Functional Abstract Interpretation

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte
Dissertation

von

Sebastian Graf

aus Osnabrück

Tag der mündlichen Prüfung:	12.12.2024
Erster Gutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweiter Gutachter:	Jun.-Prof. Dr. Jonathan I. Brachthäuser

Contents

Abstract

Functional programming languages encourage expressing large parts of a program as declarative data flow pipelines, free of side-effects such as shared mutable state. Such pipelines traverse recursive data by *pattern matching*, and share the repetitive code of these traversals by defining *higher-order functions*. Writing programs in functional style eliminates large classes of programmer errors, hence higher-order functions and pattern matching have been adopted by most general purpose programming languages today.

However, pattern matching introduces new modes of failure as well: It is easy to forget a case, and input data that is not covered leads to a crash at runtime. Thus, a compiler should integrate a *static program analysis* to warn about such uncovered pattern-matches before the program is run.

A compiler should also generate fast code for programs involving higher-order functions. More than 30 years of practical research have evolved the Glasgow Haskell Compiler (GHC) into an industrial strength tool. This evolution brought forth a number of useful and efficient compiler optimisations that are informed by static higher-order analyses. However, the more proficient a higher-order analysis becomes, the harder it gets to *explain* its implementation to maintainers, let alone convince interested bystanders of its *correctness*.

In this thesis, I present two results of my work to improve GHC: the first is a static analysis for *pattern-match coverage checking* that is both more efficient and more precise than the state of the art; the second is a design pattern for deriving *static higher-order analyses* and dynamic semantics alike from a generic *denotational interpreter*, in order to share intuition and correctness proofs. This design pattern generalises Cousot's seminal work on trace-based abstract interpretation to higher-order analyses such as GHC's Demand Analysis.

My novel approach to pattern-match coverage checking is to boil down complex pattern-matches into so-called *guard trees*. Advanced functional languages such as Haskell have many forms of patterns, some of which are influenced by previous pattern-matches through type equality constraints and lazy evaluation. Yet, guard trees comprise of only three different constructs, which vastly simplifies the coverage checking process compared to checking full Haskell.

The resulting algorithm is modular, allowing for new forms of source-language patterns to be handled with little changes to the overall structure of the algorithm, as evidenced by extending GHC with support for Or-patterns years after my work was released. It is reassuring that my new coverage checker is not only easier to maintain, it also performs better than GHC's previous coverage checker, both in terms of accuracy and runtime performance.

My second innovation is a framework in which one can explain higher-order analyses — such as those found in GHC — by formal analogy to a familiar standard semantics, via *abstract interpretation*. GHC's analyses approximate the effect of function calls by means of *function summaries*. A particularly challenging aspect of this work is to find a standard semantics that is structurally compatible with such summary-based analyses, so that it becomes feasible to draw a formal analogy between analysis and semantics in the first place.

Fortunately, my novel *denotational interpreters* are up to the challenge. Denotational interpreters are denotational semantics that produce coinductive traces of a corresponding small-step operational semantics. By parameterising a denotational interpreter over the semantic domain and then varying it, I recover dynamic semantics with different evaluation strategies as well as static analyses such as type analysis, all from the same generic interpreter.

One instance of my interpreter is a denotational semantics for lazy evaluation, the first one that is provably adequate in a strong, compositional sense. It is mathematically challenging to define such a semantics because it combines a mutable heap with higher-order functions that modify said heap. I build my semantic domain on recent advances in guarded dependent type theory; the approach should readily transfer to general purpose programming languages, including those with arbitrary side-effects.

The generated program traces lend themselves well to describe *operational properties* such as how often a variable is evaluated, and hence enable static analyses abstracting these operational properties.

In order to show that this framework is useful, I will explain and prove correct *usage analysis*, a simple form of GHC's Demand Analyser. Since static analysis and dynamic semantics share the same generic interpreter definition, the correctness proof decomposes into showing small abstraction laws about the abstract domain, thus obviating complicated ad-hoc preservation-style proof frameworks.

Thus, my work combines intraprocedural abstract interpretation of traces with higher-order function calls, enabling *functional abstract interpretation*.

Zusammenfassung

Funktionale Programmiersprachen begünstigen die Formulierung von Programmen als deklarative Datenfluss-Pipelines, die frei von Seiteneffekten wie Schreibzugriffen auf globalen Zustand sind. Solche Programme operieren auf rekursiv definierten Daten via *Pattern-Matching* und führen redundante Logik mithilfe von *Funktionen höherer Ordnung* zusammen. Da Programme im funktionalen Stil viele Programmierfehler verhindern, haben sich die meisten Allzweck-Programmiersprachen heutzutage beide Sprachfeatures einverleibt.

Pattern-Matching birgt jedoch eigenes Fehlerpotential: Es ist einfach, einen Fall unbehandelt zu lassen, der bei einer ungünstigen Eingabe zur Laufzeit zu einem Absturz führt. Daher sollte ein Compiler solche unbehandelten Fälle mithilfe einer *statischen Programmanalyse* erkennen.

Zudem sollte ein Compiler schnellen Code für Programme mit Funktionen höherer Ordnung generieren. Mehr als 30 Jahre praktischer Forschung haben den Glasgow Haskell Compiler (GHC) zu industrieller Reife geführt. Diese Evolution hat einige nützliche Compiler-Optimierungen zutage gebracht, denen statische Programmanalysen höherer Ordnung zugrunde liegen. Je raffinierter solche Analysen werden, desto schwieriger wird es, sie den pflegenden Entwicklern zu *erklären*, geschweige denn Forscher von ihrer *Korrektheit* zu überzeugen.

Diese Arbeit präsentiert zwei Ergebnisse meiner Arbeit am GHC: Das erste ist eine statische Analyse für *Pattern-Match Coverage Checking*, die sowohl effizienter als auch präziser als der Stand der Forschung ist; das zweite ist ein Entwurfsmuster zur Ableitung *statischer Analysen höherer Ordnung* und dynamischer Semantiken vom gleichen *denotationalen Interpreter*, um so Erklärungen and Korrektheitsbeweise zu teilen. So generalisiere ich Cousots Arbeit zu Trace-basierter abstrakter Interpretation auf Analysen höherer Ordnung wie im GHC.

