rangle = \mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathit{kleeneFix}\ (\lambda d \rightarrow \mathcal{S}_{\mathbf{usage}}[\![e_1]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Lookup}\ y)\ d]}))\ \mathbf{in} \\
 & (\varphi[y \mapsto \mathsf{U}_0] + (\varphi \mathbin{!?} y) * \varphi_2) \mathbin{!?} x
\end{array}
$$

$$
\begin{array}{ll}
= & \wr\ x \text{ absent in } \varphi_2\text{, see above } \int \\
& \mathbf{let}\ \langle \varphi, \_ \rangle = \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle]}\ \mathbf{in} \\
& \varphi\ !?\ x \\
\sqsubseteq & \wr\ \text{Induction hypothesis } \int \\
& \mathsf{U}_\omega * (\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_e}).\varphi\ !?\ x
\end{array}
$$

- **Case** $\mathsf{UpdateHeap}\ y\ E\ e_1$: Since x does not occur in e_1, and the initial environment is absent in x as well, we have $(\mathcal{S}_{\mathbf{usage}}[\![e_1]\!]_{\rho_E[y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle]}).\varphi\ !?\ x = \mathsf{U}_0$ by Lemma 37.

$$
\begin{array}{ll}
& (\mathcal{S}_{\mathbf{usage}}[\![(\mathsf{UpdateHeap}\ y\ E\ e_1)[e]]\!]_{\rho_E}).\varphi\ !?\ x \\
= & \wr\ \text{Definition of } \_[\_]\ \int \\
& (\mathcal{S}_{\mathbf{usage}}[\![\mathsf{Let}\ y\ E[e]\ e_1]\!]_{\rho_E}).\varphi\ !?\ x \\
= & \wr\ \text{Definition of } \mathcal{S}_{\mathbf{usage}}[\![\_]\!]_\_\ \int \\
& (\mathcal{S}_{\mathbf{usage}}[\![e_1]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathit{kleeneFix}\ (\lambda d \to \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Lookup}\ y)\ d]}))]}).\varphi\ !?\ x \\
\sqsubseteq & \wr\ \text{Abstract substitution; Lemma 7 } \int \\
& (\mathcal{S}_{\mathbf{usage}}[\![e_1]\!]_{\rho_E[y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle]})[y \Mapsto \mathit{step} \\
& \quad (\mathsf{Lookup}\ y)\ (\mathit{kleeneFix}\ (\lambda d \to \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Lookup}\ y)\ d]}))].\varphi\ !?\ x \\
= & \wr\ \text{Unfold } \_[\_ \Mapsto \_],\ \langle \varphi, v \rangle.\varphi = \varphi\ \int \\
& \mathbf{let}\ \langle \varphi, \_ \rangle = \mathcal{S}_{\mathbf{usage}}[\![e_1]\!]_{\rho_E[y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle]}\ \mathbf{in} \\
& \mathbf{let}\ \langle \varphi_2, \_ \rangle = \mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathit{kleeneFix}\ (\lambda d \to \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Lookup}\ y)\ d]}))\ \mathbf{in} \\
& (\varphi[y \mapsto \mathsf{U}_0] + (\varphi\ !?\ y) * \varphi_2)\ !?\ x \\
= & \wr\ \varphi\ !?\ y \sqsubseteq \mathsf{U}_\omega,\ x \text{ absent in } \varphi\text{, see above } \int \\
& \mathbf{let}\ \langle \varphi_2, \_ \rangle = \mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathit{kleeneFix}\ (\lambda d \to \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Lookup}\ y)\ d]}))\ \mathbf{in} \\
& \mathsf{U}_\omega * \varphi_2\ !?\ x \\
= & \wr\ \text{Refold } \langle \varphi, v \rangle.\varphi\ \int \\
& \mathsf{U}_\omega * (\mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathit{kleeneFix}\ (\lambda d \to \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Lookup}\ y)\ d]}))).\varphi\ !?\ x \\
= & \wr\ x \neq y\ \int \\
& \mathsf{U}_\omega * (\mathit{kleeneFix}\ (\lambda d \to \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto d]})).\varphi\ !?\ x \\
= & \wr\ \text{Argument below } \int \\
& \mathsf{U}_\omega * (\mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle]}).\varphi\ !?\ x \\
\sqsubseteq & \wr\ \text{Induction hypothesis, } \mathsf{U}_\omega * \mathsf{U}_\omega = \mathsf{U}_\omega\ \int \\
& \mathsf{U}_\omega * (\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_e}).\varphi\ !?\ x
\end{array}
$$

The rationale for removing the *kleeneFix* is that under the assumption that x is absent in d (such as is the case for $d \triangleq \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle$), then it is also absent in $E[e]\ \rho_E[y \mapsto d]$ per Lemma 37. Otherwise, we go to U_ω anyway.
UpdateHeap is why it is necessary to multiply with U_ω above; in the context $\mathbf{let}\ x = \square\ \mathbf{in}\ x\ x$, a variable y put in the hole would really be evaluated twice under call-by-name (where $\mathbf{let}\ x = \square\ \mathbf{in}\ x\ x$ is *not* an evaluation context).
This unfortunately means that the used-once results do not generalise to arbitrary by-need evaluation contexts and it would be unsound to elide update frames for y based on the inferred use of y in $\mathbf{let}\ y = ...\ \mathbf{in}$ e; for e $\triangleq y$ we would infer that y is used at most once, but that is wrong in context $\mathbf{let}\ x = \square\ \mathbf{in}\ x\ x$.

□

D.1 Abstract Interpretation and Denotational Interpreters

So far, we have seen how to *use* the abstraction Theorem 6, but its proof merely points to its generalisation for open terms, Theorem 56. Proving this theorem correct is the goal of this subsection and the following, where we approach the problem from the bottom up.

We begin by describing how we intend to apply abstract interpretation to our denotational interpreter, considering open expressions as well, which necessitate abstraction of environments.

Given a "concrete" (but perhaps undecidable, infinite or coinductive) semantics and a more "abstract" (but perhaps decidable, finite and inductive) semantics, when does the latter *soundly approximate* properties of the former? This question is a prominent one in program analysis, and *Abstract Interpretation* Cousot [2021] provides a generic framework to formalise this question.

Sound approximation is encoded by a Galois connection $(\mathsf{D}, \leqslant) \xleftarrow[\alpha]{\gamma} (\widehat{\mathsf{D}}, \sqsubseteq)$ between concrete and abstract semantic domains D and $\widehat{\mathsf{D}}$ equipped with a partial order. An element $\widehat{d} \in \widehat{\mathsf{D}}$ soundly approximates $d \in \mathsf{D}$ iff $d \leqslant \gamma\ \widehat{d}$, iff $\alpha\ d \sqsubseteq \widehat{d}$. This theory bears semantic significance when $(\mathsf{D}, \leqslant)$ is instantiated to the complete lattice of trace properties $(\wp(\mathbb{T}), \subseteq)$, where $\mathbb{T}$ is the set of program traces. Then the *collecting semantics* relative to a concrete, trace-generating semantics $\mathcal{S}_{\mathbb{T}}[\![\_]\!]_{\_}$, defined as $\mathcal{S}_{\mathbb{C}}[\![e]\!]_{\rho} \triangleq \{\mathcal{S}_{\mathbb{T}}[\![e]\!]_{\rho}\}$, provides the strongest trace property that a given program (e, ρ) satisfies. In this setting, we extend the original Galois connection to the signature of $\mathcal{S}_{\mathbb{T}}[\![e]\!]_{\_}$ *parametrically*,[33] to

$$((\mathsf{Name} :\rightharpoonup \wp(\mathbb{T})) \to \wp(\mathbb{T}), \dot{\subseteq}) \xleftarrow[\lambda f \to \alpha \circ f \circ (\gamma \triangleleft)]{\lambda \widehat{f} \to \gamma \circ \widehat{f} \circ (\alpha \triangleleft)} ((\mathsf{Name} :\rightharpoonup \widehat{\mathsf{D}}) \to \widehat{\mathsf{D}}, \dot{\sqsubseteq}),$$

and state soundness of the abstract semantics $\mathcal{S}_{\widehat{\mathsf{D}}}[\![\_]\!]_{\_}$ as

$$\mathcal{S}_{\mathbb{C}}[\![e]\!]_{\rho} \subseteq \gamma\ (\mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_{\alpha \triangleleft \{\_\} \triangleleft \rho}) \iff \alpha\ \{\mathcal{S}_{\mathbb{T}}[\![e]\!]_{\rho}\} \sqsubseteq \mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_{\alpha \triangleleft \{\_\} \triangleleft \rho}.$$

The statement should be read as "The concrete semantics implies the abstract semantics up to concretisation" Cousot [2021, p. 26]. It looks a bit different to what we exemplified in Theorem 6 for the following reasons: (1) $\mathcal{S}_{\mathbb{T}}[\![\_]\!]_{\_}$ and $\mathcal{S}_{\widehat{\mathsf{D}}}[\![\_]\!]_{\_}$ are in fact different type class instantiations of the same denotational interpreter $\mathcal{S}[\![\_]\!]_{\_}$ from Section 4, thus both functions share a lot of common structure. (2) The Galois connections *byName* and *nameNeed* defined below are completely determined by type class instances, even for infinite traces. (3) It turns out that we need to syntactically restrict the kind of D that occurs in an environment ρ due to the full abstraction problem [Plotkin 1977], so that the Galois connection *byName* looks a bit different. (4) By-need semantics is stateful whereas analyses such as usage analysis are rarely so; this again leads to a slightly different use of the final Galois connection *nameNeed* as exemplified in Theorem 6.

D.2 Guarded Fixpoints, Safety Properties and Safety Extension of a Galois Connection

We like to describe a semantic trace property as a "fold", in terms of a Trace instance. For example, we collect a trace into a Uses in Section 6.1 and Lemma 9. Of course such a fold (an inductive elimination procedure) has no meaning when the trace is infinite! Yet it is always clear what we mean: when the trace is infinite, we consider the meaning of the fold as the limit (i.e., least fixpoint) of its finite prefixes. In this subsection, we discuss when and why this meaning is correct.

Suppose for a second that we were only interested in the trace component of our semantic domain, thus effectively restricting ourselves to $\mathbb{T} \triangleq \mathsf{T}\ ()$, and that we were to approximate properties $P \in \wp(\mathbb{T})$ about such traces by a Galois connection $(\wp(\mathbb{T}), \subseteq) \xleftarrow[\alpha]{\gamma} (\widehat{\mathsf{D}}, \sqsubseteq)$. Alas, although the abstraction

[33]"Parametrically" in the sense of Backhouse and Backhouse [2004], i.e., the structural properties of a Galois connection follow as a free theorem.

function α is well-defined as a mathematical function, it most certainly is *not* computable at infinite inputs (in $\mathbb{T}^\infty$), for example at fix (Step (Lookup x)) = Step (Lookup x) (Step (Lookup x)...)!

Computing with such an α is of course inacceptable for a *static* analysis. Usually this is resolved by approximating the fixpoint by the least fixpoint of the abstracted iteratee, e.g., lfp ($\alpha \circ$ Step (Lookup x) $\circ\, \gamma$). It is however not the case that this yields a sound approximation of infinite traces for *arbitrary* trace properties. A classic counterexample is the property $P \triangleq \{\tau \mid \tau \text{ terminates}\}$; if P is restricted to finite traces $\mathbb{T}^*$, the analysis that constantly says "terminates" is correct; however this result doesn't carry over "to the limit", when τ may also range over infinite traces in $\mathbb{T}^\infty$. Hence it is impossible to soundly approximate P with a least fixpoint in the abstract.

Rather than making the common assumption that infinite traces are soundly approximated by $\bot$ (such as in strictness analysis [Mycroft 1980; Wadler and Hughes 1987]), thus effectively assuming that all executions are finite, our framework assumes that the properties of interest are *safety properties* [Lamport 1977]:

Definition 39 (Safety property). *A trace property $P \subseteq \mathbb{T}$ is a* safety property *iff, whenever $\tau_1 \in \mathbb{T}^\infty$ violates P (so $\tau_1 \notin P$), then there exists some proper prefix $\tau_2 \in \mathbb{T}^*$ (written $\tau_2 \lessdot \tau_1$) such that $\tau_2 \notin P$.*

Note that both well-typedness ("τ does not go wrong") and usage cardinality abstract safety properties. Conveniently, guarded recursive predicates (on traces) always describe safety properties [Birkedal and Bizjak 2023; Spies et al. 2021]. The contraposition of the above definition is

$$\forall \tau_1 \in \mathbb{T}^\infty.\ (\forall \tau_2 \in \mathbb{T}^*.\ \tau_2 \lessdot \tau_1 \Longrightarrow \tau_2 \in P) \Longrightarrow \tau_1 \in P,$$

and we can exploit safety to extend a finitary Galois connection to infinite inputs:

Lemma 40 (Safety extension). *Let $\widehat{\mathsf{D}}$ be a domain with instances for* Trace *and* Lat, $(\wp(\mathbb{T}^*), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\widehat{\mathsf{D}}, \sqsubseteq)$ *a Galois connection and $P \in \wp(\mathbb{T})$ a safety property. Then any domain element $\widehat{d}$ that soundly approximates P via γ on finite traces soundly approximates P on infinite traces as well:*

$$\forall \widehat{d}.\ P \cap \mathbb{T}^* \subseteq \gamma(\widehat{d}) \Longrightarrow P \cap \mathbb{T}^\infty \subseteq \gamma^\infty(\widehat{d}),$$

where the extension $(\wp(\mathbb{T}^*), \subseteq) \underset{\alpha^\infty}{\overset{\gamma^\infty}{\leftrightarrows}} (\widehat{\mathsf{D}}, \sqsubseteq)$ *of* $\underset{\alpha}{\overset{\gamma}{\leftrightarrows}}$ *is defined by the following abstraction function:*

$$\alpha^\infty(P) \triangleq \alpha(\{\tau_2 \mid \exists \tau_1 \in P.\ \tau_2 \lessdot \tau_1\})$$

Proof. First note that α^∞ uniquely determines the Galois connection by the representation function [Nielson et al. 1999, Section 4.3]

$$\beta^\infty(\tau_1) \triangleq \alpha(\bigcup\{\tau_2 \mid \tau_2 \lessdot \tau_1\}).$$

Now let $\tau \in P \cap \mathbb{T}^\infty$. The goal is to show that $\tau \in \gamma^\infty(\widehat{d})$, which we rewrite as follows:

$$
\begin{array}{ll}
 & \tau \in \gamma^\infty\ \widehat{d} \\
\Longleftrightarrow & \quad \wr\ \text{Galois}\ \wr \\
 & \beta^\infty\ \tau \sqsubseteq \widehat{d} \\
\Longleftrightarrow & \quad \wr\ \text{Definition of } \beta^\infty\ \wr \\
 & \alpha\ \bigcup\{\tau_2 \mid \tau_2 \lessdot \tau_1\} \sqsubseteq \widehat{d} \\
\Longleftrightarrow & \quad \wr\ \text{Galois}\ \wr \\
 & \bigcup\{\tau_2 \mid \tau_2 \lessdot \tau_1\} \subseteq \gamma\ \widehat{d} \\
\Longleftrightarrow & \quad \wr\ \text{Definition of Union}\ \wr \\
 & \forall \tau_2.\ \tau_2 \lessdot \tau \Longrightarrow \tau_2 \in \gamma\ \widehat{d}
\end{array}
$$

On the other hand, P is a safety property and $\tau \in P$, so for any prefix τ_2 of τ we have $\tau_2 \in P \cap \mathbb{T}^*$. Hence the goal follows by assumption that $P \cap \mathbb{T}^* \subseteq \gamma(\widehat{d})$. □

From now on, we tacitly assume that all trace properties of interest are safety properties, and that any Galois connection defined in Haskell has been extended to infinite traces via Lemma 40. Any such Galois connection can be used to approximate guarded fixpoints via least fixpoints:

Lemma 41 (Guarded fixpoint abstraction for safety extensions). *Let $\widehat{\mathsf{D}}$ be a domain with instances for* Trace *and* Lat*, and let $(\wp(\mathbb{T}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\widehat{\mathsf{D}}, \sqsubseteq)$ a Galois connection extended to infinite traces via Lemma 40. Then, for any guarded iteratee $f :: \blacktriangleright\mathbb{T} \to \mathbb{T}$,*

$$\alpha(\{\mathit{fix}\ f\}) \sqsubseteq \mathit{lfp}\ (\alpha \circ f^* \circ \gamma),$$

where $\mathit{lfp}\ \widehat{f}$ denotes the least fixpoint of $\widehat{f}$ and $f^* :: \wp(\blacktriangleright\mathbb{T}) \to \wp(\mathbb{T})$ is the lifting of f to powersets.

Proof. We should note that the proposition is sloppy in the treatment of ▸ and should rather have been

$$\alpha(\{\mathit{fix}\ f\}) \sqsubseteq \mathit{lfp}\ (\alpha \circ f \circ \mathit{next}^* \circ \gamma),$$

where $\mathit{next} :: \blacktriangleright\mathsf{T} \to \mathsf{T}$. Since we have proven totality in Section 5.2, the utility of being explicit in next is rather low (much more so since a pen and paper proof is not type checked) and we will admit ourselves this kind of sloppiness from now on.

