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:

1 INTRODUCTION

A static program analysis infers facts about a program, such as “this program is well-typed”, “this higher-order function is always called with argument \(\lambda x.x + 1\)” or “this program never evaluates \(x\)”. In a functional-language setting, such static analyses are often defined compositionally on the input term. For example, consider the claim “(even 42) has type Bool”. Type analysis asserts that even :: Int → Bool, 42 :: Int, and then applies the function type to the argument type to produce the result type even 42 :: Bool. The function type Int → 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...
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...
\[ A[\_]: \text{Exp} \to (\text{Var} \to \text{AbsTy}) \to \text{AbsTy} \]

\begin{align*}
A[x]_{\rho} &= \rho(x) \\
A[\lambda x. e]_{\rho} &= \text{fun}_{\lambda \theta. A[e]_{\rho[x \to \theta]}} \\
A[e]_{\rho} &= \text{app}(A[e]_{\rho})(\rho(x)) \\
A[\text{let } x = e_1 \text{ in } e_2]_{\rho} &= A[e_2]_{\rho[x \to e_1, x]} \\
\text{fun}_{\_}(f) &= \langle \varphi[x \mapsto A], \varphi(x) \vdash \zeta \rangle \\
\text{where } (\varphi, \zeta) &= f(\langle \{x \mapsto U, \text{Rep } U\} \rangle) \\
\text{app}(\langle \varphi_f, a \vdash \zeta \rangle)(\langle \varphi_a, \_ \rangle) &= \langle \varphi_f \cup (a \ast \varphi_a), \zeta \rangle
\end{align*}

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 needness 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:

<table>
<thead>
<tr>
<th>Variables x, y ∈ Var</th>
<th>Constructors K ∈ Con with arity α_K ∈ N</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values v ∈ Val :≡ _x.e</td>
<td>K _x^{α_K}</td>
</tr>
<tr>
<td>Expressions e ∈ Exp :≡ x</td>
<td>v</td>
</tr>
</tbody>
</table>

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 \( \lambda x. x \) (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 \( A[e]_{\rho} \) for lazy program semantics that conservatively approximates semantic absence.\(^1\) It takes an environment \( \rho \in \text{Var} \to \text{Absence} \) 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 \( A[\text{let } x = e_1 \text{ in } e_2]_{\rho} = A[e_2]_{\rho[x \to \text{Rep } \lambda \theta. xK, \text{Abs } e_1, x \to \theta]}. \)

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

The summary \( \text{Rep U} \) denotes an infinite repetition of \( U \), as expressed by the non-syntactic equality \( \text{Rep U} \equiv A \in \text{Rep U} \).

We illustrate the analysis at the example expression \( e \triangleq \text{let } k = \hat{\lambda} y.\hat{\lambda} x.\hat{\lambda} z.y \text{ in } k \ x_1 \ x_2 \), where the initial environment for \( e \), \( \rho_e(x) \triangleq \{ [x \mapsto U], \text{Rep U} \} \), declares the free variables of \( e \) with a pessimistic summary \( \text{Rep U} \).

\[
\begin{align*}
\mathcal{A}[\text{let } k = \hat{\lambda} y.\hat{\lambda} x.\hat{\lambda} z.y \text{ in } k \ x_1 \ x_2]_{\rho_e} & \quad \text{Unfold } \mathcal{A}[\text{let } x = e_1 \text{ in } e_2]. \text{NB: Lazy Let!} \\
= \mathcal{A}[k \ x_1 \ x_2]_{\rho_e[k \mapsto k, A \in \mathcal{A}[\hat{\lambda} y.\hat{\lambda} x.\hat{\lambda} z.y]_{\rho_e}]} & \quad \text{Unf. } \mathcal{A}[, \rho_1 \triangleq \rho_e[k \mapsto k, A \in \mathcal{A}[\hat{\lambda} y.\hat{\lambda} x.\hat{\lambda} z.y]_{\rho_e}]} & \quad (1) \\
= \text{app}(\text{app}(\rho_1(1))(\rho_1(x_1)))(\rho_1(x_2)) & \quad \text{Unfold } \rho_1(k) & \quad (2) \\
= \text{app}(\text{app}(k \in \mathcal{A}[\hat{\lambda} y.\hat{\lambda} x.\hat{\lambda} z.y]_{\rho_1})(\rho_1(x_1)))(\rho_1(x_2)) & \quad \text{Unfold fun twice, simplify} & \quad (3) \\
= \text{app}(\text{app}(k \in \mathcal{A}[, A \in \mathcal{A}[\hat{\lambda} x.e] \text{ twice, } \mathcal{A}[x]) \text{ twice, } \mathcal{A}[x]) & \quad \text{Unfold fun twice, simplify} & \quad (4) \\
= \text{app}(\text{app}(k \in U, A \in \mathcal{A}[\hat{\lambda} x.e] \text{ twice, } \mathcal{A}[x]) \text{ twice, } \mathcal{A}[x]) & \quad \text{Unfold app, } \rho_1(x_1) = \rho_e(x_1), \text{ simplify} & \quad (5) \\
= \text{app}(\text{app}(k \in U, A \in \mathcal{A}[\hat{\lambda} x.e] \text{ twice, } \mathcal{A}[x]) & \quad \text{Unfold app, simplify} & \quad (6) \\
= \text{app}(k \in U, x_1 \mapsto U, A \in \mathcal{A}[\hat{\lambda} x.e] \text{ twice, } \mathcal{A}[x]) & \quad \text{Unfold app, simplify} & \quad (7) \\
& \quad (8) \\
= \langle [k \mapsto U, x_1 \mapsto U], \text{Rep U} \rangle & \quad \text{Unfold app, simplify} & \quad (9)
\end{align*}
\]

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 \textit{app} and \textit{fun} helper functions applied to environment entries for the involved variables. Step (5) then computes the summary as part of the absence type \( \text{fun}_y(\lambda \theta_y, \text{fun}_z(\lambda \theta_z, \theta_y)) = \{ [], U \in A \in \text{Rep U} \} \). The \textit{Uses} component is empty because \( \hat{\lambda} y.\hat{\lambda} x.\hat{\lambda} z.y \) has no free variables, and \( k \) ... will add \( [k \mapsto U] \) as the single use. The \textit{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 \( U \in A \in \text{Rep U} \) as highlighted, adding the Uses from \( \rho_1(x_1) \) to \( \{ [x_1 \mapsto U], \text{Rep U} \} \) 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 \( U \) follows pointwise from the order \( A \sqcup U \), i.e., \( (\varphi_1 \sqcup \varphi_2)(x) \triangleq \varphi_1(x) \sqcup \varphi_2(x) \).

The analysis result \( [k \mapsto U, x_1 \mapsto U] \) 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 \textit{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 \textit{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 \textit{fun}
and \( app \). The former approximates a functional \((\lambda \theta. \ldots) : \text{AbsTy} \rightarrow \text{AbsTy}\) into a finite \(\text{AbsTy}\), 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 \( k = (\lambda y. \lambda z. y) \) 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 \( \text{AbsTy} \) for \( k \) into module \( A \)’s signature file. Do note that this would not have been possible for the functional \((\lambda \theta_y. \lambda \theta_z. \theta_y) : \text{AbsTy} \rightarrow \text{AbsTy} \rightarrow \text{AbsTy}\) 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 \textit{inter-module} compilation, they enable efficient \textit{intra-module} compilation for \textit{compositional} static analyses such as \( \mathcal{A}[\_] \)\(^3\). Compositionality implies that \( \mathcal{A}[f = \lambda x. e \text{big} \text{ in } f f f f] \) is a function of \( \mathcal{A}[\lambda x. e \text{big}] \), itself a function of \( \mathcal{A}[e \text{big}] \). 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 \( \mathcal{A}[e \text{big}] \) at every use site of \( f \). Thus, it is necessary to summarise \( \mathcal{A}[\lambda x. e \text{big}] \) into a finite \(\text{AbsTy}\), rather than to call the inline expansion of type \( \text{AbsTy} \rightarrow \text{AbsTy} \) 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 \( \mathcal{A}[e]_\rho = (\varphi, \zeta) \) and \( \varphi(x) = A \), then \( x \) is absent in \( e \).

What are the main obstacles to prove it? As the first step, we must define what absence \textit{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 \((\text{let } x = e’ \text{ in } e, \rho, \mu, \kappa) \leftarrow^* \text{ Look}(x) \rightarrow^* \ldots \) 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). \( \mathcal{A}[e]_\rho[x \rightarrow \rho(y)] \subseteq \mathcal{A}[\lambda x. e]_\rho \).

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 \( \text{fun} \) and \( \text{app} \) 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 \( id \subseteq \text{app} \circ \text{fun} \), one half of a Galois connection. The other half \( \text{fun} \circ \text{app} \subseteq id \) is eta-expansion.

\(^5\)A “mundane approach” according to Nielson et al. [1999, Section 4.1], applicable to \textit{trace properties}, but not to \textit{hyperproperties} [Clarkson and Schneider 2010].

generalisation of $A[\_\_]$ 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., $A[\_\_]$) 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 [Nielsen 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., $A[\_\_]$) 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. 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 $\sigma_1 \rightsquigarrow \sigma_2$ preserves analysis information in step (3), we noticed that we actually got stuck in the $\text{Upd}$ 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 $\text{EUpd}$ 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 semantics...
Addresses $a \in \text{Addr} \approx \mathbb{N}$, States $\sigma \in S = \text{Exp} \times \text{E} \times \mathbb{H} \times \mathbb{K}$, Environments $\rho \in \mathbb{E} = \text{Var} \rightarrow \text{Addr}$, Heaps $\mu \in \mathbb{H} = \text{Addr} \rightarrow \text{Var} \times \mathbb{E} \times \text{Exp}$, Continuations $\kappa \in \mathbb{K} \equiv \text{stop} \mid \text{ap}(a) \cdot \kappa \mid \text{sel}(\rho, K \backslash x^e) \rightarrow e \cdot \kappa \mid \text{upd}(a) \cdot \kappa$

\[
\begin{array}{ll}
\text{Rule} & \sigma_1 \leftarrow \sigma_2 \\
\text{LET}_1 & (\text{let } x = e_1 \text{ in } e_2, \rho, \mu, \kappa) \rightarrow (e_2, \rho', \mu[a \mapsto (x, \rho', e_1)], \kappa) \quad a \notin \text{dom}(\mu), \rho' = \rho[x \mapsto a]\\
\text{APP}_1 & (e \times, \rho, \mu, \kappa) \rightarrow (e, \rho, \mu, \text{ap}(a) \cdot \kappa) \quad a = \rho(x)\\
\text{CASE}_1 & (\text{case } e_2 \text{ of } K \rightarrow e_3, \rho, \mu, \kappa) \rightarrow (e_3, \rho, \mu, \text{sel}(\rho, K \rightarrow e_3) \cdot \kappa)\\
\text{LOOK}(y) & (x, \rho, \mu, \kappa) \rightarrow (e, \rho', \mu, \text{upd}(a) \cdot \kappa) \quad a = \rho(x), (y, \rho', e) = \mu(a)\\
\text{APP}_2 & (\hat{x} \times e, \rho, \mu, \text{ap}(a) \cdot \kappa) \rightarrow (e, \rho[x \mapsto a], \mu, \kappa)\\
\text{CASE}_2 & (K', \mu, \mu, \text{sel}(\rho', K \rightarrow e) \cdot \kappa) \rightarrow (e_3, \rho'[\hat{x} \mapsto a], \mu, \kappa) \quad K_i = K', a = \rho(y)\\
\text{UPD} & (v, \rho, \mu, \text{upd}(a) \cdot \kappa) \rightarrow (v, \rho, \mu[a \mapsto (x, \rho, v)], \kappa) \quad \mu(a) = (x, \_\_\_) \end{array}
\]

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

For these reasons, we set out to find a \textbf{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 $A[]$, as suggested above) demonstrates that we can \textbf{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 \textit{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 Case1 and Case2 in this section because we do not consider data types for now.

The configurations $\sigma$ in this transition system resemble abstract machine states, consisting of a control expression $e$, an environment $\rho$ mapping lexically-scoped variables to their current heap address, a heap $\mu$ listing a closure for each address, and a stack of continuation frames $\kappa$. 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 Let1, Look and Upd rules.

The notation $f \in A \rightarrow B$ used in the definition of $\rho$ and $\mu$ denotes a finite map from $A$ to $B$, a partial function where the domain $\text{dom}(f)$ is finite and $\text{rng}(f)$ denotes its range. The literal

\textit{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.}
notion \([a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]\) denotes a finite map with domain \([a_1, \ldots, a_n]\) 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 \(\text{init}(e) = (e, [], [], \text{stop})\) and the final machine states are of the form \((\nu, \ldots, \text{stop})\). We bake into \(\sigma \in \Sigma\) the simplifying invariant of well-addressedness: Any address a occurring in \(\rho, \kappa\) or the range of \(\mu\) must be an element of \(\text{dom}(\mu)\). 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 \(\rho\).

We conclude with two example traces. The first one evaluates \(\text{let } i = \lambda x.x\ in\ i\):

\[
\begin{align*}
\text{let } i = \lambda x.x \text{ in } i &\quad \xrightarrow{\text{let} i} \quad (i, i, \rho_1, \text{stop}) \\
(\lambda x.x, \rho_1, \mu, \text{upd}(a_1) \cdot \kappa) &\quad \xrightarrow{\text{App}_1} \quad (\lambda x.x, \rho_1, \mu, \kappa) \\
(\lambda x.x, \rho_1, \mu, \text{upd}(a_1) \cdot \text{stop}) &\quad \xrightarrow{\text{App}_2} \quad (\lambda x.x, \rho_1, \mu, \kappa)
\end{align*}
\]

(1)

where \(\kappa = \text{ap}(a_1) \cdot \text{stop}\), \(\rho_1 = [i \mapsto a_1]\), \(\rho_2 = [i \mapsto a_1, x \mapsto a_1]\), \(\mu = [a_1 \mapsto (i, \rho_1, \lambda x.x)]\)

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

\[
\begin{align*}
\text{let } i = \lambda y.\lambda x.x \text{ in } i &\quad \xrightarrow{\text{let} i} \quad (i, i, \rho_1, \mu_1, \text{stop}) \\
(\lambda y.\lambda x.x, \rho_1, \mu_1, \text{upd}(a_1) \cdot \kappa_2) &\quad \xrightarrow{\text{App}_1} \quad (\lambda y.\lambda x.x, \rho_1, \mu_1, \kappa_2) \\
(\lambda y.\lambda x.x, \rho_1, \mu_1, \text{upd}(a_1) \cdot \text{stop}) &\quad \xrightarrow{\text{App}_2} \quad (\lambda y.\lambda x.x, \rho_1, \mu_1, \kappa_2)
\end{align*}
\]

(2)

where \(\rho_1 = [i \mapsto a_1]\), \(\rho_2 = [i \mapsto a_1, y \mapsto a_1]\), \(\mu_1 = (\rho_1, (i, \lambda y.\lambda x.x) \ i), \mu_2 = [a_1 \mapsto (i, \rho_2, \lambda x.x)], \kappa_1 = \text{ap}(a_1) \cdot \text{stop}, \kappa_2 = \text{upd}(a_1) \cdot \kappa_1\)

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 \(D\) comprises semantic values such as base values (integers, strings, etc.) and functions \(D \rightarrow D\). 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.

Abstracting Denotational Interpreters

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. 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.

11If 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].
However, to derive both dynamic semantics and static analysis as instances of the same

\(\mathcal{S}[\_\_] : (\)Trace \(d\), Domain \(d\), HasBind \(d\))

\[\Rightarrow \text{Exp} \rightarrow (\text{Name} \rightarrow d) \rightarrow d\]

\(\mathcal{S}[\_\_]_\rho = \text{case e of}

\begin{align*}
\text{Var} x & \mid x \in \text{dom } \rho \rightarrow \rho ! x \\
\text{Lam} x \text{ body} & \rightarrow \text{fun} x \$ \lambda d \rightarrow \\
\text{step App}_2 (\mathcal{S}[\text{body}]_(\rho|_x \rightarrow d)) \\
\text{App} e x & \mid x \in \text{dom } \rho \rightarrow \text{step App}_1 \$
\end{align*}

\begin{align*}
\text{apply} (\mathcal{S}[e]_\rho) (\rho ! x) \\
\text{| otherwise} & \rightarrow \text{stuck}
\end{align*}

\begin{align*}
\text{ConApp} k x & \\
\text{all} (\in \text{dom } \rho) x, \text{length } xs & \equiv \text{conArity } k \\
\rightarrow \text{con } k (\text{map } (\rho !)) \text{ xs} \\
\text{| otherwise} & \rightarrow \text{stuck}
\end{align*}

\begin{align*}
\text{Case } e \text{ alts} & \rightarrow \text{step Case}_1 \$
\text{select} (\mathcal{S}[e]_\rho) (\text{cont } \triangleleft \text{alts}) \\
\text{where}
\begin{align*}
\text{cont} \ (xs, e_r) \ ds & \mid \text{length } xs \equiv \text{length } ds \\
\text{= step Case}_2 (\mathcal{S}[e]_\rho| _x \rightarrow \text{dom } d) \\
\text{| otherwise} & \rightarrow \text{stuck}
\end{align*}

\begin{align*}
\text{fun } \_ f & \Rightarrow \text{return} \ (\text{Fun } f) \\
\text{apply } d \ a & \Rightarrow d \Rightarrow \lambda v \rightarrow \text{case } v \text{ of} \\
\text{Fun } f & \Rightarrow f \ a; \_ \Rightarrow \text{stuck} \\
\text{select } dv \ a & \Rightarrow d \Rightarrow \lambda v \rightarrow \text{case } v \text{ of} \\
\text{Con } k \ ds & \Rightarrow k \in \text{dom } \text{alts} \rightarrow (\text{alts }! k) \ ds \\
\Rightarrow & \rightarrow \text{stuck}
\end{align*}

\begin{align*}
\text{bind } rhs \text{ body} & \Rightarrow \text{let } d = rhs \ d \text{ in } \text{body } d
\end{align*}

\begin{itemize}
\item[(a)] Interface of traces and values
\item[(b)] Concrete by-name semantics for \(D_{na}\)
\end{itemize}

Fig. 5. Abstract Denotational Interpreter

\(\mathcal{S}[\_\_] : (\)Trace \(d\), Domain \(d\), HasBind \(d\)) \Rightarrow \text{Exp} \rightarrow (\text{Name} \rightarrow d) \rightarrow d\)
We have parameterised the semantic domain $d$ over three type classes \texttt{Trace}, \texttt{Domain} and \texttt{HasBind}, whose signatures are given in Figure 5a.\footnote{One can think of these type classes as a fold-like final encoding \cite{Carette2007} of a domain. However, the significance is in the decomposition of the domain, not the choice of encoding.} 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 $S[\_\_\_\_\_]$ together with instances for domain $D_{\text{na}}$ that we introduced in Section 4.1. Together this is enough to actually run the denotational interpreter to produce traces. We use $\texttt{read :: String \rightarrow Exp}$ as a parsing function, and a \texttt{Show} instance for $D\tau$ that displays traces. For example, we can evaluate the expression $\texttt{let i = \lambda x.x in i}$ like this:

\[
\lambda> S[\texttt{"let i = \lambda x.x in i\"}]_\varepsilon :: D_{\text{na}}
\]

\[
\text{LET}_1 \leftarrow \text{APP}_1 \leftarrow \text{LOOK}(i) \leftarrow \text{APP}_2 \leftarrow \text{LOOK}(i) \leftarrow (\lambda),
\]

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

The definition of $S[\_\_\_\_\_\_]$ given in Figure 5, is by structural recursion over the input expression. For example, to get the denotation of $\texttt{Lam \ x \ \text{body}}$, we must recursively invoke $S[\_\_\_\_\_\_]$ on \texttt{body}, extending the environment to bind $x$ to its denotation. We wrap that body denotation in \texttt{step APP}_2, to prefix the trace of \texttt{body} with an \texttt{APP}_2 event whenever the function is invoked, where \texttt{step} is a method of class \texttt{Trace}. Finally, we use \texttt{fun} to build the returned denotation; the details necessarily depend on the \texttt{Domain}, so \texttt{fun} is a method of class \texttt{Domain}. While the lambda-bound $x :: \texttt{Name}$ passed to \texttt{fun} is ignored in in the \texttt{Domain} $D_{\text{na}}$ 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 \texttt{HasBind} type class defines a particular evaluation strategy, as we shall see in Section 4.3. The \texttt{bind} method of \texttt{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 \texttt{bind} given in Figure 5b computes a $d$ such that $d = \text{rhs} \ d$ and passes the recursively-defined $d$ to \texttt{body}.\footnote{Such a $d$ corresponds to the guarded fixpoint of \texttt{rhs}. Strict languages can define this fixpoint as $d() = \text{rhs}(d())$.} 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 \texttt{bind} instead of calling $\texttt{body}$ immediately.

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

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

\[
\text{LET}_1 \leftarrow \text{LOOK}(x) \leftarrow \text{LOOK}(x) \leftarrow \text{LOOK}(x) \leftarrow \ldots
\]

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

\[
\text{LET}_1 \leftarrow \text{APP}_1 \leftarrow \text{LOOK}(w) \leftarrow \text{APP}_2 \leftarrow \text{APP}_1 \leftarrow \text{LOOK}(w) \leftarrow \text{APP}_2 \leftarrow \text{APP}_1 \leftarrow \text{LOOK}(w) \leftarrow \ldots
\]
The reason \( S[\_\_] \) is productive is due to the coinductive nature of \( T \)'s definition in Haskell.\(^{14}\)

Productivity requires that the monadic bind operator \((\_\_\_\_\_)\) 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 \( \bot \):

\[
\lambda x. S[\text{read} \ "let zro = Z() in let one = S(zro) in case one of \{ S(z) -> z \}"_\rho] :: D_{na}
\]

\[
\text{LET}_1 \leftarrow \text{LET}_1 \leftarrow \text{CASE}_1 \leftarrow \text{LOOK}(\text{one}) \leftarrow \text{CASE}_2 \leftarrow \text{LOOK}(\text{zro}) \leftarrow (\text{Con}(\text{Z}))
\]

\[
\lambda x. S[\text{read} \ "let zro = Z() in zro zro"_\rho] :: D_{na}
\]

\[
\text{LET}_1 \leftarrow \text{APP}_1 \leftarrow \text{LOOK}(\text{zro}) \leftarrow (\bot)
\]

### 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 \( \tau \) inherits its Monad and Trace instance from \( \tau \) and in reminiscence of Darais et al. [2015]. The old \( D_{na} \) can be recovered as \( D \) (ByName \( T \)) and we refer to its interpreter instance as \( S_{\text{name}}[\_]_\rho \).

#### 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 \( \theta \) such that \( D \theta \) accommodates state? We certainly cannot perform the heap update by updating entries in \( \rho \), because those entries are immutable once inserted, and we do not want to change the generic interpreter. That rules out \( \theta \equiv T \) (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 \( \mu \) 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 \( D \theta \equiv \text{Heap} \rightarrow T \) (Value \( \theta \), 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).
Our trace transformer \( \text{ByNeed} \) in Figure 7 solves this type equation via \( \theta \triangleq \text{ByNeed} \; T \). It embeds a standard state transformer monad,\(^{15}\) whose key operations \( \text{get} \) and \( \text{put} \) are given in Figure 7.

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

The key part is again the implementation of \text{HasBind} for \( D \; (\text{ByNeed} \; \tau) \), because that is the only place where thunks are allocated. The implementation of \text{bind} designates a fresh heap address \( a \) to hold the denotation of the right-hand side. Both \textit{rhs} and \textit{body} are called with \text{fetch} \( a \), a denotation that looks up \( a \) in the heap and runs it. If we were to omit the \text{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 \text{Name} and \text{Value}. The whole purpose of the \text{memo} \( a \; d \) combinator then is to \textit{memoise} the computation of \( d \) the first time we run the computation, via \textit{fetch} \( a \) in the \textit{Var} case of \( S_{\text{need}}[\_\_](\_\_). \) So \text{memo} \( a \; d \) yields from \( d \) until it has reached a value, and then \textit{updates} the heap after an additional Update step. Repeated access to the same variable will run the replacement \text{memo} \( a \) \text{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 \text{StateT} \; (\text{Heap} \; (\text{ByNeed} \; \tau)) \; \tau \; [\text{Blöndal et al. 2018}].

\(^{16}\)More serious semantics would omit updates after the first evaluation as an optimisation, i.e., update with \( \mu[a \mapsto \text{return} \; v] \), but doing so complicates relating the semantics to Figure 2, where omission of update frames for values behaves differently.
\[ S_{\text{value}}[\cdot]_\rho = S[\cdot]_\rho : D (\text{ByValue } T) \]

newtype ByValue \( \tau \) \( v \) = ByValue \{ unByValue :: \tau \ v \}  

instance Monad \( \tau \) \( => \) Monad (ByValue \( \tau \)) where ...

instance Trace (\( \tau \) \( v \)) \( => \) Trace (ByValue \( \tau \) \( v \)) where ...

class Extract \( \tau \) where getValue :: \( \tau \) \( v \) \(-\rightarrow\) \( v \)

instance Extract T where getValue (Ret \( v \)) = \( v \); getValue (Step _ \( \tau \)) = getValue \( \tau \)

instance (Trace (D (ByValue \( \tau \))), Monad \( \tau \), Extract \( \tau \)) \( => \) HasBind (D (ByValue \( \tau \))) where

\[ \text{bind } rhs \text{ body} = \text{step Let}_0 \text{ (do } v_1 \leftarrow d; \text{ body (return } v_1)) \]

\[ \text{where } d = rhs \text{ (return } v) \]

\[ \quad \text{:: } D \text{ (ByValue } \tau) \]

\[ \quad v = \text{getValue (unByValue } d) \text{ :: } \text{Value } \text{(ByValue } \tau) \]

Fig. 8. Call-by-value

ByNeed, without modifying the main \( S[\cdot]_\rho \) function at all. In doing so, we provide the simplest encoding of a denotational by-need semantics that we know of.\(^\text{17}\)

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

\( \lambda> S_{\text{need}}[\text{read "let } i = (\lambda y. \lambda x. x) \ i \ i \text{ in } i \ i\text{"}]_{\epsilon} : T \text{ (Value } _\rho \text{ Heap } _\rho) \)

\( \text{LET}_1 \leftrightarrow \text{APP}_1 \leftrightarrow \text{LOOK}(i) \leftrightarrow \text{APP}_1 \leftrightarrow \text{APP}_2 \leftrightarrow \text{UPD} \leftrightarrow \text{APP}_2 \leftrightarrow \text{LOOK}(i) \leftrightarrow \text{UPD} \leftrightarrow ((\lambda, [0\rightarrow\_])) \)

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 \( d : D \text{ (ByValue } \tau) \) of the right-hand side by mutual recursion with \( v : \text{Value } \text{(ByValue } \tau) \) 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 \( v_1 \leftarrow d; \text{ body (return } v_1) \) yields steps from the right-hand side \( d \) until its value \( v_1 : \text{Value } \text{(ByValue } \tau) \) is reached, which is then passed returned (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 \( \tau = T \). Then return = Ret and we get \( d = rhs \text{ (Ret } v) \) for the value \( v \) at the end of the trace \( d \), as computed by the type class instance method getValue :: T \( v \) \(-\rightarrow\) \( v \).\(^\text{18}\) The effect of Ret (getValue (unByValue \( d \))) is that of stripping all Steps from \( d \).\(^\text{19}\)

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 \( i = (\lambda y. \lambda x. x) \ i \ i \) 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 \( \tau \).