Mein Ansatz zum Pattern-Match Coverage Checking reduziert komplexe Pattern-Matches auf sogenannte *Guard Trees*. Fortgeschrittene funktionale Sprachen wie Haskell haben viele verschiedene Formen von Patterns, und manche davon beeinflussen durch Typgleichheiten und faule Auswertung nachfolgende Pattern-Matches. Trotz dieser reichhaltigen Semantik kommen Guard Trees mit nur drei Konstrukten aus, was den Prüfungsprozess stark vereinfacht.

Der resultierende Algorithmus ist modular, so dass die Implementierung kaum angepasst werden muss wenn der Quellsprache neue Formen von Patterns wie z.B. Or-Patterns hinzugefügt werden, wie Jahre nach der Veröffentlichung meiner Arbeit am GHC geschehen. Der Checker ist nicht nur einfacher zu pflegen, er ist auch effizienter und präziser als das Vorgängerverfahren im GHC.

Meine zweite Neuerung ist ein Framework, das es ermöglicht, die Analysen höherer Ordnung im GHC formal in Relation zu einer bekannten Standardsemantik zu setzen, via *abstrakter Interpretation*. Die Analysen im GHC approximieren den Effekt von Funktionsaufrufen durch *Funktions-Summaries*. Daher besteht eine Herausforderung meiner Arbeit darin, eine Standardsemantik zu finden, die strukturell ähnlich zu solchen Summary-basierten Analysen ist, um einen Vergleich zwischen Analyse und Semantik überhaupt erst zu ermöglichen.

Erfreulicherweise sind meine neuen *denotationalen Interpreter* dieser Herausforderung gewachsen. Denotationale Interpreter sind denotationale Semantiken, die koinduktive Traces einer korrespondierenden small-step operationalen Semantik generieren. Indem ich die semantische Domäne des denotationalen Interpreters variiere, erhalte ich dynamische Semantiken verschiedener Auswertungsreihenfolgen als auch statische Analysen wie Typanalyse als Instanzen des gleichen generischen Interpreters.

Eine dieser Instanzen ist eine denotationale Semantik für faule Auswertung, die erste die beweisbar adäquat zur small-step Semantik ist. Es ist mathematisch anspruchsvoll eine solche Semantik zu definieren, denn sie kombiniert globalen Zustand mit Funktionen höherer Ordnung die diesen Zustand ändern können. Meine semantische Dömane bedient sich daher jüngsten Fortschritten in Guarded Dependent Type Theory, die auf Allzweck-Programmiersprachen mit beliebigen Seiteneffekten übertragbar sein sollten.

Die generierten Programm-Traces eignen sich gut zur Formulierung *operationaler Eigenschaften*, die z.B. die Anzahl an Variablenauswertungen beschreiben, und ermöglichen statische Analysen die diese Eigenschaften approximieren.

Um die Nützlichkeit meines Frameworks zu demonstrieren, werde ich *Usage Analyse*, eine Vereinfachung der Demand Analyse im GHC, erklären und korrekt beweisen. Da statische Analyse und dynamische Semantik vom gleichen generischen Interpreter abgeleitet sind, zerfällt der Korrektheitsbeweis in mehrere kleine Abstraktionsregeln über die abstrakte Domäne, die einen komplizierteren monolithischen Beweis ersetzen.

Meine Arbeit vereint intraprozedurale abstrakte Interpretation von Traces mit Funktionen höherer Ordnung: *Funktionale Abstrakte Interpration*.

Acknowledgements

Over the past six years, I had the privilege of meeting many new people who shaped me as a person and as a researcher. First and foremost, I would like to thank my advisor, Prof. Gregor Snelting, for enabling my growth over these six years in the first place, as well as for the great freedom I enjoyed while pursuing my own research agenda. May we again find the time to play organ and trumpet together.

I would also like to thank Jun.-Prof. Jonathan Brachthäuser for a great research visit in Tübingen worth repeating and the second review of this thesis. May we again find the time to play piano and trumpet together.

Of all my collaborators, I cannot stress enough the impact that my mentor Simon Peyton Jones has had on me as a researcher, technical writer, software engineer and technical leader, instilling in me the urge to find simple solutions. Time spent conversing with and learning from an ACM Fellow (to name just one of Simon's qualifiers) is always a privilege, and Simon gifted me his time aplenty, both in weekly calls and in a three-month long internship at Microsoft Research back in 2019. Without his work on GHC and Haskell, I would never have found my way to Karlsruhe in the first place, let alone attempted a PhD in programming languages. No amount of praise could meet the amount of deep gratitude I feel for Simon taking me under his wing. May we continue to find the time to improve GHC together, as friends.

Even with GHC existing back in 2014, I would never have found my way to Karlsruhe were it not for Joachim Breitner, whose work on Haskell and GHC I admired. Since then, I was fortunate enough to first experience Joachim as a teacher, then to win him as mentor for my master's thesis in 2016, introducing me to Simon at ICFP 2017, and finally to find in him a friend ever since.

I will fondly remember the productive days and fun nights spent with the members of the programming paradigms group: Johannes Bechberger, Simon Bischof, Sebastian Buchwald, Andreas Fried, Martin Hecker, Denis Lohner, Manuel Mohr, Martin Mohr, Jakob von Raumer, Brigitte Sehan-Hill, Sebastian Ullrich, Max Wagner, and Andreas Zwinkau. The reaction to my overwhelmingly half-baked thoughts was always an understanding one, and led to particularly productive

discussions with Andreas, Jakob and Sebastian Ullrich. As a LaTeX practitioner more than an enthusiast, I am grateful for the template that Martin Mohr, Joachim Breitner and ultimately Andreas Lochbihler lended me for this thesis.

My gratitude goes to Joachim Breitner, Henning Dieterichs, Lennart Graf, Martin Hecker, András Kovács and Sebastian Ullrich for proof-reading parts of the papers that constitute this thesis, and for Henning Dieterichs especially for his formalisation of the coverage checking algorithm in Chapter 3.

Five years ago I met my partner Helene, who has since shaped me as a person unlike any other. I am not sure if I could have continued my PhD through the more devastating phases without her loving support. May she let me return that support for many years to come. Finally, my whole life has been one of great privilege, thanks to my ever-loving family. I owe Birgit, Jürgen, Lennart and Henrike my deepest and most earnest gratitude for teaching me to become a decent, reliable person, equipped with the perseverance to conclude this PhD.

1

Introduction

Functional programming is a declarative programming paradigm that by now has permeated most general-purpose programming languages. Formulating a program component in functional style eschews imperative control flow and side-effects (such as global mutable state) in favour of declarative data flow pipelines built out of higher-order functions.