Let us assume that $\tau = \mathit{fix}\ f$ is finite and proceed by Löb induction.

$$\begin{array}{ll}
 & \alpha\ \{\mathit{fix}\ f\} \sqsubseteq \mathit{lfp}\ (\alpha \circ f^* \circ \gamma) \\
= & \wr\ \mathit{fix}\ f = f\ (\mathit{fix}\ f)\ \int \\
 & \alpha\ \{f\ (\mathit{fix}\ f)\} \\
= & \wr\ \text{Commute } f \text{ and } \{_\}\ \int \\
 & \alpha\ (f^*\ \{\mathit{fix}\ f\}) \\
\sqsubseteq & \wr\ \mathit{id} \sqsubseteq \gamma \circ \alpha\ \int \\
 & \alpha\ (f^*\ (\gamma\ (\alpha\ \{\mathit{fix}\ f\}))) \\
\sqsubseteq & \wr\ \text{Induction hypothesis}\ \int \\
 & \alpha\ (f^*\ (\gamma\ (\mathit{lfp}\ (\alpha \circ f^* \circ \gamma)))) \\
\sqsubseteq & \wr\ \mathit{lfp}\ \widehat{f} = \widehat{f}\ (\mathit{lfp}\ \widehat{f})\ \int \\
 & \mathit{lfp}\ (\alpha \circ f^* \circ \gamma)
\end{array}$$

When τ is infinite, the result follows by Lemma 40 and the fact that all properties of interest are safety properties. □

D.3 Abstract By-name Soundness, in Detail

We will now see how the by-name abstraction laws in Figure 13 induce an abstract interpretation of by-name evaluation. The corresponding proofs are somewhat simpler than for by-need because no heap update is involved.

As we are getting closer to the point where we reason about idealised, total Haskell code, it is important to nail down how Galois connections are represented in Haskell, and how we construct them. Following Nielson et al. [1999, Section 4.3], every *representation function* $\beta :: a \to b$ into a partial order $(b, \sqsubseteq)$ yields a Galois connection between Powersets of a and $(b, \sqsubseteq)$:

```
data GC a b = (a → b) ⇄ (b → a)
repr :: Lat b ⇒ (a → b) → GC (Pow a) b
repr β = α ⇄ γ where α (P as) = ⨆{β a | a ← as}; γ b = P {a | β a ⊑ b}
```

While the γ exists as a mathematical function, it is in general impossible to compute even for finitary inputs. Every domain $\widehat{\mathsf{D}}$ with instances $(\mathsf{Trace}\ \widehat{\mathsf{D}}, \mathsf{Domain}\ \widehat{\mathsf{D}}, \mathsf{Lat}\ \widehat{\mathsf{D}})$ induces a *trace abstraction* via the following representation function, writing f^* to map f over Pow[34]

$$
\begin{array}{l}
\textbf{type}\ (d \vdash^{\mathsf{na}}_{\mathbb{D}} \_) = d \quad \text{-- exact meaning defined below} \\
\mathit{trace} :: (\mathsf{Trace}\ \widehat{d}, \mathsf{Domain}\ \widehat{d}, \mathsf{Lat}\ \widehat{d}) \\
\qquad \Rightarrow \mathsf{GC}\ (\mathsf{Pow}\ (\mathsf{D}\ r))\ \widehat{d} \rightarrow \mathsf{GC}\ (\mathsf{Pow}\ (\mathsf{D}\ r \vdash^{\mathsf{na}}_{\mathbb{D}} \_))\ (\widehat{d} \vdash^{\mathsf{na}}_{\mathbb{D}} \_) \rightarrow \mathsf{GC}\ (\mathsf{Pow}\ (\mathsf{T}\ (\mathsf{Value}\ r)))\ \widehat{d} \\
\mathit{trace}\ (\alpha_{\mathbb{T}} \rightleftharpoons \gamma_{\mathbb{T}})\ (\alpha_{\mathbb{E}} \rightleftharpoons \gamma_{\mathbb{E}}) = \mathit{repr}\ \beta\ \textbf{where} \\
\quad \beta\ (\mathsf{Ret}\ \mathsf{Stuck}) \qquad\quad\ = \mathit{stuck} \\
\quad \beta\ (\mathsf{Ret}\ (\mathsf{Fun}\ f)) \qquad = \mathit{fun}\ (\alpha_{\mathbb{T}} \circ f^* \circ \gamma_{\mathbb{E}}) \\
\quad \beta\ (\mathsf{Ret}\ (\mathsf{Con}\ k\ ds)) = \mathit{con}\ k\ (\mathit{map}\ (\alpha_{\mathbb{E}} \circ \{\_\})\ ds) \\
\quad \beta\ (\mathsf{Step}\ e\ \widehat{d}) \qquad\quad\ = \mathit{step}\ e\ (\beta\ \widehat{d})
\end{array}
$$

Note how *trace* expects two Galois connections: The first one is applicable in the "recursive case" and the second one applies to (the powerset over) $\mathsf{D}\ (\mathsf{ByName}\ \mathsf{T}) \vdash^{\mathsf{na}}_{\mathbb{D}} \_$, a subtype of $\mathsf{D}\ (\mathsf{ByName}\ \mathsf{T})$. Every $d :: (\mathsf{ByName}\ \mathsf{T} \vdash^{\mathsf{na}}_{\mathbb{D}} \_)$ is of the form $\mathsf{Step}\ (\mathsf{Lookup}\ x)\ (\mathcal{S}[\![e]\!]_{\rho})$ for some x, e, ρ, characterising domain elements that end up in an environment or are passed around as arguments or in fields. We have seen a similar characterisation in the Agda encoding of Section 5.1. The distinction between $\alpha_{\mathbb{T}}$ and $\alpha_{\mathbb{E}}$ will be important for proving that evaluation preserves trace abstraction (comparable to Lemma 19 for a big-step-style semantics), a necessary auxiliary lemma for Theorem 44.

We utilise the *trace* combinator to define *byName* abstraction as its (guarded) fixpoint:

$$
\begin{array}{l}
\mathit{env} :: (\mathsf{Trace}\ \widehat{d}, \mathsf{Domain}\ \widehat{d}, \mathsf{Lat}\ \widehat{d}) \Rightarrow \mathsf{GC}\ (\mathsf{Pow}\ (\mathsf{D}\ (\mathsf{ByName}\ \mathsf{T}) \vdash^{\mathsf{na}}_{\mathbb{D}} \_))\ (\widehat{d} \vdash^{\mathsf{na}}_{\mathbb{D}} \_) \\
\mathit{env} = \mathit{repr}\ \beta\ \textbf{where}\ \beta\ (\mathsf{Step}\ (\mathsf{Lookup}\ x)\ (\mathcal{S}[\![e]\!]_{\rho})) = \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathcal{S}[\![e]\!]_{\beta \triangleleft \rho}) \\
\mathit{byName} :: (\mathsf{Trace}\ \widehat{d}, \mathsf{Domain}\ \widehat{d}, \mathsf{Lat}\ \widehat{d}) \Rightarrow \mathsf{GC}\ (\mathsf{Pow}\ (\mathsf{D}\ (\mathsf{ByName}\ \mathsf{T})))\ \widehat{d} \\
\mathit{byName} = (\alpha_{\mathbb{T}} \circ \mathit{unByName}^*) \rightleftharpoons (\mathsf{ByName}^* \circ \gamma_{\mathbb{T}})\ \textbf{where}\ \alpha_{\mathbb{T}} \rightleftharpoons \gamma_{\mathbb{T}} = \mathit{trace}\ \mathit{byName}\ \mathit{env}
\end{array}
$$

There is a need to clear up the domain and range of *env*. Since its domain is sets of elements from $\mathsf{D}\ (\mathsf{ByName}\ \mathsf{T}) \vdash^{\mathsf{na}}_{\mathbb{D}} \_$, its range $d \vdash^{\mathsf{na}}_{\mathbb{D}} \_$ is the (possibly infinite) join over abstracted elements that look like $\mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathcal{S}[\![e]\!]_{\beta \triangleleft \rho})$ for some "closure" x, e, ρ. Although we have "sworn off" operational semantics for abstraction, we defunctionalise environments into syntax to structure the vast semantic domain in this way, thus working around the full abstraction problem [Plotkin 1977]. More formally,

Definition 42 (Syntactic by-name environments)**.** *Let* $\widehat{\mathsf{D}}$ *be a domain satisfying* Trace, Domain *and* Lat*. We write* $\widehat{\mathsf{D}} \vdash^{\mathsf{na}}_{\mathbb{D}} d$ *(resp.* $\widehat{\mathsf{D}} \vdash^{\mathsf{na}}_{\mathbb{E}} \rho$*) to say that the denotation* d *(resp. environment* ρ*) is* syntactic, *which we define by mutual guarded recursion as*

- $\widehat{\mathsf{D}} \vdash^{\mathsf{na}}_{\mathbb{D}} d$ *iff there exists a set* Clo *of syntactic closures such that* $d = \bigsqcup\{\mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathcal{S}[\![e]\!]_{\rho_1} :: \widehat{\mathsf{D}}) \mid (x, e, \rho_1) \in \mathit{Clo} \wedge \blacktriangleright(\widehat{\mathsf{D}} \vdash^{\mathsf{na}}_{\mathbb{E}} \rho_1)\}$*, and*
- $\widehat{\mathsf{D}} \vdash^{\mathsf{na}}_{\mathbb{E}} \rho$ *iff for all* x, $\widehat{\mathsf{D}} \vdash^{\mathsf{na}}_{\mathbb{D}} (\rho\ !\ x)$.

For the remainder of this subsection, we assume a refined definition of Domain and HasBind that expects $\widehat{\mathsf{D}} \vdash^{\mathsf{na}}_{\mathbb{D}} \_$ (denoting the set of $\widehat{d} :: \widehat{\mathsf{D}}$ such that $\widehat{\mathsf{D}} \vdash^{\mathsf{na}}_{\mathbb{D}} \widehat{d}$) where we pass around denotations that end up in an environment. It is then easy to see that $\mathcal{S}[\![e]\!]_{\rho}$ preserves $\widehat{\mathsf{D}} \vdash^{\mathsf{na}}_{\mathbb{E}} \_$ in recursive invocations, and *trace* does so as well.

[34] Recall that *fun* actually takes $x :: \mathsf{Name}$ as the first argument as a cheap De Bruijn level. Every call to *fun* would need to chose a fresh x. We omit the bookkeeping here; an alternative would be to require the implementation of usage analysis/D_{U} to track their own De Bruijn levels.

Lemma 43 (By-name evaluation preserves trace abstraction). *Let $\widehat{\mathsf{D}}$ be a domain with instances for* Trace, Domain, $\mathsf{HasBind}$ *and* Lat, *satisfying the soundness properties* Step-App, Step-Sel, Beta-App, Beta-Sel, Bind-ByName *in Figure 13.*

If $\mathcal{S}_{\mathbf{name}}[\![e]\!]_{\rho_1} = \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{name}}[\![v]\!]_{\rho_2})$ *in the concrete, then* $\overline{step\ ev}\ (\mathcal{S}[\![v]\!]_{\alpha_{\mathbb{E}} \triangleleft \{\_\} \triangleleft \rho_2}) \sqsubseteq \mathcal{S}[\![e]\!]_{\alpha_{\mathbb{E}} \triangleleft \{\_\} \triangleleft \rho_1}$ *in the abstract, where* $\alpha_{\mathbb{E}} \rightleftharpoons \gamma_{\mathbb{E}} = env$.

Proof. By Löb induction and cases on e, using the representation function $\beta_{\mathbb{E}} \triangleq \alpha_{\mathbb{E}} \circ \{\_\}$.

- **Case** $\mathsf{Var}\ x$: By assumption, we know that $\mathcal{S}_{\mathbf{name}}[\![x]\!]_{\rho_1} = \mathsf{Step}\ (\mathsf{Lookup}\ y)\ (\mathcal{S}_{\mathbf{name}}[\![e']\!]_{\rho_3}) = \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{name}}[\![v]\!]_{\rho_2})$ for some y, e', ρ_3, so that $\overline{ev} = \mathsf{Lookup}\ y : \overline{ev_1}$ for some ev_1 by determinism.

$$
\begin{array}{cl}
 & \overline{step\ ev}\ (\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}} \triangleleft \rho_2}) \\
= & \quad \wr\ \overline{ev} = \mathsf{Lookup}\ y : \overline{ev_1}\ \int \\
 & step\ (\mathsf{Lookup}\ y)\ (\overline{step\ ev_1}\ (\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}} \triangleleft \rho_2})) \\
\sqsubseteq & \quad \wr\ \text{Induction hypothesis at } ev_1, \rho_3 \text{ as above } \int \\
 & step\ (\mathsf{Lookup}\ y)\ (\mathcal{S}[\![e']\!]_{\beta_{\mathbb{E}} \triangleleft \rho_3}) \\
= & \quad \wr\ \text{Refold } \beta_{\mathbb{E}}, \rho_3 \,!\, x\ \int \\
 & \beta_{\mathbb{E}}\ (\rho_1 \,!\, x) \\
= & \quad \wr\ \text{Refold } \mathcal{S}[\![x]\!]_{\beta_{\mathbb{E}} \triangleleft \rho_1}\ \int \\
 & \mathcal{S}[\![x]\!]_{\beta_{\mathbb{E}} \triangleleft \rho_1}
\end{array}
$$

- **Case** Lam, ConApp: By reflexivity of $\sqsubseteq$.
- **Case** $\mathsf{App}\ e\ x$: Then $\mathcal{S}_{\mathbf{name}}[\![e]\!]_{\rho_1} = \overline{\mathsf{Step}\ ev_1}\ (\mathcal{S}_{\mathbf{name}}[\![\mathsf{Lam}\ y\ body]\!]_{\rho_3}), \mathcal{S}_{\mathbf{name}}[\![body]\!]_{\rho_3[y \mapsto \rho_1 ! x]} = \overline{\mathsf{Step}\ ev_2}\ (\mathcal{S}_{\mathbf{name}}[\![v]\!]_{\rho_2})$.

$$
\begin{array}{cl}
 & \overline{step\ ev}\ (\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}} \triangleleft \rho_2}) \\
= & \quad \wr\ \overline{ev} = [\mathsf{App}_1] + \overline{ev_1} + [\mathsf{App}_2] + \overline{ev_2}, \text{IH at } ev_2\ \int \\
 & step\ \mathsf{App}_1\ (\overline{step\ ev_1}\ (step\ \mathsf{App}_2\ (\mathcal{S}[\![body]\!]_{(\beta_{\mathbb{E}} \triangleleft \rho_3)[y \mapsto \beta_{\mathbb{E}} \triangleleft \rho_1 ! x]}))) \\
\sqsubseteq & \quad \wr\ \text{Assumption Beta-App}\ \int \\
 & step\ \mathsf{App}_1\ (\overline{step\ ev_1}\ (apply\ (\mathcal{S}[\![\mathsf{Lam}\ y\ body]\!]_{\beta_{\mathbb{E}} \triangleleft \rho_3})\ (\beta_{\mathbb{E}} \triangleleft \rho_1 \,!\, x))) \\
\sqsubseteq & \quad \wr\ \text{Assumption Step-App}\ \int \\
 & step\ \mathsf{App}_1\ (apply\ (\overline{step\ ev_1}\ (\mathcal{S}[\![\mathsf{Lam}\ y\ body]\!]_{\beta_{\mathbb{E}} \triangleleft \rho_3}))\ (\beta_{\mathbb{E}} \triangleleft \rho_1 \,!\, x)) \\
\sqsubseteq & \quad \wr\ \text{Induction hypothesis at } ev_1\ \int \\
 & step\ \mathsf{App}_1\ (apply\ (\mathcal{S}[\![e]\!]_{\beta_{\mathbb{E}} \triangleleft \rho_1})\ (\beta_{\mathbb{E}} \triangleleft \rho_1 \,!\, x)) \\
= & \quad \wr\ \text{Refold } \mathcal{S}[\![\mathsf{App}\ e\ x]\!]_{\beta_{\mathbb{E}} \triangleleft \rho_1}\ \int \\
 & \mathcal{S}[\![\mathsf{App}\ e\ x]\!]_{\beta_{\mathbb{E}} \triangleleft \rho_1}
\end{array}
$$

- **Case** $\mathsf{Case}\ e\ alts$: Then $\mathcal{S}_{\mathbf{name}}[\![e]\!]_{\rho_1} = \overline{\mathsf{Step}\ ev_1}\ (\mathcal{S}_{\mathbf{name}}[\![\mathsf{ConApp}\ k\ ys]\!]_{\rho_3}), \mathcal{S}_{\mathbf{name}}[\![e_r]\!]_{\rho_1[\overline{xs \mapsto map\ (\rho_3 \,!)\ ys}]} = \overline{\mathsf{Step}\ ev_2}\ (\mathcal{S}_{\mathbf{name}}[\![v]\!]_{\rho_2})$, where $alts \,!\, k = (xs, e_r)$ is the matching RHS.

$$
\begin{array}{cl}
 & \overline{step\ ev}\ (\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}} \triangleleft \rho_2}) \\
\sqsubseteq & \quad \wr\ \overline{ev} = [\mathsf{Case}_1] + \overline{ev_1} + [\mathsf{Case}_2] + ev_2, \text{IH at } ev_2\ \int \\
 & step\ \mathsf{Case}_1\ (\overline{step\ ev_1}\ (step\ \mathsf{Case}_2\ (\mathcal{S}[\![e_r]\!]_{\beta_{\mathbb{E}} \triangleleft \rho_1[\overline{xs \mapsto map\ (\widehat{\rho_3} \,!)\ ys}]}))) \\
\sqsubseteq & \quad \wr\ \text{Assumption Beta-Sel}\ \int \\
 & step\ \mathsf{Case}_1\ (\overline{step\ ev_1}\ (select\ (\mathcal{S}[\![\mathsf{ConApp}\ k\ ys]\!]_{\beta_{\mathbb{E}} \triangleleft \rho_3})\ (cont \triangleleft alts))) \\
\sqsubseteq & \quad \wr\ \text{Assumption Step-Sel}\ \int \\
 & step\ \mathsf{Case}_1\ (select\ (\overline{step\ ev_1}\ (\mathcal{S}[\![\mathsf{ConApp}\ k\ ys]\!]_{\beta_{\mathbb{E}} \triangleleft \rho_3}))\ (cont \triangleleft alts)) \\
\sqsubseteq & \quad \wr\ \text{Induction hypothesis at } ev_1\ \int
\end{array}
$$

$$
\begin{aligned}
& \mathit{step}\ \mathsf{Case}_1\ (\mathit{select}\ (\mathcal{S}\llbracket e \rrbracket_{\beta_{\mathbb{E}} \lhd \rho_1})\ (\mathit{cont} \lhd \mathit{alts})) \\
= \quad & \wr\ \text{Refold } \mathcal{S}\llbracket \mathsf{Case}\ e\ \mathit{alts} \rrbracket_{\beta_{\mathbb{E}} \lhd \rho_1}\ \int \\
& \mathcal{S}\llbracket \mathsf{Case}\ e\ \mathit{alts} \rrbracket_{\beta_{\mathbb{E}} \lhd \rho_1}
\end{aligned}
$$