\(^{19}\)We could have defined \( d \) as one big guarded fixpoint fix (\( rhs \circ \text{return } \circ \text{getValue } \circ \text{unByValue } d \)), but some co-authors prefer to see the expanded form.
\[ S_{\text{vinit}}[e]_{\rho}(\mu) = \text{unByVInit} \left( S[e]_{\rho} : D \left( \text{ByVInit} \left( T \right) \right) \right) \mu \]

newtype ByVInit \( \tau \) \( v \) = ByVInit \{ \text{unByVInit} :: \text{Heap} (\text{ByVInit} \left( \tau \right)) \rightarrow \tau \ (v, \text{Heap} (\text{ByVInit} \left( \tau \right))) \}

instance (Monad \( \tau \), \( \forall \)v. Trace (\( \tau \) \( v \))) \Rightarrow \text{HasBind} (D (\text{ByVInit} \left( \tau \right))) where

\( \text{bind} \ \text{rhs} \ \text{body} = \text{do} \ \mu \leftarrow \text{get} \)

\[ \begin{align*}
\text{let} \ a &= \text{nextFree} \ \mu \\
\text{put} \ \mu[a &\mapsto \text{stuck}] \\
\text{step} \ \text{Let}_0 \ (\text{memo} \ a \ (\text{rhs} \ (\text{fetch} \ a))) &\Rightarrow \text{body} \circ \text{return}
\end{align*} \]

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

\[ S_{\text{clair}}[e]_{\rho} = \text{runClair} \ S[e]_{\rho} : T \left( \text{Value} \ (\text{Clairvoyant} \left( T \right)) \right) \]

data \ Fork \ f \ a = \text{Empty} \mid \text{Single} \ a \mid \text{Fork} (f \ a) (f \ a); \text{data} \ ParT \ m \ a = \text{ParT} \ (m \ (\text{Fork} \ (\text{ParT} \ m) \ a))

instance Monad \( \tau \) \Rightarrow \text{Alternative} (ParT \left( \tau \right)) where

\( \text{empty} = \text{ParT} \ (\text{pure} \ \text{Empty}); l \{\} r = \text{ParT} \ (\text{pure} \ (\text{Fork} \ l \ r)) \)

newtype Clairvoyant \( \tau \) \( a \) = Clairvoyant (ParT \left( \tau \right) \ a)

runClair :: D (Clairvoyant \left( T \right)) \rightarrow T \left( \text{Value} \ (\text{Clairvoyant} \left( T \right)) \right)

instance (Extract \( \tau \), Monad \( \tau \), \( \forall \)v. Trace (\( \tau \) \( v \))) \Rightarrow \text{HasBind} (D (\text{Clairvoyant} \left( \tau \right))) where

\( \text{bind} \ \text{rhs} \ \text{body} = \text{Clairvoyant} \ (\text{skip} \{\} \ \text{let}' \) \( \Rightarrow \text{body} \)

\( \text{where} \ \text{skip} = \text{return} \ (\text{Clairvoyant} \ \text{empty}) \)

\( \text{let}' = \text{fmap} \ \text{return} \ S[\text{Let}_0 \ S ... \ \text{fix} ... \ \text{rhs} ... \ \text{getValue} ... \) \]

Fig. 10. Clairvoyant Call-by-value

\( \lambda > S_{\text{value}}[\text{read} \ "\text{let} \ i = (\lambda y. \lambda x. x) \ i \ \text{in} \ i \ i"], \)

\( \text{LET}_0 \leftarrow \text{APP}_1 \leftarrow \text{APP}_2 \leftarrow \text{LET}_1 \leftarrow \text{APP}_1 \leftarrow \text{LOOK}(i) \leftarrow \text{APP}_2 \leftarrow \text{LOOK}(i) \leftarrow (\lambda) \)

The beta reduction of \((\lambda y.\lambda x. x) \ i\) now happens once within the \text{LET}_0/\text{LET}_1 bracket; the two subsequent \text{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.,

\( \lambda > \text{takeT} \ 5 \ S_{\text{value}}[\text{read} \ "\text{let} \ x = x \ \text{in} \ x \ x"], \)

\( \text{LET}_0 \leftarrow \text{LOOK}(x) \leftarrow \text{LET}_1 \leftarrow \text{APP}_1 \leftarrow \text{LOOK}(x) \leftarrow ^{\text{CInt}}\text{Interrupted} \)

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 \text{bind} as in Figure 9.

\( \lambda > S_{\text{vinit}}[\text{read} \ "\text{let} \ x = x \ \text{in} \ x \ x"],(\epsilon) :: T \left( \text{Value} \_\_ \text{Heap} \_\_ \right) \)

\( \text{LET}_0 \leftarrow \text{LOOK}(x) \leftarrow \text{LET}_1 \leftarrow \text{APP}_1 \leftarrow \text{LOOK}(x) \leftarrow \langle \langle \epsilon, [\_\_\_\_] \rangle \rangle \)
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 absorb 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:

\[
\lambda \triangleright T_{\text{clair}} \left[ \text{read } \lambda x.x \text{ in } \text{let } g = \lambda y.f \text{ in } g^\prime \right] \vdash T \text{ (Clairvoyant T)}
\]

\[
\text{LET}_1 \leftrightarrow \text{LET}_0 \leftrightarrow \text{LET}_1 \leftrightarrow \text{LOOK}(g) \leftrightarrow \langle \lambda \rangle
\]

\[
\lambda \triangleright T_{\text{clair}} \left[ \text{read } \lambda x.x \text{ in } \text{let } g = \lambda y.f \text{ in } g^\prime g^\prime \right] \vdash T \text{ (Clairvoyant T)}
\]

\[
\text{LET}_0 \leftrightarrow \text{LET}_1 \leftrightarrow \text{LET}_0 \leftrightarrow \text{LET}_1 \leftrightarrow \text{APP}_1 \leftrightarrow \text{LOOK}(g) \leftrightarrow \text{APP}_2 \leftrightarrow \text{LOOK}(f) \leftrightarrow \langle \lambda \rangle
\]

The first example discards \( f \), but the second needs it, so the trace starts with an additional \( \text{LET}_0 \) 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 \( \text{LET}_0 \) is needed can be delayed for an infinite amount of time, as exemplified by

\[
\lambda \triangleright T_{\text{clair}} \left[ \text{read } \lambda i = Z() \text{ in } \text{let } w = \lambda y.y \text{ in } w \text{ w}^\prime \right] \vdash T \text{ (Clairvoyant T)}
\]

\[\wedge \text{Interrupted}\]

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 \( S_{\text{need}}[\_ \_] \) produces small-step traces of the lazy Krivine machine and is indeed a denotational semantics.\(^{20}\) Excitingly, to our knowledge, \( S_{\text{need}}[\_ \_] \) 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 \( S_{\text{need}}[\_ \_] \)

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

\[
\alpha(\text{init}(e) \leftarrow \ldots) = S_{\text{need}}[e]_\epsilon(\epsilon),
\]

where \( \text{init}(e) \leftarrow \ldots \) denotes the maximal (i.e. longest possible) LK trace evaluating the closed expression \( e \). For example, for the LK trace (2), \( \alpha \) produces the trace at the end of Section 4.3.2.

It turns out that function \( \alpha \) 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, \( \tau \triangleq S_{\text{need}}[e]_\epsilon(\epsilon) \) the denotational by-need trace and \( \text{init}(e) \leftarrow \ldots \) the maximal lazy Krivine trace. Then

\[
\bullet \; \tau \text{ preserves the observable termination properties of } \text{init}(e) \leftarrow \ldots \text{ in the above sense.}
\]

\[\text{Similarly for } S_{\text{name}}[\_ \_] \text{ and } S_{\text{init}}[\_ \_](\_ \_).\]

\[\text{Should be derivative.}\]
• \( \tau \) preserves the length (i.e., number of steps) of init(\( e \)) \( \iff \) \( \ldots \) (i.e., number of transitions).

• every ev :: Event in \( \tau = \text{Step} ev \ldots \) corresponds to the transition rule taken in init(\( e \)) \( \iff \ldots \)

**Proof sketch.** Define \( \alpha \) by coinduction and prove \( \alpha(\text{init}(e) \iff \ldots) = S_{\text{need}}[e],\rho(e) \) by Löb induction. Then it suffices to prove that \( \alpha \) preserves the observable properties of interest. The full proof for a rigorous reformulation of this result can be found in the Appendix. \( \square \)

### 5.2 Totality of \( S_{\text{name}}[\cdot] \_ \) and \( S_{\text{need}}[\cdot] \_

**Theorem 5** (Totality). The interpreters \( S_{\text{name}}[e],\rho \) and \( S_{\text{need}}[e],\rho(\mu) \) are defined for every \( e, \rho, \mu \).

**Proof sketch.** In the Supplement, we provide an implementation of the generic interpreter \( S[\cdot] \_ \) and its instances at ByName and ByNeed in Guarded Cubical Agda, which offers a total type theory with guarded recursive types Mögelberg and Veltri [2019]. Agda enforces that all encodable functions are total, therefore \( S_{\text{name}}[\cdot] \_ \) and \( S_{\text{need}}[\cdot] \_ \) 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. \( \square \)

### 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 \( \forall \alpha_3. \text{option} (\alpha_3 \rightarrow \alpha_3) \). Polymorphic types act as summaries in the sense of the Introduction, and fixpoints are solved via unification.

- **Appendix C.2** defines OCF\( A \) 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 test suite. 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 \( \top \)

\[ \text{There is a small caveat: we did not try to optimise for compiler performance in our proof of concept and hence it regresses in a few compiler performance test cases. None of the runtime performance test cases regress and the inferred demand signatures stay unchanged.} \]
data U = U₀ | U₁ | Uω 

instance Domain_D_U where

instance HasBind_D_U where

\[ S_{usage}[\varepsilon]_ρ = S[\varepsilon]_ρ : D_U \]

\[ \text{bind rhs body} = \text{body (kleeneFix rhs)} \]

\[ \text{data Value_U = } U \uplus \text{ Value_U} | \text{Rep U} \]

\[ \text{type Uses = Name} \rightarrow U \]

\[ \text{class UVec a where} \]

\[ (+) :: a \rightarrow a \rightarrow a \]

\[ (*) :: U \rightarrow a \rightarrow a \]

\[ \text{instance UVec U where} \ldots \]

\[ \text{instance UVec Uses where} \ldots \]

\[ \text{data T_U v = (Uses, v)} \]

\[ \text{instance Trace (T_U v) where} \]

\[ \text{step (Lookup x) } \langle \varphi, v \rangle = \langle [x \mapsto U_1] + \varphi, v \rangle \]

\[ \text{step } \tau \rightarrow \tau = \tau \]

\[ \text{instance Monad T_U where} \]

\[ \text{return } a = \langle \varepsilon, a \rangle \]

\[ \langle \varphi_1, a \rangle \Rightarrow k = \text{let } \langle \varphi_2, b \rangle = k a \text{ in } \langle \varphi_1 + \varphi_2, b \rangle \]

Fig. 11. Summary-based usage analysis

in dynamic semantic domains such as Dna with a finite type such as T_U in Figure 11. A usage trace \( \langle \varphi, \text{val} \rangle :: T_U v \) is a pair of a value \( \text{val} :: v \) and a finite map \( \varphi :: \text{Uses} \), mapping variables to a usage \( U \).

The usage \( \varphi \sqsubseteq x \) assigned to \( x \) is meant to approximate the number of \( \text{Lookup } x \) events; \( U_0 \) means “at most 0 times”, \( U_1 \) means “at most 1 times”, and \( U_\omega \) means “an unknown number of times”. In this way, \( T_U \) is an abstraction of \( T \); it squashes all \( \text{Lookup } x \) events into a single entry \( \varphi \sqsubseteq x :: U \) and discards all other events.

Consider as an example the by-name trace evaluating \( e \triangleq \text{let } i = \bar{\lambda}x.x \text{ in } \text{let } j = \bar{\lambda}y.y \text{ in } i \ j \) : \[ \text{LET}_1 \leftrightarrow \text{LET}_1 \leftrightarrow \text{APP}_1 \leftrightarrow \text{APP}_1 \leftrightarrow \text{LOOK}(i) \leftrightarrow \text{APP}_2 \leftrightarrow \text{LOOK}(j) \leftrightarrow \text{APP}_2 \leftrightarrow \text{LOOK}(j) \leftrightarrow \langle \lambda \rangle \]

We would like to abstract this trace into \( \langle [i \mapsto U_1], j \mapsto U_\omega \rangle \ldots \). One plausible way to achieve this is to replace every \text{Step (Lookup } x \text{)} \ldots \) in the by-name trace with a call to \text{step (Lookup } x \text{)} \ldots \) from the \text{Trace } T_U instance in Figure 11, quite similar to \text{foldr step} on lists. The \text{step} implementation concatenates the usage of \( x \) whenever a \text{Lookup } x \text{ event occurs. The addition operation used to carry out incrementation is defined in type class instances UVec U and UVec Uses, together with scalar multiplication.}\footnote{We think that \text{UVec} models \text{U-modules}. It is not a vector space because \text{U} lacks inverses, but the intuition is close enough.}

For example, \( U_0 + u = u \) and \( U_1 + U_1 = U_\omega \) in \text{U}, as well as \( U_0 * u = U_0, U_\omega * U_1 = U_\omega \).

These operations lift to \text{Uses} pointwise, e.g., \([i \mapsto U_1] + (U_\omega * [j \mapsto U_1]) = [i \mapsto U_1, j \mapsto U_\omega] \).

Following through on the \text{foldr step} idea to abstract a \( T \) into \text{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 \text{Monad} instance on \text{U} mirroring trace concatenation and then running our interpreter at, e.g., \( D \text{ (ByName } T_U \text{)} \equiv T_U \text{ (Value } T_U \text{)} \) on
expression e from earlier:
\[ S[(\text{let } i = \tilde{l} x . x \text{ in let } j = \tilde{l} y . y \text{ in } i \ j)] = \{(i \mapsto U_1, j \mapsto U_\omega), \tilde{l}\} :: \text{D (ByName } T_U). \]

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 Value_U to replace Value_T_U.

### 6.2 Value Abstraction Value_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 Value_U to get the finite semantic domain D_U = T_U ∗ Value_U in Figure 11, and thus a static usage analysis S_usage[\_\_\_\_] when we instantiate S[\_\_] at D_U.

The definition of Value_U is just a copy of \( \varsigma \in \text{Summary in Figure 1} \) that lists argument usage \( U \) instead of Absence flags; the entire intuition transfers. For example, the Value_U summarising \( \tilde{l}y.\tilde{l}z.y = U_1 \uplus U_\omega : \text{Rep U_\omega} \), because the first argument is used once while the second is used 0 times. What we previously called absence types \( \theta \in \text{AbsTy} \) 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 \( S\text{usage}_{\_\_\_\_\_\_}[(\text{let } k = \tilde{l} y . \tilde{l} z . y \text{ in } k \ x_1 \ x_2)]_{\rho_e} = \{(k \mapsto U_1, x_1 \mapsto U_\omega), \text{Rep U_\omega}\} \), 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, \( S\text{usage}_{\_\_\_\_\_\_][(\text{let } i = \tilde{l} x . x \text{ in let } j = \tilde{l} y . y \text{ in } i \ j)]_{\rho_e} = \{(i \mapsto U_\omega, j \mapsto U_\omega), \text{Rep U_\omega}\} \) 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 \( \perp = (\epsilon, \text{Rep U_\omega}) \), the fun and apply functions play exactly the same roles as fun and apply in Figure 1. Let us briefly review how the summary for the right-hand side \( \tilde{l}x.x \) of \( i \) in the previous example is computed:

\[ S[\text{Lam } x (\text{Var } x)] = \text{fun } x (\lambda d \rightarrow \text{step } \text{App}_2 (S[\text{Var } x]_{\rho[x \mapsto d]})) = \text{case } \text{step } \text{App}_2 (S[\text{Var } x]_{\rho[x \mapsto \{x \mapsto U_1, \text{Rep U_\omega}\}]) \text{ of } (\varphi, v) \rightarrow (\varphi [x \mapsto U_\omega], \varphi \perp x \mapsto \text{Rep U_\omega}) = \text{case } \{x \mapsto U_1, \text{Rep U_\omega}\} \text{ of } (\varphi, v) \rightarrow (\varphi [x \mapsto U_\omega], \varphi \perp x \mapsto \text{Rep U_\omega}) = (\epsilon, U_1 \uplus \text{Rep U_\omega}) \]

The definition of fun x applies the lambda body to a proxy \( \{x \mapsto U_1, \text{Rep U_\omega}\} \) 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_\omega. In this case there is only a single use of x and the final usage \( \varphi \perp x = U_1 \) from the lambda body will be prepended to the summarised value. Occurrences of x must make do with the top value (Rep 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 \( \uplus \) instead of \( \sqcup \) to carry over U_1 + U_\omega = U_\omega, and an explicit peel to view a Value_U in terms of \( s \) (it is Rep \( u \equiv u \uplus \text{Rep } u \)). The usage \( u \) thus pelt from the value determines how often the actual argument was evaluated, and multiplying the uses of the argument \( \varphi_2 \) with \( u \) accounts for that.

\(^{23}\)As before, the exact identity of x is exchangeable; we use it as a De Bruijn level.

The example $S_{usage} \left[ \left( \text{let } z = Z() \text{ in case } S(z) \text{ of } S(n) \to n \right) \right] = (z \mapsto U_0), \text{Rep } U_0)$ 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 $Value_U$ intentionally simple, so our analysis assumes that every data constructor uses its fields many times.\footnote{It is clear how to do a better job at least for products; see Sergey et al. [2017].} This is achieved in con by repeatedly applying to the top value ($\text{Rep } U_0$), as if a data constructor was a lambda-bound variable. Dually, $\text{select}$ does not need to track how fields are used and can pass $\langle \varepsilon, \text{Rep } U_0 \rangle$ 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 $\text{fun } x :: (D_U \to D_U) \to D_U$ 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 $\text{HasBind}$ instance discussed next.

### 6.3 Finite Fixpoint Strategy in $\text{HasBind } D_U$ and Totality

The third and last ingredient to recover a static analysis is the fixpoint strategy in $\text{HasBind } D_U$, 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 $\text{let } d = \text{rhs } d \text{ in body } d$ in $\text{HasBind } D_{na}$ (Figure 5). At least for $S_{name}[-]_-$ and $S_{need}[-]_-$, 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 $\text{Step}$ constructor — and thus the $\text{step}$ 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 $\text{step}$ implementation of $T_U$ is not lazy, and hence the guarded fixpoint $\text{let } d = \text{rhs } d \text{ in body } d$ is not guaranteed to exist.

In general, finite trace types cannot have a lazy $\text{step}$ 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 $\text{kleeneFix}$ in Figure 12. $\text{kleeneFix}$ requires us to define an order on $D_U$, which is induced by $U_0 \sqsubseteq U_1 \sqsubseteq U_n$, in the same way that the order on $\text{AbsTy}$ in Section 2.2 was induced from the order $A \sqsubseteq U$ 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 $Value_U$ indeed contains such infinite chains, for example, $U_1 \sqsubseteq U_1 \sqsubseteq \ldots \sqsubseteq U_0$! This is easily worked around in practice by employing appropriate widening measures such as bounding the depth of $Value_U$. The resulting definition of $\text{HasBind}$ is safe for by-name and by-need semantics.\footnote{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.}
Abstracting Denotational Interpreters

Mono
\[ d_1 \sqsubseteq d_2 \quad f_1 \sqsubseteq f_2 \]
\[
\text{apply } f_1 \quad d_1 \sqsubseteq \text{apply } f_2 \quad d_2 \quad \text{and so on, for all methods of Trace, Domain, HasBind}
\]

Step-App
\[ \text{step ev (apply } d \ a) \sqsubseteq \text{apply (step ev } d \ a) \]

Step-Sel
\[ \text{step ev (select } d \ \text{alts) } \sqsubseteq \text{select (step ev } d \ \text{alts)}
\]

Unwind-Stuck
\[ \text{stuck } \sqsubseteq \bigcup \{\text{apply con } k \ ds, \text{select stuck } \text{alts}\}
\]

Intro-Stuck
\[ \text{stuck } \sqsubseteq \bigcup \{\text{apply con } k \ (\text{map } \rho_1 \ ys), \text{select (con } k \ (\text{map } \rho_1 \ ys) \ \text{alts)}\}
\]

Beta-App
\[ f \ d = \text{step App}_2 (S_D[e]\rho[x\mapsto d]) \]
\[ f \ a \sqsubseteq \text{apply (fun x f) } a \]

Beta-Sel
\[ (\text{alt ! } k) \ (\text{map } \rho_1 ! ) \ ys \sqsubseteq \text{select (con } k \ (\text{map } \rho_1 ! ) \ ys) \ \text{alts}
\]

Bind-ByName
\[ \text{rhs } d_1 = S_D[e_1]\rho[x\mapsto \text{lookup } x \ d_1] \quad \text{body } d_1 = \text{step Let}_1 (S_B[e_2]\rho[x\mapsto d_1])
\]

Step-Inc
\[ d \sqsubseteq \text{step ev } d \]

Update
\[ \text{step Update } d = d \]

Fig. 13. By-name and by-need abstraction laws for type class instances of abstract domain \( \hat{D} \)

7 GENERIC BY-NAME AND BY-NEED SOUNDNESS

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

\[ \text{abstract } (S_{\text{need }}[e]_\rho) \sqsubseteq S_B[e]_\rho. \]

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

We will instantiate the theorem at \( D_U \) in order to prove that usage analysis \( S_{\text{usage}}[e]_\rho = S_{D_U}[e]_\rho \) 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:

<table>
<thead>
<tr>
<th>MONO</th>
<th>STEP-APP</th>
<th>STEP-SEL</th>
<th>UNWIND-STUCK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intro-Stuck</td>
<td>Beta-App</td>
<td>Beta-Sel</td>
<td>Bind-ByName</td>
</tr>
</tbody>
</table>

abstract \((S_{\text{need}}[e]_\rho) \subseteq S[D][e]_\rho\).

In other words: prove the abstraction laws for an abstract domain \(\hat{D}\) of your choosing and we give you for free a proof of sound abstract by-need interpretation for the static analysis \(S[D][e]_\rho\).

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 \(\hat{D}\). The gist is as follows: \(abstract\) eliminates every \(\text{Step } \text{evt}\) in the by-need trace with a call to \(\text{step } \text{evt}\), and eliminates every concrete Value at the end of the trace with a call to the corresponding Domain method. That is, Fun turns into \(\text{fun}\), Con into con, and \(\text{Stuck}\) into \(\text{stuck}\), 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 \(\hat{D}\) 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 \(S[D][e]_\rho\) is an abstract interpreter that is sound wrt. \(abstract\), that is,

\[abstract \left( S_{\text{need}}[e]_\rho \right) \subseteq S[D][e]_\rho\]

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 \(\forall f\ a\ x.\ f\ a \subseteq \text{apply } (\text{fun } x\ f)\ a\) in the abstract domain \(\hat{D}\). This states that summarising \(f\) through \(\text{fun}\), then applying the summary to \(a\) must approximate a direct call to \(f\); it amounts to proving correct the summary mechanism. 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). \(S_{\text{usage}}[e]\rho[x\mapsto \rho!y] \subseteq S_{\text{usage}}[\text{Lam } x\ 'App' \ y]\rho\).

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

\[f\ a = \text{step } \text{App} \_2 (S[U][e]\rho[x\mapsto a]) = S[U][e]\rho[x\mapsto a] \subseteq S[U][\text{Lam } x\ 'App' \ y]\rho = \text{apply } (\text{fun } x\ f)\ a\.

Without the syntactic premise of Beta-App to rule out undefinable entities in \(D[U] \rightarrow D[U]\), the rule cannot be proved for usage analysis; we give a counterexample in the Appendix (Example 46).

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. 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” D (ByNeed T[U]), we would not need to show anything! Then apply return (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 $\alpha$. This is how our opinionated approach pays off: because both concrete semantics and $\alpha$ are known, the usual abstraction laws such as $\alpha (\text{apply } d a) \sqsubseteq \text{apply} (\alpha d) (\alpha a)$ further decompose into BETAB-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 $S_{\text{need}}[\_]$. 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 $S_{\text{need}}[E[\text{Let } x e' e]](e)$ contains a 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 ($S_{\text{usage}}[\_] - abstracts $S_{\text{need}}[\_]$). Let $e$ be a closed expression and abstract the abstraction function above. Then $\text{abstract } (S_{\text{need}}[e]) \subseteq S_{\text{usage}}[e]$.

Theorem 10 ($S_{\text{usage}}[\_]$ infers absence). Let $\rho_e \triangleq \{ y \mapsto \{ y \mapsto U_1, \text{Rep } U_0 \}\}$ be the initial environment with an entry for every free variable $y$ of an expression $e$. If $S_{\text{usage}}[e]_{\rho_e} = (\varphi, v)$ and $\varphi \notin x = U_0$, then $x$ is absent in $e$.

Proof sketch. If $x$ is used in $e$, there is a trace $S_{\text{need}}[E[\text{Let } x e e]](e)$ containing a Lookup $x$ event. The abstraction function abstract induced by $\text{DU}$ aggregates lookups in the trace into a $\varphi \exists y$ Uses, e.g., abstract $(\text{Look}(i) \leftrightarrow \text{Look}(x) \leftrightarrow \text{Look}(i) \leftrightarrow \langle \ldots \rangle) = \{ i \mapsto U_0, x \mapsto U_1, \ldots \}$. Clearly, it is $\varphi \exists x U_1$, because there is at least one Lookup $x$. Lemma 9 and a context invariance Lemma 38 prove that the computed $\varphi$ approximates $\varphi'$, so $\varphi \notin x \exists x U_1 \neq U_0$.

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 $\sigma$ 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 Reynolds’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 $k \leq 1$.