Functional solutions correlate with fewer bugs [Ray et al. 2017] and are often more concise than imperative ones [Nanz and Furia 2015], thanks to superior mechanisms for modular abstraction and composition. Fewer lines of code and good abstractions lead to easier understanding and maintenance.

Yet, *fewer* bugs does not mean *no* bugs. Code review is an effective way to prevent bugs from entering production systems, but thorough code review by a human is expensive. Thus, it is common engineering practice to employ automated linting tools — fueled by *static program analyses* — as a first line of defense. Static program analyses are not only useful to catch bugs, they also help to compile high-level data flow pipelines into efficient machine code by informing a compiler's *optimisation passes* for when said optimisations are safely applicable.

Static analysis of functional programs promises great leverage: The absence of side effects in functional languages such as Haskell [Marlow et al. 2010] means that inferred analysis results need to be invalidated far less often, guaranteeing precise results even across function calls. However, static analyses of functional programs are also extra challenging to conceive and formalise because reasoning about higher-order functions is far more difficult than first-order reasoning in imperative, intraprocedural program analyses.

This explains why industrial-strength functional language compilers, such as the Glasgow Haskell Compiler (GHC), implement several practically important static analyses, few of which have been scientifically published about to date.

Seven years ago, I was enticed by the prospect of improving and publishing about the static analyses implemented in GHC. Since then, I have made various contributions to GHC, and in this thesis I describe two results of that work.

In Chapter 3 I present an analysis for pattern-match coverage checking that is part of GHC's linting framework. The analysis detects when the programmer forgets to handle a case in a pattern-match, upon which GHC emits a warning. My analysis has been part of GHC since 2020 and improves upon its predecessor in soundness, precision, efficiency and maintainability.

The coverage checker is a nice engineering feat, exactly what I had planned for my PhD. According to this plan, this thesis would have presented two more analyses of GHC that I have worked on intensively. One of these would have been GHC's *Demand Analysis*, building on the results of my master's thesis [Graf 2017]. Alas, fate would have it otherwise. Any formal description of Demand Analysis I attempted led to pages and pages worth of complicated figures. While the improved analysis I conceived made intuitive sense to me, conveying the idea to my co-maintainer and mentor Simon Peyton Jones proved utterly impossible. If I could not explain my improvements to one of the few experts that are intimately familiar with Demand Analysis, how could I ever make my work accessible to the broader research community?

Chapter 4 is to be understood in light of this frustrating realisation. I demonstrate that Demand Analysis as well as other higher-order analyses within GHC fit into a compositional analysis framework called a *denotational interpreter*. By providing a generic implementation of a denotational interpreter that parameterises over the semantic domain, I recover both dynamic semantics as well as static analyses as instances. Thus, in order to describe a static analysis such as Demand Analysis, it suffices to describe its semantic domain, rather than a complete analysis framework, meaning that my work vastly simplifies a formal description of such analyses.

Denotational interpreters are amenable to formal reasoning. For example, one instance is the first denotational semantics for call-by-need that is proven adequate wrt. a standard small-step operational semantics. Furthermore, I formulate and prove conditions for when a static analysis conservatively approximates a trace property of the dynamic semantics, applying the well-known theory of *abstract interpretation* [Cousot and Cousot 1977] to higher-order functional language semantics, hence the title of this thesis.

Undoubtedly, denotational interpreters would have been a sublime framework to base my PhD upon. I hope it will prove useful to some colleagues in the research community.

1.1 Structure

I structured the thesis as follows. Chapter 2 briefly introduces the more unusual language features of Haskell that will become relevant in the chapters that follow. Furthermore, I will recall some fundamentals about order theory and abstract interpretation. Chapter 3 presents the coverage checking algorithm that I devised for GHC and Chapter 4 contains my work on denotational interpreters. These two chapters constitute the main content of this thesis. Both chapters can be read independently and start with dedicated introductions, ending in a list of contributions and a paragraph of acknowledgements that credit the work done by my collaborators. Chapter 4 refers to a number of Appendices: Appendix A contains any pen-and-paper proofs that were omitted in the main body. Whenever a proof for a proposition is omitted from the main body, I inserted a forward reference in the margin such as for this line that lists the □ 3 page where a restatement of the proposition and its proof can be found in the Appendix. The restatement of the proposition in turn links back to the main body, via a margin reference as in this line. Appendix B contains Agda code ↺ 3 proving that the denotational by-need interpreter is total. As a case study, I wrote a denotational interpreter for GHC Core, the intermediate representation of GHC, which can be found in Appendix C. The main body omits some Haskell definitions for brevity; their full definition can be found in Appendix D. Chapter 5 finishes with concluding remarks and ideas for future work.

2

Background

2.1 Haskell

Understanding Haskell is instrumental for coverage checking in Chapter 3 (where it appears as *object* language) as well as for defining denotational interpreters in Chapter 4 (where it appears as concise *meta* language). For this reason, I will briefly recall the more unusual language features of GHC Haskell that will become relevant in later chapters and that set Haskell apart from, say, Standard ML or Rust.

2.1.1 Lazy Evaluation and Purity

Haskell was born out of a coordinated effort to standardise a language that uses *lazy evaluation*, that is, in which every expression is only evaluated when its result is needed. Consider for example the function

```
ifThenElse :: Bool → a → a → a
ifThenElse True  t _ = t
ifThenElse False _ e = e
```

In a regular, strict language, the expression $\mathit{ifThenElse}\ (d == 0)\ n\ (\mathit{div}\ n\ d)$ would throw a division-by-zero error whenever d is zero, because the call-by-value evaluation strategy demands that $(\mathit{div}\ n\ d)$ is evaluated *eagerly*, before the function $\mathit{ifThenElse}$ is entered.

Not so in a lazy language such as Haskell: there we get the same result as if the expression was written using the builtin conditional syntax, $\mathbf{if}\ d == 0\ \mathbf{then}\ n\ \mathbf{else}\ \mathit{div}\ n\ d$, because the division is only needed in the **else** branch where d is non-zero.

On stock hardware, lazy evaluation is commonly implemented by turning the deferred computation, for example $(\mathit{div}\ n\ d)$, into a nullary function called a

thunk. This nullary function is then called (the thunk is *forced*) where the result is needed, for example in the second clause of *ifThenElse*.

Thunking inevitably leads to the allocation of a closure record to store the values of the free variables n and d (which themselves might be thunks) of the implied nullary function. Often, it is far more expensive to allocate this closure than to simply compute the result of the expression in the first place, which is why an eager by-value evaluation strategy is preferable in case the expression is going to be needed anyway (used *strictly*). For that reason, GHC's Demand Analysis computes *strictness* information and rewrites programs to evaluate an expression by-value whenever it is used strictly.