- **Case** $\mathsf{Let}\ x\ e_1\ e_2$: We make good use of Lemma 41 below:

$$
\begin{aligned}
& \overline{\mathit{step}\ \mathit{ev}}\ (\mathcal{S}\llbracket v \rrbracket_{\beta_{\mathbb{E}} \lhd \rho_2}) \\
= \quad & \wr\ \overline{\mathit{ev}} = \mathsf{Let}_1 : \overline{\mathit{ev}_1}\ \int \\
& \mathit{step}\ \mathsf{Let}_1\ (\overline{\mathit{step}\ \mathit{ev}_1}\ (\mathcal{S}\llbracket v \rrbracket_{\beta_{\mathbb{E}} \lhd \rho_2})) \\
\sqsubseteq \quad & \wr\ \text{Induction hypothesis at } \mathit{ev}_1\ \int \\
& \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}\llbracket e_2 \rrbracket_{(\beta_{\mathbb{E}} \lhd \rho_1)[x \mapsto \beta_{\mathbb{E}}\ (\mathsf{Step}\ (\mathsf{Lookup}\ x)\ (\mathit{fix}\ (\lambda d_1 \to \mathcal{S}_{\mathbf{name}}\llbracket e_1 \rrbracket_{\rho_1[x \mapsto \mathsf{Step}\ (\mathsf{Lookup}\ x)\ d_1]})))]}) \\
= \quad & \wr\ \text{Partially roll } \mathit{fix}\ \int \\
& \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}\llbracket e_2 \rrbracket_{(\beta_{\mathbb{E}} \lhd \rho_1)[x \mapsto \beta_{\mathbb{E}}\ (\mathit{fix}\ (\lambda d_1 \to \mathsf{Step}\ (\mathsf{Lookup}\ x)\ (\mathcal{S}_{\mathbf{name}}\llbracket e_1 \rrbracket_{\rho_1[x \mapsto d_1]})))]}) \\
\sqsubseteq \quad & \wr\ \text{Lemma 41}\ \int \\
& \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}\llbracket e_2 \rrbracket_{(\beta_{\mathbb{E}} \lhd \rho_1)[x \mapsto \mathit{lfp}\ (\lambda \widehat{d_1} \to \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathcal{S}\llbracket e_1 \rrbracket_{(\beta_{\mathbb{E}} \lhd \rho_1)[x \mapsto \alpha_{\mathbb{E}}\ (\gamma_{\mathbb{E}}\ \widehat{d_1})]}))]}) \\
\sqsubseteq \quad & \wr\ \alpha_{\mathbb{E}} \circ \gamma_{\mathbb{E}} \sqsubseteq \mathit{id}\ \int \\
& \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}\llbracket e_2 \rrbracket_{(\beta_{\mathbb{E}} \lhd \rho_1)[x \mapsto \mathit{lfp}\ (\lambda \widehat{d_1} \to \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathcal{S}\llbracket e_1 \rrbracket_{(\beta_{\mathbb{E}} \lhd \rho_1)[x \mapsto \widehat{d_1}]}))]}) \\
= \quad & \wr\ \text{Partially unroll } \mathit{lfp}\ \int \\
& \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}\llbracket e_2 \rrbracket_{(\beta_{\mathbb{E}} \lhd \rho_1)[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathit{lfp}\ (\lambda \widehat{d_1} \to \mathcal{S}\llbracket e_1 \rrbracket_{(\beta_{\mathbb{E}} \lhd \rho_1)[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ \widehat{d_1}]}))]}) \\
\sqsubseteq \quad & \wr\ \text{Assumption } \textsc{Bind-ByName}\ \int \\
& \mathit{bind}\ (\lambda \widehat{d_1} \to \mathcal{S}\llbracket e_1 \rrbracket_{((\beta_{\mathbb{E}} \lhd \rho_1))[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ \widehat{d_1}]}) \\
& \qquad (\lambda \widehat{d_1} \to \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}\llbracket e_2 \rrbracket_{((\beta_{\mathbb{E}} \lhd \rho_1))[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ \widehat{d_1}]})) \\
= \quad & \wr\ \text{Refold } \mathcal{S}\llbracket \mathsf{Let}\ x\ e_1\ e_2 \rrbracket_{\beta_{\mathbb{E}} \lhd \rho_1}\ \int \\
& \mathcal{S}\llbracket \mathsf{Let}\ x\ e_1\ e_2 \rrbracket_{\beta_{\mathbb{E}} \lhd \rho_1}
\end{aligned}
$$

□

We can now prove the by-name abstraction theorem:

Theorem 44 (Sound By-name Interpretation). *Let $\widehat{\mathsf{D}}$ be a domain with instances for* Trace, Domain, HasBind *and* Lat, *and let $\alpha_{\mathbb{T}} \rightleftharpoons \gamma_{\mathbb{T}} \triangleq \mathit{byName}$, $\alpha_{\mathbb{E}} \rightleftharpoons \gamma_{\mathbb{E}} \triangleq \mathit{env}$. If the by-name abstraction laws in Figure 13 hold, then $\mathcal{S}\llbracket \_ \rrbracket_{\_}$ instantiates to an abstract interpreter that is sound wrt. $\gamma_{\mathbb{E}} \to \alpha_{\mathbb{T}}$, that is,*

$$\alpha_{\mathbb{T}}\ (\{\mathcal{S}_{\mathbf{name}}\llbracket e \rrbracket_{\rho}\} :: \mathsf{Pow}\ (\mathsf{D}\ (\mathsf{ByName}\ \mathsf{T}))) \sqsubseteq \mathcal{S}_{\widehat{\mathsf{D}}}\llbracket e \rrbracket_{\alpha_{\mathbb{E}} \lhd \{\_\} \lhd \rho}.$$

Proof. We first restate the goal in terms of the *rep*resentation functions $\beta_{\mathbb{T}} \triangleq \alpha_{\mathbb{T}} \circ \{\_\}$ and $\beta_{\mathbb{E}} \triangleq \alpha_{\mathbb{E}} \circ \{\_\}$:

$$\forall \rho.\ \beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}\llbracket e \rrbracket_{\rho}) \sqsubseteq (\mathcal{S}_{\widehat{\mathsf{D}}}\llbracket e \rrbracket_{\beta_{\mathbb{E}} \lhd \rho}).$$

We will prove this goal by Löb induction and cases on e.

- **Case** $\mathsf{Var}\ x$: The stuck case follows by unfolding $\alpha_{\mathbb{T}}$. Otherwise,

$$
\begin{aligned}
& \beta_{\mathbb{T}}\ (\rho\ !\ x) \\
= \quad & \wr\ \mathsf{Pow}\ (\mathsf{D}\ (\mathsf{ByName}\ \mathsf{T})) \vdash^{\mathsf{na}}_{\mathbb{E}} \{\_\} \lhd \rho, \text{Unfold } \beta_{\mathbb{T}}\ \int \\
& \mathit{step}\ (\mathsf{Lookup}\ y)\ (\beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}\llbracket e' \rrbracket_{\rho'})) \\
\sqsubseteq \quad & \wr\ \text{Induction hypothesis}\ \int \\
& \mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathcal{S}\llbracket e' \rrbracket_{\beta_{\mathbb{E}} \lhd \rho'}) \\
= \quad & \wr\ \text{Refold } \beta_{\mathbb{E}}\ \int \\
& \beta_{\mathbb{E}}\ (\rho\ !\ x)
\end{aligned}
$$

- **Case** $\mathsf{Lam}\ x\ body$:

$$\begin{array}{ll}
 & \beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}[\![\mathsf{Lam}\ x\ body]\!]_{\rho}) \\
= & \{\!\!\{\ \text{Unfold } \mathcal{S}[\![_]\!]_{_},\ \beta_{\mathbb{T}}\ \}\!\!\} \\
 & \mathit{fun}\ (\lambda\widehat{d} \to \bigsqcup\{\mathit{step}\ \mathsf{App}_2\ (\beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}[\![body]\!]_{\rho[x\mapsto d]}))\mid \beta_{\mathbb{E}}\ d \sqsubseteq \widehat{d}\}) \\
\sqsubseteq & \{\!\!\{\ \text{Induction hypothesis}\ \}\!\!\} \\
 & \mathit{fun}\ (\lambda\widehat{d} \to \bigsqcup\{\mathit{step}\ \mathsf{App}_2\ (\mathcal{S}[\![body]\!]_{\beta_{\mathbb{E}}\triangleleft\rho[x\mapsto d]})\mid \beta_{\mathbb{E}}\ d \sqsubseteq \widehat{d}\}) \\
\sqsubseteq & \{\!\!\{\ \text{Least upper bound / } \alpha_{\mathbb{E}}\circ\gamma_{\mathbb{E}}\sqsubseteq \mathit{id}\ \}\!\!\} \\
 & \mathit{fun}\ (\lambda\widehat{d} \to \mathit{step}\ \mathsf{App}_2\ (\mathcal{S}[\![body]\!]_{((\beta_{\mathbb{E}}\triangleleft\rho))[x\mapsto\widehat{d}]})) \\
= & \{\!\!\{\ \text{Refold } \mathcal{S}[\![_]\!]_{_}\ \}\!\!\} \\
 & \mathcal{S}[\![\mathsf{Lam}\ x\ body]\!]_{\beta_{\mathbb{E}}\triangleleft\rho}
\end{array}$$

- **Case** $\mathsf{ConApp}\ k\ ds$:

$$\begin{array}{ll}
 & \beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}[\![\mathsf{ConApp}\ k\ xs]\!]_{\rho}) \\
= & \{\!\!\{\ \text{Unfold } \mathcal{S}[\![_]\!]_{_},\ \beta_{\mathbb{T}}\ \}\!\!\} \\
 & \mathit{con}\ k\ (\mathit{map}\ ((\beta_{\mathbb{E}}\triangleleft\rho)\,!)\ xs) \\
= & \{\!\!\{\ \text{Refold } \mathcal{S}[\![_]\!]_{_}\ \}\!\!\} \\
 & \mathcal{S}[\![\mathsf{Lam}\ x\ body]\!]_{\beta_{\mathbb{E}}\triangleleft\rho}
\end{array}$$

- **Case** $\mathsf{App}\ e\ x$: The stuck case follows by unfolding $\beta_{\mathbb{T}}$.
Our proof obligation can be simplified as follows

$$\begin{array}{ll}
 & \beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}[\![\mathsf{App}\ e\ x]\!]_{\rho}) \\
= & \{\!\!\{\ \text{Unfold } \mathcal{S}[\![_]\!]_{_},\ \beta_{\mathbb{T}}\ \}\!\!\} \\
 & \mathit{step}\ \mathsf{App}_1\ (\beta_{\mathbb{T}}\ (\mathit{apply}\ (\mathcal{S}_{\mathbf{name}}[\![e]\!]_{\rho})\ (\rho\,!\,x))) \\
= & \{\!\!\{\ \text{Unfold } \mathit{apply}\ \}\!\!\} \\
 & \mathit{step}\ \mathsf{App}_1\ (\beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}[\![e]\!]_{\rho} \gg\!= \lambda\mathbf{case}\ \mathsf{Fun}\ f \to f\ (\rho\,!\,x);\ _ \to \mathit{stuck})) \\
\sqsubseteq & \{\!\!\{\ \text{By cases, see below}\ \}\!\!\} \\
 & \mathit{step}\ \mathsf{App}_1\ (\mathit{apply}\ (\mathcal{S}[\![e]\!]_{\beta_{\mathbb{E}}\triangleleft\rho})\ ((\beta_{\mathbb{E}}\triangleleft\rho)\,!\,x)) \\
= & \{\!\!\{\ \text{Refold } \mathcal{S}[\![_]\!]_{_}\ \}\!\!\} \\
 & \mathcal{S}[\![\mathsf{App}\ e\ x]\!]_{\beta_{\mathbb{E}}\triangleleft\rho}
\end{array}$$

When $\mathcal{S}_{\mathbf{name}}[\![e]\!]_{\rho}$ diverges, we have

$$\begin{array}{ll}
= & \{\!\!\{\ \mathcal{S}_{\mathbf{name}}[\![e]\!]_{\rho} \text{ diverges, unfold } \beta_{\mathbb{T}}\ \}\!\!\} \\
 & \mathit{step}\ \mathit{ev}_1\ (\mathit{step}\ \mathit{ev}_2\ (...)) \\
\sqsubseteq & \{\!\!\{\ \text{Assumption } \textsc{Step-App}\ \}\!\!\} \\
 & \mathit{apply}\ (\mathit{step}\ \mathit{ev}_1\ (\mathit{step}\ \mathit{ev}_2\ (...)))\ ((\beta_{\mathbb{E}}\triangleleft\rho)\,!\,x) \\
= & \{\!\!\{\ \text{Refold } \beta_{\mathbb{T}},\ \mathcal{S}_{\mathbf{name}}[\![e]\!]_{\rho}\ \}\!\!\} \\
 & \mathit{apply}\ (\beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}[\![e]\!]_{\rho}))\ ((\beta_{\mathbb{E}}\triangleleft\rho)\,!\,x) \\
\sqsubseteq & \{\!\!\{\ \text{Induction hypothesis}\ \}\!\!\} \\
 & \mathit{apply}\ (\mathcal{S}[\![e]\!]_{\beta_{\mathbb{E}}\triangleleft\rho})\ ((\beta_{\mathbb{E}}\triangleleft\rho)\,!\,x)
\end{array}$$

Otherwise, $\mathcal{S}_{\mathbf{name}}[\![e]\!]_{\rho}$ must produce a value v. If $v = \mathsf{Stuck}$ or $v = \mathsf{Con}\ k\ ds$, we set $d \triangleq \mathit{stuck}$ (resp. $d \triangleq \mathit{con}\ k\ (\mathit{map}\ \beta_{\mathbb{E}}\ ds)$) and have

$$\begin{array}{ll}
 & \beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}[\![e]\!]_{\rho} \gg\!= \lambda\mathbf{case}\ \mathsf{Fun}\ f \to f\ (\rho\,!\,x);\ _ \to \mathit{stuck}) \\
= & \{\!\!\{\ \mathcal{S}_{\mathbf{name}}[\![e]\!]_{\rho} = \overline{\mathsf{Step}\ \mathit{ev}}\ (\mathit{return}\ v),\ \text{unfold } \beta_{\mathbb{T}}\ \}\!\!\} \\
 & \overline{\mathit{step}\ \mathit{ev}}\ (\beta_{\mathbb{T}}\ (\mathit{return}\ v \gg\!= \lambda\mathbf{case}\ \mathsf{Fun}\ f \to f\ (\rho\,!\,x);\ _ \to \mathit{stuck}))
\end{array}$$

$$
\begin{array}{ll}
= & \wr\ v \text{ not } \mathsf{Fun}\text{, unfold } \beta_{\mathbb{T}}\ \int \\
& \overline{\mathit{step}\ \mathit{ev}}\ \mathit{stuck} \\
\sqsubseteq & \wr\ \text{Assumptions } \textsc{Unwind-Stuck}, \textsc{Intro-Stuck} \text{ where } d \triangleq \mathit{stuck} \text{ or } d \triangleq \mathit{con}\ k\ (\mathit{map}\ \beta_{\mathbb{T}}\ \mathit{ds})\ \int \\
& \overline{\mathit{step}\ \mathit{ev}}\ (\mathit{apply}\ d\ a) \\
\sqsubseteq & \wr\ \text{Assumption } \textsc{Step-App}\ \int \\
& \mathit{apply}\ (\overline{\mathit{step}\ \mathit{ev}}\ d)\ ((\beta_{\mathbb{E}} \lhd \rho)\ !\ x) \\
= & \wr\ \text{Refold } \beta_{\mathbb{T}}, \mathcal{S}_{\mathbf{name}}\llbracket e \rrbracket_{\rho}\ \int \\
& \mathit{apply}\ (\beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}\llbracket e \rrbracket_{\rho}))\ ((\beta_{\mathbb{E}} \lhd \rho)\ !\ x) \\
\sqsubseteq & \wr\ \text{Induction hypothesis}\ \int \\
& \mathit{apply}\ (\mathcal{S}\llbracket e \rrbracket_{\beta_{\mathbb{E}} \lhd \rho})\ ((\beta_{\mathbb{E}} \lhd \rho)\ !\ x)
\end{array}
$$

In the final case, we have $v = \mathsf{Fun}\ f$, which must be the result of some call $\mathcal{S}_{\mathbf{name}}\llbracket \mathsf{Lam}\ y\ \mathit{body} \rrbracket_{\rho_1}$; hence $f \triangleq \lambda d \rightarrow \mathsf{Step}\ \mathsf{App}_2\ (\mathcal{S}_{\mathbf{name}}\llbracket \mathit{body} \rrbracket_{\rho_1[y \mapsto d]})$.