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 ($D_r$) instance depending on Monad $r$. 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 $\infty$-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.
Abstracting Denotational Interpreters

START OF APPENDIX

A PROOFS FOR SECTION 2 (THE PROBLEM WE SOLVE)

Theorem 1 (A[·] infers absence). If \( A[e]_{\rho_e} = \langle \varphi, \zeta \rangle \) and \( \varphi(x) = A \), 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 \( (\text{let } x = e' \text{ in } e, \rho, \mu, \kappa) \xrightarrow{\text{Look}(x)} ... \xrightarrow{\text{absent}} \) ... 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

\[
A[\text{let } x = e_1 \text{ in } e_2]_{\rho} = A[e_2]_{\rho[\{x\rightarrow \text{lpf}(\lambda \theta. \text{let } x = e_1 \text{ in } e_2, \rho_1, \rho(x, \theta))\}]}
\]

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

Abbreviation 11. The syntax \( \theta.\varphi \) for an AbsTy \( \theta = \langle \varphi, \zeta \rangle \) returns the \( \varphi \) component of \( \theta \). The syntax \( \theta.\zeta \) returns the \( \zeta \) component of \( \theta \).

Definition 12 (Abstract substitution). We call \( \varphi[x \mapsto \varphi'] \equiv \varphi[x \mapsto A] \sqcup (\varphi(x) \ast \varphi') \) the abstract substitution operation on \( \text{Uses and overload this notation for AbsTy, so that } (\langle \varphi, \zeta \rangle)[x \mapsto \varphi_y] \equiv \langle \varphi[x \mapsto \varphi_y], \zeta \rangle \).

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

Lemma 13. \( A[(\lambda x.e) y]_{\rho} = (A[e]_{\rho[\{x\rightarrow \text{Rep } U\}]})[x \mapsto \rho(y).\varphi] \).

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

\( (A[e]_{\rho}).\varphi(x) = A \).

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

Lemma 15. \( A[(\lambda x.\lambda y.e) z]_{\rho} = A[\lambda y.(\lambda x.e) z]_{\rho} \).

Proof.\[
A[(\lambda x.\lambda y.e) z]_{\rho} = (\text{fun} \ (\lambda \theta_y.\ A[e]_{\rho[\{x\rightarrow \text{Rep } U\}, y\rightarrow \theta_y]}))[x \mapsto \rho(z).\varphi] \]
\[
\begin{aligned}
\text{Unfold } A[\cdot], \text{Lemma 13} \\
\rho(z)(y) = A \text{ by Lemma 14, } x \neq y \neq z \\
\text{Refold } A[\cdot]
\end{aligned}
\]

Lemma 16. \( A[(\lambda x.e) y z]_{\rho} = A[(\lambda x.e z) y]_{\rho} \).

Proof. \[
A[(\lambda x.e) y z]_{\rho} = \text{app}(A[e]_{\rho[\{x\rightarrow \text{Rep } U\}]})[x \mapsto \rho(y).\varphi](\rho(z)) \]
\[
\begin{aligned}
\text{Unfold } A[\cdot], \text{Lemma 13} \\
\rho(z)(x) = A \text{ by Lemma 14, } y \neq x \neq z \\
\text{Refold } A[\cdot]
\end{aligned}
\]

Lemma 17. \( A[\text{let } z = (\lambda x.e_1) y \text{ in } (\lambda x.e_2) y]_{\rho} = A[(\lambda x.\text{let } z = e_1 \text{ in } e_2) y]_{\rho} \).

Lemma 3 (Substitution). \( A[e_1 \rightarrow \lambda x. e_2] \sqsubseteq A[(\lambda x. e_1) y] \).

Proof. By induction on \( e_1 \).

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

  \[
  A[z]_{\rho[x \rightarrow \rho(y)]} = A[z]_{\rho[x \rightarrow (U \rightarrow U) \mapsto \varphi]} = (A[z]_{\rho[x \rightarrow (U \rightarrow U) \mapsto \varphi]})[x \mapsto \rho(y) \cdot \varphi] = (A[\lambda x. z]_{\rho} \mapsto A[(\lambda x. z) y]_{\rho})
  \]

  Otherwise, we have \( x = z \), thus \( \rho(x) = (\mapsto V) \cdot \varphi \), and thus

  \[
  A[z]_{\rho[x \rightarrow \rho(y)]} = A[z]_{\rho[x \rightarrow \varphi]} = (A[z]_{\rho[x \rightarrow \varphi]})[x \mapsto \rho(y) \cdot \varphi] = (A[z]_{\rho[x \rightarrow \mapsto V] \cdot \varphi})[x \mapsto \rho(y) \cdot \varphi] = (A[\lambda x. z]_{\rho} \mapsto A[(\lambda x. z) y]_{\rho})
  \]

  **Case** \( \lambda z.e' \):

  \[
  A[\lambda z.e']_{\rho[x \rightarrow \rho(y)]} = \text{fun}_x \cdot \lambda \theta_2. A[e']_{\rho[y \rightarrow \theta_2, x \rightarrow \rho(y)]} \sqsubseteq A[(\lambda x.e') y]_{\rho[x \rightarrow \theta_2, y \mapsto \rho(y)]} \sqsubseteq A[(\lambda x.e') y]_{\rho[\theta_2, y \mapsto \rho(y)]} = (A[\lambda z.(\lambda x.e') y]_{\rho[\theta_2, y \mapsto \rho(y)]})[x \mapsto \rho(y) \cdot \varphi] = (A[\lambda z.(\lambda x.e') y]_{\rho})[x \mapsto \rho(y) \cdot \varphi] = (A[\lambda z.(\lambda x.e') y]_{\rho})[x \mapsto A[(\lambda x.e') y]_{\rho}] = (A[\lambda z.e]_{\rho})[x \mapsto \rho(y) \cdot \varphi] = A[(\lambda z.e) y]_{\rho}
  \]
• **Case** $e': z$: When $x = z$:

\[
\mathcal{A}[e' z]_{\rho[x \mapsto \rho(y)]} = \text{app}(\mathcal{A}[e']_{\rho[x \mapsto \rho(y)]})(\rho(y)) \\
\subseteq \text{app}(\mathcal{A}[\lambda x.e']_{\rho[y]})(\rho(y)) \\
= \mathcal{A}[\lambda x.e'] y_{\rho} \\
= \mathcal{A}[\lambda x.e'] y_{\rho} \\
= \mathcal{A}[\lambda x.e'] y_{\rho} \\
= x = z
\]

When $x \neq z$:

\[
\mathcal{A}[e' z]_{\rho[x \mapsto \rho(y)]} = \text{app}(\mathcal{A}[e']_{\rho[x \mapsto \rho(y)]})(\rho(z)) \\
\subseteq \text{app}(\mathcal{A}[\lambda x.e']_{\rho[y]})(\rho(z)) \\
= \mathcal{A}[\lambda x.e'] y_{\rho} \\
= \mathcal{A}[\lambda x.e'] y_{\rho} \\
= \mathcal{A}[\lambda x.e'] y_{\rho}
\]

• **Case** let $z = e_1$ in $e_2$:

\[
\mathcal{A}[\text{let } z = e_1 \text{ in } e_2]_{\rho[x \mapsto \rho(y)]} = \mathcal{A}[e_2]_{\rho[x \mapsto \rho(y)]} \text{lfp}(\lambda \theta. \mathcal{A}[e_1]_{\rho[x \mapsto \rho(y)]}) \\
\subseteq \mathcal{A}[\lambda x.e_2] y_{\rho[\text{lfp}(\lambda \theta. \mathcal{A}[e_1]_{\rho[x \mapsto \rho(y)]})]} \\
= \mathcal{A}[\text{let } z = \lambda x.e_1 \text{ in } \lambda x.e_2] y_{\rho} \\
= \mathcal{A}[\lambda x.\text{let } z = \lambda x.e_1 \text{ in } e_2] y_{\rho}
\]

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

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

\[
\mathcal{A}[e]_{\rho} = (\mathcal{A}[e]_{\rho_\Delta})[x \mapsto \rho(x).\varphi]
\]

**Proof.** By induction on $e$.

• **Case** $e = y$: We assert $\mathcal{A}[y]_{\rho} = \rho(y) = \rho_\Delta(y)[y \mapsto \rho(y).\varphi]$ by simple unfolding.

• **Case** $e = e': y$:

\[
\mathcal{A}[e' y]_{\rho} = \text{app}(\mathcal{A}[e']_{\rho, \rho(y)}) \\
= \text{app}(\mathcal{A}[\text{let } z = \lambda x.e_1 \text{ in } e_2] y_{\rho}) \text{lfp}(\lambda \theta. \mathcal{A}[e_1]_{\rho[x \mapsto \rho(y)]}) \\
= \mathcal{A}[\lambda x.\text{let } z = \lambda x.e_1 \text{ in } e_2] y_{\rho} \\
= (\mathcal{A}[e']_{\rho_\Delta})[x \mapsto \rho(x).\varphi]
\]
C[\_] : S \to AbsTy

C[(e, \rho, \mu, \kappa)] = apps_\mu(\kappa, A[e]_\alpha(\mu)_{\rho})
\alpha(\mu) = lfp(\lambda\hat{\mu}. [a \mapsto x \& A[e']_{\rho_{\rho'}} | \mu(a) = (x, \rho', e')])
apps_\rho(stop, \theta) = \theta
apps_\rho(ap(a) \cdot \kappa, \theta) = apps_\rho(\kappa, app(\theta, \alpha(\mu)(a)))
apps_\rho(upd(a) \cdot \kappa, \theta) = apps_\rho(\kappa, \theta)

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

• Case e = \lambda y.e': Note that x \neq y because y is not free in e.
A[\lambda y.e']_{\rho}
= lam_y(\lambda\theta. A[e']_{\rho[y\mapsto\theta]})
= lam_y(\lambda\theta. (A[e']_{\rho[y\mapsto((y\mapsto\theta),\text{Rep U})]}))

Induction hypothesis

Property of lam_y

Unfold A[\_]

\theta[y] \Rightarrow [y \mapsto \theta] = \theta
\theta[y] \Rightarrow [y \mapsto U] = \theta

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

In Figure 14, we give the extension of C[\_] to whole machine configurations \sigma. Although C[\_] looks like an entirely new definition, it is actually derivative of A[\_] via a context lemma à la Moran and Sands [1999, Lemma 3.2]: The environments \rho 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 A[\_] and invoke the summary mechanism. Update frames are ignored because our analysis is not heap-sensitive.
Now we can prove that $C[\cdot]$ is preserved/improves during reduction:

**Lemma 19** (Preservation of $C[\cdot]$). If $\sigma_1 \leftrightarrow \sigma_2$, then $C[\sigma_1] \sqsupseteq C[\sigma_2]$.

**Proof.** By cases on the transition.

- **Case LET:** Then $e = \text{let } y = e_1 \text{ in } e_2$ and
  \[
  \text{let } y = e_1 \text{ in } e_2, \rho, \mu, \kappa \quad \leftrightarrow \quad (e_2, \rho[y \mapsto a], \mu[a \mapsto (y, \rho[y \mapsto a], e_1)], \kappa).
  \]

  Abbreviating $\rho_1 = \rho[y \mapsto a], \mu_1 = \mu[a \mapsto (y, \rho_1, e_1)]$, we have
  \[
  C[\sigma_1] = apps(y)(\langle e_2 \rangle_{\alpha(\mu) \circ \rho}) \quad \text{Unfold } C[\sigma_1]
  \]
  \[
  = apps(y)(\langle e_2 \rangle_{\alpha(\mu) \circ \rho}) \quad \text{Unfold } \langle e \rangle_{\alpha(\mu) \circ \rho}
  \]
  \[
  = apps(y)(\langle e \rangle_{\alpha(\mu) \circ \rho}) \quad \text{Unfold RHS of Lemma 3}
  \]
  \[
  = C[\sigma_1] \quad \text{Refold } C[\sigma_1]
  \]

- **Case APP:** Then $(e', y, \rho, \mu, \kappa) \leftrightarrow (e', \rho[y \mapsto a], \mu, \kappa)$.

  \[
  C[\sigma_1] = apps(y)(\langle e' \rangle_{\alpha(\mu) \circ \rho}) \quad \text{Unfold } C[\sigma_1]
  \]
  \[
  = apps(y)(\langle e' \rangle_{\alpha(\mu) \circ \rho}) \quad \text{Unfold } \langle e \rangle_{\alpha(\mu) \circ \rho}
  \]
  \[
  = apps(y)(\langle e \rangle_{\alpha(\mu) \circ \rho}) \quad \text{Unfold RHS of Lemma 3}
  \]
  \[
  = C[\sigma_1] \quad \text{Refold } C[\sigma_1]
  \]

- **Case LOOK:** Then $e = y, a \triangleq \rho(y), (z, \rho', e') \triangleq \mu(a)$ and $(y, \rho, \mu, \kappa) \leftrightarrow (e', \rho', \mu, \text{upd}(a) \cdot \kappa)$.

  \[
  C[\sigma_1] = apps(y)(\langle y \rangle_{\alpha(\mu) \circ \rho}) \quad \text{Unfold } C[\sigma_1]
  \]
  \[
  = apps(y)(\langle y \rangle_{\alpha(\mu) \circ \rho}) \quad \text{Unfold } \langle y \rangle_{\alpha(\mu) \circ \rho}
  \]
  \[
  = apps(y)(\langle y \rangle_{\alpha(\mu) \circ \rho}) \quad \text{Unfold } \langle e \rangle_{\alpha(\mu) \circ \rho}
  \]
  \[
  = apps(y)(\langle y \rangle_{\alpha(\mu) \circ \rho}) \quad \text{Definition of } apps(y)
  \]
  \[
  = C[\sigma_1] \quad \text{Refold } C[\sigma_1]
  \]

- **Case UPD:** Then $(v, \rho, \mu[a \mapsto (y, \rho, e')], \text{upd}(a) \cdot \kappa) \leftrightarrow (v, \rho, \mu[a \mapsto (y, \rho, v)], \kappa)$.

  \[
  C[\sigma_1] = apps(y)(\langle e' \rangle_{\alpha(\mu) \circ \rho}) \quad \text{Unfold } C[\sigma_1]
  \]
  \[
  = apps(y)(\langle e' \rangle_{\alpha(\mu) \circ \rho}) \quad \text{Unfold } \langle e \rangle_{\alpha(\mu) \circ \rho}
  \]
  \[
  = apps(y)(\langle e \rangle_{\alpha(\mu) \circ \rho}) \quad \text{Unfold RHS of Lemma 3}
  \]
  \[
  = C[\sigma_1] \quad \text{Refold } C[\sigma_1]
  \]
This case is a bit hand-wavy and shows how heap update during by-need evaluation is dreadfully complicated to handle, even though $A[\_]$ is heap-less and otherwise correct wrt. by-name evaluation. The culprit is that in order to show $C[\sigma_2] \subseteq C[\sigma_1]$, we have to show
\begin{equation}
A[v]_{\alpha(\mu) \circ \rho} \subseteq A[e']_{\alpha(\mu') \circ \rho'}.
\end{equation}

Intuitively, this is somewhat clear, because $\mu$ "evaluates to" $\mu'$ and $v$ is the value of $e'$, in the sense that there exists $\sigma' = (e', \rho', \mu', \kappa)$ such that $\sigma' \leadsto^* \sigma_1 \leadsto \sigma_2$. Alas, who guarantees that such a $\sigma'$ 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
\[ \forall n. \sigma_0 \vdash^n \sigma_2 \implies C[\sigma_2] \subseteq C[\sigma_0] \]
has been proved for all $n < k$ and proceed to prove it for $n = k$. So we presume $\sigma_0 \vdash^{k-1} \sigma_1 \leadsto \sigma_2$ and $C[\sigma_1] \subseteq C[\sigma_0]$ to arrive at a similar setup as before, only with a stronger assumption about $\sigma_1$. Specifically, due to the balanced stack discipline we know that $\sigma_0 \vdash^{k-1}$ factors over $\sigma'$ 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 $\sigma' \leadsto^* \sigma_1$ to show Equation (1). This reasoning was not specific to $A[\_]$ 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{align*}
C[\sigma_1] &= \text{apps}_\mu(\text{upd}(a) \cdot \kappa)(A[v]_{\alpha(\mu) \circ \rho}) \quad \text{Unfold } C[\sigma_1] \\
&= \text{apps}_\mu(\kappa)(A[v]_{\alpha(\mu) \circ \rho}) \quad \text{Definition of apps}_\mu \\
&\supseteq \text{apps}_{\mu[\alpha \mapsto (y,\rho,v)]}(\kappa)(A[v]_{\alpha(\mu[\alpha \mapsto (y,\rho,v)]) \circ \rho}) \quad \text{Above argument that } A[v]_{\alpha(\mu) \circ \rho} \subseteq A[e']_{\alpha(\mu') \circ \rho'} \\
&= C[\sigma_2] \quad \text{Refold } C[\sigma_2]
\end{align*}

We conclude with the proof for Theorem 1:

**Proof.** We show the contraposition, that is, if $x$ is used in $e$, then $\phi(x) = U$.

Since $x$ is used in $e$, there exists a trace
\[
(\text{let } x = e' \text{ in } e, \rho, \mu, \kappa) \leadsto (e, \rho_1, \mu_1, \kappa) \leadsto^* (y, \rho', [y \mapsto a], \mu', \kappa') \xrightarrow{\text{Look(x)}} \ldots,
\]
where $\rho_1 \triangleq \rho[x \mapsto a]$, $\mu_1 \triangleq \mu[a \mapsto (x, \rho[x \mapsto a], e')]$. Without loss of generality, we assume the trace prefix ends at the first lookup at $a$, so $\mu'(a) = \mu_1(a) = (x, \rho_1, e')$. 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, $\tilde{\rho}(y).\phi(x) = U$ implies $x = y$ for all $y$, because $\mu_1(a)$ is the only heap entry in which $x$ occurs by our shadowing assumptions on syntax. By unfolding $C[\_]$ and $A[\_]$, we can see that
\[
[x \mapsto U] \subseteq \alpha(\mu_1)(a).\phi = \alpha(\mu')(a).\phi = A[y]_{\alpha(\mu') \circ \rho'[y \mapsto a], \phi} \subseteq (C[(y, \rho'[y \mapsto a], \mu', \kappa')]).\phi.
\]

By Lemma 19, we also have
\[
(C[(y, \rho'[y \mapsto a], \mu', \kappa')]).\phi \subseteq (C[(e, \rho_1, \mu_1, \kappa)]).\phi.
\]
And with transitivity, we get \([x \overset{\iota}{\Rightarrow} U] \subseteq (C[[e, \rho_1, \mu_1, \kappa]]) \cdot \phi\). Since there was no other heap entry for \(x\) and \(\iota\) cannot occur in \(\kappa\) or \(\rho_1\) due to well-addressedness, we have \([x \overset{\iota}{\Rightarrow} U] \subseteq (C[[e, \rho_1, \mu_1, \kappa]]) \cdot \phi\) if and only if \([x \overset{\iota}{\Rightarrow} U] \subseteq (\mathcal{A}[e]^{\hat{\rho}_e}) \cdot \phi\). With Lemma 18, we can decompose

\[
[x \overset{\iota}{\Rightarrow} U] \subseteq (\mathcal{A}[e]^{\hat{\rho}_e}) \cdot \phi
\]

by definition of \([\cdot] \Rightarrow \cdot\). But since \(\hat{\rho}(y) \cdot \phi(x) = U\) implies \(x = y\) (refer to definition of \(\hat{\rho}\)), we must have \((\mathcal{A}[e]^{\hat{\rho}_e}) \cdot \phi(x) = U\), as required.

\[\square\]

### B PROOFS FOR SECTION 5 (TOTALITY AND SEMANTIC ADEQUACY)

**Theorem 4 (Strong Adequacy).** Let \(e\) be a closed expression, \(\tau \triangleq S_{\text{need}}[e]_{x}(\epsilon)\) the denotational by-need trace and \(\text{init}(e) \Leftarrow \cdots\) the maximal lazy Krivine trace. Then

\begin{itemize}
  \item \(\tau\) preserves the observable termination properties of \(\text{init}(e) \Leftarrow \cdots\) in the above sense.
  \item \(\tau\) preserves the length (i.e., number of Steps) of \(\text{init}(e) \Leftarrow \cdots\) (i.e., number of transitions).
  \item every \(\text{ev} :: \text{Event in } \tau = \text{Stop ev} \ldots\) corresponds to the transition rule taken in \(\text{init}(e) \Leftarrow \cdots\).
\end{itemize}

**Proof.** We formally define as \(\alpha(\text{init}(e) \Leftarrow \cdots) \triangleq \alpha_{\text{Ev}}(\text{init}(e) \Leftarrow \cdots, \text{stop})\), where \(\alpha_{\text{Ev}}\) is defined in Figure 15.

Then \(S_{\text{need}}[e]_{x}(\epsilon) = \alpha(\text{init}(e) \Leftarrow \cdots)\) follows directly from Theorem 27. The preservation results in are a consequence of Lemma 25 and theorem 28; function \(\alpha_{\text{Ev}}\) in Figure 15 encodes the intuition in which \(\text{LK}\) transitions abstract into \(\text{Events}\).

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 \(S[\cdot] \Rightarrow \cdot\).

Formally, an \(\text{LK}\) trace is a trace in \((\overset{\iota}{\Rightarrow})\) from Figure 2, i.e., a non-empty and potentially infinite sequence of \(\text{LK}\) states \((\sigma_i)_{i \in \mathbb{N}}\) (where \(\mathbb{N} = \{m \in \mathbb{N} \mid m < n\}\) when \(n \in \mathbb{N}, \mathbb{N} = \mathbb{N}\)), such that \(\sigma_i \overset{\iota}{\Rightarrow} \sigma_{i+1}\) for \(i, (i + 1) \in \mathbb{N}\). The source state \(\sigma_0\) exists for finite and infinite traces, while the target state \(\sigma_\omega\) is only defined when \(n \neq \omega\) is finite. When the control expression of a state \(\sigma\) (selected via \(\text{ctrl}(\sigma)\)) is a value \(v\), we call \(\sigma\) a \(\text{return state}\) and say that the continuation (selected via \(\text{cont}(\sigma)\)) drives evaluation. Otherwise, \(\sigma\) is an \(\text{evaluation state}\) and \(\text{ctrl}(\sigma)\) 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 \(\text{LK}\) trace \((\sigma_i)_{i \in \mathbb{N}}\) is \(\kappa\)-deep if every intermediate continuation \(\kappa_i \triangleq \text{cont}(\sigma_i)\) extends \(\kappa\) (so \(\kappa_0 = \kappa\) or \(\kappa_i = \cdots \kappa\), abbreviated \(\kappa_i = \cdots \kappa\)). A trace \((\sigma_i)_{i \in \mathbb{N}}\) is called interior if it is \(\text{cont}(\sigma_0)\)-deep. Furthermore, an interior trace \((\sigma_i)_{i \in \mathbb{N}}\) is balanced \([\text{Sestoft 1997}]\) if the target state exists and is a return state with continuation \(\text{cont}(\sigma_0)\).

We notate \(\kappa\)-deep and interior traces as \(\kappa\) deep \((\sigma_i)_{i \in \mathbb{N}}\) and \((\sigma_i)_{i \in \mathbb{N}}\) 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 \(\text{LOOK}\) transitions, and it is confusing when the heap-bound expression is a variable \(x\), e.g., \((y, \rho, x)\).

**Example 21.** Let \(\rho = [x \mapsto a_1], \mu = [a_1 \mapsto (\_ \_ [\_ \_ \_]) \lambda y .y]\) and \(\kappa\) an arbitrary continuation. The trace

\[
(x, \rho, \mu, \kappa) \Leftarrow (\lambda y .y, \rho, \mu, \text{upd}(a_1) \cdot \kappa) \Leftarrow (\lambda y .y, \rho, \mu, \kappa)
\]
is interior and balanced. Its proper prefixes are interior but not balanced. The trace suffix
\[ (\lambda y.y, \rho, \mu, \text{upd}(a_1) \cdot \kappa) \leftrightarrow (\lambda y.y, \rho, \mu, \kappa) \]
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 \( e \jtr v \) 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 \( (\sigma_i)_{i \in \mathbb{N}} \) is maximal if and only if it is interior and there is no \( \sigma_{i+1} \) such that \( (\sigma_i)_{i \in \mathbb{N}} \) is interior. More formally,

\[ (\sigma_i)_{i \in \mathbb{N}} \max \triangleq (\sigma_i)_{i \in \mathbb{N}} \text{inter} \wedge (\exists \sigma_{i+1} \cdot \sigma_n \leftrightarrow \sigma_{i+1} \wedge \text{cont}(\sigma_{i+1}) = \ldots \text{cont}(\sigma_0)). \]

We notate maximal traces as \( (\sigma_i)_{i \in \mathbb{N}} \max. \) 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
\[ (\text{tt } x, [x \mapsto a_1], [a_1 \mapsto \ldots, \kappa]) \leftrightarrow (\text{tt }, [x \mapsto a_1], [a_1 \mapsto \ldots], \text{ap}(a_1) \cdot \kappa), \]
where \( \text{tt } \) is a data constructor. It is stuck, but its singleton suffix is balanced. An example for a diverging trace, where \( \rho = [x \mapsto a_1] \) and \( \mu = [a_1 \mapsto (\rho, x)] \), is
\[ (\text{let } x = x \text{ in } x, [], [], \kappa) \leftrightarrow (x, \rho, \mu, \kappa) \leftrightarrow (x, \rho, \mu, \text{upd}(a_1) \cdot \kappa) \leftrightarrow \ldots \]

**Lemma 24** (Characterisation of maximal traces). An LK trace \( (\sigma_i)_{i \in \mathbb{N}} \) is maximal if and only if it is balanced, diverging or stuck.

**Proof.** \( \Rightarrow \): Let \( (\sigma_i)_{i \in \mathbb{N}} \) be maximal. If \( n = \omega \) is infinite, then it is diverging due to interiority, and if \( (\sigma_i)_{i \in \mathbb{N}} \) is stuck, the goal follows immediately. So we assume that \( (\sigma_i)_{i \in \mathbb{N}} \) 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 \( (\sigma_i)_{i \in \mathbb{N}} \) is interior and infinite, hence \( n = \omega \). Indeed \( (\sigma_i)_{i \in \mathbb{N}} \) is maximal, because the expression \( \sigma_\omega \) is undefined and hence does not exist. \( \Box \)

Interiority guarantees that the particular initial stack \( \kappa \) 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 \( \text{init}(e)! \)
Whether it is infinite, stuck or balanced is the defining termination observable of \( e \). If we can show
that \( S[\ldots] \), 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 \( S[\ldots] \) adequate at
D (ByNeed T). It encodes that an abstraction of every maximal LK trace can be recovered by running \( S[\ldots] \) 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 \( S[\ldots] \) and the
syntactic objects in the transition system.
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 \( \tilde{\mu} \overset{\Delta}{=} \tilde{\alpha}_\Sigma(\mu) \) denotes a semantic heap which in this instance is defined to be the abstraction of a syntactic heap \( \mu \).