There are in fact two evaluation strategies modelling lazy evaluation. So far, I have described the *call-by-name* evaluation strategy; however GHC implements the *call-by-need* evaluation strategy, which is an essential optimisation of by-name evaluation and discussed next.

Consider the expression $sum\ (map\ (+x)\ [1\,..\,99])$. Under the by-name thunk forcing strategy described above, the thunk bound to x will be forced 99 times, each time calling a potentially expensive nullary function, the result of which is the same for each call. Compared to call-by-value, where the computation that defines the value of x is evaluated exactly once, this repetition is exceedingly wasteful.

Call-by-need avoids such repeated work by utilising a mutable heap to implement a *memoising* thunk forcing strategy: After x has been forced and its result computed, call-by-need will memoise this result by *overwriting* the heap object that represents the nullary function and its closure with a different nullary function that returns the result right away.

For this optimisation to be safe, it is crucial that x is *pure*, that is: Evaluating x has no side-effects or hidden inputs, so that its re-evaluation can be discarded. Indeed, Haskell's type and runtime system is designed in a way that isolates side-effects to computations in the monadic type constructor IO. There is no (safe) way to run an IO computation other than by binding it to *main*, and a side-effecting computation wrapped in IO is implemented in a way that its result is never subject to memoisation. Since all non-IO computations are pure, memoisation is safe.

Purity is *the* essential language feature of Haskell, because it enables efficient lazy evaluation through memoisation and remorseless refactorings. While recent years indicate that lazy evaluation has not attracted much uptake in language design, purity had lasting impact on the world of functional programming languages.

2.1.2 Higher-Kinded Types and Quantified Constraints

It is very easy and convenient to define parametrically polymorphic functions (that is, "generic functions" in Java terms) in Haskell. Function *ifThenElse* above is one such example which is polymorphic in the result type.

However, Haskell does not only allow to parameterise over *types* of kind Type, it also allows parameterisation over *type constructors*, such as Maybe or lists [ ], of kind Type → Type.

It is sometimes useful to say that some data type which has a type constructor parameter preserves the instance of this parameter, for example to define a type class for monad transformers:

```
class (∀m. Monad m ⇒ Monad (t m)) ⇒ MonadTrans t where
  lift :: m a → t m a
```

The super class constraint (∀ *m*. Monad *m* ⇒ Monad (*t m*)) of MonadTrans says: "A monad transformer *t* is a type constructor such that *t m* is a monad for any monad *m*." Note that a constraint of the form (∀*m*. ... ⇒ ...) is called a *quantified* constraint [Bottu et al. 2017], a non-standard extension of Haskell. Section 4.3 uses quantified constraints to similar effect.

2.1.3 Generalised Algebraic Datatypes

Generalised Algebraic Datatypes (GADTs) [Xi, Chen, et al. 2003] allow capturing of type information at data constructor invokation site with the goal of unleashing it at a pattern-match of that constructor. For example, consider the declaration

```
data IntOrBool a where
  IsInt  :: Int  → IntOrBool Int
  IsBool :: Bool → IntOrBool Bool

negate :: IsIntOrBool a → a
negate (IsInt n)  = −n
negate (IsBool b) = not b
```

Note that *negate* is defined *for all* types *a*, such as Char. Neither integer negation (−) nor boolean negation *not* are defined at Char, yet the definition of *negate* is well-typed. This is because the match on constructor IsInt unleashes a *type equality constraint* *a* ∼ Int on the type parameter *a*, which effectively rewrites *a* to Int (similar for IsBool). This type equality constraint is provided at construction sites of IsInt. That is, the declaration

```
nono :: Int → IsInt a
nono n = IsInt n
```

is ill-typed, because a must be `Int` but could not be proven so. In this way, the type equality constraint $a \sim \mathsf{Int}$ can be thought of as an additional, invisible constructor field of `IsInt` that is packed and unpacked automatically.

GADTs and type equality constraints are instrumental to pattern-match coverage checking and thus to Chapter 3.

2.2 Order Theory

Let me briefly recall the essentials of order theory to appreciate the soundness results in Section 4.6.

The definitions of the algebraic structures *preorder*, *partial order* and *(complete) lattice* are standard and can be looked up in canonical works such as Nielson et al. [1999, Appendix A] or Cousot [2021, Chapter 10].

However, for the purposes of introducing notation, let me recall a few definitions. The notation $l \triangleq r$ defines l via r. For example, $\wp(S) \triangleq \{ U \mid U \subseteq S \}$ defines $\wp(S)$ as the powerset of S. A pair $(L, \leqslant)$ is a *preorder* if and only if L is a set equipped with a binary relation $\leqslant \in \wp(L \times L)$ on L that is reflexive and transitive. When the context is unambiguous, I will often omit the binary relation and simply say that L *is* a preorder, and similarly for partial orders and lattices.

A function $f : (A, \leqslant) \to (B, \sqsubseteq)$ between preorders is *monotone* if it preserves the ordering, that is, $\forall a_1, a_2.\ a_1 \leqslant a_2 \implies f(a_1) \sqsubseteq f(a_2)$.

A pair of monotone functions $\alpha : (C, \leqslant) \to (A, \sqsubseteq)$, $\gamma : (A, \sqsubseteq) \to (C, \leqslant)$ forms a *Galois connection*, abbreviated $\alpha : (C, \leqslant) \rightleftarrows (A, \sqsubseteq) : \gamma$ or even just $\alpha \rightleftarrows \gamma$, if and only if, for all $a \in A$ and $c \in C$, $\alpha(c) \sqsubseteq a \iff c \leqslant \gamma(a)$. That is, I assume the *monotone* notion of Galois connection.

A particularly convenient way for Section 4.6 to define a Galois connection is by giving a representation function. When a function $\beta : C \to A$ maps into a complete lattice $(A, \sqsubseteq, \sqcup)$, we can define

$$\begin{aligned} \alpha(C) &\triangleq \bigsqcup \{ \beta(c) \mid c \in C \} \\ \gamma(a) &\triangleq \{ c \mid \beta(c) \sqsubseteq a \} \end{aligned}$$

Then $\alpha : (\wp(C), \subseteq) \rightleftarrows (A, \sqsubseteq) : \gamma$ form a Galois connection, and β is called the *representation function* of the Galois connection [Nielson et al. 1999, Section 4.3]. (Cousot [2021, Exercise 11.8] calls this construction "partitioning abstraction" instead.)