$$
\begin{array}{ll}
& \beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}\llbracket e \rrbracket_{\rho} \ggg \lambda\mathbf{case}\ \mathsf{Fun}\ f \rightarrow f\ (\rho\ !\ x); \_ \rightarrow \mathit{stuck}) \\
= & \wr\ \mathcal{S}_{\mathbf{name}}\llbracket e \rrbracket_{\rho} = \overline{\mathsf{Step}\ \mathit{ev}}\ (\mathit{return}\ v)\text{, unfold } \beta_{\mathbb{T}}\ \int \\
& \overline{\mathit{step}\ \mathit{ev}}\ (\beta_{\mathbb{T}}\ (\mathit{return}\ v \ggg \lambda\mathbf{case}\ \mathsf{Fun}\ f \rightarrow f\ (\rho\ !\ x); \_ \rightarrow \mathit{stuck})) \\
= & \wr\ v = \mathsf{Fun}\ f\text{, with } f \text{ as above; unfold } \beta_{\mathbb{T}}\ \int \\
& \overline{\mathit{step}\ \mathit{ev}}\ (\mathit{step}\ \mathsf{App}_2\ (\beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}\llbracket \mathit{body} \rrbracket_{\rho_1[y \mapsto \rho\ !\ x]}))) \\
\sqsubseteq & \wr\ \text{Induction hypothesis}\ \int \\
& \overline{\mathit{step}\ \mathit{ev}}\ (\mathit{step}\ \mathsf{App}_2\ (\mathcal{S}\llbracket \mathit{body} \rrbracket_{\beta_{\mathbb{E}} \lhd \rho_1[y \mapsto \rho\ !\ x]})) \\
= & \wr\ \text{Rearrange}\ \int \\
& \overline{\mathit{step}\ \mathit{ev}}\ (\mathit{step}\ \mathsf{App}_2\ (\mathcal{S}\llbracket \mathit{body} \rrbracket_{(\beta_{\mathbb{E}} \lhd \rho_1)[y \mapsto (\beta_{\mathbb{E}} \lhd \rho)\ !\ x]})) \\
\sqsubseteq & \wr\ \text{Assumption } \textsc{Beta-App}\ \int \\
& \overline{\mathit{step}\ \mathit{ev}}\ (\mathit{apply}\ (\mathcal{S}\llbracket \mathsf{Lam}\ y\ \mathit{body} \rrbracket_{\beta_{\mathbb{E}} \lhd \rho_1})\ ((\beta_{\mathbb{E}} \lhd \rho)\ !\ x)) \\
\sqsubseteq & \wr\ \text{Assumption } \textsc{Step-App}\ \int \\
& \mathit{apply}\ (\overline{\mathit{step}\ \mathit{ev}}\ (\mathcal{S}\llbracket \mathsf{Lam}\ y\ \mathit{body} \rrbracket_{\beta_{\mathbb{E}} \lhd \rho_1}))\ ((\beta_{\mathbb{E}} \lhd \rho)\ !\ x) \\
\sqsubseteq & \wr\ \text{Lemma 43 applied to } \overline{\mathit{ev}}\ \int \\
& \mathit{apply}\ (\mathcal{S}\llbracket e \rrbracket_{\beta_{\mathbb{E}} \lhd \rho})\ ((\beta_{\mathbb{E}} \lhd \rho)\ !\ x)
\end{array}
$$

- **Case** $\mathsf{Case}\ e\ \mathit{alts}$: The stuck case follows by unfolding $\beta_{\mathbb{T}}$. When $\mathcal{S}_{\mathbf{name}}\llbracket e \rrbracket_{\rho}$ diverges or does not evaluate to $\mathcal{S}_{\mathbf{name}}\llbracket \mathsf{ConApp}\ k\ \mathit{ys} \rrbracket_{\rho_1}$, the reasoning is similar to $\mathsf{App}\ e\ x$, but in a *select* context. So assume that $\mathcal{S}_{\mathbf{name}}\llbracket e \rrbracket_{\rho} = \overline{\mathsf{Step}\ \mathit{ev}}\ (\mathcal{S}_{\mathbf{name}}\llbracket \mathsf{ConApp}\ k\ \mathit{ys} \rrbracket_{\rho_1})$ and that there exists $((\mathit{cont} \lhd \mathit{alts})\ !\ k)\ \mathit{ds} = \mathsf{Step}\ \mathsf{Case}_2\ (\mathcal{S}_{\mathbf{name}}\llbracket e_r \rrbracket_{\rho[\overline{\mathit{xs} \mapsto \mathit{ds}}]})$.

$$
\begin{array}{ll}
& \beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}\llbracket \mathsf{Case}\ e\ \mathit{alts} \rrbracket_{\rho}) \\
= & \wr\ \text{Unfold } \mathcal{S}\llbracket \_ \rrbracket_{\_}, \beta_{\mathbb{T}}\ \int \\
& \mathit{step}\ \mathsf{Case}_1\ (\beta_{\mathbb{T}}\ (\mathit{select}\ (\mathcal{S}_{\mathbf{name}}\llbracket e \rrbracket_{\rho})\ (\mathit{cont} \lhd \mathit{alts}))) \\
= & \wr\ \text{Unfold } \mathit{select}\ \int \\
& \mathit{step}\ \mathsf{Case}_1\ (\beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}\llbracket e \rrbracket_{\rho} \ggg \lambda\mathbf{case}\ \mathsf{Con}\ k\ \mathit{ds} \mid k \in \mathit{dom}\ \mathit{alts} \rightarrow ((\mathit{cont} \lhd \mathit{alts})\ !\ k)\ \mathit{ds})) \\
= & \wr\ \mathcal{S}_{\mathbf{name}}\llbracket e \rrbracket_{\rho} = \overline{\mathsf{Step}\ \mathit{ev}}\ (\mathcal{S}_{\mathbf{name}}\llbracket \mathsf{ConApp}\ k\ \mathit{ys} \rrbracket_{\rho_1})\text{, unfold } \beta_{\mathbb{T}}\ \int \\
& \mathit{step}\ \mathsf{Case}_1\ (\overline{\mathit{step}\ \mathit{ev}}\ (\beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}\llbracket \mathsf{ConApp}\ k\ \mathit{ys} \rrbracket_{\rho_1}) \ggg \lambda\mathbf{case}\ \mathsf{Con}\ k\ \mathit{ds} \mid k \in \mathit{dom}\ (\mathit{cont} \lhd \mathit{alts}) \rightarrow ((c \\
= & \wr\ \text{Simplify } \mathit{return}\ (\mathsf{Con}\ k\ \mathit{ds}) \ggg f = f\ (\mathsf{Con}\ k\ \mathit{ds}), (\mathit{cont} \lhd \mathit{alts})\ !\ k \text{ as above}\ \int \\
& \mathit{step}\ \mathsf{Case}_1\ (\overline{\mathit{step}\ \mathit{ev}}\ (\beta_{\mathbb{T}}\ (\mathsf{Step}\ \mathsf{Case}_2\ (\mathcal{S}_{\mathbf{name}}\llbracket e_r \rrbracket_{\rho[\overline{\mathit{xs} \mapsto \mathit{map}\ (\rho_1\ !)\ \mathit{ys}}]})))) \\
= & \wr\ \text{Unfold } \beta_{\mathbb{T}}\ \int \\
& \mathit{step}\ \mathsf{Case}_1\ (\overline{\mathit{step}\ \mathit{ev}}\ (\mathit{step}\ \mathsf{Case}_2\ (\beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}\llbracket e_r \rrbracket_{\rho[\overline{\mathit{xs} \mapsto \mathit{map}\ (\rho_1\ !)\ \mathit{ys}}]}))))
\end{array}
$$

$$
\begin{aligned}
\sqsubseteq\quad & \langle\ \text{Induction hypothesis}\ \rangle \\
& \mathit{step}\ \mathsf{Case}_1\ (\overline{\mathit{step}\ \mathit{ev}}\ (\mathit{step}\ \mathsf{Case}_2\ (\mathcal{S}\llbracket e_r \rrbracket_{(\beta_{\mathbb{E}} \lhd \rho)[\overline{xs \mapsto \mathit{map}\ ((\beta_{\mathbb{E}} \lhd \rho_1)\,!)\ ys}]}))) \\
=\quad & \langle\ \text{Refold } \mathit{cont}\ \rangle \\
& \mathit{step}\ \mathsf{Case}_1\ (\mathit{cont}\ (\mathit{alts}\,!\,k)\ (\mathit{map}\ ((\beta_{\mathbb{E}} \lhd \rho_1)\,!)\ \mathit{xs})) \\
\sqsubseteq\quad & \langle\ \text{Assumption } \textsc{Beta-Sel}\ \rangle \\
& \mathit{step}\ \mathsf{Case}_1\ (\overline{\mathit{step}\ \mathit{ev}}\ (\mathit{select}\ (\mathcal{S}\llbracket \mathsf{ConApp}\ k\ \mathit{ys} \rrbracket_{\beta_{\mathbb{E}} \lhd \rho_1})\ (\mathit{cont} \lhd \mathit{alts}))) \\
\sqsubseteq\quad & \langle\ \text{Assumption } \textsc{Step-Sel}\ \rangle \\
& \mathit{step}\ \mathsf{Case}_1\ (\mathit{select}\ (\overline{\mathit{step}\ \mathit{ev}}\ (\mathcal{S}\llbracket \mathsf{ConApp}\ k\ \mathit{ys} \rrbracket_{\beta_{\mathbb{E}} \lhd \rho_1}))\ (\mathit{cont} \lhd \mathit{alts})) \\
\sqsubseteq\quad & \langle\ \text{Lemma 43 applied to } \overline{\mathit{ev}}\ \rangle \\
& \mathit{step}\ \mathsf{Case}_1\ (\mathit{select}\ (\mathcal{S}\llbracket e \rrbracket_{\beta_{\mathbb{E}} \lhd \rho})\ (\mathit{cont} \lhd \mathit{alts})) \\
=\quad & \langle\ \text{Refold } \mathcal{S}\llbracket\_\rrbracket_{\_}\ \rangle \\
& \mathcal{S}\llbracket \mathsf{Case}\ e\ \mathit{alts} \rrbracket_{\beta_{\mathbb{E}} \lhd \rho}
\end{aligned}
$$

- **Case** $\mathsf{Let}\ x\ e_1\ e_2$:

$$
\begin{aligned}
& \beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}\llbracket \mathsf{Let}\ x\ e_1\ e_2 \rrbracket_{\rho}) \\
=\quad & \langle\ \text{Unfold } \mathcal{S}\llbracket\_\rrbracket_{\_}\ \rangle \\
& \beta_{\mathbb{T}}\ (\mathit{bind}\ (\lambda d_1 \to \mathcal{S}_{\mathbf{name}}\llbracket e_1 \rrbracket_{\rho[x \mapsto \mathsf{Step}\ (\mathsf{Lookup}\ x)\ d_1]}) \\
& \qquad\qquad (\lambda d_1 \to \mathsf{Step}\ \mathsf{Let}_1\ (\mathcal{S}_{\mathbf{name}}\llbracket e_2 \rrbracket_{\rho[x \mapsto \mathsf{Step}\ (\mathsf{Lookup}\ x)\ d_1]}))) \\
=\quad & \langle\ \text{Unfold } \mathit{bind}, \beta_{\mathbb{T}}\ \rangle \\
& \mathit{step}\ \mathsf{Let}_1\ (\beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{name}}\llbracket e_2 \rrbracket_{\rho[x \mapsto \mathsf{Step}\ (\mathsf{Lookup}\ x)\ (\mathit{fix}\ (\lambda d_1 \to \mathcal{S}_{\mathbf{name}}\llbracket e_1 \rrbracket_{\rho[x \mapsto \mathsf{Step}\ (\mathsf{Lookup}\ x)\ d_1]}))]})) \\
\sqsubseteq\quad & \langle\ \text{Induction hypothesis}\ \rangle \\
& \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}\llbracket e_2 \rrbracket_{(\beta_{\mathbb{E}} \lhd \rho)[x \mapsto \beta_{\mathbb{E}}\ (\mathsf{Step}\ (\mathsf{Lookup}\ x)\ (\mathit{fix}\ (\lambda d_1 \to \mathcal{S}_{\mathbf{name}}\llbracket e_1 \rrbracket_{\rho[x \mapsto \mathsf{Step}\ (\mathsf{Lookup}\ x)\ d_1]})))]})
\end{aligned}
$$

And from hereon, the proof is identical to the Let case of Lemma 43:

$$
\begin{aligned}
\sqsubseteq\quad & \langle\ \text{By Lemma 41, as in the proof for Lemma 43}\ \rangle \\
& \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}\llbracket e_2 \rrbracket_{(\beta_{\mathbb{E}} \lhd \rho)[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathit{lfp}\ (\lambda \widehat{d_1} \to \mathcal{S}\llbracket e_1 \rrbracket_{(\beta_{\mathbb{E}} \lhd \rho)[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ \widehat{d_1}]}))]}) \\
\sqsubseteq\quad & \langle\ \text{Assumption } \textsc{Bind-ByName}, \text{ with } \widehat{\rho} = \beta_{\mathbb{E}} \lhd \rho\ \rangle \\
& \mathit{bind}\ (\lambda d_1 \to \mathcal{S}\llbracket e_1 \rrbracket_{(\beta_{\mathbb{E}} \lhd \rho)[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ d_1]}) \\
& \qquad (\lambda d_1 \to \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}\llbracket e_2 \rrbracket_{(\beta_{\mathbb{E}} \lhd \rho)[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ d_1]})) \\
=\quad & \langle\ \text{Refold } \mathcal{S}\llbracket \mathsf{Let}\ x\ e_1\ e_2 \rrbracket_{\beta_{\mathbb{E}} \lhd \rho}\ \rangle \\
& \mathcal{S}\llbracket \mathsf{Let}\ x\ e_1\ e_2 \rrbracket_{\beta_{\mathbb{E}} \lhd \rho}
\end{aligned}
$$

□

We can now show a generalisation to open expressions of the by-name version of Lemma 9:

Lemma 45 ($\mathcal{S}_{\mathbf{usage}}\llbracket\_\rrbracket_{\_}$ abstracts $\mathcal{S}_{\mathbf{name}}\llbracket\_\rrbracket_{\_}$, open). *Usage analysis $\mathcal{S}_{\mathbf{usage}}\llbracket\_\rrbracket_{\_}$ is sound wrt. $\mathcal{S}_{\mathbf{name}}\llbracket\_\rrbracket_{\_}$, that is,*

$$\alpha_{\mathbb{T}}\ \{\mathcal{S}_{\mathbf{name}}\llbracket e \rrbracket_{\rho}\} \sqsubseteq (\mathcal{S}_{\mathbf{usage}}\llbracket e \rrbracket_{\alpha_{\mathbb{E}} \lhd \{\_\} \lhd \rho} :: \mathsf{D}_{\mathsf{U}})\ \textbf{where}\ \alpha_{\mathbb{T}} \rightleftharpoons \_ = \mathit{byName}; \alpha_{\mathbb{E}} \rightleftharpoons \_ = \mathit{env}.$$

Proof. By Theorem 44, it suffices to show the abstraction laws in Figure 13 as done in the proof for Lemma 9. □

The following example shows why we need syntactic premises in Figure 13. It defines a monotone, but non-syntactic $f :: \mathsf{D}_{\mathsf{U}} \to \mathsf{D}_{\mathsf{U}}$ for which $f\ a\ \not\sqsubseteq\ \mathit{apply}\ (\mathit{fun}\ x\ f)\ a$. So if we did not have the syntactic premises, we would not be able to prove usage analysis correct.

Example 46. *Let $z \neq x \neq y$. The monotone function f defined as follows*

$$
\begin{array}{l}
\mathit{freezeHeap} :: (\mathsf{Trace}\ \widehat{d}, \mathsf{Domain}\ \widehat{d}, \mathsf{Lat}\ \widehat{d}) \Rightarrow \vdash^{\mathsf{ne}}_{\mathbb{H}} \_ \to \mathsf{GC}\ (\vdash^{\mathsf{ne}}_{\mathbb{D}} \_)\ (\widehat{d} \vdash^{\mathsf{na}}_{\mathbb{D}} \_) \\
\mathit{freezeHeap}\ \mu = \mathit{repr}\ \beta\ \mathbf{where}\ \beta\ (\mathsf{Step}\ (\mathsf{Lookup}\ x)\ (\mathit{fetch}\ a))\ \mid\ \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho}) \leftarrow \mu\ !\ a \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad = \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathcal{S}[\![e]\!]_{\beta \triangleleft \rho}) \\
\\
\mathit{nameNeed} :: (\mathsf{Trace}\ \widehat{d}, \mathsf{Domain}\ \widehat{d}, \mathsf{Lat}\ \widehat{d}) \Rightarrow \mathsf{GC}\ (\mathsf{Pow}\ (\mathsf{T}\ (\mathsf{Value}\ (\mathsf{ByNeed}\ \mathsf{T}), \vdash^{\mathsf{ne}}_{\mathbb{H}} \_)))\ \widehat{d} \\
\mathit{nameNeed} = \mathit{repr}\ \beta\ \mathbf{where} \\
\quad \beta\ (\mathsf{Step}\ e\ d) \qquad\qquad\quad = \mathit{step}\ e\ (\beta\ d) \\
\quad \beta\ (\mathsf{Ret}\ (\mathsf{Stuck}, \mu)) \qquad\ \ = \mathit{stuck} \\
\quad \beta\ (\mathsf{Ret}\ (\mathsf{Fun}\ f, \mu)) \qquad = \mathit{fun}\ (\lambda \widehat{d} \to \bigsqcup\{\beta\ (f\ \ d\ \mu) \mid d \in \gamma_{\mathbb{E}}\ \widehat{d}\})\ \mathbf{where}\ \_ \ \rightleftharpoons \gamma_{\mathbb{E}} = \mathit{freezeHeap}\ \mu \\
\quad \beta\ (\mathsf{Ret}\ (\mathsf{Con}\ k\ ds, \mu)) = \mathit{con}\ k\ (\mathit{map}\ (\alpha_{\mathbb{E}} \circ \{\_\})\ ds) \qquad\qquad \mathbf{where}\ \alpha_{\mathbb{E}} \rightleftharpoons \_\ = \mathit{freezeHeap}\ \mu
\end{array}
$$

Fig. 18. Galois connection for sound by-name and by-need abstraction

$$
\begin{array}{l}
f :: \mathsf{D_U} \to \mathsf{D_U} \\
f\ \langle \varphi, \_ \rangle = \mathbf{if}\ \varphi\ !?\ y \sqsubseteq \mathsf{U}_0\ \mathbf{then}\ \langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_\omega \rangle\ \mathbf{else}\ \langle [z \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle
\end{array}
$$

violates $f\ \ a \sqsubseteq \mathit{apply}\ (\mathit{fun}\ x\ f)\ \ a$. *To see that, let* $a \triangleq \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle :: \mathsf{D_U}$ *and consider*

$$f\ \ a = \langle [z \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle\ \ \not\sqsubseteq \langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_\omega \rangle = \mathit{apply}\ (\mathit{fun}\ x\ f)\ \ a.$$

D.4 Abstract By-need Soundness, in Detail

Now that we have gained some familiarity with the proof framework while proving Theorem 44 correct, we will tackle the proof for Theorem 56, which is applicable for analyses that are sound both wrt. to by-name as well as by-need, such as usage analysis or perhaps type analysis in Appendix C.1 (we have however not proven it so).