Note first that \( \alpha_{\Sigma^{\infty}} \) is defined by guarded recursion over the LK trace, in the following sense:

We regard \( (\sigma_i)_{i \in \mathbb{N}} \) as a Sigma type \( \Sigma^{\infty} \overset{\Delta}{=} \exists n \in \mathbb{N}_\omega. \pi \rightarrow S \), where \( \mathbb{N}_\omega \) is defined by guarded recursion as \( \text{data } \mathbb{N}_\omega = Z | S \uplus (\bullet \mathbb{N}_\omega) \). Now \( \mathbb{N}_\omega \) contains all natural numbers (where \( n \) is encoded as \( S \circ \text{pure} \)) and the transfinite limit ordinal \( \omega = S \uplus (\text{pure} \circ S \uplus \text{pure} \cdots) \). We will assume that addition and subtraction are defined as on Peano numbers, and \( \omega + \_ = \_ + \omega = \omega \). When \( (\sigma_i)_{i \in \mathbb{N}} \in \Sigma^{\infty} \) is an LK trace and \( n > 1 \), then \( (\sigma_{i+1})_{i \in \mathbb{N}^{-1}} \in \Sigma^{\infty} \) is the guarded tail of the trace with an associated coinduction principle.

As such, the expression \( \langle \alpha_{\Sigma^{\infty}}((\sigma_{i+1})_{i \in \mathbb{N}^{-1}}, \kappa) \rangle \) has type \( (T \uplus (\text{Value}(\text{ByNeed} T), \text{Heap}(\text{ByNeed} T))) \) (the \( \uplus \) in the type of \( (\sigma_{i+1})_{i \in \mathbb{N}^{-1}} \) maps through \( \alpha_{\Sigma^{\infty}} \) via the idiom brackets). Definitional equality = on \( T \uplus (\text{Value}(\text{ByNeed} T), \text{Heap}(\text{ByNeed} T)) \) 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 \( \alpha_{\Sigma^{\infty}}(\sigma) \) 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 \( \text{Lookup } y \) events that carry with them the \( y::\text{Name} \) of the let binding that allocated the heap entry. This event corresponds precisely to a \text{Look}(y) transition, so \( \alpha_{\Sigma^{\infty}}(\sigma) \) maps \( \sigma \) to \( \text{Lookup } y \) when \( \sigma \) is about to make a \text{Look}(y) transition. In that case, the focus expression must be \( x \) and \( y \) is the first component of the heap entry \( \mu(\rho(x)) \). 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 \( \alpha_{\Sigma^{\infty}} \) and in which way. Let us warm up by defining a length function on traces:
**Lemma 25** (Preservation of length). Let \((σ_i)_{i∈\mathbb{N}}\) be a trace. Then \(\text{len} (α\subseteq\omega((σ_i)_{i∈\mathbb{N}}, \text{cont}(σ_0))) = n\).

**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:

\[ \text{löb} = \text{fix} : \forall P. (\text{⇒} P \implies P) \implies P \]

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

\[ IH \in (\text{⇒} P \triangleq \forall n ∈ \mathbb{N}. ∀ (σ_i)_{i∈\mathbb{N}}. \text{len} (α\subseteq\omega((σ_i)_{i∈\mathbb{N}}, \text{cont}(σ_0))) = n) \]

and use \(IH\) to prove \(P\). Let us assume \(n\) and \((σ_i)_{i∈\mathbb{N}}\) are given, define \(τ \triangleq α\subseteq\omega((σ_i)_{i∈\mathbb{N}}, \text{cont}(σ_0))\) and proceed by case analysis over \(n\):

- **Case \(Z\):** Then we have either \(τ = \text{Ret} (α\subseteq(σ_0))\) or \(τ = \text{Ret Stuck}\), both of which map to \(Z\) under \(\text{len}\).
- **Case \(S\ \{m\}\):** Then \(τ = \text{Step} \{α\subseteq\omega((σ_{i+1})_{i∈\mathbb{N}}, \text{cont}(σ_0))\},\) where \((σ_{i+1})_{i∈\mathbb{N}} \in (\text{⇒} S\subseteq\omega)\) is the guarded tail of the LK trace \((σ_i)_{i∈\mathbb{N}}\). Now we apply the inductive hypothesis, as follows:

\[ (IH \circ m \circ (σ_{i+1})_{i∈\mathbb{N}}) ∈ (\text{⇒} \text{len} (α\subseteq\omega((σ_{i+1})_{i∈\mathbb{N}}, \text{cont}(σ_0))) = m) \]

We use this fact and congruence to prove

\[ n = S \{m\} = S (\text{len} (α\subseteq\omega((σ_{i+1})_{i∈\mathbb{N}}, \text{cont}(σ_0)))) = \text{len} (α\subseteq\omega((σ_i)_{i∈\mathbb{N}}, \text{cont}(σ_0))). \]

**Lemma 26** (Abstraction preserves termination observable). Let \((σ_i)_{i∈\mathbb{N}}\) be a maximal trace. Then \(α\subseteq\omega((σ_i)_{i∈\mathbb{N}}, \text{cont}(σ_0))\) is ...

- ... ending with \(\text{Ret} (\text{Fun } \_\_)\) or \(\text{Ret} (\text{Con } \_\_)\) if and only if \((σ_i)_{i∈\mathbb{N}}\) is balanced.
- ... infinite if and only if \((σ_i)_{i∈\mathbb{N}}\) is diverging.
- ... ending with \(\text{Ret Stuck}\) if and only if \((σ_i)_{i∈\mathbb{N}}\) 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 \((σ_i)_{i∈\mathbb{N}}\) is balanced, then \(σ_n\) is a return state with continuation \(\text{cont}(σ_0)\), so its control expression is a value. Then \(α\subseteq\omega\) will conclude with \(\text{Ret} (α\subseteq(\_\_),\) and the latter is never \(\text{Ret Stuck}\). Conversely, if the trace ended with \(\text{Ret} (\text{Fun } \_\_)\) or \(\text{Ret} (\text{Con } \_\_)\), then \(\text{cont}(σ_n) = \text{cont}(σ_0)\) and \(\text{ctrl}(σ_n)\) is a value, so \((σ_i)_{i∈\mathbb{N}}\) 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 \(τ : T a\) 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 \(\text{Ret} v\) for a \(v \not\in \text{Stuck}\), stuck if ending in \(\text{Ret Stuck}\) and diverging if infinite.

We are now ready to prove the main soundness predicate, proving that \(S_{\text{need}}[\_\_]\) is an exact abstract interpretation of the LK machine:

**Theorem 27** \((S_{\text{need}}[\_\_])\) abstracts LK machine. \(C\) from Figure 15 holds. That is, whenever \((σ_i)_{i∈\mathbb{N}}\) is a maximal LK trace with source state \((e, ρ, μ, κ)\), we have \(α\subseteq\omega((σ_i)_{i∈\mathbb{N}}, κ) = S_{\text{need}}[\_\_][τ]_{αE(μ, ρ)}(αT:\mu)).\)

PROOF. By Löb induction, with $IH \in \triangledown C$ as the hypothesis.
We will say that an LK state $\sigma$ is stuck if there is no applicable rule in the transition system (i.e., the singleton LK trace $\sigma$ is maximal and stuck).

Now let $(\sigma_i)_{i \in \overline{1,n}}$ be a maximal LK trace with source state $\sigma_0 = (e, \rho, \mu, \kappa)$ and let $\tau = S_{\text{need}}(e)_{\sigma(\mu, \rho)}(\alpha_\tau(\mu))$.
Then the goal is to show $\alpha(\overline{1,n}, \kappa) = \tau$. We do so by cases over $e$, abbreviating $\tilde{\mu} \triangleq \alpha_\tau(\mu)$ and $\tilde{\rho} \triangleq \alpha_\tau(\mu, \rho)$:

- **Case $x$:** Let us assume first that $\sigma_0$ is stuck. Then $x \notin \text{dom}(\rho)$ (because Look is the only transition that could apply), so $\tau = \text{Ret Stuck}$ and the goal follows from Lemma 26.

\[
\text{Otherwise, } \sigma_1 \triangleq (e', \rho_1, \mu, \text{upd}(a) \cdot \kappa), \sigma_0 \hookrightarrow \sigma_1 \text{ via } \text{Look}(y), \text{ and } \rho(x) = a, \mu(a) = (y, \rho_1, e').
\]

This matches the head of the action of $\tilde{\rho} \cdot x$, which is of the form step (Lookup $y$) (fetch $a$).

To show that the tails equate, it suffices to show that they equate later.
We can infer that $\tilde{\mu} \cdot a = \text{memo } a (S_{\text{need}}(e')_{\tilde{\rho}})$ from the definition of $\alpha_\tau$, so

\[
\text{fetch } a \cdot \tilde{\mu} \cdot a \cdot \tilde{\mu} = S_{\text{need}}(e')_{\tilde{\rho}}(\tilde{\mu}) \Rightarrow \lambda \text{case}
\]

\[
\begin{align*}
\text{(Stuck, } \tilde{\mu} \text{) } & \rightarrow \text{Ret (Stuck, } \tilde{\mu} \text{)} \quad \\
\text{(val, } \tilde{\mu} \text{) } & \rightarrow \text{Step Update (Ret (val, } \tilde{\mu}[a \mapsto \text{memo } a \text{ (return val)]})
\end{align*}
\]

Let us define $\tau^* \triangleq \|S_{\text{need}}(e')_{\tilde{\rho}}(\tilde{\mu})\|$ and apply the induction hypothesis $IH$ to the maximal trace starting at $\sigma_1$. This yields an equality

\[
IH \otimes (\sigma_{i+1})_{i \in \overline{1,m}} \in \|\alpha(\overline{1,m}, \text{ upd}(a) \cdot \kappa) = \tau^*\|
\]

When $\tau^*$ is infinite, we are done. Similarly, if $\tau^*$ ends in Ret Stuck then the continuation of $\Rightarrow$ will return Ret Stuck, indicating by Lemma 25 and Lemma 26 that $(\sigma_{i+1})_{i \in \overline{1,n-1}}$ is stuck and hence $(\sigma_i)_{i \in \overline{1,n}}$ is, too.

Otherwise $\tau^*$ ends after $m - 1$ Steps with Ret (val, $\tilde{\mu}_m$) and by Lemma 26 $(\sigma_{i+1})_{i \in \overline{1,m}}$ is balanced; hence $\text{cont}(\sigma_m) = \text{upd}(a) \cdot \kappa$ and $\text{ctrl}(\sigma_m)$ is a value. So $\sigma_m = (v, \rho_m, \mu_m, \text{upd}(a) \cdot \kappa)$ and the UPD transition fires, reaching $(v, \rho_m, \mu_m[a \mapsto (y, \rho_m, v)], \kappa)$ and this must be the target state $\sigma_n$ (so $m = n - 2$), because it remains a return state and has continuation $\kappa$, so $(\sigma_i)_{i \in \overline{1,n}}$ is balanced. Likewise, the continuation argument of $\Rightarrow$ does a Step Update on Ret (val, $\tilde{\mu}_m$), updating the heap. By cases on $v$ and the Domain (D (ByNeed T)) instance we can see that

\[
\begin{align*}
\text{Ret (val, } \tilde{\mu}_m[a \mapsto \text{memo } a \text{ (return val)]}) = \\
\text{Ret (val, } \tilde{\mu}_m[a \mapsto \text{memo } a (S_{\text{need}}(v)_{\tilde{\rho}_m})]) = \\
\alpha(\overline{1,m}, \mu_m) = \end{align*}
\]

and this equality concludes the proof.

- **Case $e$:** The cases where $\tau$ gets stuck or diverges before finishing evaluation of $e$ are similar to the variable case. So let us focus on the situation when $\tau^* \triangleq \|S_{\text{need}}(e')_{\tilde{\rho}}(\tilde{\mu})\|$ returns and let $\sigma_m$ be LK state at the end of the balanced trace $(\sigma_{i+1})_{i \in \overline{m-1}}$ through $e$ starting in stack $\text{ap}(a) \cdot \kappa$.

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 $\text{ap}(a) \cdot \kappa$ due to maximality, necessarily by an $\text{App}_2$ transition. That in turn means that the value in $\text{ctrl}(\sigma_m)$ must be a lambda $\lambda y.e'$, and

\[
\sigma_{m+1} = (e', \rho_m[y \mapsto \rho(x)], \mu_m, \kappa).
\]

Likewise, $\tau^*$ ends in $\alpha(\overline{1,m}) = \text{Ret (Fun } (\lambda d \rightarrow \text{ step } \text{App}_2 (S_{\text{need}}(e')_{\tilde{\rho}_m[y \mapsto d]})), \tilde{\mu}_m)$ (where $\tilde{\mu}_m$ corresponds to the heap in $\sigma_m$ in the usual way). The $\text{fun}$ implementation of Domain (D (ByNeed T)) applies the $\text{Fun}$ value to the argument denotation $\tilde{\rho} \cdot x$, hence it
remains to show that \( t^* : S_{\text{need}}[\epsilon'] \rho_m[\alpha \rightarrow \hat{\rho} x] (\hat{\mu}_m) \) is equal to \( \alpha_{\leq \epsilon}(\sigma_{i+m+1})_{i \in K} \) later, where \((\sigma_{i+m+1})_{i \in K}\) is the maximal trace starting at \(\sigma_{m+1}\).

We can apply the induction hypothesis to this situation. From this and our earlier equalities, we get \( \alpha_{\leq \epsilon}(\sigma_i)_{i \in \mathbb{N}} = \tau \), concluding the proof of the case where there exists a transition \(\sigma_m \xrightarrow{\epsilon} \sigma_{m+1}\).

When \(\sigma_m \not\xrightarrow{\epsilon}\), then \(\text{ctrl}(\sigma_m)\) is not a lambda, otherwise \(\text{App}_2\) would apply. In this case, \(\text{fun}\) gets to see a \text{Stuck} or \text{Con \_\_\_} value, for which it is \text{Stuck} as well.

\* Case case \(e_s\) of \(\overline{K} \xrightarrow{\epsilon} e_r\): Similar to the application and lookup case.

\* Cases \(\lambda x.e, K \xrightarrow{\epsilon}\): The length of both traces is \(n = 0\) and the goal follows by simple calculation.

\* Case let \(x = e_1\) in \(e_2\): Let \(\sigma_0 = (\text{let } x = e_1 \text{ in } e_2, \rho, \mu, \kappa)\). Then \(\sigma_1 = (e_2, \rho_1, \mu', \kappa)\) by \(\text{LET}_1\), where \(\rho_1 = \rho[\{x \mapsto a_{k_{i}}\}], \mu' = \mu[a_{k_{i}} \mapsto (x, \rho_1, e_1)]\). Since the stack does not grow, maximality from the tail \((\sigma_{i+1})_{i \in \mathbb{N} - 1}\) transfers to \((\sigma_i)_{i \in \mathbb{N}}\). Straightforward application of the induction hypothesis to \((\sigma_{i+1})_{i \in \mathbb{N} - 1}\) yields the equality for the tail (after a bit of calculation for the updated environment and heap), which concludes the proof.

\[\square\]

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 \(\sigma\) is \text{final} when \(\text{ctrl}(\sigma)\) is a value and \(\text{cont}(\sigma) = \text{stop}\).)

**Theorem 28 (Adequacy of \(S_{\text{need}}\))**. Let \(\tau \triangleq S_{\text{need}}[\epsilon'](\epsilon)\).

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

**Proof.** There exists a maximal trace \((\sigma_i)_{i \in \mathbb{N}}\) starting from \(\sigma_0 = \text{init}(\epsilon)\), and by Theorem 27 we have \(\alpha_{\leq \epsilon}(\sigma_i)_{i \in \mathbb{N}, \text{stop}} = \tau\). The correctness of \(\text{Events}\) emitted follows directly from \(\alpha_{\leq \epsilon}\).

\[
\begin{align*}
\Rightarrow & \quad \text{If } (\sigma_i)_{i \in \mathbb{N}} \text{ is balanced, its target state } \sigma_n \text{ is a return state that must also have the empty continuation, hence it is a final state.} \\
& \quad \text{If } (\sigma_i)_{i \in \mathbb{N}} \text{ is stuck, it is finite and maximal, but not balanced, so its target state } \sigma_n \text{ cannot be a return state; otherwise maximality implies } \sigma_n \text{ has an (initial) empty continuation and the trace would be balanced. On the other hand, the only returning transitions apply to return states, so maximality implies there is no } \sigma' \text{ such that } \sigma \xrightarrow{\epsilon} \sigma' \text{ whatsoever.} \\
& \quad \text{If } (\sigma_i)_{i \in \mathbb{N}} \text{ is diverging, } n = \omega \text{ and for every } \sigma \text{ with } \text{init}(\epsilon) \xrightarrow{\ast} \sigma \text{ there exists an } i \text{ such that } \sigma = \sigma_i \text{ by determinism.} \\
\Leftarrow & \quad \text{If } \sigma_n \text{ is a final state, it has } \text{cont}(\sigma) = \text{cont} \text{(init}(\epsilon)) = [\_\_\_], \text{so the trace is balanced.} \\
& \quad \text{If } \sigma \text{ is not a final state, } \tau' \text{ is not balanced. Since there is no } \sigma' \text{ such that } \sigma \xrightarrow{\epsilon} \sigma' \text{, it is still maximal; hence it must be stuck.} \\
& \quad \text{Suppose that } n \in \mathbb{N}_0 \text{ was finite. Then, if for every choice of } \sigma \text{ there exists } \sigma' \text{ such that } \sigma \xrightarrow{\epsilon} \sigma', \text{ then there must be } \sigma_{n+1} \text{ with } \sigma_n \xrightarrow{\epsilon} \sigma_{n+1} \text{, violating maximality of the trace. Hence it must be infinite. It is also interior, because every stack extends the empty stack, hence it is diverging.} \\
\end{align*}
\]
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 ▶ which allows to define coinductive data types with negative recursive occurrences such as in the data constructor Fun :: (D τ → D τ) → Value τ (recall that D τ = τ (Value τ)), as first realised by Nakano [2000]. The way that is achieved is roughly as follows: The type ▶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 fix : ∀A. (▶A → A) → 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:

\[
\text{Str} = \mathbb{N} \times \triangleright \text{Str} \quad \text{ones} = \text{fix}(r : \triangleright \text{Str}). (1, r),
\]

where ones : Str is the constant stream of 1. In particular, Str is the fixpoint of a locally contractive functor F(X) = \(\mathbb{N} \times \triangleright X\). According to Birkedal and Mogelberg [2013], any type expression in simply typed lambda calculus defines a locally contractive functor as long as any occurrence of X is under a ▶. The most exciting consequence is that changing the Fun data constructor to

\[
\text{Fun} :: (\triangleright (D \tau \rightarrow D \tau) \rightarrow \text{Value} \tau) \rightarrow \text{Value} \tau
\]

makes Value \(\tau\) a well-defined coinductive data type,\(^{30}\) whereas syntactic approaches to coinduction reject any negative recursive occurrence.

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

\[
\text{next} : \forall A. A \rightarrow \triangleright A \\
\text{⋯} \odot \text{⋯} : \forall A, B. \triangleright (A \rightarrow B) \rightarrow \triangleright A \rightarrow \triangleright B,
\]

allowing us to apply a familiar framework of reasoning around ▶. 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 ▶ “effects” happen.

We will now outline the changes necessary to encode \(S[\_\_]\) in Guarded Cubical Agda, a system implementing Ticked Cubical Type Theory [Mogelberg 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 step :: Event → ▶ d → d.
- All Ds that will be passed to lambdas, put into the environment or stored in fields need to have the form step (Lookup x) d for some x :: Name and a delayed d :: ▶ (D \(\tau\)). This is enforced as follows:
  1. The Domain type class gains an additional predicate parameter p :: D → Set that will be instantiated by the semantics to a predicate that checks that the D has the required form step (Lookup x) d for some x :: Name, d :: ▶ (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 (Σ D p → D) → D.

\(^{30}\)The reason why the positive occurrence of 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., Value \(\tau\) = fix X. ifp Y... | Fun (X → Y) | ...
(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 Fun :: (Name × ▶ (D τ) → D τ) → Value τ, breaking the previously discussed negative recursive cycle by a ▶, and expecting x :: Name, d :: ▶ (D τ) such that the original D τ can be recovered as step (Lookup x) d. This is in contrast to the original definition Fun :: (D τ → D τ) → Value τ which would not type-check. One can understand Fun as carrying the “closure” resulting from defunctionalising [Reynolds 1972] a Σ 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 bind :: ▶ (▷ D → D) → (▷ D → D) → D. We tried rolling up step (Lookup x) _ in the definition of S[[_]_ to get a simpler type bind :: (Σ D p → D) → (Σ D p → D) → 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, Heap τ = Addr := ▶ (D τ).
- We need to pass around Tick binders in 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 S[[_]_ is a total, mathematical function, and fast and loose equational reasoning about 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 S[[_]_, we could have defined it in a strict language (lowering ▶ a as () → 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
Abstracting Denotational Interpreters

Table 1. Examples for type analysis.

<table>
<thead>
<tr>
<th>#</th>
<th>e</th>
<th>closedType (S[e],)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>let i = λx.x in i i i i i</td>
<td>∀α_{11}. α_{11} → α_{11}</td>
</tr>
<tr>
<td>(2)</td>
<td>λx.let y = x in y x</td>
<td>\textbf{wrong}</td>
</tr>
<tr>
<td>(3)</td>
<td>let i = λx.x in let o = Some(i) in o</td>
<td>∀α_{6}. option (α_{6} → α_{6})</td>
</tr>
<tr>
<td>(4)</td>
<td>let x = x in x</td>
<td>∀α_{1}. α_{1}</td>
</tr>
</tbody>
</table>

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 ∀α. θ for an expression, where θ ranges over a Type that can be either a type variable TyVar α, a function type θ₁ →:θ₂, 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 offv(ρ) 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 rhsTy of the shape α₁. The result of this call in turn is instantiated to a non-PolyType rhsTy, perhaps turning a type-scheme ∀α₂. option (α₂ → α₂) into the shape option (α₃ → α₃) for fresh α₃. Then a constraint is emitted to unify α₁ with option (α₃ → α₃). Ultimately, the type rhsTy is returned and generalised to ∀α₃. option (α₃ → α₃), because α₃ 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 T) → da ⇒ λa → lub <$> traverse (λℓ → cachedCall ℓ a) (Set.toList T)

...

Fig. 17. 0CFA

<table>
<thead>
<tr>
<th>#</th>
<th>e</th>
<th>runCFA (S[e]_ℓ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>let i = (\lambda x.x) in let j = (\lambda y.y) in i j</td>
<td>{ly}_ℓ</td>
</tr>
<tr>
<td>2</td>
<td>let i = (\lambda x.x) in let j = (\lambda y.y) in i j j</td>
<td>{lx.., ly}_ℓ</td>
</tr>
<tr>
<td>3</td>
<td>let (\omega = \lambda x.x) x in (\omega \omega)</td>
<td>{}</td>
</tr>
<tr>
<td>4</td>
<td>let (x = let y = S(x)) in (S(y)) in x</td>
<td>{S(y)}_ℓ</td>
</tr>
</tbody>
</table>

Table 2. Examples for control-flow analysis.

We omit how to forward labels appropriately in \(S[e]_\ell\) and how to adjust Domain instances.

Figure 17 gives a rough outline of how we use this extension to define a 0CFA. An abstract Value_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 Value_Cs for the fields. For a lambda value, the shape is its abstract control-flow transformer, of type D_C → D_C (populated by updFunCache), plus a single point \((v_1, v_2)\) of its graph \((k\text{-CFA would have one point per contour})\), 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 \(v \subseteq v_1\), 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 D_C → D_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.

31As 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 $\widehat{D}$ be a domain with instances for Trace, Domain, HasBind and Lat, and let $\text{abstract}$ be the abstraction function described above. If the abstraction laws in Figure 13 hold, then $S_{D}[\_]$ is an abstract interpreter that is sound wrt. $\text{abstract}$, that is,

$$\text{abstract} \ (S_{\text{need}}[\varepsilon]) \subseteq S_{D}[\varepsilon].$$

Proof. The definition of $\text{abstract}$ is in terms of the Galois connection $\text{nameNeed}$ from Figure 18. Let $\alpha$ be the abstraction function from $\text{nameNeed}$; then we define

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

i.e., we simply run $d$ in the initial empty heap. Do note that $\text{abstract}$ does not work for open expressions because of this.

When we inline $\text{abstract}$, 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 $\varphi[x \mapsto \varphi'] \triangleq \varphi[x \mapsto \text{U}_b] + (\varphi \cdot x) * \varphi'$ the abstract substitution operation on $\text{Uses}$ and overload this notation for $\text{Trace}$, so that $\langle \varphi, \nu \rangle [x \mapsto \varphi'] \triangleq \langle \varphi[x \mapsto \varphi'], \nu \rangle$.