2.3 Abstract Interpretation

The powerset over a set S can be regarded as the set of *properties* of elements in S. Under this view, an element $s \in S$ has a property $P \in \wp(S)$ if and only if $s \in P$. The stronger the property, the smaller the set, so $\subseteq$ models implication.

Abstract interpretation applies this view to *semantic properties*, that is, properties of the semantics of programs. Given a program semantics $\mathcal{S}[\![\mathrm{e}]\!]$ that assigns expression e meaning in some semantic domain D, we can define the *collecting semantics* as the function $\mathcal{S}_{\mathbb{C}}[\![\mathrm{e}]\!] \triangleq \{\mathcal{S}[\![\mathrm{e}]\!]\}$ that maps e to the *strongest semantic property* $\{\mathcal{S}[\![\mathrm{e}]\!]\} \in \wp(D)$ it satisfies.

If the program e has some semantic property P of interest, say "n is always even", then $\mathcal{S}_{\mathbb{C}}[\![\mathrm{e}]\!] \subseteq P$ must be provable. However, such an implication is in general undecidable by Rice's theorem, hence a static program analysis $\mathcal{A}[\![\mathrm{e}]\!]$ for property P must approximate the decision problem in a suitable sense. Furthermore, the elements of D are potentially infinite and thus difficult to compute with, so $\mathcal{A}[\![\mathrm{e}]\!]$ operates in some symbolic domain A. This domain is equipped with a complete lattice structure $(A, \sqsubseteq, \sqcup)$ and relates to the complete lattice of semantic properties $(\wp(D), \subseteq, \cup)$ via a monotone concretisation function $\gamma : A \to \wp(D)$.

The analysis $\mathcal{A}[\![\_]\!]$ is conservative, or *sound*, wrt. P if

$$\forall \mathrm{e}.\ \gamma(\mathcal{A}[\![\mathrm{e}]\!]) \subseteq P \implies \mathcal{S}_{\mathbb{C}}[\![\mathrm{e}]\!] \subseteq P. \tag{2.1}$$

In other words: If $\mathcal{A}[\![\_]\!]$ can prove that e has property P, then e really has that property according to $\mathcal{S}_{\mathbb{C}}[\![\_]\!]$. Conversely, $\mathcal{A}[\![\_]\!]$ may be *incomplete* wrt. P: If $\mathcal{A}[\![\_]\!]$ is *unable* to prove that e has property P, e may still have property P.

When $\mathcal{A}[\![\_]\!]$ is simultaneously sound for any property in the image of γ we say that $\mathcal{A}[\![\_]\!]$ is *sound* wrt. γ and the soundness criterion (2.1) simplifies to

$$\forall \mathrm{e}.\ \mathcal{S}_{\mathbb{C}}[\![\mathrm{e}]\!] \subseteq \gamma(\mathcal{A}[\![\mathrm{e}]\!]). \tag{2.2}$$

In this way, the choice of γ determines the semantic properties P that can be soundly inferred by $\mathcal{A}[\![\_]\!]$.

The essence of *abstract interpretation* is the study of conservative approximation by recognising soundness relationships such as (2.2). Often, for any property P there exists a "best abstract property" (i.e. *least*) $\widehat{P}$ such that $P \subseteq \gamma(\widehat{P})$, in which case there exists an abstraction function α such that $\alpha : (\wp(D), \subseteq) \rightleftarrows (A, \sqsubseteq) : \gamma$ forms a Galois connection. In this case, soundness of $\mathcal{A}[\![\_]\!]$ can be equivalently expressed as

$$\forall \mathrm{e}.\ \alpha(\mathcal{S}_{\mathbb{C}}[\![\mathrm{e}]\!]) \sqsubseteq \mathcal{A}[\![\mathrm{e}]\!],$$

and that is the form of the statement I will prove in Section 4.6.

3

Lower Your Guards: A Compositional Pattern-Match Coverage Checker

Program definition by pattern matching is a tremendously useful feature in Haskell and many other programming languages, but it must be used with care. Consider this example of a function defined by pattern matching:

```
f :: Int → Bool
f 0 = True
f 0 = False
```

The function f has two serious flaws. One obvious problem is that there are two clauses that match on 0, and due to the top-to-bottom semantics of pattern matching, this makes the `f 0 = False` clause completely unreachable. Even worse is that f never matches on any patterns besides 0, rendering its definition incomplete. Attempting to invoke `f 1`, for instance, will crash.

To avoid these mishaps, compilers for languages with pattern matching often emit warnings (or errors) if a function is missing clauses (i.e. if it is *non-exhaustive*), if one of its right-hand sides will never be entered (i.e. if it is *inaccessible*), or if one of its equations can be deleted altogether (i.e. if it is *redundant*). Let us refer to the combination of checking for exhaustivity, redundancy, and accessibility as *pattern-match coverage checking*. Coverage checking is the first line of defense in catching programmer mistakes when defining code that uses pattern matching.

Coverage checking for a set of equations matching on algebraic data types is a well studied (although still surprisingly tricky) problem—see Section 3.7 for this related work. But the coverage-checking problem becomes *much* harder when one includes the raft of innovations that have become part of a modern

programming language like Haskell, including: view patterns, pattern guards, pattern synonyms, overloaded literals, bang patterns, lazy patterns, as-patterns, strict data constructors, empty case expressions, and long-distance effects (Section 3.3). Particularly tricky are: *Generalised Algebraic Datatypes* (*GADTs*) where the *type* of a match can determine what *values* can possibly appear [Xi, Chen, et al. 2003]; and *local type-equality constraints* brought into scope by pattern matching [Vytiniotis et al. 2011].

The state of the art[1] for coverage checking in a richer language of this sort was *GADTs Meet Their Match* [Karachalias et al. 2015], or GMTM for short. It presents an algorithm that handles the intricacies of checking GADTs, lazy patterns, and pattern guards. However, GMTM is monolithic and does not account for a number of important language features; it gives incorrect results in certain cases; its formulation in terms of structural pattern matching makes it hard to avoid some serious performance problems; and its implementation in the Glasgow Haskell Compiler (GHC), while a big step forward over its predecessors, has proved complex and hard to maintain.

Contributions. In this chapter, I propose a new, compositional coverage-checking algorithm, called Lower Your Guards (LYG), that is simpler, more modular, *and* more powerful than GMTM (see Section 3.7.1). Moreover, it avoids GMTM's performance pitfalls.