A sound by-name analysis must only satisfy the two additional abstraction laws Step-Inc and Update in Figure 13 to yield sound results for by-need as well. These laws make intuitive sense, because Update events cannot be observed in a by-name trace and hence must be ignored. Other than Update steps, by-need evaluation makes fewer steps than by-name evaluation, so Step-Inc asserts that dropping steps never invalidates the result.

In order to formalise this intuition, we must find a Galois connection that does so, starting with its domain. Although in Section 4.3 we considered a $d :: \mathsf{D}\ (\mathsf{ByNeed}\ \mathsf{T})$ as an atomic denotation, such a denotation actually only makes sense when it travels together with an environment ρ that ties free variables to their addresses in the heap that d expects.

For our purposes, the key is that a by-need environment ρ and a heap μ can be "frozen" into a corresponding by-name environment. This operation forms a Galois connection *freezeHeap* in Figure 18, where $\vdash^{\mathsf{ne}}_{\mathbb{D}} \_$ serves a similar purpose as $\widehat{d} \vdash^{\mathsf{na}}_{\mathbb{D}} \_$ from Definition 42, restricting environment entries to the syntactic by-need form $\mathsf{Step}\ \ (\mathsf{Lookup}\ \ x)\ \ (\mathit{fetch}\ \ a)$ and heap entries in $\vdash^{\mathsf{ne}}_{\mathbb{H}} \_$ to $\mathit{memo}\ a\ (\mathcal{S}[\![e]\!]_{\rho})$.

Definition 47 (Syntactic by-need heaps and environments, address domain). *We write* $\vdash^{\mathsf{ne}}_{\mathbb{E}} \rho$ *(resp.* $\vdash^{\mathsf{ne}}_{\mathbb{H}} \mu$*) to say that the by-need environment* $\rho :: \mathsf{Name} :\rightharpoonup \mathsf{Pow}\ (\mathsf{D}\ (\mathsf{ByNeed}\ \mathsf{T}))$ *(resp. by-need heap* μ*) is* syntactic, *defined by mutual guarded recursion as*

- $\vdash^{\mathsf{ne}}_{\mathbb{D}} d$ *iff there exists a set* Clo *of syntactic closures such that* $d = \bigcup\{\mathsf{Step}\ (\mathsf{Lookup}\ x)\ (\mathit{fetch}\ a) \mid (x, a) \in \mathit{Clo}\}$.
- $\vdash^{\mathsf{ne}}_{\mathbb{E}} \rho$ *iff for all* x, $\vdash^{\mathsf{ne}}_{\mathbb{D}} \rho\ !\ x$.
- $\mathit{adom}\ d \triangleq \{a \mid \mathsf{Step}\ (\mathsf{Lookup}\ y)\ (\mathit{fetch}\ a) \in d\}$

$$\boxed{\mu_1 \rightsquigarrow \mu_2}$$

$$\rightsquigarrow\textsc{-Refl}\ \frac{\vdash^{\mathsf{ne}}_{\mathbb{H}} \mu}{\mu \rightsquigarrow \mu} \qquad \rightsquigarrow\textsc{-Trans}\ \frac{\mu_1 \rightsquigarrow \mu_2 \quad \mu_2 \rightsquigarrow \mu_3}{\mu_1 \rightsquigarrow \mu_3} \qquad \rightsquigarrow\textsc{-Ext}\ \frac{a \notin dom\ \mu \quad adom\ \rho \subseteq dom\ \mu \cup \{a\}}{\mu \rightsquigarrow \mu[a \mapsto memo\ a\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho})]}$$

$$\rightsquigarrow\textsc{-Memo}\ \frac{\mu_1 \mathbin{!} a = memo\ a\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}) \quad \blacktriangleright(\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}(\mu_1) = \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2)))}{\mu_1 \rightsquigarrow \mu_2[a \mapsto memo\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2})]}$$

Fig. 19. Heap progression relation

- $adom\ \rho \triangleq \bigcup\{adom\ (\rho \mathbin{!} x) \mid x \in dom\ \rho\}$.
- $\vdash^{\mathsf{ne}}_{\mathbb{H}} \mu$ *iff for all* a*, there is a set* Clo *of syntactic closures such that* $\mu \mathbin{!} a = \bigcup\{memo\ a\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho}) \mid \blacktriangleright((e,\rho) \in Clo \wedge \vdash^{\mathsf{ne}}_{\mathbb{E}} \rho \wedge adom\ \rho \subseteq dom\ \mu)\}$.

We refer to $adom\ d$ *(resp.* $adom\ \rho$*) as the* address domain *of* d *(resp.* ρ*).*

We assume that all concrete environments $\mathsf{Name} :\rightharpoonup \mathsf{D}\ (\mathsf{ByNeed}\ \mathsf{T})$ and heaps $\mathsf{Heap}\ (\mathsf{ByNeed}\ \mathsf{T})$ satisfy $\vdash^{\mathsf{ne}}_{\mathbb{E}}\ \_$ resp. $\vdash^{\mathsf{ne}}_{\mathbb{H}}\ \_$. It is easy to see that syntacticness is preserved by $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_{\_}(\_)$ whenever the environment or heap is extended, assuming that Domain and HasBind are adjusted accordingly.

The environment abstraction $\alpha_{\mathbb{E}}\ \mu \rightleftharpoons \_ = freezeHeap\ \mu$ improves the more "evaluated" μ is. E.g., when μ_1 *progresses* into μ_2 during evaluation, written $\mu_1 \rightsquigarrow \mu_2$, it is $\alpha_{\mathbb{E}}\ \mu_2\ d \sqsubseteq \alpha_{\mathbb{E}}\ \mu_1\ d$ for all d. The heap progression relation is formally defined (on syntactic heaps $\vdash^{\mathsf{ne}}_{\mathbb{H}}\ \_$) in Figure 19, and we will now work toward a proof for the approximation statement about $\alpha_{\mathbb{E}}$ in Lemma 54.

Transitivity and reflexivity of $(\rightsquigarrow)$ are definitional by rules $\rightsquigarrow$-Refl and $\rightsquigarrow$-Trans; antisymmetry is not so simple to show for a lack of full abstraction.

Corollary 48. *$(\rightsquigarrow)$ is a preorder.*

The remaining two rules express how a heap can be modified during by-need evaluation: Evaluation of a Let extends the heap via $\rightsquigarrow$-Ext and evaluation of a Var will memoise the evaluated heap entry, progressing it along $\rightsquigarrow$-Memo.

Lemma 49 (Evaluation progresses the heap). *If* $\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}(\mu_1) = \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))$*, then* $\mu_1 \rightsquigarrow \mu_2$.

Proof. By Löb induction and cases on e. Since there is no approximation yet, all occurring closure sets in $\vdash^{\mathsf{ne}}_{\mathbb{E}}\ \_$ are singletons.

- **Case** $\mathsf{Var}\ x$: Let $\overline{ev_1} \triangleq tail\ (init\ (\overline{ev}))$.

$$\begin{array}{ll} & (\rho_1 \mathbin{!} x)\ \mu_1 \\ = & \langle\!\langle \vdash^{\mathsf{ne}}_{\mathbb{E}} \rho_1, \text{some } y, a \rangle\!\rangle \\ & \mathsf{Step}\ (\mathsf{Lookup}\ y)\ (fetch\ a\ \mu_1) \\ = & \langle\!\langle \text{Unfold } fetch \rangle\!\rangle \\ & \mathsf{Step}\ (\mathsf{Lookup}\ y)\ ((\mu_1 \mathbin{!} a)\ \mu_1) \\ = & \langle\!\langle \vdash^{\mathsf{ne}}_{\mathbb{H}} \mu, \text{some } e, \rho_3 \rangle\!\rangle \\ & \mathsf{Step}\ (\mathsf{Lookup}\ y)\ (memo\ a\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_3}(\mu_1))) \\ = & \langle\!\langle \text{Unfold } memo \rangle\!\rangle \\ & \mathsf{Step}\ (\mathsf{Lookup}\ y)\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_3}(\mu_1) \gg= upd) \\ = & \langle\!\langle \mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_3}(\mu_1) = \overline{\mathsf{Step}\ ev_1}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_3)) \text{ for some } \mu_3, \text{unfold } \gg=, upd \rangle\!\rangle \end{array}$$

$$\begin{array}{ll}
& \mathsf{Step}\ (\mathsf{Lookup}\ y)\ (\overline{\mathsf{Step}\ ev_1}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_3) \ggg \lambda v\ \mu_3 \rightarrow \\
& \quad \mathsf{Step}\ \mathsf{Update}\ (\mathsf{Ret}\ (v, \mu_3[a \mapsto \mathit{memo}\ a\ (\mathit{return}\ v)]))))
\end{array}$$

Now let sv::$\mathsf{Value}\ (\mathsf{ByNeed}\ \mathsf{T})$ be the semantic value such that $\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_3) = \mathsf{Ret}\ (sv, \mu_3)$.

$$\begin{array}{ll}
= & \wr\ \mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_3) = \mathsf{Ret}\ (sv, \mu_3)\ \wr \\
& \mathsf{Step}\ (\mathsf{Lookup}\ y)\ (\overline{\mathsf{Step}\ ev_1}\ (\mathsf{Step}\ \mathsf{Update}\ (\mathsf{Ret}\ (sv, \mu_3[a \mapsto \mathit{memo}\ a\ (\mathit{return}\ sv)])))) \\
= & \wr\ \text{Refold } \mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(_), \overline{ev} = [\mathsf{Lookup}\ y] + \overline{ev_1} + [\mathsf{Update}]\ \wr \\
& \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_3[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2})])) \\
= & \wr\ \text{Determinism of } \mathcal{S}_{\mathbf{need}}[\![_]\!]_{_}(_), \text{assumption}\ \wr \\
& \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))
\end{array}$$

We have

$$\mu_1\ !\ a = \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_3}) \tag{10}$$

$$\blacktriangleright (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_3}(\mu_1) = \overline{\mathsf{Step}\ ev_1}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_3))) \tag{11}$$

$$\mu_2 = \mu_3[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2})] \tag{12}$$

We can apply rule $\rightsquigarrow$-MEMO to Equation (10) and Equation (11) to get $\mu_1 \rightsquigarrow \mu_3[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2})]$, and rewriting along Equation (12) proves the goal.

- **Case** $\mathsf{Lam}\ x\ \mathit{body}$, $\mathsf{ConApp}\ k\ \mathit{xs}$: Then $\mu_1 = \mu_2$ and the goal follows by $\rightsquigarrow$-REFL.
- **Case** $\mathsf{App}\ e_1\ x$: Let us assume that $\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_1}(\mu_1) = \overline{\mathsf{Step}\ ev_1}\ (\mathcal{S}_{\mathbf{need}}[\![\mathsf{Lam}\ y\ e_2]\!]_{\rho_3}(\mu_3))$ and $\mathcal{S}_{\mathbf{need}}[\![e_2]\!]_{\rho_3[y \mapsto \rho\,!\,x]}(\mu_3) = \overline{\mathsf{Step}\ ev_2}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))$, so that $\mu_1 \rightsquigarrow \mu_3$, $\mu_3 \rightsquigarrow \mu_2$ by the induction hypothesis. The goal follows by $\rightsquigarrow$-TRANS, because $\overline{ev} = [\mathsf{App}_1] + \overline{ev_1} + [\mathsf{App}_2] + \overline{ev_2}$.
- **Case** $\mathsf{Case}\ e_1\ \mathit{alts}$: Similar to $\mathsf{App}\ e_1\ x$.
- **Case** $\mathsf{Let}\ x\ e_1\ e_2$:

$$\begin{array}{ll}
& \mathcal{S}_{\mathbf{need}}[\![\mathsf{Let}\ x\ e_1\ e_2]\!]_{\rho_1}(\mu_1) \\
= & \wr\ \text{Unfold } \mathcal{S}_{\mathbf{need}}[\![_]\!]_{_}(_)\ \wr \\
& \mathit{bind}\ (\lambda d_1 \rightarrow \mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_1[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ d_1]}(_)) \\
& \qquad (\lambda d_1 \rightarrow \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}_{\mathbf{need}}[\![e_2]\!]_{\rho_1[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ d_1]}(_))) \\
& \qquad \mu_1 \\
= & \wr\ \text{Unfold } \mathit{bind}, a \triangleq \mathit{nextFree}\ \mu \text{ with } a \notin \mathit{dom}\ \mu\ \wr \\
& \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}_{\mathbf{need}}[\![e_2]\!]_{\rho_1[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathit{fetch}\ a)]}(\\
& \quad \mu_1[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_1[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathit{fetch}\ a)]})]))
\end{array}$$

At this point, we can apply the induction hypothesis to $\mathcal{S}_{\mathbf{need}}[\![e_2]\!]_{\rho_1[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathit{fetch}\ a)]}(\_)$ to conclude that $\mu_1[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_1[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathit{fetch}\ a)]})] \rightsquigarrow \mu_2$.
On the other hand, we have $\mu_1 \rightsquigarrow \mu_1[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_1[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathit{fetch}\ a)]})]$ by rule $\rightsquigarrow$-EXT (note that $a \notin \mathit{dom}\ \mu$), so the goal follows by $\rightsquigarrow$-TRANS.

□

Lemma 49 exposes nested structure in $\rightsquigarrow$-MEMO. For example, if $\mu_1 \rightsquigarrow \mu_2[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2})]$ is the result of applying rule $\rightsquigarrow$-MEMO, then we obtain a proof that the memoised expression $\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}\ \mu_1 = \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}\ \mu_2)$, and this evaluation in turn implies that $\mu_1 \rightsquigarrow \mu_2$.

Heap progression is useful to state a number of semantic properties, for example the "update once" property of memoisation and that a heap binding is semantically irrelevant when it is never updated:

Lemma 50 (Update once). *If* $\mu_1 \rightsquigarrow \mu_2$ *and* $\mu_1 \mathbin{!} a = \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_\rho)$, *then* $\mu_2 \mathbin{!} a = \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_\rho)$.

Proof. Simple proof by induction on $\mu_1 \rightsquigarrow \mu_2$. The only case updating a heap entry is $\rightsquigarrow$-Memo, and there we can see that $\mu_2 \mathbin{!} a = \mathit{memo}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_\rho)$ because evaluating v in μ_1 does not make a step. □

Lemma 51 (No update implies semantic irrelevance). *If* $\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}(\mu_1) = \overline{\mathit{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))$ *and* $\mu_1 \mathbin{!} a = \mu_2 \mathbin{!} a = \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_3})$, e_1 *not a value, then*

$$\forall d.\ \mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}(\mu_1[a \mapsto d]) = \overline{\mathit{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2[a \mapsto d]))$$

as well.

Proof. By Löb induction and cases on e.

- **Case** $\mathit{Var}\ x$: It is $\mathcal{S}_{\mathbf{need}}[\![x]\!]_{\rho_1}(\mu_1) = \mathit{Step}\ (\mathit{Lookup}\ y)\ (\mathit{memo}\ a_1\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_3}(\mu_1)))$ for the suitable a_1,y. Furthermore, it must be $a \neq a_1$, because otherwise, $\mathit{memo}\ a$ would have updated a with $\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}$. Then we also have

  $$\mathcal{S}_{\mathbf{need}}[\![x]\!]_{\rho_1}(\mu_1[a \mapsto d]) = \mathit{Step}\ (\mathit{Lookup}\ y)\ (\mathit{memo}\ a_1\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_3}(\mu_1[a \mapsto d]))).$$

  The goal follows from applying the induction hypothesis and realising that $\mu_2 \mathbin{!} a_1$ has been updated consistently with $\mathit{memo}\ a_1\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2})$.
- **Case** $\mathit{Lam}\ x\ e$, $\mathit{ConApp}\ k\ xs$: Easy to see for $\mu_1 = \mu_2$.
- **Case** $\mathit{App}\ e\ x$: We can apply the induction hypothesis twice, to both of

  $$\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}(\mu_1) = \overline{\mathit{step}\ ev_1}\ (\mathcal{S}_{\mathbf{need}}[\![\mathit{Lam}\ y\ \mathit{body}]\!]_{\rho_3}(\mu_3))$$
  $$\mathcal{S}_{\mathbf{need}}[\![\mathit{body}]\!]_{\rho_3[y \mapsto \rho_1 \,!\, x]}(\mu_3) = \overline{\mathit{step}\ ev_2}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))$$

  to show the goal.
- **Case** $\mathit{Case}\ e\ \mathit{alts}$: Similar to App.
- **Case** $\mathit{Let}\ x\ e_1\ e_2$: We have $\mathcal{S}_{\mathbf{need}}[\![\mathit{Let}\ x\ e_1\ e_2]\!]_{\rho_1}(\mu_1) = \mathit{step}\ \mathit{Let}_1\ (\mathcal{S}_{\mathbf{need}}[\![e_2]\!]_{\rho_1'}(\mu_1'))$, where $\rho_1' \triangleq \rho_1[x \mapsto \mathit{step}\ (\mathit{Lookup}\ x)\ (\mathit{fetch}\ a_1)]$, $a_1 \triangleq \mathit{nextFree}\ \mu_1$, $\mu_1' \triangleq \mu_1[a_1 \mapsto \mathit{memo}\ a_1\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_1'})]$. We have $a \neq a_1$ by a property of $\mathit{nextFree}$, and applying the induction hypothesis yields $\mathit{step}\ \mathit{Let}_1\ (\mathcal{S}_{\mathbf{need}}[\![e_2]\!]_{\rho_1'}(\mu_1'[a \mapsto d])) = \overline{\mathit{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))$ as required.