Lemma 31. $S[\text{Lam} \ x \ e \ 'App' \ y] \rho = (S[e] \rho[x \mapsto \text{U}_b, \text{Rep} \ U_a]) [x \mapsto (\rho ! y) \cdot \varphi].$

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. $S_{\text{usage}}[\text{Lam} \ y \ (\text{Lam} \ x \ e \ 'App' \ z)] \rho = S_{\text{usage}}[\text{Lam} \ x \ (\text{Lam} \ y \ e \ 'App' \ z)] \rho.$

Lemma 33. $S_{\text{usage}}[\text{Lam} \ x \ e \ 'App' \ y \ 'App' \ z] \rho = S_{\text{usage}}[\text{Lam} \ x \ (e \ 'App' \ z) \ 'App' \ y] \rho.$

Lemma 34. $S_{\text{usage}}[\text{Case} \ (\text{Lam} \ x \ e \ 'App' \ y) \ (\text{alts} \ (\text{Lam} \ x \ e_r \ 'App' \ y))] \rho[x \mapsto \rho ! y]$

$\ = \ S_{\text{usage}}[\text{Lam} \ x \ (\text{Case} \ e \ (\text{alts} \ e_r)) \ 'App' \ y] \rho.$

Lemma 35. $S_{\text{usage}}[\text{Lam} \ x \ e_1 \ 'App' \ y \ (\text{Lam} \ x \ e_2 \ 'App' \ y)] \rho = S_{\text{usage}}[\text{Lam} \ x \ (\text{Lam} \ z \ e_1 \ e_2) \ 'App' \ y] \rho.$

Now we can finally prove the substitution lemma:

Lemma 7 (Substitution). $S_{\text{usage}}[e] \rho[x \mapsto \rho ! y] \subseteq S_{\text{usage}}[\text{Lam} \ x \ e \ 'App' \ y] \rho.$

Proof. We need to assume that $x$ is absent in the range of $\rho$. This is a “freshness assumption” relating to the identifier of $x$ that in practice is always respected by $S_{\text{usage}}[\_].$

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

- Case Var $z$: When $x \neq z$, we have

$$S_{\text{usage}}[z] \rho[x \mapsto \rho ! y] \rho ! z$$

\[
\begin{align*}
\text{Case } \text{Lam } z \ e: & \\
S_{\text{usage}}[\text{Lam } z \ e]_{\rho[x \mapsto \rho ! y]} &= \{ \text{Unfold } S_{\text{usage}}[\_ \_ ] \} \\
& \text{fun } z (\lambda d \rightarrow \text{step } A p p_2 \$ S_{\text{usage}}[e]_{\rho[x \mapsto \rho ! y]}[z \mapsto d]) \\
& = \{ \text{Rearrange, } x \neq z \} \\
& \text{fun } z (\lambda d \rightarrow \text{step } A p p_2 \$ S_{\text{usage}}[e]_{\rho[z \mapsto d]}[x \mapsto \rho ! y]) \\
& \leq \{ \text{Induction hypothesis, } x \neq z \} \\
& \text{fun } z (\lambda d \rightarrow \text{step } A p p_2 \$ S_{\text{usage}}[\text{Lam } x \ e ' A p p ' y]_{\rho[z \mapsto d]}) \\
& = \{ \text{Unfold } S_{\text{usage}}[\_ \_ ] \} \\
& S_{\text{usage}}[\text{Lam } z (\text{Lam } x \ e ' A p p ' y)]_{\rho} \\
& = \{ x \neq z, \text{Lemma 32 } \} \\
& S_{\text{usage}}[\text{Lam } x (\text{Lam } z e ' A p p ' y)]_{\rho} \\
\text{Case } \text{App } e z: & \\
S_{\text{usage}}[\text{App } e z]_{\rho[x \mapsto \rho ! y]} &= \{ \text{Unfold } S_{\text{usage}}[\_ \_ ] \} \\
& \text{apply } (S_{\text{usage}}[e]_{\rho[x \mapsto \rho ! y]})(\rho ! y) \\
& \leq \{ \text{Induction hypothesis } \} \\
& \text{apply } (S_{\text{usage}}[\text{Lam } x e ' A p p ' y]_{\rho})(\rho ! y) \\
& = \{ \text{Unfold } apply, S_{\text{usage}}[\_ \_ ] \} \\
& \text{let } \langle \varphi, v \rangle = (S_{\text{usage}}[e]_{\rho[x \mapsto \rho ! y]}[x \mapsto (\rho ! y).\varphi]) \text{ in case peel } v \text{ of } (u, v_2) \rightarrow \langle \varphi + u * ((\rho ! y).\varphi), v_2 \rangle \\
& = \{ \text{Unfold } \_ \_ \rightarrow \_ \_ \} \\
& \text{let } \langle \varphi, v \rangle = S_{\text{usage}}[e]_{\rho[x \mapsto \rho ! y]} \text{ in }
\end{align*}
\]
case peel v of (u, v2) → ⟨φ[x ← U_0] + (φ !? x) * ((ρ ! y).φ) + u * ((ρ ! y).φ), v2⟩
 = { Refold [ ] ⇒ [ ] }
let ⟨φ, v⟩ = S_{usage}[e][ρ[x → prx x]] in

case peel v of (u, v2) → ⟨φ + u * ((prx x).φ), v2⟩[x ⇒ (ρ ! y).φ]
 = { Move out [x ⇒ [ ]], refold apply }
(apply (S_{usage}[e][ρ[x → prx x]]) (prx x))[x ⇒ (ρ ! y).φ]
 = { Refold S_{usage}[ ] [ ] }
S_{usage}[Lam x (App e z) ‘App’ y]_ρ

When x ≠ z:

S_{usage}[App e z]_ρ[x → ρ ! y]
 = { Unfold S_{usage}[ ] x ≠ z }
apply (S_{usage}[e][ρ[x → ρ ! y]]) (ρ ! z)
 = { Induction hypothesis }
apply (S_{usage}[Lam x e ‘App’ y]_ρ) (ρ ! z)
 = { Refold S_{usage}[ ] [ ] }
S_{usage}[Lam x e ‘App’ y ‘App’ z]_ρ
 = { Lemma 33 }
S_{usage}[Lam x (e ‘App’ z) ‘App’ y]_ρ

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

S_{usage}[ConApp k [z]]_ρ[x → ρ ! y]
 = { Unfold S_{usage}[ ] [ ] }
foldl apply (e, Rep U_ω) [ρ[x → ρ ! y] ! z]
 = { Similar to Var case }
foldl apply (e, Rep U_ω) [(ρ[x → prx x] ! z)[x ⇒ (ρ ! y).φ]]
 = { x dead in (e, Rep U_ω), push out substitution }
(foldl apply (e, Rep U_ω) [(prx x) ! z])[x ⇒ (ρ ! y).φ]
 = { Refold S_{usage}[ ] [ ] }
S_{usage}[Lam x (ConApp k [z]) ‘App’ y]_ρ

- **Case Case e alts:** We concentrate on the single alternative e_r, single field binder z case.

S_{usage}[Case e [k → [z], e_r]]_ρ[x → ρ ! y]
 = { Unfold S_{usage}[ ] [ ] step Case2 = id }
select (S_{usage}[e][ρ[x → ρ ! y]]) [k → λ[d] → S_{usage}[e_r][ρ[x → ρ ! y][z → d]]
 = { Unfold select }
S_{usage}[e][ρ[x → ρ ! y] ⇒ S_{usage}[e_r][ρ[x → ρ ! y][z → (e, Rep U_ω)]]
 = { Induction hypothesis }
S_{usage}[Lam x e ‘App’ y]_ρ ⇒ S_{usage}[Lam x e_r ‘App’ y]_ρ[z → (e, Rep U_ω)]
 = { Refold select, S_{usage}[ ] [ ] }
S_{usage}[Case (Lam x e ‘App’ y) alts]_ρ[x → ρ ! y]
 = { Refold select, S_{usage}[ ] [ ] }
S_{usage}[Case (Lam x e ‘App’ y) [k → [z], Lam x e_r ‘App’ y]]_ρ[x → ρ ! y]
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 \( S_{\text{need}}[E[\text{let } x \mapsto e']_\epsilon] \) contains a Lookup \( x \) event. (Otherwise, \( x \) is absent in \( e \).)

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

\[
\begin{align*}
(\text{let } x = e' \text{ in } e, \rho, \mu, \kappa) \xrightarrow{\text{Look}(x)} \cdots \xrightarrow{\text{Look}(x)} \cdots
\end{align*}
\]

We proceed as follows:

\[
\begin{align*}
\iff init(E[\text{let } x = e' \text{ in } e]) \xrightarrow{\text{Look}(x)} \cdots \xrightarrow{\text{Look}(x)} \cdots \\
\iff \alpha_{\text{trans}}(\text{init}(E[\text{let } x = e' \text{ in } e]) \xrightarrow{\star}, []) = \ldots \text{Step (Lookup } x) \ldots \\
\iff S_{\text{need}}[E[\text{let } x \mapsto e']_\epsilon]_\epsilon = \ldots \text{Step (Lookup } x) \ldots
\end{align*}
\]

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 \text{trans} (cf. Lemma 36) such that

\[
\text{init}(E[\text{let } x = e' \text{ in } e]) \xrightarrow{\star} (\text{let } x = e' \text{ in } e, \rho, \mu, \kappa)
\]

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

The next two steps apply adequacy of \( S_{\text{need}}[..] \) to the trace, making the shift from LK trace to denotational interpreter.

\[\square\]

Lemma 9 (\( S_{\text{usage}}[..] \) abstracts \( S_{\text{need}}[..] \)). Let \( e \) be a closed expression and \text{abstract} the abstraction function above. Then \text{abstract} \( (S_{\text{need}}[e]_\epsilon) \subseteq S_{\text{usage}}[e]_\epsilon \).

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

• \text{MONO}: Always immediate, since \( \cup \) and \( + \) are the only functions matching on \( U \), and these are monotonic.

• \text{UNWIND-STUCK, INTRO-STUCK}: Trivial, since \( \text{stuck} = \perp \).
Abstracting Denotational Interpreters 1:49

- **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 (S_{usage}[\_\_\_ infers absence)**. Let \( \rho_e \triangleq [y \mapsto \langle y \mapsto U_1, \text{Rep } U_\omega \rangle] \) be the initial environment with an entry for every free variable \( y \) of an expression \( e \). If \( S_{usage}[\_\_\_ \rho_e = (\varphi, v) \) and \( \varphi !? x = U_0 \), then \( x \) is absent in \( e \).

**Proof.** We show the contraposition, that is, if \( x \) is used in \( e \), then \( \varphi !? x \neq U_0 \).

By **Lemma 8**, there exists \( E, e' \) such that

\[ S_{need}[E[Let x e' e]](\_\_\_ \varphi = \text{... Step (Lookup x) ... .} \]

This is the big picture of how we prove \( \varphi !? x \neq U_0 \) from this fact:

\[
\begin{align*}
  S_{need}[E[Let x e' e]](\varepsilon) = \text{... Step (Lookup x) ...} & \text{ Usage instrumentation} \\
  \Rightarrow (\alpha \{ S_{need}[E[Let x e' e]](\varepsilon) \}).\varphi \ni [x \mapsto U_1] & \text{ Lemma 9} \\
  \Rightarrow (S_{usage}[E[Let x e' e]](\varepsilon)).\varphi \ni [x \mapsto U_1] & \text{ Lemma 38} \\
  \Rightarrow U_\omega \star (S_{usage}[\_\_\_ \rho_e]).\varphi = U_\omega \star \varphi \ni [x \mapsto U_1] & \text{ (8) } \\
  \Rightarrow \varphi !? x \neq U_0 & \text{ (9) }
\end{align*}
\]

Step (5) instruments the trace by applying the usage abstraction function \( \alpha \Rightarrow \_ \triangleq \text{nameNeed}. \) This function will replace every **Step** constructor with the **step** implementation of \( T_U \); The **Lookup x** event on the right-hand side implies that its image under \( \alpha \) is at least \( [x \mapsto U_1] \).

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 \( U_\omega \star S_{usage}[\_\_\_ \rho_e = S_{usage}[E[\_\_\_ \rho_e]] \). 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 \( (\Box, \mu, \kappa) \) of the Sestoft Mark I machine; a variant of Figure 2 that uses syntactic substitution of variables instead of delayed substitution and addresses, so \( \mu \in Var \rightarrow \text{Exp} \) and no closures are needed.

We follow their approach, but inline applicative contexts,\(^{32}\) thus defining the by-need evaluation contexts with hole \( \Box \) for our language as

\[
E \in E \in \begin{cases}
\Box | E \times | \text{case } E \text{ of } K \overline{x} \rightarrow e | \text{let } x = e \text{ in } E | \text{let } x = E \text{ in } E[x]
\end{cases}
\]

\(^{32}\text{The result is that of Ariola et al. [1995, Figure 3] in A-normal form and extended with data types.}\)


[git] • Branch: main @ 5d26cc9 • Release: (2024-03-05)
The correspondence to Mark I machine contexts (□, µ, κ) is encoded by the following translation function \textit{trans} that translates from mark I machine contexts (□, µ, κ) to evaluation contexts \(E\).

\[
\text{trans(}E, [x \mapsto e], \kappa) = \begin{cases} \text{let } x = e \text{ in trans}(E, [], \kappa) \\
\text{trans}(E, [], \text{ap}(x) \cdot \kappa) = \text{trans}(E, x, [], \kappa) \\
\text{trans}(E, [], \text{sel}(\overline{K} \mapsto e) \cdot \kappa) = \text{trans}(\text{case } E \text{ of } \overline{K} \mapsto e, [], \kappa) \\
\text{trans}(E, [], \text{upd}(x) \cdot \kappa) = \begin{cases} \text{let } x = E \text{ in trans}(\square, [], \kappa)[x] \\
\end{cases} \\
\text{trans}(E, [], \text{stop}) = E
\end{cases}
\]

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

**Lemma 36** (Translation, without proof). \(\text{init}(\text{trans}(\square, \mu, \kappa)[e]) \rightsquigarrow^* (\mu, \kappa, \kappa)\), and all transitions in this trace are search transitions (\textit{App\textsubscript{1}}, \textit{Case\textsubscript{1}}, \textit{LET\textsubscript{1}}, \textit{LOOK}).

In other words: every machine configuration \(\sigma\) corresponds to an evaluation context \(E\) and a focus expression \(e\) such that there exists a trace \(\text{init}(E[e]) \rightsquigarrow^* \sigma\) 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 \(\_[]\):

\[
\text{data ECtxt} = \text{ Hole } | \text{ Apply } \text{ ECtxt } \text{ Name } | \text{ Select } \text{ ECtxt } \text{ Alts} \\
| \text{ ExtendHeap } \text{ Name } \text{ Expr } \text{ ECtxt } | \text{ UpdateHeap } \text{ ECtxt } \text{ Expr}
\]

\(\_[] :: \text{ECtxt} \rightarrow \text{Expr} \rightarrow \text{Expr}\)

\(\text{ Hole}[e] = e\)

\(\text{ Apply } E x) [e] = \text{ App } E[e] x\)

\(\text{ Select } E \text{ alts}[e] = \text{ Case } E[e] \text{ alts}\)

\(\text{ ExtendHeap } x e_1 E) [e_2] = \text{ Let } x \ e_1 \ E[e_2]\)

\(\text{ UpdateHeap } x E e_1) [e_2] = \text{ Let } x \ E[e_1] \ e_2\)

**Lemma 37** (Used variables are free). If \(x\) does not occur in \(e\) and in \(\rho\) (that is, \(\forall y. ((\rho \cdot y) . \varphi) ?! x = U_0\)), then \((S_{\text{usage}}[e]_{\rho_E}) . \varphi) ?! x = U_0\).

**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 \((S_{\text{usage}}[E[e]_{\rho_E}) . \varphi) ?! x \subseteq U_{\omega} * (S_{\text{usage}}[e]_{\rho_E}) . \varphi) ?! x\), where \(\rho_E\) and \(\rho_e\) are the initial environments that map free variables \(z\) to their proxy \(([z \mapsto U_1], \text{Rep } U_{\omega})\).

**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** \textbf{Hole}:

  \[
  (S_{\text{usage}}[\text{Hole}[e]_{\rho_E}) . \varphi) ?! x\]

  \[
  \subseteq \begin{cases} 
  \rho_e = \rho_E \\
  U_{\omega} * (S_{\text{usage}}[e]_{\rho_E}) . \varphi) ?! x
  \end{cases}
  \]

  By reflexivity.
• **Case Apply** $E y$: Since $y$ occurs in $E$, it must be different to $x$.

\[
\begin{align*}
(S_{usage}[\text{Apply } E y][e])_{\rho_E}.\varphi \neq x \\
= \quad \text{(Definition of } \_ \langle \_ \rangle \text{)} \\
(S_{usage}[\text{App } E[e] y]_{\rho_E}).\varphi \neq x \\
= \quad \text{(Definition of } S_{usage}[\_ \langle \_ \rangle \text{)} \\
(apply (S_{usage}[E[e]]_{\rho_E}) (\rho_E y) ).\varphi \neq x \\
= \quad \text{(Definition of apply)} \\
\text{let } \langle \varphi, v \rangle = S_{usage}[E[e]]_{\rho_E} \text{ in} \\
\text{case peel } v \text{ of } (u, v_2) \to (\langle \varphi + u * ((\rho_E y).\varphi), v_2 \rangle.\varphi \neq x \\
= \quad \text{(Induction hypothesis)} \\
\text{let } \langle \varphi, v \rangle = S_{usage}[E[e]]_{\rho_E} \text{ in} \\
\text{case peel } v \text{ of } (u, v_2) \to \varphi \neq x \\
= \quad \text{(Refold )} \\
(\langle \rho, v \rangle.\varphi = \varphi \neq x \\
(\rho_{usage}[E[e]]_{\rho_E}).\varphi \neq x \\
= \quad \text{(Induction hypothesis)} \\
U_{\rho_0} \times (S_{usage}[e])_{\rho_0}.\varphi \neq x \\
\end{align*}
\]

• **Case Select** $E alts$: Since $x$ does not occur in $alts$, it is absent in $alts$ as well by Lemma 37.

(Recall that select analyses $alts$ with $\langle \ell, \text{Rep } U_{\omega} \rangle$ as field proxies.)

\[
\begin{align*}
(S_{usage}[\text{Select } E alts][e])_{\rho_E}.\varphi \neq x \\
= \quad \text{(Definition of } \_ \langle \_ \rangle \text{)} \\
(S_{usage}[\text{Case } E[e] alts]_{\rho_E}).\varphi \neq x \\
= \quad \text{(Definition of } S_{usage}[\_ \langle \_ \rangle \text{)} \\
(\text{select } (S_{usage}[E[e]]_{\rho_E}) (\text{cont } \ell alts)).\varphi \neq x \\
= \quad \text{(Definition of select)} \\
(\rho_{usage}[E[e]]_{\rho_E} \triangleright \text{ lub } \langle \ldots alts \ldots \rangle).\varphi \neq x \\
= \quad \text{(x absent in lub } \langle \ldots alts \ldots \rangle) \\
(\rho_{usage}[E[e]]_{\rho_E}).\varphi \neq x \\
= \quad \text{(Induction hypothesis)} \\
U_{\rho_0} \times (S_{usage}[e])_{\rho_0}.\varphi \neq x \\
\end{align*}
\]

• **Case 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 $(S_{usage}[e_1])_{\rho_E}.\varphi \neq x = U_0$ by Lemma 37.

\[
\begin{align*}
(S_{usage}[\text{ExtendHeap } y e_1 E][e])_{\rho_E}.\varphi \neq x \\
= \quad \text{(Definition of } \_ \langle \_ \rangle \text{)} \\
(S_{usage}[\text{Let } y e_1 E[e]]_{\rho_E}).\varphi \neq x \\
= \quad \text{(Definition of } S_{usage}[\_ \langle \_ \rangle \text{)} \\
(\rho_{usage}[E[e]]_{\rho_E} \triangleright \text{ step } \text{(Lookup } y \rangle \text{ (kleeneFix } \lambda d \rightarrow S_{usage}[e_1][\rho_E]_{\rho_E}[\gamma \triangleright \text{ step } \text{(Lookup } y \rangle \text{ d)}])).\varphi \neq x \\
= \quad \text{(Abstract substitution; Lemma 7)} \\
(\rho_{usage}[E[e]]_{\rho_E} \triangleright \text{ step } \text{(Lookup } y \rangle \text{ (kleeneFix } \lambda d \rightarrow S_{usage}[e_1][\rho_E]_{\rho_E}[\gamma \triangleright \text{ step } \text{(Lookup } y \rangle \text{ d)}])).\varphi \neq x \\
= \quad \text{(Unfold } \_ \langle \_ \rangle \text{)} \\
\text{let } \langle \varphi, v \rangle = S_{usage}[E[e]]_{\rho_E}[\gamma \triangleright \text{ step } \text{(Lookup } y \rangle \text{ d)}] \text{ in} \\
\text{let } \langle \varphi_2, v_2 \rangle = \text{step } \text{(Lookup } y \rangle \text{ (kleeneFix } \lambda d \rightarrow S_{usage}[e_1][\rho_E]_{\rho_E}[\gamma \triangleright \text{ step } \text{(Lookup } y \rangle \text{ d)}]) \text{ in} \\
(\varphi[y \mapsto U_0] + (\varphi \neq \varphi_2)) \neq x \\
\end{align*}
\]
= \{ x \text{ absent in } \varphi_2, \text{ see above } \}
\text{let } \langle \varphi, \_ \rangle = S_{usage}[E[e]]_{\rho_E[y \mapsto \{y \mapsto U_1\}, \text{Rep } U_{\omega})} \text{ in } \\
\varphi \ ? \ x \\
\{ \text{ Induction hypothesis } \}
U_{\omega} * (S_{usage}[e]_{\rho_\omega}).\varphi \ ? \ x
\}
• Case 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 (S_{usage}[e_1]_{\rho_E[y \mapsto \{y \mapsto U_1\}, \text{Rep } U_{\omega})}).\varphi \ ? \ x = U_0 \text{ by Lemma 37.}

(S_{usage}[(\text{UpdateHeap } y E e_1)[e]]_{\rho_E}).\varphi \ ? \ x
= \{ \text{ Definition of } \_\_ \}
(S_{usage}[\text{Let } y E[e] e_1]_{\rho_E}).\varphi \ ? \ x
= \{ \text{ Definition of } S_{usage} \}
(S_{usage}[e_1]_{\rho_E[y \mapsto \text{step (Lookup y) (kleeneFix } (\lambda d \to S_{usage}[E[e]]_{\rho_E[y \mapsto \text{step (Lookup y) d})]))}).\varphi \ ? \ x
\{ \text{ Abstract substitution; Lemma 7 } \}
\{ \text{ Unfold } \_\_ \Rightarrow \_ , \langle \varphi, v \rangle . \varphi = \varphi \}
\text{let } \langle \varphi, \_ \rangle = S_{usage}[e_1]_{\rho_E[y \mapsto \{y \mapsto U_1\}, \text{Rep } U_{\omega})} \text{ in } \\
\text{let } \langle \varphi_2, \_ \rangle = \text{step (Lookup y) (kleeneFix } (\lambda d \to S_{usage}[E[e]]_{\rho_E[y \mapsto \text{step (Lookup y) d})])) \text{ in } \\
\langle \varphi[y \mapsto U_0] + (\varphi \ ? \ y) * \varphi_2 \rangle \ ? \ x
= \{ \varphi \ ? \ y \in U_{\omega}, x \text{ absent in } \varphi, \text{ see above } \}
\text{let } \langle \varphi_2, \_ \rangle = \text{step (Lookup y) (kleeneFix } (\lambda d \to S_{usage}[E[e]]_{\rho_E[y \mapsto \text{step (Lookup y) d})])) \text{ in } \\
U_{\omega} * \varphi_2 \ ? \ x
= \{ \text{ Refold } \langle \varphi, v \rangle . \varphi \}
U_{\omega} * \text{(step (Lookup y) (kleeneFix } (\lambda d \to S_{usage}[E[e]]_{\rho_E[y \mapsto \text{step (Lookup y) d})]))).\varphi \ ? \ x
= \{ x \neq y \}
U_{\omega} * \text{(kleeneFix } (\lambda d \to S_{usage}[E[e]]_{\rho_E[y \mapsto d}))).\varphi \ ? \ x
= \{ \text{ Argument below } \}
U_{\omega} * (S_{usage}[E[e]]_{\rho_E[y \mapsto \{y \mapsto U_1\}, \text{Rep } U_{\omega})}).\varphi \ ? \ x
\{ \text{ Induction hypothesis, } U_{\omega} * U_{\omega} = U_{\omega} \}
U_{\omega} * (S_{usage}[e]_{\rho_\omega}).\varphi \ ? \ x

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 \{y \mapsto U_1\}, \text{Rep } U_{\omega}), then it is also absent in E[e]_{\rho_E[y \mapsto d]} per Lemma 37. Otherwise, we go to U_{\omega} anyway.
UpdateHeap is why it is necessary to multiply with U_{\omega} above; in the context let x = \Box in x x, a variable y put in the hole would really be evaluated twice under call-by-name (where let x = \Box 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 let y = ... in e; for e \triangleq y we would infer that y is used at most once, but that is wrong in context let x = \Box in x x.

[]}
D.1 Abstract Interpretation and Denotational Interpreters

So far, we have seen how to use the abstract interpretation 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. 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 inductive) 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 \((D, \leq) \xrightarrow{\gamma} (\widehat{D}, \subseteq)\) between concrete and abstract semantic domains \(D\) and \(\widehat{D}\) equipped with a partial order. An element \(\widehat{d} \in \widehat{D}\) soundly approximates \(d \in D\) iff \(d \leq \gamma \widehat{d}\), iff \(\alpha d \subseteq \widehat{d}\). This theory bears semantic significance when \((D, \leq)\) 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 \(S_T[-]\), defined as \(S_C[e]_\rho \triangleq \{S_T[e]_\rho\}\), provides the strongest trace property that a given program \((e, \rho)\) satisfies. In this setting, we extend the original Galois connection to the signature of \(S_T[-]_\rho^\text{parametrically}\),\(^{33}\) to

\[
((\text{Name} :\rightarrow \wp(\mathbb{T})) \rightarrow \wp(\mathbb{T}), \subseteq) 
\xrightarrow{\alpha \rightarrow \wp(\mathbb{T})} \left( ((\text{Name} :\rightarrow \widehat{D}) \rightarrow \widehat{D}, \subseteq) \right),
\]

and state soundness of the abstract semantics \(S_{\widehat{D}}[-]_\rho^\text{parametrically}\) as

\[
S_C[e]_\rho \subseteq \gamma (S_{\widehat{D}}[e]_{\alpha \triangleleft \rho}) \iff \alpha (S_T[e]_\rho) \subseteq S_{\widehat{D}}[e]_{\alpha \triangleleft \rho}.
\]

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

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 \(\mathbb{T} \triangleq T()\), and that we were to approximate properties \(P \in \wp(\mathbb{T})\) about such traces by a Galois connection \((\wp(\mathbb{T}), \subseteq) \xrightarrow{\gamma} (\widehat{D}, \subseteq)\). Alas, although the abstraction

\[\text{...}
\]

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

Computing with such an \(a\) is of course unacceptable for a static analysis. Usually this is resolved by approximating the fixpoint by the least fixpoint of the abstracted iteratee, e.g., \(lfp\ (a \circ \text{Step}\ (\text{Lookup}\ x) \circ \gamma)\). 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 : \tau \text{ terminates}\}; if \(P\) is restricted to finite traces \(T^*\), the analysis that constantly says “terminates” is correct; however this result doesn’t carry over “to the limit”, when \(\tau\) may also range over infinite traces in \(T^\infty\). 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 [Lampert 1977]:

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

Note that both well-typedness (“\(\tau\) 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

\[
\forall \tau_1 \in T^\infty. \ (\forall \tau_2 \in T^*, \tau_2 \ll \tau_1 \implies \tau_2 \in P) \implies \tau_1 \in P,
\]

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

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

\[
\forall \widehat{d}. \ P \cap T^* \subseteq \gamma(\widehat{d}) \implies P \cap T^\infty \subseteq \gamma^\infty(\widehat{d}),
\]

where the extension \((\varphi(T^*), \subseteq) \xleftarrow{\gamma} (\widehat{D}, \sqsubseteq)\) of \((\varphi(T), \subseteq)\) is defined by the following abstraction function:

\[
a^\infty(P) \triangleq a(\{\tau_2 : \exists \tau_1 \in P. \tau_2 \ll \tau_1\})
\]

**Proof.** First note that \(a^\infty\) uniquely determines the Galois connection by the representation function [Nielson et al. 1999, Section 4.3]

\[
\beta^\infty(\tau_1) \triangleq a(\{\tau_2 : \tau_2 \ll \tau_1\}).
\]

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

\[
\tau \in \gamma^\infty(\widehat{d})
\]

\[
\iff \langle \text{Galois} \rangle \widehat{d}
\]

\[
\iff \langle \text{Definition of} \ \beta^\infty \rangle \widehat{d}
\]

\[
\iff \langle \text{Definition of} \ \beta^\infty \rangle \widehat{d}
\]

\[
\iff \langle \text{Union} \ \rangle \widehat{d}
\]

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

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:

\[ \alpha(\{\text{fix } f\}) \subseteq \text{lfp} (\alpha \circ f^* \circ \gamma), \]

where \( \text{lfp} \tilde{f} \) denotes the least fixpoint of \( \tilde{f} \) and \( f^* : \varphi(\mathbb{T}) \rightarrow \varphi(\mathbb{T}) \) is the lifting of \( f \) to powersets.