- I characterise some nuances of coverage checking that not even GMTM handles (Section 3.1). I also identify issues in GHC's implementation of GMTM.
- I describe a new, compositional coverage checking algorithm, LYG, in Section 3.2. The key insight is to abandon the notion of structural pattern matching altogether, and instead desugar all the complexities of pattern matching into a very simple language of *guard trees*, with just three constructs (Section 3.2.1). Coverage checking on these guard trees becomes remarkably simple, returning an *annotated tree* (Section 3.2.2) decorated with *refinement types*. Finally, provided there is a suitable way to find inhabitants of a refinement type, one can report accurate coverage errors (Section 3.2.3).

[1] Before this work appeared at ICFP 2020, that is!

- I demonstrate the compositionality of LYG by augmenting it with several language extensions (Section 3.3). Although these extensions can change the source language in significant ways, the effort needed to incorporate them into the algorithm is comparatively small.
- I discuss how to optimize the performance of LYG (Section 3.4) and implement a proof of concept in GHC (Section 3.5).
- The evaluation against a large number of Haskell packages (Section 3.5) provides evidence that LYG is sound. In order to discuss soundness formally in Section 3.6, I turn the informal semantics of guard trees and refinement types in Section 3.2 into a formal semantics. I also list mechanisms that render LYG incomplete in order to guarantee good performance.
- The wealth of related work is discussed in Section 3.7.

Acknowledgements. The work in this chapter is an extended version of Graf, Peyton Jones, et al. [2020]. It is the result of a research internship with Simon Peyton Jones at Microsoft Research Cambridge in 2019, in which I completely overhauled GHC's neglected pattern-match coverage checker, following ideas that Simon and I developed and which I implemented. Our third author, Ryan Scott, had previously improved parts of the coverage checker and was of great help in improving the technical writing, as well as contributing the evaluation (Section 3.5), Related Work (Section 3.7) and introductory examples.

Since this work appeared at ICFP 2020, its implementation in GHC evolved as well. I describe how LYG accommodates a new, unanticipated language extension for *Or-patterns* (Section 3.3.9) and report a useful structural pattern to model guard trees from the trenches (Section 3.4.6). Furthermore, Section 3.6 summarises a formalisation of significant parts of LYG that have been formalised by Dieterichs [2021], a thesis supervised by Sebastian Ullrich and me.

3.1 Problem Statement

What makes coverage checking so difficult in a language like Haskell? At first glance, implementing a coverage checking algorithm might appear simple: just check that every function matches on every possible combination of data constructors exactly once. A function must match on every possible combination of

constructors in order to be exhaustive, and it must match on them exactly once to avoid redundant matches.

This algorithm, while concise, leaves out many nuances. What constitutes a "match"? Haskell has multiple matching constructs, including function definitions, **case** expressions, and guards. How does one count the number of possible combinations of data constructors? This is not a simple exercise since term and type constraints can make some combinations of constructors unreachable if matched on, and some combinations of data constructors can overlap others. Moreover, what constitutes a "data constructor"? In addition to traditional data constructors, GHC features *pattern synonyms* [Pickering et al. 2016], which provide an abstract way to embed arbitrary computation into patterns. Matching on a pattern synonym is syntactically identical to matching on a data constructor, which makes coverage checking in the presence of pattern synonyms challenging.

Prior work on coverage checking (discussed in Section 3.7) accounts for some of these nuances, but not all of them. In this section I identify some key language features that make coverage checking difficult. While these features may seem disparate at first, I will later show in Section 3.2 that these ideas can all fit into a unified framework.

3.1.1 Guards

Guards are a flexible form of control flow in Haskell. Here is a function that demonstrates various capabilities of guards:

```
guardDemo :: Char → Char → Int
guardDemo c1 c2 | c1 == 'a'                          = 0
                | 'b' ← c1                           = 1
                | let c1' = c1, 'c' ← c1', c2 == 'd' = 2
                | otherwise                          = 3
```

This function has four *guarded right-hand sides* or GRHSs for short. The first GRHS has a *boolean guard*, (c_1 == 'a'), that succeeds if the expression in the guard returns True. The second GRHS has a *pattern guard*, ('b' ← c_1), that succeeds if the pattern in the guard successfully matches. The next line illustrates that each GRHS may have multiple guards, and that guards include **let** bindings, such as **let** $c_1' = c_2$. The fourth GRHS uses *otherwise*, which is simply defined as True.

Guards can be thought of as a generalisation of patterns, and a useful coverage checker should include them. Checking guards is significantly more complicated

than checking ordinary structural pattern-matches, however, since guards can contain arbitrary expressions. Consider this implementation of the *signum* function:

```
signum :: Int → Int
signum x | x > 0  = 1
         | x == 0 = 0
         | x < 0  = −1
```

Intuitively, *signum* is exhaustive since the combination of (>), (==), and (<) covers all possible Ints. This is hard for a machine to check, because doing so requires knowledge about the properties of Int inequalities. Clearly, coverage checking for guards is undecidable in general. However, while we cannot accurately check *all* uses of guards, we can at least give decent warnings for some common cases. For instance, take the following functions:

```
not :: Bool → Bool
not b | False ← b = True
      | True ← b  = False

not2 :: Bool → Bool
not2 False = True
not2 True  = False

not3 :: Bool → Bool
not3 x | False ← x = True
not3 True          = False
```

Clearly all are equivalent. A coverage checking algorithm should find that all three are exhaustive, and indeed, LYG does so.

3.1.2 Programmable Patterns

Expressions in guards are not the only source of undecidability that the coverage checker must cope with. GHC extends the pattern language in other ways that are also impossible to check in the general case. We consider two such extensions here: view patterns and pattern synonyms.

View Patterns

View patterns allow arbitrary computation to be performed while pattern matching. When a value v is matched against a view pattern $(f \to p)$, the match is successful when $f\ v$ successfully matches against the pattern p. For example, one can use view patterns to succinctly define a function that computes the length of Haskell's opaque Text data type:

```
Text.null :: Text → Bool
Text.uncons :: Text → Maybe (Char, Text)
```

```
length :: Text → Int
length (Text.null → True)          = 0
length (Text.uncons → Just (_, xs)) = 1 + length xs
```

Function Text.*null* returns True when the given Text is empty. When it is non-empty, it must have a first character x :: Char and the remainder xs :: Text, in which case Text.*uncons* returns Just (x, xs).