□

Now we move on to proving auxiliary lemmas about *freezeHeap*.

Lemma 52 (Heap extension preserves frozen entries). *Let* $\alpha_{\mathbb{E}}\ \mu \rightleftharpoons \gamma_{\mathbb{E}}\ \mu = \mathit{freezeHeap}\ \mu$. *If* $\mathit{adom}\ d \subseteq \mathit{dom}\ \mu$ *and* $a \notin \mathit{dom}\ \mu$, *then* $\alpha_{\mathbb{E}}\ \mu\ d = \alpha_{\mathbb{E}}\ \mu[a \mapsto d_2]\ d$.

Proof. By Löb induction. Since $\vdash^{\mathit{ne}}_{\mathbb{D}} d$, we have $d = \bigsqcup\{\mathit{step}\ (\mathit{Lookup}\ y)\ (\mathit{fetch}\ a_1)\}$ and $a_1 \in \mathit{dom}\ \mu$. Let $\mathit{memo}\ a_1\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_\rho) \triangleq \mu \mathbin{!} a_1 = \mu[a \mapsto d_2] \mathbin{!} a$. Then $\mathit{adom}\ \rho \subseteq \mathit{dom}\ \mu$ due to $\vdash^{\mathit{ne}}_{\mathbb{H}} \mu$ and the goal follows by the induction hypothesis:

$$\begin{aligned}\alpha_{\mathbb{E}}\ \mu\ d &= \bigsqcup\{\mathit{step}\ (\mathit{Lookup}\ y)\ (\mathcal{S}[\![e]\!]_{\alpha_{\mathbb{E}}\ \mu \triangleleft \rho})\}\\ &= \bigsqcup\{\mathit{step}\ (\mathit{Lookup}\ y)\ (\mathcal{S}[\![e]\!]_{\alpha_{\mathbb{E}}\ \mu[a \mapsto d_2] \triangleleft \rho})\} = \alpha_{\mathbb{E}}\ \mu[a \mapsto d_2]\ d\end{aligned}$$

□

An by-name analysis that is sound wrt. by-need must improve when an expression reduces to a value, which in particular will happen after the heap update during memoisation.

The following pair of lemmas corresponds to the Upd step of the preservation Lemma 19 where we (and Sergey et al. [2017]) resorted to hand-waving. Its proof is suprisingly tricky, but it will pay off; in a moment, we will hand-wave no more!

Lemma 53 (Preservation of heap update). *Let $\widehat{\mathsf{D}}$ be a domain with instances for* Trace, Domain, HasBind *and* Lat, *satisfying the abstraction laws* Beta-App, Beta-Sel, Bind-ByName *and* Step-Inc *from Figure 13. Furthermore, let $\alpha_{\mathbb{E}}\ \mu \rightleftharpoons \gamma_{\mathbb{E}}\ \mu = \mathit{freezeHeap}\ \mu$ for all μ and $\beta_{\mathbb{E}}\ \mu \triangleq \alpha_{\mathbb{E}}\ \mu \circ \{\_\}$ the representation function.*

(a) If $\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}(\mu_1) = \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))$ and $\mu_1\ !\ a = \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1})$, then $\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_2[a\mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2})]\triangleleft\rho_2} \sqsubseteq \mathcal{S}[\![e]\!]_{\beta_{\mathbb{E}}\ \mu_2\triangleleft\rho_1}$.

(b) If $\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}(\mu_1) = \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))$ and $\mu_2 \rightsquigarrow \mu_3$, then $\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_2} \sqsubseteq \mathcal{S}[\![e]\!]_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_1}$.

Proof. By Löb induction, we assume that both properties hold *later*.

- 53.(a): We assume that $\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}(\mu_1) = \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))$ and $\mu_1\ !\ a = \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1})$ to show $\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_2[a\mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2})]\triangleleft\rho_2} \sqsubseteq \mathcal{S}[\![e]\!]_{\beta_{\mathbb{E}}\ \mu_2\triangleleft\rho_1}$.
  We can use the IH 53.(a) to prove that $\beta_{\mathbb{E}}\ \mu_2[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2})]\ d \sqsubseteq \beta_{\mathbb{E}}\ \mu_2\ d$ for all d such that $\mathit{adom}\ d \subseteq \mathit{adom}\ \mu_2$. This is simple to see unless $d = \mathsf{Step}\ (\mathsf{Lookup}\ y)\ (\mathit{fetch}\ a)$, in which case we have:

$$
\begin{array}{ll}
& \beta_{\mathbb{E}}\ \mu_2[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2})]\ (\mathsf{Step}\ (\mathsf{Lookup}\ y)\ (\mathit{fetch}\ a)) \\
= & \wr\ \text{Unfold } \beta_{\mathbb{E}}\ \int \\
& \mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_2[a\mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2})]\triangleleft\rho_2}) \\
\sqsubseteq & \wr\ \text{IH 53.(a)}\ \int \\
& \mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathcal{S}[\![e]\!]_{\beta_{\mathbb{E}}\ \mu_2\triangleleft\rho_1}) \\
= & \wr\ \text{Refold } \beta_{\mathbb{E}}\ \int \\
& \beta_{\mathbb{E}}\ \mu_2\ (\mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathit{fetch}\ a))
\end{array}
$$

  This is enough to show the goal:

$$
\begin{array}{ll}
& \mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_2[a\mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2})]\triangleleft\rho_2} \\
\sqsubseteq & \wr\ \beta_{\mathbb{E}}\ \mu_2[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2})] \sqsubseteq \beta_{\mathbb{E}}\ \mu_2\ \int \\
& \mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_2\triangleleft\rho_2} \\
\sqsubseteq & \wr\ \text{IH 53.(b) applied to } \mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}(\mu_1) = \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))\ \int \\
& \mathcal{S}[\![e]\!]_{\beta_{\mathbb{E}}\ \mu_2\triangleleft\rho_1}
\end{array}
$$

- 53.(b) $\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}(\mu_1) = \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2)) \wedge \mu_2 \rightsquigarrow \mu_3 \Longrightarrow \mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_2} \sqsubseteq \mathcal{S}[\![e]\!]_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_1}$:
  By Löb induction and cases on e.
  - **Case** $\mathsf{Var}\ x$: Let a be the address such that $\rho_1\ !\ x = \mathsf{Step}\ (\mathsf{Lookup}\ y)\ (\mathit{fetch}\ a)$. Note that $\mu_1\ !\ a = \mathit{memo}\ a\ \_$, so the result has been memoised in μ_2, and by Lemma 50 in μ_3 as well. Hence the entry in μ_3 must be of the form $\mu_3\ !\ a = \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2})$.

$$
\begin{array}{ll}
& \mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_2} \\
\sqsubseteq & \wr\ \text{Assumption Step-Inc}\ \int \\
& \mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_2}) \\
= & \wr\ \text{Refold } \beta_{\mathbb{E}} \text{ for the appropriate } y\ \int \\
& (\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_1)\ !\ x \\
= & \wr\ \text{Refold } \mathcal{S}[\![\_]\!]_{\_}\ \int \\
& \mathcal{S}[\![x]\!]_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_1}
\end{array}
$$

  - **Case** $\mathsf{Lam}\ x\ \mathit{body}$, $\mathsf{ConApp}\ k\ \mathit{xs}$: Follows by reflexivity.

- **Case** $\mathsf{App}\ e\ x$: Then $\mathcal{S}_{\mathbf{need}}\llbracket e\rrbracket_{\rho_1}(\mu_1) = \overline{\mathsf{Step}\ ev_1}\ (\mathcal{S}_{\mathbf{need}}\llbracket \mathsf{Lam}\ y\ body\rrbracket_{\rho_3}(\mu_4))$ and $\mathcal{S}_{\mathbf{need}}\llbracket body\rrbracket_{\rho_3[y\mapsto\rho_1\,!\,x]}(\mu_4) = \overline{\mathsf{Step}\ ev_2}\ (\mathcal{S}_{\mathbf{need}}\llbracket v\rrbracket_{\rho_2}(\mu_2))$. Note that $\mu_4 \leadsto \mu_2$ by Lemma 49, hence $\mu_4 \leadsto \mu_3$ by $\leadsto$-Trans.

$$
\begin{aligned}
& \mathcal{S}\llbracket v\rrbracket_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_2} \\
\sqsubseteq\quad & \wr\ \text{IH 53.(b) at } body \text{ and } \mu_2 \leadsto \mu_3\ \int \\
& \mathcal{S}\llbracket body\rrbracket_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_3[y\mapsto\rho_1\,!\,x]} \\
\sqsubseteq\quad & \wr\ \text{Assumption } \textsc{Step-Inc}\ \int \\
& step\ \mathsf{App}_2\ (\mathcal{S}\llbracket body\rrbracket_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_3[y\mapsto\rho_1\,!\,x]}) \\
\sqsubseteq\quad & \wr\ \text{Assumption } \textsc{Beta-App}\text{, refold } \mathsf{Lam} \text{ case}\ \int \\
& apply\ (\mathcal{S}\llbracket \mathsf{Lam}\ y\ body\rrbracket_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_3})\ (\beta_{\mathbb{E}}\ \mu_3\ (\rho_1\,!\,x)) \\
\sqsubseteq\quad & \wr\ \text{IH 53.(b) at } e \text{ and } \mu_4 \leadsto \mu_3\ \int \\
& apply\ (\mathcal{S}\llbracket e\rrbracket_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_1})\ (\beta_{\mathbb{E}}\ \mu_3\ (\rho_1\,!\,x)) \\
\sqsubseteq\quad & \wr\ \text{Assumption } \textsc{Step-Inc}\ \int \\
& step\ \mathsf{App}_1\ (apply\ (\mathcal{S}\llbracket e\rrbracket_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_1})\ (\beta_{\mathbb{E}}\ \mu_3\ (\rho_1\,!\,x))) \\
=\quad & \wr\ \text{Refold } \mathcal{S}\llbracket \mathsf{App}\ e\ x\rrbracket_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_1}\ \int \\
& \mathcal{S}\llbracket \mathsf{App}\ e\ x\rrbracket_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_1}
\end{aligned}
$$

- **Case** $\mathsf{Case}\ e\ alts$: Similar to App.
- **Case** $\mathsf{Let}\ x\ e_1\ e_2$: Then $\mathcal{S}_{\mathbf{need}}\llbracket \mathsf{Let}\ x\ e_1\ e_2\rrbracket_{\rho_1}(\mu_1) = \mathsf{Step}\ \mathsf{Let}_1\ (\mathcal{S}_{\mathbf{need}}\llbracket e_2\rrbracket_{\rho_4}(\mu_4))$, where $a \triangleq nextFree\ \mu_1$, $\rho_4 \triangleq \rho_1[x \mapsto \mathsf{Step}\ (\mathsf{Lookup}\ x)\ (fetch\ a)]$, $\mu_4 \triangleq \mu_1[a \mapsto memo\ a\ (\mathcal{S}_{\mathbf{need}}\llbracket e_1\rrbracket_{\rho_4})]$. Observe that $\mu_4 \leadsto \mu_2 \leadsto \mu_3$.
The first first half of the proof is simple enough:

$$
\begin{aligned}
& \mathcal{S}\llbracket v\rrbracket_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_2} \\
\sqsubseteq\quad & \wr\ \text{IH 53.(b) at } e_2 \text{ and } \mu_2 \leadsto \mu_3\ \int \\
& \mathcal{S}\llbracket e_2\rrbracket_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_4} \\
\sqsubseteq\quad & \wr\ \text{Assumption } \textsc{Step-Inc}\ \int \\
& step\ \mathsf{Let}_1\ (\mathcal{S}\llbracket e_2\rrbracket_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_4}) \\
=\quad & \wr\ \text{Unfold } \rho_4\ \int \\
& step\ \mathsf{Let}_1\ (\mathcal{S}\llbracket e_2\rrbracket_{(\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_1)[x\mapsto\beta_{\mathbb{E}}\ \mu_3\ (\rho_4\,!\,x)]})
\end{aligned}
$$

We proceed by case analysis on whether $\mu_4\,!\,a = \mu_3\,!\,a$.
If that is the case, we get

$$
\begin{aligned}
=\quad & \wr\ \text{Unfold } \beta_{\mathbb{E}}\ \mu_3\ (\rho_4\,!\,x),\ \mu_3\,!\,a = \mu_4\,!\,a\ \int \\
& step\ \mathsf{Let}_1\ (\mathcal{S}\llbracket e_2\rrbracket_{(\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_1)[x\mapsto lfp\ (\lambda\widehat{d_1}\to step\ (\mathsf{Lookup}\ x)\ (\mathcal{S}\llbracket e_1\rrbracket_{(\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_1)[x\mapsto\widehat{d_1}]}))]}) \\
\sqsubseteq\quad & \wr\ \text{Assumption } \textsc{Bind-ByName}\ \int \\
& bind\ (\lambda\widehat{d_1} \to \mathcal{S}\llbracket e_1\rrbracket_{((\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_1))[x\mapsto step\ (\mathsf{Lookup}\ x)\ \widehat{d_1}]}) \\
& \qquad (\lambda\widehat{d_1} \to step\ \mathsf{Let}_1\ (\mathcal{S}\llbracket e_2\rrbracket_{((\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_1))[x\mapsto step\ (\mathsf{Lookup}\ x)\ \widehat{d_1}]})) \\
=\quad & \wr\ \text{Refold } \mathcal{S}\llbracket\_\rrbracket_\_\ \int \\
& \mathcal{S}\llbracket \mathsf{Let}\ x\ e_1\ e_2\rrbracket_{\beta_{\mathbb{E}}\ \mu_3\triangleleft\rho_1}
\end{aligned}
$$

Otherwise, we have $\mu_3\,!\,a \neq \mu_4\,!\,a$, implying that $\mu_4 \leadsto \mu_3$ contains an application of $\leadsto$-Memo updating $\mu_3\,!\,a$.
By rule inversion, $\mu_3\,!\,a$ is the result of updating it to the form $memo\ a\ (\mathcal{S}_{\mathbf{need}}\llbracket v_1\rrbracket_{\rho_3})$, where $\mathcal{S}_{\mathbf{need}}\llbracket e_1\rrbracket_{\rho_4}(\mu_4') = \overline{\mathsf{Step}\ ev_1}\ (\mathcal{S}_{\mathbf{need}}\llbracket v_1\rrbracket_{\rho_3}(\mu_3'))$ such that $\mu_4 \leadsto \mu_4' \leadsto \mu_3'[a \mapsto memo\ a\ (\mathcal{S}_{\mathbf{need}}\llbracket v_1\rrbracket_{\rho_3})] \leadsto \mu_3$ and $\mu_4\,!\,a = \mu_4'\,!\,a = \mu_3'\,!\,a \neq \mu_3\,!\,a$. (NB: if there are

multiple such occurrences of $\leadsto$-Memo in $\mu_4 \leadsto \mu_3$, this must be the first one, because afterwards it is $\mu_4 ! a \neq \mu'_4 ! a$.)