**Proof.** We should note that the proposition is sloppy in the treatment of \( \triangleright \) and should rather have been

\[ \alpha(\{\text{fix } f\}) \subseteq \text{lfp} (\alpha \circ f \circ \text{next}^* \circ \gamma), \]

where \( \text{next} :: \triangleright \mathbb{T} \rightarrow \mathbb{T} \). Since we have proven totality in Section 5.2, the utility of being explicit in \( \text{next} \) 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 \( \tau = \text{fix } f \) is finite and proceed by L"ob induction.

\[
\begin{align*}
\alpha \{\text{fix } f\} &\subseteq \text{lfp} (\alpha \circ f^* \circ \gamma) \\
= &\quad \{\text{fix } f = f \ (\text{fix } f\}) \} \\
= &\quad \{\text{Commute } f \text{ and } (.) \} \} \\
\subseteq &\quad \{\text{id } \subseteq \gamma \circ \alpha \} \} \\
\alpha (f^* \{\text{fix } f\}) \} &\subseteq \{\text{Induction hypothesis } \} \} \\
\alpha (f^* (\gamma (\alpha \{\text{fix } f\}))) \} &\subseteq \{\text{Induction hypothesis } \} \} \\
\text{lfp } \tilde{f} = \tilde{f} (\text{lfp } \tilde{f}) \} &\subseteq \{\text{lfp } (\alpha \circ f^* \circ \gamma) \} \}
\end{align*}
\]

When \( \tau \) 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 \( \beta :: a \rightarrow b \) into a partial order \((b, \sqsubseteq)\) yields a Galois connection between Powersets of \( a \) and \((b, \sqsubseteq)\):
While the $\gamma$ exists as a mathematical function, it is in general impossible to compute even for finitary inputs. Every domain $\widehat{D}$ with instances $\langle \text{Trace } \widehat{D}, \text{Domain } \widehat{D}, \text{Lat } \widehat{D} \rangle$ induces a trace abstraction via the following representation function, writing $f^n$ to map $f$ over $\text{Pow}$:

$$\text{type } (d \vdash_{D}^n) = d \quad \text{-- exact meaning defined below}$$

$$\text{trace } :: (\text{Trace } \widehat{d}, \text{Domain } \widehat{d}, \text{Lat } \widehat{d})$$

$$\Rightarrow \text{GC } (\text{Pow } (D \ r) \ \widehat{d}) \to \text{GC } (\text{Pow } (D \ r \ \vdash_{D}^n)) (\widehat{d} \ \vdash_{D}^n) \to \text{GC } (\text{Pow } (T \ (\text{Value } r))) \ \widehat{d}$$

$$\text{trace } (\alpha_T \Rightarrow \gamma_T) \ (\alpha_E \Rightarrow \gamma_E) = \text{repr } \beta \text{ where}$$

$$\beta \text{ (Ret Stuck) } = \text{stuck}$$

$$\beta \text{ (Ret (Fun } f)) = \text{fun } (\alpha_T \circ f^n \circ \gamma_E)$$

$$\beta \text{ (Ret (Con } k \ ds)) = \text{con } k \ (\text{map } (\alpha_E \circ \gamma_E) \ ds)$$

$$\beta \text{ (Step } e \ \widehat{d}) = \text{step } e \ (\beta \ \widehat{d})$$

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) $D \ (\text{ByName } T) \ \vdash_{D}^n$ a subtype of $D \ (\text{ByName } T)$. Every $d :: (\text{ByName } T \ \vdash_{D}^n)$ is of the form $\text{Step } (\text{Lookup } x) \ (S[e]_\rho)$ for some $x, e, \rho$, 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 $\alpha_T$ and $\alpha_E$ 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:

$$\text{env } :: (\text{Trace } \widehat{d}, \text{Domain } \widehat{d}, \text{Lat } \widehat{d}) \Rightarrow \text{GC } (\text{Pow } (D \ (\text{ByName } T) \ \vdash_{D}^n)) (\widehat{d} \ \vdash_{D}^n)$$

$$\text{env } = \text{repr } \beta \text{ where } \beta \ (\text{Step } (\text{Lookup } x) \ (S[e]_\rho)) = \text{step } (\text{Lookup } x) \ (S[e]_{\beta \circ \rho})$$

$$\text{byName } :: (\text{Trace } \widehat{d}, \text{Domain } \widehat{d}, \text{Lat } \widehat{d}) \Rightarrow \text{GC } (\text{Pow } (D \ (\text{ByName } T))) \ \widehat{d}$$

$$\text{byName } = (\alpha_T \circ \text{unByName}) \Rightarrow (\text{ByName}^* \circ \gamma_T) \text{ where } \alpha_T \Rightarrow \gamma_T = \text{trace } \text{byName } \text{env}$$

There is a need to clear up the domain and range of env. Since its domain is a set of elements from $D \ (\text{ByName } T) \ \vdash_{D}^n$, its range $d \ \vdash_{D}^n$ is the (possibly infinite) join over abstracted elements that look like $\text{step } (\text{Lookup } x) \ (S[e]_{\beta \circ \rho})$ for some “closure” $x, e, \rho$. 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 $\widehat{D}$ be a domain satisfying Trace, Domain and Lat. We write $\widehat{D} \ \vdash_{D}^n \ d$ (resp. $\widehat{D} \ \vdash_{E}^n \ \rho$) to say that the denotation $d$ (resp. environment $\rho$) is syntactic, which we define by mutual guarded recursion as

- $\widehat{D} \ \vdash_{D}^n \ d$ iff there exists a set $\text{Clo}$ of syntactic closures such that

  $$d = \bigcup \{ \text{step } (\text{Lookup } x) \ (S[e]_{\rho_1} :: \widehat{D}) \mid (x, e, \rho_1) \in \text{Clo} \land \bullet (\widehat{D} \ \vdash_{E}^n \ \rho_1) \},$$

- $\widehat{D} \ \vdash_{E}^n \ \rho$ iff for all $x$, $\widehat{D} \ \vdash_{E}^n \ (\rho ! x)$.

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

---

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

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

If \( S_{\text{name}}[e]_{\rho_1} = \text{Step } ev (S_{\text{name}}[v]_{\rho_2}) \) in the concrete, then \( \text{step } ev (S[v]_{\alpha_E \leftarrow (-) \& \rho_1}) \subseteq S[e]_{\alpha_E \leftarrow (-) \& \rho_1} \) in the abstract, where \( \alpha_E \vdash \gamma_E = \text{ev} \).

Proof. By Löb induction and cases on \( e \), using the representation function \( \beta_E \triangleq \alpha_E \circ \{ \} \).

- **Case** \( \text{Var } x \): By assumption, we know that \( S_{\text{name}}[x]_{\rho_1} = \text{Step } (\text{Lookup } y) (S_{\text{name}}[e']_{\rho_1}) = \text{Step } ev (S_{\text{name}}[v]_{\rho_2}) \) for some \( y, e', \rho_3 \), so that \( \text{ev} = \text{Lookup } y : \text{ev}_1 \) for some \( \text{ev}_1 \) by determinism.

\[
\begin{align*}
\text{step } ev (S[v]_{\beta_E \& \rho_1}) &= \{ \text{ev} = \text{Lookup } y : \text{ev}_1 \} \\
\text{step } (\text{Lookup } y) (\text{step } ev_1 (S[v]_{\beta_E \& \rho_2})) &= \{ \text{Induction hypothesis at } \text{ev}_1, \rho_3 \text{ as above} \} \\
\text{step } (\text{Lookup } y) (S[e']_{\beta_E \& \rho_1}) &= \{ \text{Refold } \beta_E, \rho_3 \mid x \} \\
\beta_E (\rho_1 \mid x) &= \{ \text{Refold } S[x]_{\beta_E \& \rho_1} \}
\end{align*}
\]

- **Case** \( \text{Lam, ConApp} \): By reflexivity of \( \subseteq \).

- **Case** \( \text{App } e \, x \): Then \( S_{\text{name}}[e]_{\rho_1} = \text{Step } ev_1 (S_{\text{name}}[\text{Lam } y \, \text{body}]_{\rho_1}), S_{\text{name}}[\text{body}]_{\rho_3[y \mapsto \rho_1 \mid x]} = \text{Step } ev_2 (S_{\text{name}}[v]_{\rho_2}) \).

\[
\begin{align*}
\text{step } ev (S[v]_{\beta_E \& \rho_1}) &= \{ \text{ev} = [\text{App}_1] + \text{ev}_1 + [\text{App}_2] + \text{ev}_2, \text{IH at } \text{ev}_2 \} \\
\text{step } \text{App}_1 (\text{step } ev_1 (\text{step } \text{App}_2 (S[\text{body}]_{(\beta_E \& \rho_1)[y \mapsto \beta_E \& \rho_1 \mid x]}))) &= \{ \text{Assumption Beta-App} \} \\
\text{step } \text{App}_1 (\text{step } ev_1 (\text{apply } (S[\text{Lam } y \, \text{body}]_{\beta_E \& \rho_1}) (\beta_E \& \rho_1 \mid x))) &= \{ \text{Assumption Beta-App} \} \\
\text{step } \text{App}_1 (\text{apply } (\text{step } ev_1 (S[\text{Lam } y \, \text{body}]_{\beta_E \& \rho_1})) (\beta_E \& \rho_1 \mid x)) &= \{ \text{Induction hypothesis at } \text{ev}_1 \} \\
\text{step } \text{App}_1 (\text{apply } (S[e]_{\beta_E \& \rho_1})) (\beta_E \& \rho_1 \mid x) &= \{ \text{Refold } S[\text{App } e \, x]_{\beta_E \& \rho_1} \}
\end{align*}
\]

- **Case** \( \text{Case } e \, \text{alts} \): Then \( S_{\text{name}}[e]_{\rho_1} = \text{Step } ev_1 (S_{\text{name}}[\text{ConApp } k \, \text{ys}]_{\rho_1}), S_{\text{name}}[e_r]_{\rho_1[\text{map } (\rho_3 \mid y)]} = \text{Step } ev_2 (S_{\text{name}}[v]_{\rho_2}) \), where \( \text{alts } k = (x, e_r) \) is the matching RHS.

\[
\begin{align*}
\text{step } ev (S[v]_{\beta_E \& \rho_1}) &= \{ \text{ev} = [\text{Case}_1] + \text{ev}_1 + [\text{Case}_2] + \text{ev}_2, \text{IH at } \text{ev}_2 \} \\
\text{step } \text{Case}_1 (\text{step } ev_1 (\text{step } \text{Case}_2 (S[e_r]_{\beta_E \& \rho_1[\text{map } (\rho_3 \mid y)]}))) &= \{ \text{Assumption Beta-App} \} \\
\text{step } \text{Case}_1 (\text{select } (S[\text{ConApp } k \, \text{ys}]_{\beta_E \& \rho_1}) (\text{cont } \& \text{alts})) &= \{ \text{Assumption Beta-App} \} \\
\text{step } \text{Case}_1 (\text{select } (\text{step } ev_1 (S[\text{ConApp } k \, \text{ys}]_{\beta_E \& \rho_1})) (\text{cont } \& \text{alts})) &= \{ \text{Induction hypothesis at } \text{ev}_1 \}
\end{align*}
\]
\[
\text{step Case}_1 \quad (\text{select } (S[e])_{\beta_E \triangleright} (\text{cont } \triangleleft \text{ alts}))
\]
\[
= \left\{ \begin{array}{l}
\text{Refold } S[\text{Case } e \text{ alts}]_{\beta_E \triangleright} \\
S[\text{Case } e \text{ alts}]_{\beta_E \triangleright}
\end{array} \right.
\]

- **Case** Let \( x \in e \in e_2 \): We make good use of **Lemma 41** below:

\[
\text{step } ev \quad (S[v]_{\beta_E \triangleright})
\]
\[
= \left\{ \begin{array}{l}
\text{ev } = \text{Let}_1 : ev_1 \\
\text{step } \text{Let}_1 \quad (\text{step } ev_1) (S[v]_{\beta_E \triangleright})
\end{array} \right.
\]

\[
\subseteq \quad \text{Induction hypothesis at } ev_1
\]

\[
\text{step } \text{Let}_1 \quad (S[e_2]_{\beta_E \triangleright}) (x \mapsto S[\text{Step } (\text{Lookup } x) \ (\text{fix } (\lambda d_4 \mapsto S[\text{name } e_1]_{\rho_4 (x \mapsto \text{Step } (\text{Lookup } x) \ d_4)))))
\]

\[
\subseteq \quad \text{Partially roll } fix
\]

\[
\text{step } \text{Let}_1 \quad (S[e_2]_{\beta_E \triangleright}) (x \mapsto S[\text{Step } (\text{Lookup } x) \ (S[\text{name } e_1]_{\rho_4 (x \mapsto d_4)))))
\]

\[
\subseteq \quad \text{Lemma } 41
\]

\[
\text{step } \text{Let}_1 \quad (S[e_2]_{\beta_E \triangleright}) (x \mapsto \text{lfp } \lambda d_4 \mapsto \text{Step } (\text{Lookup } x) \ (S[e_1]_{\beta_E \triangleright}(x \mapsto \text{Step } (\text{Lookup } x) \ d_4)))
\]

\[
\subseteq \quad \text{Partially unroll } \text{lfp}
\]

\[
\text{step } \text{Let}_1 \quad (S[e_2]_{\beta_E \triangleright}) (x \mapsto \text{Step } (\text{Lookup } x) \ (\text{lfp } \lambda d_4 \mapsto S[e_1]_{\beta_E \triangleright}(x \mapsto \text{Step } (\text{Lookup } x) \ d_4)))
\]

\[
\subseteq \quad \text{Assumption BIND-BYNAME}
\]

\[
\text{bind } (\lambda d_4 \mapsto S[e_1]_{\beta_E \triangleright}(x \mapsto \text{Step } (\text{Lookup } x) \ d_4)) (\lambda d_4 \mapsto \text{Let}_1 \quad (S[e_2]_{\beta_E \triangleright}(x \mapsto \text{Step } (\text{Lookup } x) \ d_4)))
\]

\[
= \left\{ \begin{array}{l}
\text{Refold } S[\text{Let } x \ e_1 e_2]_{\beta_E \triangleright}
\end{array} \right.
\]

We can now prove the by-name abstraction theorem:

**Theorem 44** (Sound By-name Interpretation). Let \( \mathcal{D} \) be a domain with instances for Trace, Domain, HasBind and Lat, and let \( \alpha_T \models \gamma_T \triangleq \text{by Name}, \alpha_E \models \gamma_E \triangleq \text{env} \). If the by-name abstraction laws in **Figure 13** hold, then \( S[.] \) instantiates to an abstract interpreter that is sound wrt. \( \gamma_E \rightarrow \alpha_T \), that is,

\[
\alpha_T \quad (S[\text{name } e]_{\rho}) :: \text{Pow } (\mathcal{D} \ (\text{ByName T})) \subseteq S[\mathcal{D}[e]]_{\alpha_T \triangleright} \triangleleft \rho.
\]

**Proof.** We first restate the goal in terms of the representation functions \( \beta_T \triangleq \alpha_T \circ \{\_\} \) and \( \beta_E \triangleq \alpha_E \circ \{\_\} \):

\[
\forall \rho. \quad \beta_T \ (S[\text{name } e]_{\rho}) \subseteq (S[\mathcal{D}[e]]_{\beta_E \triangleright} \rho).
\]

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

- **Case** \( \text{Var } x \): The stuch case follows by unfolding \( \alpha_T \). Otherwise,

\[
\beta_T \quad (\rho \ x)
\]

\[
= \left\{ \begin{array}{l}
\text{Pow } (\mathcal{D} \ (\text{ByName T})) \downarrow e \ {\_} \triangleleft \rho, \text{ Unfold } \beta_E
\end{array} \right.
\]

\[
\text{step } (\text{Lookup } y) \quad (\beta_T \ (S[\text{name } e]_{\rho'}))
\]

\[
\subseteq \quad \text{Induction hypothesis}
\]

\[
\text{step } (\text{Lookup } y) \quad (S[e']_{\beta_E \triangleright} \rho')
\]

\[
= \left\{ \begin{array}{l}
\text{Refold } \beta_E
\end{array} \right.
\]

\[
\beta_E \quad (\rho \ x)
\]
Abstracting Denotational Interpreters 1:59

- **Case Lam x body:**
  
  \[ \beta_T \left( S_{name[Lam \times body]} \rho \right) \]
  
  = \{ Unfold \[ S[-] \], \beta_T \}
  
  \[
  \text{fun} \left( \lambda d \rightarrow \bigcup \{ \text{step App}_2 \left( \beta_T \left( S_{name[body]} \rho [x \mapsto d] \right) \right) \mid \beta_E \ d \subseteq \lambda d \} \right) 
  \]
  
  \[ \subseteq \{ \text{Induction hypothesis} \} \]
  
  \[
  \text{fun} \left( \lambda d \rightarrow \bigcup \{ \text{step App}_2 \left( S[body] \beta_E \rho [x \mapsto d] \right) \mid \beta_E \ d \subseteq \lambda d \} \right) 
  \]
  
  \[ \subseteq \{ \text{Least upper bound} / \alpha_E \circ \gamma_E \subseteq \text{id} \} \]
  
  \[
  \text{fun} \left( \lambda d \rightarrow \text{step App}_2 \left( S[body] \left( \left( \beta_E \rho \right) [x \mapsto d] \right) \right) \right) 
  \]
  
  \[ = \{ \text{Refold} \ S[-] \} \]
  
  \[ S[Lam \times body] \beta_E \rho \]

- **Case ConApp k ds:**

  \[
  \beta_T \left( S_{name[ConApp k x]} \rho \right) 
  \]
  
  = \{ Unfold \[ S[-] \], \beta_T \}
  
  \[
  \text{con} \ k \left( \text{map} \left( \left( \beta_E \rho \right)! \right) \ x s \right) 
  \]
  
  \[ = \{ \text{Refold} \ S[-] \} \]
  
  \[ S[Lam \times body] \beta_E \rho \]

- **Case App e x:** The stuck case follows by unfolding \( \beta_T \).

  Our proof obligation can be simplified as follows

  \[
  \beta_T \left( S_{name[App e x]} \rho \right) 
  \]
  
  = \{ Unfold \[ S[-] \], \beta_T \}
  
  \[
  \text{step App}_1 \left( \beta_T \left( \text{apply} \left( S_{name[e]} \rho \right) \left( \rho! x \right) \right) \right) 
  \]
  
  \[ = \{ \text{Unfold} \ \text{apply} \} \]
  
  \[
  \text{step App}_1 \left( \beta_T \left( S_{name[e]} \rho \right) \Rightarrow \lambda \text{case Fun} \ f \rightarrow f \left( \rho! x \right) ; \_ \rightarrow \text{stuck} \right) 
  \]
  
  \[ \subseteq \{ \text{By cases, see below} \} \]
  
  \[
  \text{step App}_1 \left( \text{apply} \left( S[e] \beta_E \rho \right) \left( \left( \beta_E \rho \right)! x \right) \right) 
  \]
  
  \[ = \{ \text{Refold} \ S[-] \} \]
  
  \[ S[App e x] \beta_E \rho \]

  When \( S_{name[e]} \rho \) diverges, we have

  \[
  = \{ S_{name[e]} \rho \text{ diverges, unfold} \beta_T \} \]
  
  \[
  \text{step ev}_1 \left( \text{step ev}_2 \left( \ldots \right) \right) 
  \]
  
  \[ \subseteq \{ \text{Assumption} \ \text{STEP-APP} \} \]
  
  \[
  \text{apply} \left( \text{step ev}_1 \left( \text{step ev}_2 \left( \ldots \right) \right) \left( \beta_E \rho \right)! x \right) 
  \]
  
  \[ = \{ \text{Refold} \ \beta_T, S_{name[e]} \rho \} \]
  
  \[
  \text{apply} \left( \beta_T \left( S_{name[e]} \rho \right) \right) \left( \beta_E \rho \right)! x 
  \]
  
  \[ \subseteq \{ \text{Induction hypothesis} \} \]
  
  \[
  \text{apply} \left( S[e] \beta_E \rho \right) \left( \beta_E \rho \right)! x 
  \]

  Otherwise, \( S_{name[e]} \rho \) must produce a value \( v \). If \( v = \text{Stuck} \) or \( v = \text{Con} \ k \ ds \), we set \( d \triangleq \text{stuck} \) (resp. \( d \triangleq \text{con} \ k \left( \text{map} \ \beta_E \ ds \right) \)) and have

  \[
  \beta_T \left( S_{name[e]} \rho \right) \Rightarrow \lambda \text{case Fun} \ f \rightarrow f \left( \rho! x \right) ; \_ \rightarrow \text{stuck} 
  \]
  
  \[
  = \{ S_{name[e]} \rho \} = \text{Step ev} \left( \text{return} \ v \right), \text{unfold} \beta_E \}
  \]
  
  \[
  \text{step ev} \left( \beta_T \left( \text{return} \ v \Rightarrow \lambda \text{case Fun} \ f \rightarrow f \left( \rho! x \right) ; \_ \rightarrow \text{stuck} \right) \right) 
  \]
\[ \begin{align*}
&\text{Assumptions UNWIND-STUCK, INTRO-STUCK where } d \triangleq \text{stuck or } d \triangleq \text{con } k \text{ (map } \beta_T \text{ ds)} \set
&\text{Assumption STEP-APP } \set
&\text{Refold } \beta_T, S_{\text{name}}[e]_\rho \set
&\text{Unfold hypothesis } \set
&\lambda \text{ Case } f \rightarrow f (\rho ! x) ; \_ \rightarrow \text{stuck} \set
&\text{Case } e \text{ alts: The stuck case follows by unfolding } \beta_T. \text{ When } S_{\text{name}}[e]_\rho \text{ diverges or does not evaluate to } S_{\text{name}}[\text{ConApp } k \text{ ys}]_\rho, \text{ the reasoning is similar to App } e \text{ x, but in a select context. So assume that } S_{\text{name}}[e]_\rho = \text{Step ev } (S_{\text{name}}[\text{ConApp } k \text{ ys}]_\rho, k) \text{ and that there exists } ((\text{cont }< \text{ alts}) ! k) \text{ ds = Step Case } 2 (S_{\text{name}}[e_r]_\rho |(\text{ys } \rightarrow d_1)) \set
&\text{Unfold } \beta_T, S_{\text{name}}[\text{Case } e \text{ alts}]_\rho \set
&\text{Unfold select } \set
&\lambda \text{ Case } k \text{ ds | } k \in \text{dom alts } \rightarrow ((\text{cont }< \text{ alts}) ! k) \text{ ds) } \set
&\text{Simplify return } (\text{Con } k \text{ ds}) \Rightarrow f = f \text{ (Con } k \text{ ds), (cont }< \text{ alts}) ! k \text{ as above } \set
&\text{Unfold } \beta_T \set
&\text{Step Case } 1 \text{ (Step Case } 2 (S_{\text{name}}[e_r]_\rho |(\text{ys } \rightarrow \text{map } (\rho_1) \text{ ys}))))\end{align*} \]
\[ \{ \text{Induction hypothesis} \} \]
\[
\text{step Case}_1 \ (\text{step ev} \ (\text{step Case}_2 \ (S[e_1]_{(\beta_{\mathcal{E}}<\rho_p)}[x\mapsto\text{map} \ ((\beta_{\mathcal{E}}<\rho_p) \triangleright y)])))
\]
\[ = \{ \text{Refold cont} \} \]
\[
\text{step Case}_1 \ (\text{cont} \ (\text{alts} \ k) \ (\text{map} \ ((\beta_{\mathcal{E}}<\rho_k) \triangleright k) \ xs))
\]
\[ \sqsubseteq \{ \text{Assumption } \text{BETA-SEL} \} \]
\[
\text{step Case}_1 \ (\text{step ev} \ (\text{select} \ (S[\text{ConApp} \ k \ yz]_{\beta_{\mathcal{E}}<\rho_1})) \ (\text{cont} \ < \text{alts}))
\]
\[ \sqsubseteq \{ \text{Assumption } \text{STEP-SEL} \} \]
\[
\text{step Case}_1 \ (\text{select} \ (S[e]_{\beta_{\mathcal{E}}<\rho_p}) \ (\text{cont} \ < \text{alts}))
\]
\[ = \{ \text{Refold } S[-] \} \]
\[
S[e]_{\beta_{\mathcal{E}}<\rho_p}
\]

- \[ \text{Case} \ Let \ x \ e_1 \ e_2; \]
\[
\beta_\Gamma (S[\text{Name}][\text{Let} \ x \ e_1 \ e_2]_{\rho})
\]
\[ = \{ \text{Unfold } S[-] \} \]
\[
\beta_\Gamma (\text{bind} \ (\lambda d_1 \rightarrow S[\text{Name}]_{e_1} \rho[x\mapsto\text{Step} \ (\text{Lookup} \ x) \ d_1])
\]
\[ (\lambda d_1 \rightarrow \text{Step} \ 
\text{Let}_1 \ (S[\text{Name}]_{e_2} \rho[x\mapsto\text{Step} \ (\text{Lookup} \ x) \ d_1]))
\]
\[ = \{ \text{Unfold } \text{bind}, \beta_\Gamma \} \]
\[
\text{step} \ \text{Let}_1 \ (\beta_\Gamma (S[\text{Name}]_{e_1} \rho[x\mapsto\text{Step} \ (\text{Lookup} \ x) \ d_1]) \ (\text{fix} \ (\lambda d_1 \rightarrow S[\text{Name}]_{e_1} \rho[x\mapsto\text{Step} \ (\text{Lookup} \ x) \ d_1])))
\]
\[ = \{ \text{Induction hypothesis} \} \]
\[
\text{step} \ \text{Let}_1 \ (S[e_2]_{\beta_{\mathcal{E}}<\rho_p}[x\mapsto\text{Step} \ (\text{Lookup} \ x) \ (\text{fix} \ (\lambda d_1 \rightarrow S[\text{Name}]_{e_1} \rho[x\mapsto\text{Step} \ (\text{Lookup} \ x) \ d_1])))
\]