Again, it would be unreasonable to expect a coverage checking algorithm to prove that the definition of *length* is exhaustive, but one might hope for a coverage checking algorithm that handles some common usage patterns. For example, LYG indeed *is* able to prove that the *safeLast* function is exhaustive:

```
safeLast :: [a] → Maybe a
safeLast (reverse → [])     = Nothing
safeLast (reverse → (x : _)) = Just x
```

Pattern Synonyms

Pattern synonyms [Pickering et al. 2016] allow abstraction over patterns themselves. Pattern synonyms and view patterns can be useful in tandem, as the pattern synonym can present an abstract interface to a view pattern that does complicated things under the hood. For example, one can define *length* with pattern synonyms like so:

```
pattern Nil :: Text
pattern Nil ← (Text.null → True)
pattern Cons :: Char → Text → Text
pattern Cons x xs ← (Text.uncons → Just (x, xs))

length :: Text → Int
length Nil = 0
length (Cons _ xs) = 1 + length xs
```

The pattern synonym Nil matches precisely when the view pattern Text.*null* → True would match, and similarly for Cons.

How should a coverage checker handle pattern synonyms? One idea is to simply "look through" the definitions of each pattern synonym and verify whether the underlying patterns are exhaustive. This would be undesirable, however, because (1) we would like to avoid leaking the implementation details

of abstract pattern synonyms, and (2) even if we *did* look at the underlying implementation, it would be challenging to automatically check that the combination of Text.*null* and Text.*uncons* is exhaustive.

Nevertheless, Text.*null* and Text.*uncons* together are in fact exhaustive, and GHC allows programmers to communicate this fact to the coverage checker using a `COMPLETE` pragma [GHC team 2020]. A `COMPLETE` set is a combination of data constructors and pattern synonyms that should be regarded as exhaustive when a function matches on all of them. For example, declaring `{-# COMPLETE Nil, Cons #-}` is sufficient to make the definition of *length* above compile without any exhaustivity warnings. Since GHC does not (and cannot, in general) check that all of the members of a `COMPLETE` set actually comprise a complete set of patterns, the burden is on the programmer to ensure that this invariant is upheld.

3.1.3 Strictness

The evaluation order of pattern matching can impact whether a pattern is reachable or not. Consider:

```
f :: Bool → Bool → Int
f _     False = 1
f True  False = 2
f _     _     = 3
```

Is the second clause redundant? In a strict language such as OCaml or Lean the answer is “Yes”, but in lazy Haskell the answer is “No”. To see that, consider the call *f* (*error* `"boom"`) True, an expression that in a strict language would immediately evaluate the error in the argument (*error* `"boom"`) by-value before making the call.

In Haskell, however, the first argument is not evaluated until it is needed. Concretely, after falling through the first clause that does not match in the second argument, the second clause will evaluate the first argument in order to match against True. Doing so forces the error, to much the same effect as in a strict language, and we see that *f* (*error* `"boom"`) True evaluates to *error* `"boom"`. However, if we *remove* the second clause, *f* (*error* `"boom"`) True would evaluate to 3, because the first argument is never needed during pattern matching. Since removing the clause changes the semantics of the function, it cannot be redundant, but its right-hand side is *inaccessible* still (Section 3.1.3).

We could have used a different error such as *undefined* or a non-terminating expression such as *loop* = *loop* in the example above. The Haskell Language Report [Marlow et al. 2010] does not distinguish between these different kinds of divergence, referring to them as ⊥, and I do the same here. There are a number of language features which interact with ⊥ in the context of pattern matching.

Redundancy versus Inaccessibility

The example function *f* above demonstrates that when reporting unreachable equations, we must distinguish between *redundant* and *inaccessible* cases. A redundant equation can be removed from a function without changing its semantics, whereas an inaccessible equation cannot, even though its right-hand side is unreachable. The examples below illustrate the challenges for LYG in more detail:

```
u :: () → Int
u () | False = 1
     | True  = 2
u _          = 3
```

```
u' :: () → Int
u' () | False = 1
      | False = 2
u' _          = 3
```

Within *u*, the equations that return 1 and 3 could be deleted without changing the semantics of *u*, so they are classified as *redundant*. Within *u*′, the right-hand sides of the equations that return 1 and 2 are inaccessible, but they cannot both be redundant because their clause evaluates the parameter; *u*′ ⊥ = ⊥. As a result, LYG picks the first equation in *u*′ as inaccessible to keep alive the pattern-match on the parameter, and the second equation as redundant. Inaccessibility suggests to the programmer that *u*′ might benefit from a refactor to render the first equation redundant as well (e.g. *u*′ () = 3).

Observe that *u* and *u*′ have completely different warnings, but the only difference between the two functions is whether the second equation uses True or False in its guard. Moreover, this second equation affects the warnings for *other* equations. This demonstrates that determining whether code is redundant or inaccessible is a non-local problem. Inaccessibility may seem like a corner case, but GHC's users have reported many bugs of this sort (Section 3.5.2).

Strict Fields

Just like Haskell function parameters, fields of data constructors may hide arbitrary computations as well. However, Haskell programmers can opt into extra strict evaluation by giving a data type strict fields, such as in this example:

```
data Void  -- No data constructors; only inhabitant is bottom
data SMaybe a = SJust !a | SNothing

v :: SMaybe Void → Int
v SNothing = 0
v (SJust _)  = 1  -- Redundant!
```

The "!" in the definition of SJust makes the constructor strict, so (SJust ⊥) ≡ ⊥ semantically, in contrast to the regular lazy Just constructor.

Curiously, the strict field semantics of SJust makes the second equation of v redundant! Since ⊥ is the only inhabitant of type Void, the only inhabitants of SMaybe Void are SNothing and ⊥. The former will match on the first equation; the latter will make the first equation diverge. In neither case will execution flow to the second equation, so it is redundant and can be deleted.

Bang Patterns

Strict data-constructor fields are one mechanism for adding extra strictness in ordinary Haskell, but GHC adds another in the form of *bang patterns*. When a value v is matched against a bang pattern !*pat*, first v is evaluated to weak-head normal form (WHNF), a step that might diverge, and then v is matched against *pat*. Here is a variant of v, this time using the standard, lazy Maybe data type:

```
v' :: Maybe Void → Int
v' Nothing = 0
v' (Just !_) = 1  -- Not redundant, but GRHS is inaccessible
```

The inhabitants of the type Maybe Void are ⊥, Nothing, and (Just ⊥). The input ⊥ makes the first equation diverge; Nothing matches on the first equation; and (Just ⊥) makes the second equation diverge because of the bang pattern. Therefore, none of the three inhabitants will result in the right-hand side of the second equation being reached. Note that the second equation is inaccessible, but not redundant.