It is not useful to apply the IH 53.(a) to this situation directly, because $\mu'_3 \leadsto \mu_3$ does not hold. However, note that $\leadsto$-Memo contains proof that evaluation of $\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_4}(\mu'_4)$ succeeded, and it must have done so without looking at $\mu'_4 ! a$ (because that would have led to an infinite loop). Furthermore, e_1 cannot be a value; otherwise, $\mu_4 ! a = \mu_3 ! a$, a contradiction. Since e_1 is not a value and $\mu'_4 ! a = \mu'_3 ! a$, we can apply Lemma 51 to get the useful reduction

$$\begin{aligned} & \mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_4}(\mu'_4[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v_1]\!]_{\rho_3})]) \\ =& \overline{\mathsf{Step}\ \mathit{ev}_1}\ (\mathcal{S}_{\mathbf{need}}[\![v_1]\!]_{\rho_3}(\mu'_3[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v_1]\!]_{\rho_3})])). \end{aligned}$$

This reduction is so useful because we know something about $\mu'_3[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v_1]\!]_{\rho_3})]$; namely that $\mu'_3[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v_1]\!]_{\rho_3})] \leadsto \mu_3$. This allows us to apply the induction hypothesis 53.(a) to the reduction, which yields

$$\mathcal{S}[\![v_1]\!]_{\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_3} \sqsubseteq \mathcal{S}[\![e_1]\!]_{\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_4}$$

We this identity below:

$$\begin{aligned} = & \quad \wr\ \text{Unfold } \beta_{\mathbb{E}}\ \mu_3\ (\rho_4 ! x),\ \mu_3 ! a = \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v_1]\!]_{\rho_3})\ \wr \\ & \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}[\![e_2]\!]_{(\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_1)[x \mapsto \mathit{lfp}\ (\lambda \widehat{d_1} \to \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathcal{S}[\![v_1]\!]_{(\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_3)[x \mapsto \widehat{d_1}]}))]}) \\ \sqsubseteq & \quad \wr\ \mathcal{S}[\![v_1]\!]_{\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_3} \sqsubseteq \mathcal{S}[\![e_1]\!]_{\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_4},\ \text{unfold } \beta_{\mathbb{E}}\ \mu_3\ (\rho_4 ! x)\ \wr \\ & \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}[\![e_2]\!]_{(\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_1)[x \mapsto \mathit{lfp}\ (\lambda \widehat{d_1} \to \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathcal{S}[\![e_1]\!]_{(\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_1)[x \mapsto \widehat{d_1}]}))]}) \\ \sqsubseteq & \quad \wr\ \dots \text{ as above } \dots\ \wr \\ & \mathcal{S}[\![\mathsf{Let}\ x\ e_1\ e_2]\!]_{\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_1} \end{aligned}$$

□

With that, we can finally prove that heap progression preserves environment abstraction:

Lemma 54 (Heap progression preserves abstraction). *Let $\widehat{\mathsf{D}}$ be a domain with instances for* Trace, Domain, HasBind *and* Lat, *satisfying the abstraction laws* Beta-App, Beta-Sel, Bind-ByName *and* Step-Inc *in Figure 13. Furthermore, let $\alpha_{\mathbb{E}}\ \mu \rightleftharpoons \gamma_{\mathbb{E}}\ \mu = \mathit{freezeHeap}\ \mu$ for all μ.*

If $\mu_1 \leadsto \mu_2$ and $\mathit{adom}\ d \subseteq \mathit{dom}\ \mu_1$, then $\alpha_{\mathbb{E}}\ \mu_2\ d \sqsubseteq \alpha_{\mathbb{E}}\ \mu_1\ d$.

Proof. By Löb induction. Let us assume that $\mu_1 \leadsto \mu_2$ and $\mathit{adom}\ d \subseteq \mathit{dom}\ \mu_1$. Since $\vdash^{\mathsf{ne}}_{\mathbb{D}} d$, we have $d = \bigsqcup\{\mathsf{Step}\ (\mathsf{Lookup}\ y)\ (\mathit{fetch}\ a)\}$. Similar to Theorem 44, it suffices to show the goal for a single $d = \mathsf{Step}\ (\mathsf{Lookup}\ y)\ (\mathit{fetch}\ a)$ for some y, a and the representation function $\beta_{\mathbb{E}}\ \mu \triangleq \alpha_{\mathbb{E}}\ \mu \triangleleft \{\_\}$.

Furthermore, let us abbreviate $\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_i]\!]_{\rho_i}) \triangleq \mu_i ! a$. The goal is to show

$$\mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathcal{S}[\![e_2]\!]_{\beta_{\mathbb{E}}\ \mu_2 \triangleleft \rho_2}) \sqsubseteq \mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathcal{S}[\![e_1]\!]_{\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1}),$$

Monotonicity allows us to drop the $\mathit{step}\ (\mathsf{Lookup}\ x)$ context

$$\blacktriangleright (\mathcal{S}[\![e_2]\!]_{\beta_{\mathbb{E}}\ \mu_2 \triangleleft \rho_2} \sqsubseteq \mathcal{S}[\![e_1]\!]_{\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1}).$$

Now we proceed by induction on $\mu_1 \leadsto \mu_2$, which we only use to prove correct the reflexive and transitive closure in $\leadsto$-Refl and $\leadsto$-Trans. By contrast, the $\leadsto$-Memo and $\leadsto$-Ext cases make use of the Löb induction hypothesis, which is freely applicable under the ambient $\blacktriangleright$.

- **Case** $\leadsto$-Refl: Then $\mu_1 = \mu_2$ and hence $\alpha_{\mathbb{E}}\ \mu_1 = \alpha_{\mathbb{E}}\ \mu_2$.
- **Case** $\leadsto$-Trans: Apply the induction hypothesis to the sub-derivations and apply transitivity of $\sqsubseteq$.

- **Case** $\rightsquigarrow$-Ext $\dfrac{a_1 \notin \mathit{dom}\ \mu_1 \quad \mathit{adom}\ \rho \subseteq \mathit{dom}\ \mu_1 \cup \{a_1\}}{\mu \rightsquigarrow \mu_1[a_1 \mapsto \mathit{memo}\ a_1\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho})]}$:
  We get to refine $\mu_2 = \mu_1[a_1 \mapsto \mathit{memo}\ a_1\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho})]$. Since $a \in \mathit{dom}\ \mu_1$, we have $a_1 \neq a$ and thus $\mu_1\ !\ a = \mu_2\ !\ a$, thus $e_1 = e_2$, $\rho_1 = \rho_2$. The goal can be simplified to $\blacktriangleright (\mathcal{S}[\![e_1]\!]_{\beta_{\mathbb{E}}\ \mu_2 \triangleleft \rho_1} \sqsubseteq \mathcal{S}[\![e_1]\!]_{\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1})$. We can apply the induction hypothesis to get $\blacktriangleright (\beta_{\mathbb{E}}\ \mu_2 \sqsubseteq \beta_{\mathbb{E}}\ \mu_1)$, and the goal follows by monotonicity.
- **Case** $\rightsquigarrow$-Memo $\dfrac{\mu_1\ !\ a_1 = \mathit{memo}\ a_1\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_3}) \quad \blacktriangleright (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_3}(\mu_1) = \overline{\mathsf{Step}\ \mathit{ev}}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_3)))}{\mu_1 \rightsquigarrow \mu_3[a_1 \mapsto \mathit{memo}\ a_1\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2})]}$:
  We get to refine $\mu_2 = \mu_3[a_1 \mapsto \mathit{memo}\ a_1\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2})]$. When $a_1 \neq a$, we have $\mu_1\ !\ a = \mu_2\ !\ a$ and the goal follows as in the $\rightsquigarrow$-Ext case. Otherwise, $a = a_1$, $e_1 = e$, $\rho_3 = \rho_1$, $e_2 = v$.
  We can use Lemma 53.(a) to prove that $\beta_{\mathbb{E}}\ \mu_2\ d \sqsubseteq \beta_{\mathbb{E}}\ \mu_3\ d$ for all d such that $\mathit{adom}\ d \subseteq \mathit{adom}\ \mu_2$. This is simple to see unless $d = \mathsf{Step}\ (\mathsf{Lookup}\ y)\ (\mathit{fetch}\ a)$, in which case we have:

$$
\begin{array}{ll}
& \beta_{\mathbb{E}}\ \mu_2\ (\mathsf{Step}\ (\mathsf{Lookup}\ y)\ (\mathit{fetch}\ a)) \\
= & \wr\ \text{Unfold } \beta_{\mathbb{E}}\ \int \\
& \mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_2 \triangleleft \rho_2}) \\
\sqsubseteq & \wr\ \text{Lemma 53.(a)}\ \int \\
& \mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathcal{S}[\![e]\!]_{\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_1}) \\
= & \wr\ \text{Refold } \beta_{\mathbb{E}}\ \int \\
& \beta_{\mathbb{E}}\ \mu_3\ (\mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathit{fetch}\ a))
\end{array}
$$

  We can finally show the goal $\beta_{\mathbb{E}}\ \mu_2\ d \sqsubseteq \beta_{\mathbb{E}}\ \mu_1\ d$ for all d such that $\mathit{adom}\ d \subseteq \mathit{dom}\ \mu_1$:

$$
\begin{array}{ll}
& \beta_{\mathbb{E}}\ \mu_2\ d \\
\sqsubseteq & \wr\ \beta_{\mathbb{E}}\ \mu_2 \sqsubseteq \beta_{\mathbb{E}}\ \mu_3\ \int \\
& \beta_{\mathbb{E}}\ \mu_3\ d \\
\sqsubseteq & \wr\ \text{Löb induction hypothesis at } \mu_1 \rightsquigarrow \mu_3 \text{ by Lemma 49}\ \int \\
& \beta_{\mathbb{E}}\ \mu_1\ d
\end{array}
$$

□

Lemma 55 (By-need evaluation preserves by-name trace abstraction). *Let $\widehat{\mathsf{D}}$ be a domain with instances for* Trace, Domain, HasBind *and* Lat*, satisfying the abstraction laws* Step-App, Step-Sel, Beta-App, Beta-Sel, Bind-ByName, Step-Inc *and* Update *in Figure 13. Furthermore, let $\alpha_{\mathbb{E}}\ \mu \rightleftharpoons \gamma_{\mathbb{E}}\ \mu = \mathit{freezeHeap}\ \mu$ for all μ.*

If $\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}(\mu_1) = \overline{\mathsf{Step}\ \mathit{ev}}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))$, then $\overline{\mathit{step}\ \mathit{ev}}\ (\mathcal{S}[\![v]\!]_{\alpha_{\mathbb{E}}\ \mu_2 \triangleleft \{\_\} \triangleleft \rho_2}) \sqsubseteq \mathcal{S}[\![e]\!]_{\alpha_{\mathbb{E}}\ \mu_1 \triangleleft \{\_\} \triangleleft \rho_1}$.

Proof. By Löb induction and cases on e, using the representation function $\beta_{\mathbb{E}} \triangleq \alpha_{\mathbb{E}} \circ \{\_\}$.

- **Case** $\mathsf{Var}\ x$: By assumption, we know that

$$\mathcal{S}_{\mathbf{need}}[\![x]\!]_{\rho_1}(\mu_1) = \mathsf{Step}\ (\mathsf{Lookup}\ y)\ (\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_3}(\mu_1))) = \overline{\mathsf{Step}\ \mathit{ev}}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))$$

  for some y, a, e_1, ρ_3, such that $\rho_1 = \mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathit{fetch}\ a)$, $\mu_1\ !\ a = \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_3})$ and $\overline{\mathit{ev}} = [\mathsf{Lookup}\ y] + \overline{\mathit{ev}_1} + [\mathsf{Update}]$ for some ev_1 by determinism.
  The step below that uses Item 53.(b) does so at e_1 and $\mu_2 \rightsquigarrow \mu_2$ to get $\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_2 \triangleleft \rho_2} \sqsubseteq \mathcal{S}[\![e_1]\!]_{\beta_{\mathbb{E}}\ \mu_2 \triangleleft \rho_3}$, in order to prove that $(\beta_{\mathbb{E}}\ \mu_2 \triangleleft \rho_2) \sqsubseteq (\beta_{\mathbb{E}}\ \mu_2[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_3})] \triangleleft \rho_2)$.

$$
\begin{array}{ll}
& \overline{\mathit{step}\ \mathit{ev}}\ (\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_2 \triangleleft \rho_2}) \\
= & \wr\ \overline{\mathit{ev}} = [\mathsf{Lookup}\ y] + \overline{\mathit{ev}_1} + [\mathsf{Update}]\ \int \\
& \mathit{step}\ (\mathsf{Lookup}\ y)\ (\overline{\mathit{step}\ \mathit{ev}_1}\ (\mathit{step}\ \mathsf{Update}\ (\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_2 \triangleleft \rho_2}))) \\
= & \wr\ \text{Assumption Update}\ \int
\end{array}
$$

$$
\begin{array}{ll}
& \mathit{step}\ (\mathsf{Lookup}\ y)\ (\overline{\mathit{step}\ \mathit{ev}_1}\ (\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_2 \triangleleft \rho_2})) \\
\sqsubseteq & \{\ \text{Item 53.(b) at } e_1 \text{ implies } (\beta_{\mathbb{E}}\ \mu_2 \triangleleft \rho_2) \sqsubseteq (\beta_{\mathbb{E}}\ \mu_2[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_3})] \triangleleft \rho_2)\ \} \\
& \mathit{step}\ (\mathsf{Lookup}\ y)\ (\overline{\mathit{step}\ \mathit{ev}_1}\ (\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_2[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_3})] \triangleleft \rho_2})) \\
\sqsubseteq & \{\ \text{Lemma 55}\ \} \\
& \mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathcal{S}[\![e_1]\!]_{\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_3}) \\
= & \{\ \text{Refold } \beta_{\mathbb{E}},\ \rho_3\,!\,x\ \} \\
& \beta_{\mathbb{E}}\ (\rho_1\,!\,x) \\
= & \{\ \text{Refold } \mathcal{S}[\![x]\!]_{\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1}\ \} \\
& \mathcal{S}[\![x]\!]_{\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1}
\end{array}
$$

- **Case** $\mathsf{Let}\ x\ e_1\ e_2$: We can make one step to see

  $\mathcal{S}_{\mathbf{need}}[\![\mathsf{Let}\ x\ e_1\ e_2]\!]_{\rho_1}(\mu_1) = \mathsf{Step}\ \mathsf{Let}_1\ (\mathcal{S}_{\mathbf{need}}[\![e_2]\!]_{\rho_3}(\mu_3)) = \mathsf{Step}\ \mathsf{Let}_1\ (\overline{\mathsf{Step}\ \mathit{ev}_1}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2)))$,

  where $\rho_3 \triangleq \rho_1[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathit{fetch}\ a)]$, $a \triangleq \mathit{nextFree}\ \mu_1$, $\mu_3 \triangleq \mu_1[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_3})]$.
  Then $(\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_3)\,!\,y = (\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1)\,!\,y$ whenever $x \neq y$ by Lemma 52, and $(\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_3)\,!\,x = \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathcal{S}[\![e_1]\!]_{\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_3})$.
  We prove the goal, thus

$$
\begin{array}{ll}
& \overline{\mathit{step}\ \mathit{ev}}\ (\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_2 \triangleleft \rho_2}) \\
= & \{\ \overline{\mathit{ev}} = \mathsf{Let}_1 : \overline{\mathit{ev}_1}\ \} \\
& \mathit{step}\ \mathsf{Let}_1\ (\overline{\mathit{step}\ \mathit{ev}_1}\ (\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_2 \triangleleft \rho_2})) \\
\sqsubseteq & \{\ \text{Induction hypothesis at } \mathit{ev}_1\ \} \\
& \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}[\![e_2]\!]_{\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_3}) \\
= & \{\ \text{Rearrange } \beta_{\mathbb{E}}\ \mu_3 \text{ by above reasoning}\ \} \\
& \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}[\![e_2]\!]_{(\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1)[x \mapsto \beta_{\mathbb{E}}\ \mu_3\ (\rho_3\,!\,x)]}\ \mu_3) \\
= & \{\ \text{Expose fixpoint, rewriting } \beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_3 \text{ to } (\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1)[x \mapsto \beta_{\mathbb{E}}\ \mu_3\ (\rho_3\,!\,x)]\ \} \\
& \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}[\![e_2]\!]_{(\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1)[x \mapsto \mathit{lfp}\ (\lambda \widehat{d_1} \rightarrow \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathcal{S}[\![e_1]\!]_{(\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1)[x \mapsto \widehat{d_1}]}))]}) \\
= & \{\ \text{Partially unroll } \mathit{lfp}\ \} \\
& \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}[\![e_2]\!]_{(\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1)[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathit{lfp}\ (\lambda \widehat{d_1} \rightarrow \mathcal{S}[\![e_1]\!]_{(\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1)[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ \widehat{d_1}]}))]}) \\
\sqsubseteq & \{\ \text{Assumption } \textsc{Bind-ByName}\ \} \\
& \mathit{bind}\ (\lambda \widehat{d_1} \rightarrow \mathcal{S}[\![e_1]\!]_{((\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1))[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ \widehat{d_1}]}) \\
& \qquad (\lambda \widehat{d_1} \rightarrow \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}[\![e_2]\!]_{((\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1))[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ \widehat{d_1}]})) \\
= & \{\ \text{Refold } \mathcal{S}[\![\mathsf{Let}\ x\ e_1\ e_2]\!]_{\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1}\ \} \\
& \mathcal{S}[\![\mathsf{Let}\ x\ e_1\ e_2]\!]_{\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1}
\end{array}
$$

- **Case** Lam, ConApp: By reflexivity.
- **Case** $\mathsf{App}\ e\ x$: Very similar to Lemma 43, since the heap is never updated or extended. There is one exception: We must apply Lemma 54 to argument denotations.
  We have $\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}(\mu_1) = \overline{\mathsf{Step}\ \mathit{ev}_1}\ (\mathcal{S}_{\mathbf{need}}[\![\mathsf{Lam}\ y\ \mathit{body}]\!]_{\rho_3}(\mu_3))$ and $\mathcal{S}_{\mathbf{need}}[\![\mathit{body}]\!]_{\rho_3[y \mapsto \rho_1\,!\,x]}(\mu_3) = \overline{\mathsf{Step}\ \mathit{ev}_2}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))$. We have $\mu_1 \rightsquigarrow \mu_3$ by Lemma 49.