And from hereon, the proof is identical to the \text{Let} case of Lemma 43:

\[ \sqsubseteq \{ \text{By Lemma } 41, \text{as in the proof for Lemma } 43 \} \]
\[
\text{step} \ \text{Let}_1 \ (S[e_2]_{\beta_{\mathcal{E}}<\rho_p}[x\mapsto\text{Step} \ (\text{Lookup} \ x) \ (\text{fix} \ (\lambda d_1 \rightarrow S[e_1]_{\beta_{\mathcal{E}}<\rho_p}[x\mapsto\text{Step} \ (\text{Lookup} \ x) \ d_1])))
\]
\[ \sqsubseteq \{ \text{Assumption } \text{BIND-BY-NAME}, \text{with } \rho = \beta_{\mathcal{E}} < \rho \} \]
\[
\text{bind} \ (\lambda d_1 \rightarrow S[e_1]_{\beta_{\mathcal{E}}<\rho_p}[x\mapsto\text{Step} \ (\text{Lookup} \ x) \ d_1])
\]
\[ (\lambda d_1 \rightarrow \text{step} \ \text{Let}_1 \ (S[e_2]_{\beta_{\mathcal{E}}<\rho_p}[x\mapsto\text{Step} \ (\text{Lookup} \ x) \ d_1]))
\]
\[ = \{ \text{Refold } S[\text{Let} \ x \ e_1 \ e_2]_{\beta_{\mathcal{E}}<\rho_p} \]
\[
S[e]_{\beta_{\mathcal{E}}<\rho_p}
\]

\[ \square \]

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

\textbf{Lemma 45} \ (S[\text{usage}][-] \text{- abstracts } S[\text{name}][-], \text{open}). Usage analysis \text{S[usage][-]} \text{- is sound wrt. } S[\text{name}][-],

that is,

\[ \alpha_\Gamma \ \{ S[\text{name}]_{\rho} \} \sqsubseteq \{ S[\text{usage}][e]_{\alpha_{\mathcal{E}}<\rho} :: D_U \} \text{ where } \alpha_\Gamma \models \_ = \text{byName}; \alpha_\Xi \models \_ = \text{env}. \]

\textbf{Proof.} By Theorem 44, it suffices to show the abstraction laws in Figure 13 as done in the proof for Lemma 9. \[\square\]

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

\textbf{Example 46. Let } z \neq x \neq y. \text{ The monotone function } f \text{ defined as follows}

freezeHeap :: (Trace \( \hat{d} \), Domain \( \hat{d} \), Lat \( \hat{d} \)) \( \Rightarrow \) nameNeed \( \mu \)

freezeHeap \( \mu \) = repr \( \beta \) where \( \beta \) (Step (Lookup \( x \)) (fetch \( a \))) | memo \( a \) (S\( \text{need}[e] \rho \)) \( \Leftrightarrow \) \( \mu ! a \)

= step (Lookup \( x \)) (S\( [e] \beta_\rho \))

nameNeed :: (Trace \( \hat{d} \), Domain \( \hat{d} \), Lat \( \hat{d} \)) \( \Rightarrow \) GC (Pow (T (Value (ByNeed T), r\( \text{ne} \rho \)))) \( \hat{d} \)

nameNeed = repr \( \beta \) where

\( \beta \) (Step \( e \ d \)) = step \( e \) (\( \beta \) \( d \))

\( \beta \) (Ret (Stuck, \( \mu \))) = stuck

\( \beta \) (Ret (Fun \( f \), \( \mu \))) = fun (\( \lambda \hat{d} \rightarrow \{ \beta (f \ d \ \mu) \mid d \in \gamma_\hat{d} \}) \) where \( \_ \) \( \Rightarrow \) \( \gamma_\hat{d} = \text{freezeHeap} \mu \)

\( \beta \) (Ret (Con \( k \) \( ds \), \( \mu \))) = con \( k \) (map (\( \alpha_\hat{e} \circ \{ \_ \}) \) \( ds \)) where \( \alpha_\hat{e} \_ \_ = \text{freezeHeap} \mu \)

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

\[ f : \text{D}_U \rightarrow \text{D}_U \]
\[ f \langle \varphi, \_ \rangle = \text{if} \ \varphi \ ? ! \ y \in U_0 \ \text{then} \langle e, \text{Rep U}_0 \rangle \ \text{else} \langle [z \mapsto U_1], \text{Rep U}_0 \rangle \]

violates \( f \ a \subseteq \text{apply} (\text{fun} | \text{x} \ f) \ a \). To see that, let \( a \triangleq \langle [y \mapsto U_1], \text{Rep U}_0 \rangle : \text{D}_U \) and consider

\[ f \ a = \langle [z \mapsto U_1], \text{Rep U}_0 \rangle \not\subseteq \langle e, \text{Rep U}_0 \rangle = \text{apply} (\text{fun} | \text{x} \ f) \ a. \]

D.4 Abstract By-need Soundness, in Detail

Now that we have gained some familiarity with the proof framework while proving Theorem 44, 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 \( d : \text{D} (\text{ByNeed T}) \) as an atomic denotation, such a denotation actually only makes sense when it travels together with an environment \( \rho \) that ties free variables to their addresses in the heap that \( d \) expects.

For our purposes, the key is that a by-name environment \( \rho \) and a heap \( \mu \) can be “frozen” into a corresponding by-name environment. This operation forms a Galois connection freezeHeap in Figure 18, where \( \text{r}^{\text{ne}}_{\text{D}} \) serves a similar purpose as \( \text{d} \text{r}^{\text{na}}_{\text{D}} \) from Definition 42, restricting environment entries to the syntactic by-need form Step (Lookup \( x \)) (fetch \( a \)) and heap entries in \( \text{r}^{\text{ne}}_{\text{D}} \) to memo \( a \) (\( S_\text{need}[e] \rho \)).

Definition 47 (Syntactic by-name heaps and environments, address domain). We write \( \text{r}^{\text{ne}}_{\text{E}} \rho \) (resp. \( \text{r}^{\text{ne}}_{\text{D}} \mu \)) to say that the by-name environment \( \rho : \text{Name} \rightarrow \text{Pow} (\text{D (ByNeed T)}) \) (resp. by-need heap \( \mu \)) is syntactic, defined by mutual guarded recursion as

- \( \text{r}^{\text{ne}}_{\text{E}} \ d \) iff there exists a set \( \text{Clo} \) of syntactic closures such that

  \( d \triangleq \{ \text{Step (Lookup } x \) (fetch \( a \)) \mid (x, a) \in \text{Clo} \}. \)

- \( \text{r}^{\text{ne}}_{\text{E}} \rho \) iff for all \( x, \text{r}^{\text{ne}}_{\text{D}} \rho \! x. \)

- \( \text{adam } d \triangleq \{ a \mid \text{Step (Lookup } y \) (fetch \( a \)) \in d \} \)

We refer to $\text{dom } d$ (resp. $\text{dom } a$) as the address domain of $d$ (resp. $a$).

We assume that all concrete environments Name:→ D (By Need T) and heaps Heap (By Need T) satisfy $\text{extr } E \rightarrow \text{extr } H$ resp. $\text{extr } E \rightarrow \text{extr } H$. It is easy to see that syntactic closeness is preserved by $S_{\text{need }}[\_ ](\_)$ whenever the environment or heap is extended, assuming that Domain and Has Bind are adjusted accordingly.

The environment abstraction $\alpha E \mu \rightarrow \gamma = \text{freezeHeap } \mu$ improves the more "evaluated" $\gamma$ is. E.g., when $\mu_1$ progresses into $\mu_2$ during evaluation, written $\mu_1 \rightarrow_\mu \mu_2$, it is $\alpha E \mu_2 \theta \in \alpha E \mu_1 \theta$ for all $\theta$. The heap progression relation is formally defined (on syntactic heaps $\text{extr } E \rightarrow \text{extr } H$) in Figure 19, and we will now work toward a proof for the approximation statement about $\alpha E$ in Lemma 54.

Transitivity and reflexivity of ($\rightarrow$) are definitional by rules $\rightarrow\rightarrow \text{Ref}l$ and $\rightarrow\rightarrow \text{Trans}$; antisymmetry is not so simple to show for a lack of full abstraction.

Corollary 48. ($\rightarrow$) 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 $\rightarrow\rightarrow \text{Ext}$ and evaluation of a Var will memoise the evaluated heap entry, progressing it along $\rightarrow\rightarrow \text{Memo}$.

Lemma 49 (Evaluation progresses the heap). If $S_{\text{need }}[e ](\mu_1) = \text{Step } ev (S_{\text{need }}[v ](\mu_2))$, then $\mu_1 \rightarrow \mu_2$.

Proof. By Löb induction and cases on $e$. Since there is no approximation yet, all occurring closure sets in $\text{extr } E \rightarrow \text{extr } H$ are singletons.

• Case Var $x$: Let $\overline{\tau v}_1 \rightarrow \text{tail } (\text{init } (\overline{\tau v}))$.

  \[
  (\mu_1 ! x) \mu_1
  = \begin{cases}
    \text{extr } \mu_1, \text{ some } y, a \rightarrow \mu_1 \\
    \text{Step } (\text{Lookup } y ) (\text{fetch } a \mu_1) \\
    \text{Step } (\text{Lookup } y ) ((\mu_1 ! a) \mu_1) \\
    \text{ extr } \mu_1, \text{ some } e, \mu_3 \rightarrow \mu_1 \\
    \text{Step } (\text{Lookup } y ) (\text{memo } a (S_{\text{need }}[e ](\mu_1))) \\
    \text{ extr } \mu, \text{ some } e, \mu_3 \rightarrow \mu_1 \\
    \text{Step } (\text{Lookup } y ) (S_{\text{need }}[e ](\mu_1) \Rightarrow \text{upd}) \\
    \text{Step } (\text{Lookup } y ) (S_{\text{need }}[e ](\mu_1)) \Rightarrow \text{upd}
  \end{cases}
\]
Step (Lookup y) (Step ev₁ (Sneed[v]₁ (µ₃) ⇒ λv µ₃ →)
Step Update (Ret (v, µ₃[a ← memo a (return v)])))

Now let sv::Value (ByNeed T) be the semantic value such that Sneed[v]₁ (µ₃) = Ret (sv, µ₃).

= \{ Sneed[v]₁ (µ₃) = Ret (sv, µ₃) \}

Step (Lookup y) (Step ev₁ (Step Update (Ret (sv, µ₃[a ← memo a (return sv)]))))

= \{ Refold Sneed[v]₁ (µ₃), ev = [Lookup y] + ev₁ + [Update] \}

Step ev (Sneed[v]₁ (µ₃[a ← memo a (Sneed[v]₁ (µ₃))])

= \{ Determinism of Sneed[.] (.), assumption \}

Step ev (Sneed[v]₁ (µ₂))

We have

\[ µ₁ \triangleright a = memo a (Sneed[e]₁ (µ₁)) \tag{10} \]

- \( (Sneed[e]₁ (µ₁) = Step ev₁ (Sneed[v]₁ (µ₃))) \)

\[ µ₂ = µ₃[a ← memo a (Sneed[v]₁ (µ₂))] \tag{12} \]

We can apply rule \( \sim \rightarrow \text{-MEMO} \) to Equation (10) and Equation (11) to get \( µ₁ \sim µ₃[a ← memo a (Sneed[v]₁ (µ₂))] \), and rewriting along Equation (12) proves the goal.

- **Case** Lam x body, ConApp k xs: Then \( µ₁ = µ₂ \) and the goal follows by \( \sim \rightarrow \text{-REFL} \).

- **Case** App e₁ x: Let us assume that \( Sneed[e₁]₁ (µ₁) = Step ev₁ (Sneed[Lam y e₂]₁ (µ₃)) \) and \( Sneed[e₂]₁ (µ₃) = Step ev₂ (Sneed[v]₁ (µ₂)) \), so that \( µ₁ \sim µ₃, µ₃ \sim µ₂ \) by the induction hypothesis. The goal follows by \( \sim \rightarrow \text{-TRANS} \), because \( ev = [App₁] + ev₁ + [App₂] + ev₂ \).

- **Case** Case e₁ alts: Similar to App e₁ x.

- **Case** Let x e₁ e₂:

\[ Sneed[Let x e₁ e₂]₁ (µ₁) \]

\[ = \{ Unfold Sneed[.] (.) \}

\[ bind (λd₁ → Sneed[e₁]₁ (µ₁)[x → step (Lookup x) d₁] (.)

(λd₁ → step Let₁ (Sneed[e₂]₁ (µ₂)[x → step (Lookup x) d₁] (.)

\[ µ₁ \]

\[ = \{ Unfold bind, a ≅ nextFree µ with a ∉ dom µ \}

\[ step Let₁ (Sneed[e₂]₁ [x → step (Lookup x) (fetch a)] (µ₁[a ← memo a (Sneed[e₁]₁ [x → step (Lookup x) (fetch a)])])

At this point, we can apply the induction hypothesis to \( Sneed[e₂]₁ [x → step (Lookup x) (fetch a)] (.) \) to conclude that \( µ₁[a ← memo a (Sneed[e₁]₁ [x → step (Lookup x) (fetch a)])] \sim µ₂ \). On the other hand, we have \( µ₁ \sim µ₁[a ← memo a (Sneed[e₁]₁ [x → step (Lookup x) (fetch a)])] \)

by rule \( \sim \rightarrow \text{-EXT} \) (note that \( a ∉ dom µ \)), so the goal follows by \( \sim \rightarrow \text{-TRANS} \).

Lemma 49 exposes nested structure in \( \sim \rightarrow \text{-MEMO} \). For example, if \( µ₁ \sim µ₂[a ← memo a (Sneed[v]₁ (µ₂))] \) is the result of applying rule \( \sim \rightarrow \text{-MEMO} \), then we obtain a proof that the memoised expression \( Sneed[e]₁ (µ₁) = Step ev (Sneed[v]₁ (µ₂)) \), and this evaluation in turn implies that \( µ₁ \sim µ₂ \).

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 \( \mu_1 \leadsto \mu_2 \) and \( \mu_2 \neq \mu_1 \), then \( \mu_1 \neq \mu_2 \).

Proof. Simple proof by induction on \( \mu_1 \leadsto \mu_2 \). The only case updating a heap entry is \( \leadsto \) \( \text{memo} \) because evaluating \( v \) in \( \mu_1 \) does not make a step.

Lemma 51 (No update implies semantic irrelevance). If \( S_{\text{need}}[e]_{\rho_1}(\mu_1) = \text{Step} \; e \; v \; (S_{\text{need}}[v]_{\rho_2}(\mu_2)) \) and \( \mu_1 \neq \mu_2 \), then \( \mu_1 \neq \mu_2 \).

Proof. By Löb induction and cases on \( e \).

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

  \[
  S_{\text{need}}[x]_{\rho_1}(\mu_1) = \text{Step} \; (\text{Lookup} \; y) \; (\text{memo} \; a \; (S_{\text{need}}[e_1]_{\rho_1}(\mu_1[a \mapsto d])))
  \]

  The goal follows from applying the induction hypothesis and realising that \( \mu_2 \neq \mu_1 \) has been updated consistently with \( \text{memo} \; a \; (S_{\text{need}}[v]_{\rho_2}) \).

- Case \( \text{Lam} \; x \; e \), \( \text{ConApp} \; k \; x \): Easy to see for \( \mu_1 = \mu_2 \).

- Case \( \text{App} \; e \; x \): We can apply the induction hypothesis twice, to both of

  \[
  S_{\text{need}}[e]_{\rho_1}(\mu_1) = \text{Step} \; e \; v \; (S_{\text{need}}[\text{Lam} \; y \; \text{body}]_{\rho_1}(\mu_3))
  \]

  \[
  S_{\text{need}}[\text{body}]_{\rho_1}[y \mapsto \rho_1 \cdot x](\mu_3) = \text{Step} \; e \; v \; (S_{\text{need}}[v]_{\rho_2}(\mu_2))
  \]

to show the goal.

- Case \( \text{Case} \; e \; \text{als} \): Similar to \( \text{App} \).

- Case \( \text{Let} \; x \; e_1 \; e_2 \): We have \( S_{\text{need}}[\text{Let} \; x \; e_1 \; e_2]_{\rho_1}(\mu_1) = \text{Step} \; \text{Let}_1 \; (S_{\text{need}}[e_2]_{\rho'_1}(\mu'_1)) \), where

  \[
  \rho'_1 \triangleq \rho_1[x \mapsto \text{Step} \; (\text{Lookup} \; x) \; (\text{fetch} \; a_1), a_1 \triangleq \text{nextFree} \; \mu_1', \mu'_1 \triangleq \mu_1[a_1 \mapsto \text{memo} \; a_1 \; (S_{\text{need}}[e_1]_{\rho'_1})] \].

  We have \( a \neq a_1 \) by a property of \( \text{nextFree} \), and applying the induction hypothesis yields

  \[
  \text{Step} \; \text{Let}_1 \; (S_{\text{need}}[e_2]_{\rho'_1}(\mu'_1[a_1 \mapsto d])) = \text{Step} \; e \; v \; (S_{\text{need}}[v]_{\rho_2}(\mu_2))
  \]
as required.

Now we move on to proving auxiliary lemmas about \( \text{freezeHeap} \).

Lemma 52 (Heap extension preserves frozen entries). Let \( \alpha_\Xi \; \mu \leadsto \gamma_\Xi \; \mu = \text{freezeHeap} \; \mu \). If \( \alpha \in dom \; \mu \) and \( a \notin dom \; \mu \), then \( \alpha \; \mu \; d = \alpha \; \mu \; [a \mapsto d] \).

Proof. By Löb induction. Since \( \vdash_{\Xi} \alpha \; \mu \) \( d \), we have \( d = \bigcup \{ \text{step} \; (\text{Lookup} \; y) \; (\text{fetch} \; a_1) \} \) and \( a_1 \in \text{dom} \; \mu \). Let \( \text{memo} \; a_1 \; (S_{\text{need}}[e]_{\rho_1}(\mu_1)) \triangleq \mu \; a_1 \triangleq \mu[a \mapsto d_2] \). Then \( \alpha \in \text{dom} \; \mu \) due to \( \vdash_{\Xi} \alpha \; \mu \) and the goal follows by the induction hypothesis:

\[
\alpha \; \mu \; d = \bigcup \{ \text{step} \; (\text{Lookup} \; y) \; (S[e]_{\alpha \; \mu \; \rho_1}) \}
\]
\[
= \bigcup \{ \text{step} \; (\text{Lookup} \; y) \; (S[e]_{\alpha \; \mu \; [a \mapsto d_2 \; \rho_1]} \}) = \alpha \; \mu \; [a \mapsto d_2] \]

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 \( \text{upd} \) step of the preservation Lemma 19 where we (and Sergey et al. [2017]) resorted to hand-waving. Its proof is surprisingly tricky, but it will pay off; in a moment, we will hand-wave no more!
Lemma 53 (Preservation of heap update). Let \( \widehat{D} \) be a domain with instances for \( \text{Trace}, \text{Domain}, \text{HasBind} \) and \( \text{Lat} \), satisfying the abstraction laws \( \text{BETA-APP}, \text{BETA-SEL}, \text{BIND-BY-NAME} \) and \( \text{STEP-INC} \) from Figure 13. Furthermore, let \( \alpha \in \mu \Rightarrow \forall \mu = \text{freezeHeap} \mu \) for all \( \mu \) and \( \beta \in \mu = \alpha \circ \ldots \) the representation function.

(a) If \( S_{\text{need}}(\varepsilon)_{\rho_1}(\mu_1) = \text{Step ev} (S_{\text{need}}(\varepsilon)_{\rho_2}(\mu_2)) \) and \( \mu_1 ! \ a = \text{memo a} (S_{\text{need}}(\varepsilon)_{\rho_1}) \), then \( S[\varepsilon]_{\beta \in \mu_2}[\alpha \rightarrow \text{memo a} (S_{\text{need}}(\varepsilon)_{\rho_2})] \subseteq S[\varepsilon]_{\beta \in \mu_2 \rho_1} \).

(b) If \( S_{\text{need}}(\varepsilon)_{\rho_1}(\mu_1) = \text{Step ev} (S_{\text{need}}(\varepsilon)_{\rho_2}(\mu_2)) \) and \( \mu_2 \rightsquigarrow \mu_3 \), then \( S[\varepsilon]_{\beta \in \mu_3 \rho_2} \subseteq S[\varepsilon]_{\beta \in \mu_3 \rho_1} \).

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

- 53(a): We assume that \( S_{\text{need}}(\varepsilon)_{\rho_1}(\mu_1) = \text{Step ev} (S_{\text{need}}(\varepsilon)_{\rho_2}(\mu_2)) \) and \( \mu_1 ! \ a = \text{memo a} (S_{\text{need}}(\varepsilon)_{\rho_1}) \) to show \( S[\varepsilon]_{\beta \in \mu_2}[\alpha \rightarrow \text{memo a} (S_{\text{need}}(\varepsilon)_{\rho_2})] \subseteq S[\varepsilon]_{\beta \in \mu_2 \rho_1} \).

We can use the IH 53(a) to prove that \( \beta \in \mu_2[a \mapsto \text{memo a} (S_{\text{need}}(\varepsilon)_{\rho_2})] d \subseteq \beta \in \mu_2 d \) for all \( d \) such that \( \text{adom} d \subseteq \text{adom} \mu_2 \). This is simple to see unless \( d = \text{Step (Lookup y)} (\text{fetch a}) \), in which case we have:

\[
\begin{align*}
\beta \in \mu_2[a \mapsto \text{memo a} (S_{\text{need}}(\varepsilon)_{\rho_2})] (\text{Step (Lookup y)} (\text{fetch a}))
= & \quad \{ \text{Unfold } \beta \in \mu_2 \} \\
= & \quad \{ \text{Step (Lookup y)} (S[\varepsilon]_{\beta \in \mu_2[a \mapsto \text{memo a} (S_{\text{need}}(\varepsilon)_{\rho_2})] \rho_2}) \}
\end{align*}
\]

This is enough to show the goal:

\[
\begin{align*}
S[\varepsilon]_{\beta \in \mu_2[a \mapsto \text{memo a} (S_{\text{need}}(\varepsilon)_{\rho_2})] \rho_2} \\
\subseteq & \quad \{ \text{IH 53(a)} \} \\
S[\varepsilon]_{\beta \in \mu_2 \rho_2} \\
\subseteq & \quad \{ \text{Refold } \beta \in \mu_2 \} \\
S[\varepsilon]_{\beta \in \mu_2 \rho_1}
\end{align*}
\]

- 53(b) \( S_{\text{need}}(\varepsilon)_{\rho_1}(\mu_1) = \text{Step ev} (S_{\text{need}}(\varepsilon)_{\rho_2}(\mu_2)) \wedge \mu_2 \rightsquigarrow \mu_3 \Rightarrow S[\varepsilon]_{\beta \in \mu_3 \rho_2} \subseteq S[\varepsilon]_{\beta \in \mu_3 \rho_1} \).

By Löb induction and cases on \( e \).

- Case \( \text{Var } x \): Let \( a \) be the address such that \( \rho_1 ! x = \text{Step (Lookup y)} (\text{fetch a}) \). Note that \( \mu_1 ! a = \text{memo a } x \), so the result has been memoised in \( \mu_2 \), and by Lemma 50 in \( \mu_3 \) as well. Hence the entry in \( \mu_3 \) must be of the form \( \mu_3 ! a = \text{memo a} (S_{\text{need}}(\varepsilon)_{\rho_2}) \).

\[
\begin{align*}
S[\varepsilon]_{\beta \in \mu_3 \rho_2} \\
\subseteq & \quad \{ \text{Assumption STEP-INC} \} \\
S[\varepsilon]_{\beta \in \mu_3 \rho_1}
\end{align*}
\]

- Case \( \text{Lam x body, ConApp k xs} \): Follows by reflexivity.
- Case \texttt{App e x}: Then $S_{\text{need}}[e]_{\rho_1}(\mu_1) = \text{Step } ev_1 (S_{\text{need}}[\text{Lam } y \text{ body}]_{\rho_1}(\mu_4))$
and $S_{\text{need}}[\text{body}]_{\rho_1[y\rightarrow \mu_1 \! x]}(\mu_4) = \text{Step } ev_2 (S_{\text{need}}[v]_{\rho_2}(\mu_2))$. Note that $\mu_4 \leadsto \mu_2$ by Lemma 49, hence $\mu_4 \leadsto \mu_3$ by $\leadsto$-Trans.

\[
S[v]_{\beta \in \mu_3} \leadsto \mu_2 \\
\mathcal{H} \text{ IH 53.b at } \text{body and } \mu_2 \leadsto \mu_3 \\
S[\text{body}]_{\beta \in \mu_3} \leadsto \mu_1 \! x \\
\mathcal{H} \text{ Assumption STEP-INC } \\
\text{step } \text{App}_2 (S[\text{body}]_{\beta \in \mu_3} \leadsto \mu_1 \! x) \\
\mathcal{H} \text{ Assumption BETTA-APP, refold Lam case } \\
\text{apply } (S[\text{Lam } y \text{ body}]_{\beta \in \mu_3}) (\beta \in \mu_3 (\rho_1 \! x)) \\
\mathcal{H} \text{ IH 53.b at } e \text{ and } \mu_4 \leadsto \mu_3 \\
\text{apply } (S[e]_{\beta \in \mu_3}) (\beta \in \mu_3 (\rho_1 \! x)) \\
\mathcal{H} \text{ Assumption STEP-INC } \\
\text{step } \text{App}_1 (\text{apply } (S[e]_{\beta \in \mu_3}) (\beta \in \mu_3 (\rho_1 \! x))) \\
= \{ \text{Refold } S[\text{App } e x]_{\beta \in \mu_3} \}
\]

- Case \texttt{Case e alts: Similar to App.}
- Case \texttt{Let x e1 e2}: Then $S_{\text{need}}[\text{Let } x \text{ e1 e2}]_{\rho_1}(\mu_1) = \text{Step } \text{Let}_1 (S_{\text{need}}[e2]_{\rho_4}(\mu_4))$, where $a \triangleq \text{nextFree } \mu_1, \rho_4 \triangleq \rho_1 \! x \mapsto \text{Step } (\text{Lookup } x \mapsto \text{Step } (\text{Lookup } x \mapsto \text{fetch } a), \mu_4 \triangleq \mu_1 \! [a \mapsto \text{memo } a (S_{\text{need}}[e1]_{\rho_1})])$. Observe that $\mu_4 \leadsto \mu_2 \leadsto \mu_3$.