$$
\begin{array}{ll}
& \mathit{step}\ \mathsf{App}_1\ (\overline{\mathsf{Step}\ \mathit{ev}_1}\ (\mathit{step}\ \mathsf{App}_2\ (\overline{\mathsf{Step}\ \mathit{ev}_2}\ (\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_2 \triangleleft \rho_2})))) \\
= & \{\ \text{Induction hypothesis at } \overline{\mathit{ev}_2}\ \} \\
& \mathit{step}\ \mathsf{App}_1\ (\overline{\mathit{step}\ \mathit{ev}_1}\ (\mathit{step}\ \mathsf{App}_2\ (\mathcal{S}[\![\mathit{body}]\!]_{\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_3[y \mapsto \rho_1\,!\,x]}))) \\
\sqsubseteq & \{\ \text{Assumption } \textsc{Beta-App}\text{, refold } \mathsf{Lam} \text{ case}\ \}
\end{array}
$$

$$
\begin{array}{ll}
 & step\ \mathsf{App}_1\ (\overline{step\ ev_1}\ (apply\ (\mathcal{S}[\![\mathsf{Lam}\ y\ body]\!]_{\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_3})\ ((\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_1)\,!\,x))) \\
\sqsubseteq & \langle\ \text{Assumption } \textsc{Step-App}\ \rangle \\
 & step\ \mathsf{App}_1\ (apply\ (\overline{step\ ev_1}\ (\mathcal{S}[\![\mathsf{Lam}\ y\ body]\!]_{\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_3}))\ ((\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_1)\,!\,x)) \\
\sqsubseteq & \langle\ \text{Induction hypothesis at } \overline{ev_1}\ \rangle \\
 & step\ \mathsf{App}_1\ (apply\ (\mathcal{S}[\![e]\!]_{\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1})\ ((\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_1)\,!\,x)) \\
\sqsubseteq & \langle\ \text{Lemma 54}\ \rangle \\
 & step\ \mathsf{App}_1\ (apply\ (\mathcal{S}[\![e]\!]_{\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1})\ ((\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1)\,!\,x)) \\
= & \langle\ \text{Refold } \mathcal{S}[\![\_]\!]_{\_}\ \rangle \\
 & \mathcal{S}[\![\mathsf{App}\ e\ x]\!]_{\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1}
\end{array}
$$

- **Case** $\mathsf{Case}\ e\ alts$: The same as in Lemma 43.
  We have $\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}(\mu_1) = \overline{\mathsf{Step}\ ev_1}\ (\mathcal{S}_{\mathbf{need}}[\![\mathsf{ConApp}\ k\ ys]\!]_{\rho_3}(\mu_3)), \mathcal{S}_{\mathbf{need}}[\![e_r]\!]_{\rho_1[\overline{xs \mapsto map\ (\rho_3\,!)\ ys}]}(\mu_3) = \overline{\mathsf{Step}\ ev_2}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))$, where $alts\,!\,k = (xs, e_r)$ is the matching RHS.

$$
\begin{array}{ll}
 & \overline{step\ ev}\ (\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}} \triangleleft \rho_2}\ m2) \\
\sqsubseteq & \langle\ \overline{ev} = [\,\mathsf{Case}_1\,] + \overline{ev_1} + [\,\mathsf{Case}_2\,] + ev_2, \text{IH at } ev_2\ \rangle \\
 & step\ \mathsf{Case}_1\ (\overline{step\ ev_1}\ (step\ \mathsf{Case}_2\ (\mathcal{S}[\![e_r]\!]_{\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_1[\overline{xs \mapsto map\ (\widehat{\rho_3}\,!)\ ys}]}))) \\
\sqsubseteq & \langle\ \text{Assumption } \textsc{Beta-Sel}\ \rangle \\
 & step\ \mathsf{Case}_1\ (\overline{step\ ev_1}\ (select\ (\mathcal{S}[\![\mathsf{ConApp}\ k\ ys]\!]_{\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_3})\ (cont \triangleleft alts))) \\
\sqsubseteq & \langle\ \text{Assumption } \textsc{Step-Sel}\ \rangle \\
 & step\ \mathsf{Case}_1\ (select\ (\overline{step\ ev_1}\ (\mathcal{S}[\![\mathsf{ConApp}\ k\ ys]\!]_{\beta_{\mathbb{E}}\ \mu_3 \triangleleft \rho_3}))\ (cont \triangleleft alts)) \\
\sqsubseteq & \langle\ \text{Induction hypothesis at } ev_1\ \rangle \\
 & step\ \mathsf{Case}_1\ (select\ (\mathcal{S}[\![e]\!]_{\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1})\ (cont \triangleleft alts)) \\
= & \langle\ \text{Refold } \mathcal{S}[\![\mathsf{Case}\ e\ alts]\!]_{\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1}\ \rangle \\
 & \mathcal{S}[\![\mathsf{Case}\ e\ alts]\!]_{\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1}
\end{array}
$$

□

Using *freezeHeap*, we can give a Galois connection expressing correctness of a by-name analysis wrt. by-need semantics:

Theorem 56 (Sound By-need Interpretation). *Let $\widehat{\mathsf{D}}$ be a domain with instances for* Trace, Domain, HasBind *and* Lat, *and let $\alpha_{\mathbb{T}} \rightleftharpoons \gamma_{\mathbb{T}} = nameNeed$, as well as $\alpha_{\mathbb{E}}\ \mu \rightleftharpoons \gamma_{\mathbb{E}}\ \mu = freezeHeap\ \mu$ from Figure 18. If the abstraction laws in Figure 13 hold, then $\mathcal{S}[\![\_]\!]_{\_}$ instantiates at $\widehat{\mathsf{D}}$ to an abstract interpreter that is sound wrt. $\gamma_{\mathbb{E}} \to \alpha_{\mathbb{T}}$, that is,*

$$\alpha_{\mathbb{T}}\ \{\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho}(\mu)\} \sqsubseteq (\mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_{\alpha_{\mathbb{E}}\ \mu \triangleleft \{\_\} \triangleleft \rho})$$

Proof. As in Theorem 44, we simplify our proof obligation to the single-trace case:

$$\forall \rho.\ \beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho}(\mu)) \sqsubseteq (\mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_{\beta_{\mathbb{E}}\ \mu \triangleleft \rho}),$$

where $\beta_{\mathbb{T}} \triangleq \alpha_{\mathbb{T}} \circ \{\_\}$ and $\beta_{\mathbb{E}}\ \mu \triangleq \alpha_{\mathbb{E}}\ \mu \circ \{\_\}$ are the representation functions corresponding to $\alpha_{\mathbb{T}}$ and $\alpha_{\mathbb{E}}$. We proceed by Löb induction.

Whenever $\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho}(\mu) = \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))$ yields a balanced trace and makes at least one step, we can reuse the proof for Lemma 55 as follows:

$$
\begin{array}{ll}
 & \beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho}(\mu)) \\
= & \langle\ \mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho}(\mu) = \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2)), \text{unfold } \beta_{\mathbb{T}}\ \rangle \\
 & \overline{step\ ev}\ (\beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))) \\
\sqsubseteq & \langle\ \text{Induction hypothesis (needs non-empty } \overline{ev})\ \rangle
\end{array}
$$

$$\begin{array}{ll} & \overline{\mathit{step}\ \mathit{ev}}\ (\mathcal{S}[\![v]\!]_{\beta_{\mathbb{E}}\ \mu_2 \triangleleft \rho_2}) \\ \sqsubseteq & \{\ \text{Lemma 55}\ \} \\ & \mathcal{S}[\![e]\!]_{\beta_{\mathbb{E}}\ \mu \triangleleft \rho} \end{array}$$

Thus, without loss of generality, we may assume that if e is not a value, then either the trace diverges or is stuck. We proceed by cases over e.

- **Case** $\mathsf{Var}\ x$: The stuck case follows by unfolding $\beta_{\mathbb{T}}$.

$$\begin{array}{ll} & \beta_{\mathbb{T}}\ ((\rho\,!\,x)\ \mu) \\ = & \{\ \vdash^{\mathsf{ne}}_{\mathbb{E}} \rho, \text{Unfold } \beta_{\mathbb{T}}\ \} \\ & \mathit{step}\ (\mathsf{Lookup}\ y)\ (\beta_{\mathbb{T}}\ (\mathit{fetch}\ a\ \mu)) \\ = & \{\ \vdash^{\mathsf{ne}}_{\mathbb{H}} \mu\ \} \\ & \mathit{step}\ (\mathsf{Lookup}\ y)\ (\beta_{\mathbb{T}}\ (\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_1}(\mu)))) \end{array}$$

  By assumption, $\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_1}(\mu))$ diverges or gets stuck and the result is equivalent to $\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_1}(\mu)$.

$$\begin{array}{ll} = & \{\ \text{Diverging or stuck}\ \} \\ & \mathit{step}\ (\mathsf{Lookup}\ y)\ (\beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_2}(\mu))) \\ \sqsubseteq & \{\ \text{Induction hypothesis}\ \} \\ & \mathit{step}\ (\mathsf{Lookup}\ y)\ (\mathcal{S}[\![e_1]\!]_{\beta_{\mathbb{E}}\ \mu \triangleleft \rho_1}) \\ = & \{\ \text{Refold } \beta_{\mathbb{E}}\ \} \\ & \beta_{\mathbb{E}}\ \mu\ (\rho\,!\,x) \end{array}$$

- **Case** $\mathsf{Lam}\ x\ \mathit{body}$:

$$\begin{array}{ll} & \beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{need}}[\![\mathsf{Lam}\ x\ \mathit{body}]\!]_{\rho}(\mu)) \\ = & \{\ \text{Unfold } \mathcal{S}_{\mathbf{need}}[\![\_]\!]_{\_}(\_), \beta_{\mathbb{T}}\ \} \\ & \mathit{fun}\ (\lambda \widehat{d} \to \bigsqcup\{\mathit{step}\ \mathsf{App}_2\ (\beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{need}}[\![\mathit{body}]\!]_{\rho[x \mapsto d]}(\mu)))\ \mid\ \beta_{\mathbb{E}}\ \mu\ d \sqsubseteq \widehat{d}\}) \\ \sqsubseteq & \{\ \text{Induction hypothesis}\ \} \\ & \mathit{fun}\ (\lambda \widehat{d} \to \bigsqcup\{\mathit{step}\ \mathsf{App}_2\ (\mathcal{S}[\![\mathit{body}]\!]_{\beta_{\mathbb{E}}\ \mu \triangleleft \rho[x \mapsto d]})\ \mid\ \beta_{\mathbb{E}}\ \mu\ d \sqsubseteq \widehat{d}\}) \\ \sqsubseteq & \{\ \text{Least upper bound / } \alpha_{\mathbb{E}} \circ \gamma_{\mathbb{E}} \sqsubseteq \mathit{id}\ \} \\ & \mathit{fun}\ (\lambda \widehat{d} \to \mathit{step}\ \mathsf{App}_2\ (\mathcal{S}[\![\mathit{body}]\!]_{((\beta_{\mathbb{E}}\ \mu \triangleleft \rho))[x \mapsto \widehat{d}]})) \\ = & \{\ \text{Refold } \mathcal{S}[\![\_]\!]_{\_}\ \} \\ & \mathcal{S}[\![\mathsf{Lam}\ x\ \mathit{body}]\!]_{\beta_{\mathbb{E}}\ \mu \triangleleft \rho} \end{array}$$

- **Case** $\mathsf{ConApp}\ k\ \mathit{xs}$:

$$\begin{array}{ll} & \beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{need}}[\![\mathsf{ConApp}\ k\ \mathit{xs}]\!]_{\rho}(\mu)) \\ = & \{\ \text{Unfold } \mathcal{S}_{\mathbf{need}}[\![\_]\!]_{\_}(\_), \beta_{\mathbb{T}}\ \} \\ & \mathit{con}\ k\ (\mathit{map}\ ((\beta_{\mathbb{E}}\ \mu \triangleleft \rho)\,!)\ \mathit{xs}) \\ = & \{\ \text{Refold } \mathcal{S}[\![\_]\!]_{\_}\ \} \\ & \mathcal{S}[\![\mathsf{Lam}\ x\ \mathit{body}]\!]_{\beta_{\mathbb{E}}\ \mu \triangleleft \rho} \end{array}$$

- **Case** $\mathsf{App}\ e\ x$, **Case** $\mathsf{Case}\ e\ \mathit{alts}$: The same steps as in Theorem 44.
- **Case** $\mathsf{Let}\ x\ e_1\ e_2$: We can make one step to see

$$\mathcal{S}_{\mathbf{need}}[\![\mathsf{Let}\ x\ e_1\ e_2]\!]_{\rho}(\mu) = \mathsf{Step}\ \mathsf{Let}_1\ (\mathcal{S}_{\mathbf{need}}[\![e_2]\!]_{\rho_1}(\mu_1)),$$

  where $\rho_1 \triangleq \rho[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathit{fetch}\ a)]$, $a \triangleq \mathit{nextFree}\ \mu$, $\mu_1 \triangleq \mu[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_1})]$. Then $(\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1)\,!\,y = (\beta_{\mathbb{E}}\ \mu \triangleleft \rho)\,!\,y$ whenever $x \neq y$ by Lemma 52, and $(\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1)\,!\,x = \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathcal{S}[\![e_1]\!]_{\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho_1})$.

$$
\begin{aligned}
& \beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{need}}\llbracket \mathsf{Let}\ x\ e_1\ e_2 \rrbracket_{\rho}(\mu)) \\
= \quad & \wr \text{ Unfold } \mathcal{S}_{\mathbf{need}}\llbracket \_ \rrbracket_{\_}(\_) \ \wr \\
& \beta_{\mathbb{T}}\ (\mathit{bind}\ (\lambda d_1 \rightarrow \mathcal{S}_{\mathbf{need}}\llbracket e_1 \rrbracket_{\rho_1})\ (\lambda d_1 \rightarrow \mathsf{Step}\ \mathsf{Let}_1\ (\mathcal{S}_{\mathbf{need}}\llbracket e_2 \rrbracket_{\rho_1}))\ \mu) \\
= \quad & \wr \text{ Unfold } \mathit{bind},\ a \notin \mathit{dom}\ \mu, \text{ unfold } \beta_{\mathbb{T}} \ \wr \\
& \mathit{step}\ \mathsf{Let}_1\ (\beta_{\mathbb{T}}\ (\mathcal{S}_{\mathbf{need}}\llbracket e_2 \rrbracket_{\rho_1}(\mu_1))) \\
\sqsubseteq \quad & \wr \text{ Induction hypothesis, unfolding } \rho_1 \ \wr \\
& \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}\llbracket e_2 \rrbracket_{(\beta_{\mathbb{E}}\ \mu_1 \triangleleft \rho)[x \mapsto \beta_{\mathbb{E}}\ \mu_1\ (\rho_1\, !\, x)]}) \\
= \quad & \wr \text{ Expose fixpoint, rewriting } \beta_{\mathbb{E}}\ \mu_1\ (\rho_1\, !\, x) \text{ to } (\beta_{\mathbb{E}}\ \mu \triangleleft \rho)[x \mapsto \beta_{\mathbb{E}}\ \mu_1\ (\rho_1\, !\, x)] \text{ using Lemma 52} \ \wr \\
& \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}\llbracket e_2 \rrbracket_{(\beta_{\mathbb{E}}\ \mu \triangleleft \rho)[x \mapsto \mathit{lfp}\ (\lambda \widehat{d_1} \rightarrow \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathcal{S}\llbracket e_1 \rrbracket_{(\beta_{\mathbb{E}}\ \mu \triangleleft \rho)[x \mapsto \widehat{d_1}]}))]}) \\
= \quad & \wr \text{ Partially unroll fixpoint } \wr \\
& \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}\llbracket e_2 \rrbracket_{(\beta_{\mathbb{E}}\ \mu \triangleleft \rho)[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ (\mathit{lfp}\ (\lambda \widehat{d_1} \rightarrow \mathcal{S}\llbracket e_1 \rrbracket_{(\beta_{\mathbb{E}}\ \mu \triangleleft \rho)[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ \widehat{d_1}]}))]}) \\
\sqsubseteq \quad & \wr \text{ Assumption } \textsc{Bind-ByName}, \text{ with } \widehat{\rho} = \beta_{\mathbb{E}}\ \mu \triangleleft \rho \ \wr \\
& \mathit{bind}\ (\lambda d_1 \rightarrow \mathcal{S}\llbracket e_1 \rrbracket_{(\beta_{\mathbb{E}}\ \mu \triangleleft \rho)[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ d_1]}) \\
& \qquad (\lambda d_1 \rightarrow \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}\llbracket e_2 \rrbracket_{(\beta_{\mathbb{E}}\ \mu \triangleleft \rho)[x \mapsto \mathit{step}\ (\mathsf{Lookup}\ x)\ d_1]})) \\
= \quad & \wr \text{ Refold } \mathcal{S}\llbracket \mathsf{Let}\ x\ e_1\ e_2 \rrbracket_{\beta_{\mathbb{E}}\ \mu \triangleleft \rho} \ \wr \\
& \mathcal{S}\llbracket \mathsf{Let}\ x\ e_1\ e_2 \rrbracket_{\beta_{\mathbb{E}}\ \mu \triangleleft \rho}
\end{aligned}
$$

□

We can apply this by-need abstraction theorem to usage analysis on open expressions, just as before:

Lemma 57 ($\mathcal{S}_{\mathbf{usage}}\llbracket \_ \rrbracket_{\_}$ abstracts $\mathcal{S}_{\mathbf{need}}\llbracket \_ \rrbracket_{\_}(\_)$, open)**.** *Usage analysis $\mathcal{S}_{\mathbf{usage}}\llbracket \_ \rrbracket_{\_}$ is sound wrt. $\mathcal{S}_{\mathbf{need}}\llbracket \_ \rrbracket_{\_}(\_)$, that is,*

$$\alpha_{\mathbb{T}}\ \{\mathcal{S}_{\mathbf{need}}\llbracket e \rrbracket_{\rho}(\mu)\} \sqsubseteq \mathcal{S}_{\mathbf{usage}}\llbracket e \rrbracket_{\alpha_{\mathbb{E}} \triangleleft \{\_\} \triangleleft \rho} \ \textbf{where}\ \alpha_{\mathbb{T}} \rightleftharpoons \_ = \mathit{nameNeed}; \alpha_{\mathbb{E}}\ \mu \rightleftharpoons \_ = \mathit{freezeHeap}\ \mu$$

Proof. By Theorem 56, it suffices to show the abstraction laws in Figure 13 as done in the proof for Lemma 9. □