The first first half of the proof is simple enough:

\[
S[v]_{\beta \in \mu_3} \leadsto \mu_2 \\
\mathcal{H} \text{ IH 53.b at } e_2 \text{ and } \mu_2 \leadsto \mu_3 \\
S[e_2]_{\beta \in \mu_3} \leadsto \mu_4 \\
\mathcal{H} \text{ Assumption STEP-INC } \\
\text{step } \text{Let}_1 (S[e_2]_{\beta \in \mu_3} \leadsto \mu_4) \\
= \{ \text{Unfold } \mu_4 \}
\]

\[
\text{step } \text{Let}_1 (S[e_2]_{\beta \in \mu_3} \leadsto \mu_4 \! x) \\
\mathcal{H} \text{ Assumption STEP-INC } \\
\text{step } \text{Let}_1 (S[e_2]_{\beta \in \mu_3} \leadsto \mu_4 \! x \mapsto \mu_3 \! x) \\
\mathcal{H} \text{ Unfold } \mu_4 \! a \triangleq \mu_4 \! a \\
\text{step } \text{Let}_1 (S[e_2]_{\beta \in \mu_3} \leadsto \mu_4 \! a) \\
\mathcal{H} \text{ Assumption STEP-INC } \\
\text{step } \text{Let}_1 (S[e_2]_{\beta \in \mu_3} \leadsto \mu_4 \! a \mapsto \mu_3 \! a) \\
\mathcal{H} \text{ Unfold } \mu_4 \! a \\
S[\text{Let } x e1 e2]_{\beta \in \mu_3} \leadsto \mu_4 \! a \\
\]

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 \text{memo } a (S_{\text{need}}[v_1]_{\rho_1})$, where $S_{\text{need}}[e_1]_{\rho_1}(\mu'_4) = \text{Step } ev_1 (S_{\text{need}}[v_1]_{\rho_1}(\mu'_4))$ such that $\mu_4 \leadsto \mu'_4 \leadsto \mu'_3 \! a \mapsto \text{memo } a (S_{\text{need}}[v_1]_{\rho_1}) \leadsto \mu_3$ and $\mu_4 \! a = \mu'_4 \! a = \mu'_3 \! a \neq \mu_3 \! a$. (NB: if there are
multiple such occurrences of \(\rightsquigarrow\)-Memo in \(\mu_4 \rightsquigarrow \mu_3\), this must be the first one, because afterwards it is \(\mu_4 \neq \mu'_4 \neq a\).

It is not useful to apply the IH 53.(a) to this situation directly, because \(\mu'_4 \rightsquigarrow \mu_3\) does not hold. However, note that \(\rightsquigarrow\)-Memo contains proof that evaluation of \(S_{\text{need}}[e_1]_{\rho_1}(\mu'_4)\) succeeded, and it must have done so without looking at \(\mu'_4 \neq a\) (because that would have led to an infinite loop). Furthermore, \(e_1\) cannot be a value; otherwise, \(\mu_4 \neq a = \mu'_3 \neq a\), a contradiction. Since \(e_1\) is not a value and \(\mu'_4 \neq a = \mu'_3 \neq a\), we can apply Lemma 51 to get the useful reduction

\[
S_{\text{need}}[e_1]_{\rho_1}(\mu'_4[a \mapsto \text{memo } a (S_{\text{need}}[v_1]_{\rho_3}))
\]

\[
= \text{Step } ev_1 (S_{\text{need}}[v_1]_{\rho_1}(\mu'_3[a \mapsto \text{memo } a (S_{\text{need}}[v_1]_{\rho_3})))).
\]

This reduction is so useful because we know something about \(\mu'_4[a \mapsto \text{memo } a (S_{\text{need}}[v_1]_{\rho_3})]\); namely that \(\mu'_3[a \mapsto \text{memo } a (S_{\text{need}}[v_1]_{\rho_3})] \rightsquigarrow \mu_3\). This allows us to apply the induction hypothesis 53.(a) to the reduction, which yields

\[
S[v_1]_E \beta \mu_1 \triangleleft \mu_3 \subseteq S[e_1]_E \beta \mu_1 \triangleleft \mu_3.
\]

We this identity below:

\[
\begin{align*}
\text{Let } & \text{ Step } \text{Let }_1 (S[e_2]_E (\mu_3 \triangleleft \mu)(\beta \mu_3 \triangleleft \mu_3)) \subseteq \text{Step } \text{Let }_1 (S[v_1]_E \beta \mu_3 \triangleleft \mu_3) \triangleleft \text{Step } (\text{Lookup } x) (S[v_1]_E (\beta \mu_3 \triangleleft \mu_3)) \\
\text{Let } & \text{ as above } \mu, \mu_3 \triangleleft \mu_3.
\end{align*}
\]

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

**Lemma 54 (Heap progression preserves abstraction).** Let \(\widehat{D}\) 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 \(\alpha_E \mu \equiv \gamma_E \mu = \text{freezeHeap } \mu\) for all \(\mu\).

If \(\mu_1 \rightsquigarrow \mu_2\) and \(\text{adm } d \subseteq \text{dom } \mu_1\), then \(\alpha_E \mu_2 \triangleleft \mu_1\).

**Proof.** By Löb induction. Let us assume that \(\mu_1 \rightsquigarrow \mu_2\) and \(\text{adm } d \subseteq \text{dom } \mu_1\). Since \(\Rightarrow_D d\), we have \(d = \bigcup \{\text{Step } (\text{Lookup } y) (\text{fetch } a)\}\). Similar to Theorem 44, it suffices to show the goal for a single \(d = \text{Step } (\text{Lookup } y) (\text{fetch } a)\) for some \(y, a\) and the representation function \(\beta_E \mu \triangleleft \alpha_E \mu < \{\}\).

Furthermore, let us abbreviate \(\text{memo } a (S_{\text{need}}[e_1]_{\rho_1}) \neq \mu_1 \neq a\). The goal is to show

\[
\text{Step } (\text{Lookup } y) (S[e_2]_E \beta \mu_1 \triangleleft \mu_2) \subseteq \text{Step } (\text{Lookup } y) (S[e_1]_E \beta \mu_1 \triangleleft \mu_2).
\]

Monotonicity allows us to drop the step (Lookup x) context

\[
(\beta \mu_2 \triangleleft \mu_2 \subseteq S[e_1]_E \beta \mu_1 \triangleleft \mu_1).
\]

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

- **Case** \(\rightsquigarrow\)-RefL: Then \(\mu_1 = \mu_2\) and hence \(\alpha_E \mu_1 = \alpha_E \mu_2\).
- **Case** \(\rightsquigarrow\)-Trans: Apply the induction hypothesis to the sub-derivations and apply transitivity of \(\subseteq\).
Step \( \beta_\text{app} \)

**Abstracting Denotational Interpreters**

3378

3375

3373

3369

3368

3362

3360

3350

3349

3343

3335

By Löb induction and cases on

We can finally show the goal

\( \mu \)

\( \Rightarrow \)

\( \beta_\text{app} \mu_2 \subseteq \beta_\text{app} \mu_1 \), and the goal follows by monotonicity.

**Case \( \rightsquigarrow \text{-MEMO} \)**

\( \mu_1 \overset{1}{\Rightarrow} \mu_3 \left( a_1 \mapsto \text{memo} a_1 \left( \text{S}_\text{need} [\text{eval}] \beta \right) \right) \)

We get to refine \( \mu_2 = \mu_1 \left( a_1 \mapsto \text{memo} a_1 \left( \text{S}_\text{need} \left[ \text{eval} \right] \beta \right) \right) \). Since \( a \in \text{dom} \, \mu_1 \), we have \( a_1 \neq a \) and thus \( \mu_1 \left( a \right) = \mu_2 \left( a \right) \), thus \( e_1 = e_2, \rho_1 = \rho_2 \). The goal can be simplified to \( \left( \text{S}_1 \right) \beta \mu_2 \rho_1 \subseteq \text{S}_1 \rho_1 \). We can apply the induction hypothesis to get \( \left( \beta_\text{app} \mu_2 \subseteq \beta_\text{app} \mu_1 \right) \), and the goal follows by monotonicity.

**Case \( \rightsquigarrow \text{-EXT} \)**

\[ a_1 \not\in \text{dom} \, \mu_1 \quad \text{dom} \, \rho \subseteq \text{dom} \, \mu_1 \cup \left\{ a_1 \right\} \]

\[ \mu_1 \overset{1}{\Rightarrow} \mu_1 \left( a_1 \mapsto \text{memo} a_1 \left( \text{S}_\text{need} \left[ \text{eval} \right] \beta \right) \right) \]

We get to refine \( \mu_2 = \mu_1 \left( a_1 \mapsto \text{memo} a_1 \left( \text{S}_\text{need} \left[ \text{eval} \right] \beta \right) \right) \). When \( a_1 \neq a \), we have \( \mu_1 \left( a \right) = \mu_2 \left( a \right) \) and the goal follows as in the \( \rightsquigarrow \text{-EXT} \) case. Otherwise, \( a = a_1, e_1 = e, \rho_3 = \rho_1, e_2 = \nu \). We can use Lemma 53.(a) to prove that \( \beta_\text{app} \mu_2 \rho_3 \subseteq \beta_\text{app} \mu_2 \rho_3 d \) for all \( d \) such that \( \text{dom} \, d \subseteq \text{dom} \, \mu_2 \).

This is simple to see unless \( d = \text{Step} \left( \text{Lookup} \, e \right) \left( \text{fetch} \, a \right) \), in which case we have:

\[ \beta_\text{app} \mu_2 \left( \text{Step} \left( \text{Lookup} \, y \right) \left( \text{fetch} \, a \right) \right) \]

\[ \beta_\text{app} \mu_2 \left( \text{Refold} \left( \text{Lookup} \, y \right) \left( \text{fetch} \, a \right) \right) \]

We can finally show the goal \( \beta_\text{app} \mu_2 \rho_3 \subseteq \beta_\text{app} \mu_2 \rho_3 d \) for all \( d \) such that \( \text{dom} \, d \subseteq \text{dom} \, \mu_1 \):

\[ \beta_\text{app} \mu_2 d \]

\[ \beta_\text{app} \mu_2 \rho_3 \subseteq \beta_\text{app} \mu_3 \]

\[ \beta_\text{app} \mu_3 d \]

\[ \beta_\text{app} \mu_1 \quad \text{Löb induction hypothesis at} \quad \beta_\text{app} \mu_1 \quad \text{by} \quad \text{Lemma 49} \]

\[ \beta_\text{app} \mu_1 \]

\[ \square \]

**Lemma 55** (By-need evaluation preserves by-name trace abstraction). Let \( \beta \) be a domain with instances for Trace, Domain, HasBind and Lat, satisfying the abstraction laws Step-App, Step-Sel, Beta-App, Beta-Sel, Bind-By-Name, Step-Inc and Update in Figure 13. Furthermore, let \( \alpha_\text{app} \mu = \gamma_\text{app} \mu = \text{freezHeap} \mu \) for all \( \mu \).

If \( \text{S}_\text{need} \left[ \text{eval} \right] \beta \left( \mu_1 \right) = \text{Step} \left( \text{Lookup} \, e \right) \left( \text{memo} \, a \left( \text{S}_\text{need} \left[ \text{eval} \right] \beta \left( \mu_1 \right) \right) \right) \) and \( \text{Step} \left( \text{Lookup} \, e \right) \left( \text{fetch} \, a \right), \mu_1 \left( a \right) = \text{memo} \, a \left( \text{S}_\text{need} \left[ \text{eval} \right] \beta \left( \mu_1 \right) \right) \) and \( \text{add} = \left[ \text{Lookup} \, e \right] + \text{add} + \left[ \text{Update} \right] \) for some ev1 by determinism.

The step below that uses Item 53.(b) does so at \( e_1 \) and \( \mu_2 \overset{1}{\Rightarrow} \mu_2 \) to get \( \text{S}_1 \beta \mu_2 \rho_1 \subseteq \text{S}_1 \left( e_1 \right) \beta \mu_2 \rho_1 \), in order to prove that \( \beta_\text{app} \mu_2 \rho_1 \subseteq \beta_\text{app} \mu_2 \left( a \mapsto \text{memo} \, a \left( \text{S}_\text{need} \left[ \text{eval} \right] \beta \left( \mu_1 \right) \right) \right) \left( e_1 \right) \beta \mu_2 \rho_1 \rho_2 \).

\[ \text{Step} \left( \text{Lookup} \, e \right) \left( \text{Step} \left( \text{Lookup} \, e \right) \left( \text{Update} \left( \text{Step} \left( \text{Lookup} \, e \right) \right) \right) \right) \]

\[ \text{Assumption Update} \]

\[ \text{Step (Lookup } y) \ (\text{Step } ev_1 (S[v]_{\beta_E \mu_1 \rho_1})) \]
\[ \subseteq (\text{Lemma 55}) \]
\[ \text{Step (Lookup } y) \ (\text{Step } ev_1 (S[v]_{\beta_E \mu_2 \rho_2})) \]
\[ \subseteq (\text{Memoization}) \]
\[ \text{Step (Lookup } y) \ (S[e_1]_{\beta_E \mu_1 \rho_1}) \]
\[ \Rightarrow \]
\[ \text{Refold } \beta_E, \rho_3 \mapsto x \]
\[ \beta_E(\rho_1 ! x) \]
\[ \Rightarrow \]
\[ \text{Refold } S[x]_{\beta_E \mu_1 \rho_1} \]
\[ S[x]_{\beta_E \mu_1 \rho_1} \]

**Case** Let \( e_1, e_2 \): We can make one step to see
\[ S_{\text{need}}[\text{Let } x e_1 e_2]_{\rho_1}(\mu_1) = \text{Step Let}_1 (S_{\text{need}}[e_2]_{\rho_1}(\mu_3)) = \text{Step Let}_1 (\text{Step } ev_1 (S_{\text{need}}[v]_{\rho_2}(\mu_2))), \]
where \( \rho_3 \triangleq \rho_1 [x \mapsto \text{Step (Lookup } x) (\text{fetch } a)], a \triangleq \text{nextFree } \mu_1, \mu_3 \triangleq \mu_1 [a \mapsto \text{memo } a (S_{\text{need}}[e_1]_{\rho_1})]. \]
Then \( (\beta_E \mu_3 \triangleq \rho_3) \): \( y = (\beta_E \mu_1 \triangleq \rho_1) \! \) whenever \( x \neq y \) by \( \text{Lemma 52} \), and \( (\beta_E \mu_3 \triangleq \rho_3) ! x = \text{Step (Lookup } x) (S[e_1]_{\beta_E \mu_1 \rho_1}). \]
We prove the goal, thus
\[ \text{Step } ev_1 (S[v]_{\beta_E \mu_2 \rho_2}) \]
\[ \Rightarrow \]
\[ \text{Step Let}_1 (S[e_2]_{\beta_E \mu_1 \rho_1}) \]
\[ \Rightarrow \]
\[ \text{Rearrange } \beta_E \mu_3 \text{ by above reasoning} \]
\[ \text{Step Let}_1 (S[e_2]_{\beta_E \mu_1 \rho_1}) \]
\[ \Rightarrow \]
\[ \text{Expose fixpoint, rewriting } \beta_E \mu_3 \triangleq \rho_3 \text{ to } (\beta_E \mu_1 \triangleq \rho_1) [x \mapsto \beta_E \mu_3 (\rho_3 ! x)] \]
\[ \text{Step Let}_1 (S[e_2]_{\beta_E \mu_1 \rho_1}) \]
\[ \Rightarrow \]
\[ \text{Partially unroll lfp} \]
\[ \text{Step Let}_1 (S[e_2]_{\beta_E \mu_1 \rho_1}) \]
\[ \Rightarrow \]
\[ \text{Assumption Bind-ByNAME} \]
\[ \text{bind } (\lambda \tilde{a}_1 \rightarrow S[e_1]|(\beta_E \mu_1 \triangleq \rho_1)) \rightarrow \text{Step (Lookup } x) \tilde{a}_1 \]
\[ (\lambda \tilde{a}_1 \rightarrow \text{Step Let}_1) (S[e_2]|(\beta_E \mu_1 \triangleq \rho_1)) \rightarrow \text{Step (Lookup } x) \tilde{a}_1 \]
\[ \Rightarrow \]
\[ \text{Refold } S[\text{Let } x e_1 e_2]_{\beta_E \mu_1 \rho_1} \]
\[ S[\text{Let } x e_1 e_2]_{\beta_E \mu_1 \rho_1} \]

**Case** Lam, ConApp: By reflexivity.

**Case** App \( e \ x \): Very similar to \( \text{Lemma 43} \), since the heap is never updated or extended.

There is one exception: We must apply \( \text{Lemma 54} \) to argument denotations.

We have \( S_{\text{need}}[e]_{\rho_1}(\mu_1) = \text{Step } ev_1 (S_{\text{need}}[\text{Lam } y \text{ body}]_{\rho_1}(\mu_3)) \) and \( S_{\text{need}}[\text{body}]_{\rho_1[y \mapsto \rho_1 ! x]}(\mu_3) = \text{Step } ev_2 (S_{\text{need}}[v]_{\rho_2}(\mu_2)). \) We have \( \mu_1 \rightsquigarrow \mu_3 \) by \( \text{Lemma 49} \).
\[ \text{Step App}_1 (\text{Step } ev_1 (\text{Step App}_2 (\text{Step } ev_2 (S[v]_{\beta_E \mu_2 \rho_2})))) \]
\[ \Rightarrow \]
\[ \text{Induction hypothesis at } ev_2 \]
\[ \text{Step App}_1 (\text{Step } ev_1 (\text{Step App}_2 (S[\text{body}]_{\beta_E \mu_3 \rho_3[y \mapsto \rho_1 ! x]}))) \]
\[ \Rightarrow \]
\[ \text{Assumption Beta-App, refold } \text{Lam case} \]
Theorem 56 (Sound By-name Interpretation). Let \( \mathcal{D} \) be a domain with instances for \( \text{Trace}, \text{Domain}, \text{HasBind}, \text{and Lat} \), and let \( \alpha_T \models \gamma_T = \text{nameNeed} \), as well as \( \alpha_E \mu \models \gamma_E \mu = \text{freezeHeap} \mu \) from Figure 18. If the abstraction laws in Figure 13 hold, then \( S[\_] \) instantiates at \( \mathcal{D} \) to an abstract interpreter that is sound wrt. \( \gamma_E \to \alpha_T \), that is, 
\[
\forall \rho. \beta_T (S_{\text{need}}[e]\rho(\mu)) \models (S_{\mathcal{D}}[e]_{\alpha_E \mu(\_)}). \]

Proof. As in Theorem 44, we simplify our proof obligation to the single-trace case: 
\[
\forall \rho. \beta_T (S_{\text{need}}[e]\rho(\mu)) \models (S_{\mathcal{D}}[e]_{\beta_E \mu(\_)}), \]
where \( \beta_T \triangleq \alpha_T \circ \{ \} \) and \( \beta_E \mu \triangleq \alpha_E \mu \circ \{ \} \) are the representation functions corresponding to \( \alpha_T \) and \( \alpha_E \). We proceed by Löb induction.

Whenever \( S_{\text{need}}[e]\rho(\mu) = \text{Step ev} (S_{\text{need}}[v]\rho_2(\mu_2)) \) yields a balanced trace and makes at least one step, we can reuse the proof for Lemma 55 as follows:
\[
\beta_T (S_{\text{need}}[e]\rho(\mu)) = \text{Step ev} (S_{\text{need}}[v]\rho_2(\mu_2)), \text{unfold } \beta_T \]
\[
\text{Step ev} (\beta_T (S_{\text{need}}[v]\rho_2(\mu_2))) \]
\[
\subseteq \text{Induction hypothesis (needs non-empty } \overline{ev}) \]

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

\[
\text{Theorem 56 (Sound By-name Interpretation). Let } \mathcal{D} \text{ be a domain with instances for } \text{Trace}, \text{Domain}, \text{HasBind} \text{ and } \text{Lat}, \text{ and let } \alpha_T (\models \gamma_T = \text{nameNeed}), \text{ as well as } \alpha_E \mu (\models \gamma_E \mu = \text{freezeHeap} \mu) \text{ from Figure 18. If the abstraction laws in Figure 13 hold, then } S[\_] \text{ instantiates at } \mathcal{D} \text{ to an abstract interpreter that is sound wrt. } \gamma_E \to \alpha_T, \text{ that is, }
\]
\[
\forall \rho, \beta_T (S_{\text{need}}[e]\rho(\mu)) \models (S_{\mathcal{D}}[e]_{\beta_E \mu(\_)}),
\]

where \( \beta_T \triangleq \alpha_T \circ \{ \} \) and \( \beta_E \mu \triangleq \alpha_E \mu \circ \{ \} \) are the representation functions corresponding to \( \alpha_T \) and \( \alpha_E \). We proceed by Löb induction.

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

\[
\beta_T (S_{\text{need}}[e]\rho(\mu)) = \text{Step ev} (S_{\text{need}}[v]\rho_2(\mu_2)), \text{ unfold } \beta_T \]
\[
\text{Step ev} (\beta_T (S_{\text{need}}[v]\rho_2(\mu_2)))
\]
\[
\subseteq \text{Induction hypothesis (needs non-empty } \overline{ev}) \]
\[
\begin{align*}
\text{step } e & \in (S[e]_{\beta_E \mu \triangleleft \rho}) \\
\subseteq & \text{ Lemma 55} \\
S[e] & \in (S[e]_{\beta_E \mu \triangleleft \rho})
\end{align*}
\]

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

- **Case Var** \( x \): The stuck case follows by unfolding \( \beta_T \).

\[
\begin{align*}
\beta_T ((\rho ! x) \mu) & = \{ \text{Unfold } \beta_T \} \\
\]\[
\begin{align*}
\text{step } \text{(Lookup } y & \text{)} (\beta_T (\text{fetch } a \mu)) \\
& = \{ \text{Unfold } \beta_T \} \\
\text{step } \text{(Lookup } y & \text{)} (\beta_T (\text{memo } a (S_{\text{need}}[e_1]_{\rho_1}(\mu))))
\end{align*}
\]

By assumption, \( \text{memo } a (S_{\text{need}}[e_1]_{\rho_1}(\mu)) \) diverges or gets stuck and the result is equivalent to \( S_{\text{need}}[e_1]_{\rho_1}(\mu) \).

\[
\begin{align*}
\beta_T (\beta_T (S_{\text{need}}[e_1]_{\rho_1}(\mu))) & = \{ \text{Induction hypothesis} \} \\
\beta_T (\beta_T (\text{lookups } y \mu)) & = \{ \text{Refold } \beta_T \}
\end{align*}
\]

- **Case Lam** \( \lambda x \): body:

\[
\begin{align*}
\beta_T (S_{\text{need}}[\lambda x \text{ body}]_{\rho}(\mu)) & = \{ \text{Unfold } S_{\text{need}}[\lambda x \text{ body}]_{\rho}(\mu) \} \\
\text{fun } (\lambda \bar{d} \rightarrow \| \{ \text{step } \text{App}_2 \} (\beta_T (S_{\text{need}}[\text{body}]_{\rho}(\mu) | \beta_E \mu d \equiv \bar{d}) | \beta_E \mu d \equiv \bar{d}) | \beta_E \mu d \equiv \bar{d}) & = \{ \text{Induction hypothesis} \} \\
\text{fun } (\lambda \bar{d} \rightarrow \| \{ \text{step } \text{App}_2 \} (S[\text{body}]_{\beta_E \mu \triangleleft \rho_1}(\mu) | \beta_E \mu d \equiv \bar{d}) | \beta_E \mu d \equiv \bar{d}) & = \{ \text{Refold } S[\lambda x \text{ body}]_{\beta_E \mu \triangleleft \rho} \}
\end{align*}
\]

- **Case ConApp** \( k \) xs:

\[
\begin{align*}
\beta_T (S_{\text{need}}[\text{ConApp } k]_{\rho}(\mu)) & = \{ \text{Unfold } S_{\text{need}}[\text{ConApp } k]_{\rho}(\mu) \} \\
\text{con } k \text{ (map } ((\beta_E \mu \triangleleft \rho) \mapsto \text{xs}) & = \{ \text{Refold } S[\text{ConApp } k]_{\beta_E \mu \triangleleft \rho} \}
\end{align*}
\]

- **Case App** \( e \), **Case e alts**: The same steps as in Theorem 44.

- **Case Let** \( e_1 \) \( e_2 \): We can make one step to see

\[
S_{\text{need}}[\text{Let } e_1 \ e_2]_{\rho}(\mu) = \text{Step } \text{Let}_1 ((S_{\text{need}}[e_2]_{\rho_1}(\mu_1)), \]

where \( \rho_1 \triangleq \rho[x \rightarrow \text{step } \text{(Lookup } x) (\text{fetch } a)] \), \( a \triangleq \text{nextFree } \mu, \mu_1 \triangleq \mu[a \mapsto \text{memo } a (S_{\text{need}}[e_1]_{\rho_1})] \).

Then \( (\beta_E \mu_1 \triangleleft \rho_1) ! y = (\beta_E \mu_2 \triangleleft \rho_2) ! y \) whenever \( x \neq y \) by Lemma 52, and \( (\beta_E \mu_1 \triangleleft \rho_1) ! x = \text{step } \text{(Lookup } x) (S[e_1]_{\beta_E \mu_1 \triangleleft \rho_1}). \)
We can apply this by-need abstraction theorem to usage analysis on open expressions, just as before:

**Lemma 57** ($S_{\text{usage}}[\_](\_)$ abstracts $S_{\text{need}}[\_](\_)$, open). Usage analysis $S_{\text{usage}}[\_](\_)$ is sound wrt. $S_{\text{need}}[\_](\_)$, that is, $\alpha_{\Rightarrow} (S_{\text{need}}[e]_{\rho}(\mu)) \subseteq S_{\text{usage}}[e]_{\alpha_{\Rightarrow}(\_)} \Rightarrow_{\rho}$ where $\alpha_{\Rightarrow} \Rightarrow_{=} = \text{nameNeed}$; $\alpha_{\Rightarrow} \Rightarrow_{=} = \text{freezeHeap} \mu$

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