3.1.4 Type-Equality Constraints

Besides strictness, another way for pattern-matches to be rendered unreachable is by way of *type equality constraints*. A popular method for introducing equalities between types is matching on GADTs [Xi, Chen, et al. 2003]. The following examples demonstrate the interaction between GADTs and coverage checking:

```
data T a b where      g1 :: T Int b → b → Int    g2 :: T a b → T a b → Int
  T1 :: T Int Bool    g1 T1 False = 0            g2 T1 T1 = 0
  T2 :: T Char Bool   g1 T1 True = 1             g2 T2 T2 = 1
```

When g_1 matches against T1, the b in the type T Int b is known to be a Bool, which is why matching the second argument against False or True will typecheck. Phrased differently, matching against T1 brings into scope an *equality constraint* between the types b and Bool. GHC has a powerful type inference engine that is equipped to reason about type equalities of this sort [Vytiniotis et al. 2011].

Just as important as the code used in the g_1 function is the code that is *not* used in g_1. One might wonder if g_1 not matching its first argument against T2 is an oversight. In fact, the exact opposite is true: matching on T2 would be rejected by the typechecker. This is because T2 is of type T Char Bool, but the first argument to g_1 must be of type T Int b. Matching against T2 would be tantamount to saying that Int and Char are the same type, which is not the case. As a result, g_1 is exhaustive even though it does not match on all of T's data constructors.

The presence of type equalities is not always as clear-cut as it is in g_1. Consider the more complex g_2 function, which matches on two arguments of the type T a b. While matching the arguments against T1 T1 or T2 T2 is possible, it is not possible to match against T1 T2 or T2 T1. To see why, suppose the first argument is matched against T1, giving rise to an equality between a and Int. If the second argument were then matched against T2, we would have that a equals Char. By the transitivity of type equality, we would have that Int equals Char. This cannot be true, so matching against T1 T2 is impossible (and similarly for T2 T1).

Concluding that g_2 is exhaustive requires some non-trivial reasoning about equality constraints. In GHC, the same engine that typechecks GADT pattern-matches is also used to rule out cases made unreachable by type equalities, and LYG adopts a similar approach. Besides GHC's previous coverage checker [Karachalias et al. 2015], there are a variety of other coverage checking algorithms that account for GADTs, including those for OCaml [Garrigue and Normand 2011], Dependent ML [Xi 1998a,b, 2003], and Stardust [Dunfield 2007].

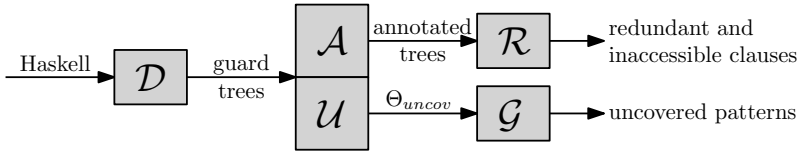


Fig. 3.1: Bird's eye view of pattern-match checking

Meta variables

x, y, z, f, g, h	Term variables
a, b, c	Type variables
K	Data constructors
P	Pattern synonyms
T	Type constructors
l	Literal
$expr$	Expressions

Pattern syntax

$$
\begin{array}{rcl}
defn & ::= & \overline{clause} \\
clause & ::= & f\ \overline{pat}\ match \\
pat & ::= & x \mid \_ \mid K\ \overline{pat} \mid x@pat \\
 & \mid & !\,pat \mid expr \to pat \\
match & ::= & = expr \mid \overline{grhs} \\
grhs & ::= & \mid \overline{guard} = expr \\
guard & ::= & pat \leftarrow expr \mid expr \\
 & \mid & \texttt{let}\ x = expr
\end{array}
$$

Fig. 3.2: Source syntax: A desugared Haskell

3.2 Lower Your Guards: A New Coverage Checker

In this section, I describe the new coverage checking algorithm, LYG. Figure 3.1 depicts a high-level overview, which divides into three steps:

- First, we desugar the complex source Haskell syntax (cf. Figure 3.2) into a ***guard tree*** $t \in \mathrm{Gdt}$ (Section 3.2.1). The language of guard trees is tiny but expressive, and allows the subsequent passes to be entirely independent of the source syntax. LYG can readily be adapted to other languages simply by changing the desugaring algorithm.
- Next, the resulting guard tree is then processed by two different functions (Section 3.2.2). The function $\mathcal{A}(t)$ produces an ***annotated tree*** $u \in \mathrm{Ant}$, which has the same general branching structure as t but describes which clauses are accessible, inaccessible, or redundant. The function $\mathcal{U}(t)$, on the other hand, returns a ***refinement type*** Θ [Rushby et al. 1998; Xi and Pfenning 1998] that describes the set of ***uncovered values***, which are not matched by any of the clauses.

Guard syntax

$$
\begin{array}{rcl}
k, n, m & \in & \mathbb{N} \\
K & \in & \mathsf{Con} \\
x, y, a, b & \in & \mathsf{Var} \\
\tau, \sigma & \in & \mathsf{Type} ::= a \mid \ldots \\
e & \in & \mathsf{Expr} ::= x \mid K\ \overline{\tau}\ \overline{\gamma}\ \overline{e} \mid \ldots
\end{array}
\qquad
\begin{array}{rcl}
\gamma & \in & \mathsf{TyCt} ::= \tau_1 \sim \tau_2 \mid \ldots \\
p & \in & \mathsf{Pat} ::= \_ \mid K\ \overline{p} \mid \ldots \\
g & \in & \mathsf{Grd} ::= \mathsf{let}\ x : \tau = e \\
 & & \quad \mid\ K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x \\
 & & \quad \mid\ !x
\end{array}
$$

Clause tree syntax

$$
\begin{array}{rcl}
t & \in & \mathsf{Gdt} ::= \longrightarrow k \mid \top\!\!\sqsubset \begin{array}{l} t_1 \\ t_2 \end{array} \mid \dashv g \text{---} t \\
u & \in & \mathsf{Ant} ::= \longrightarrow \Theta\, k \mid \top\!\!\sqsubset \begin{array}{l} u_1 \\ u_2 \end{array} \mid \text{---}\, \Theta \,\lightning \text{---} u
\end{array}
$$

Refinement type syntax

$$
\begin{array}{rcll}
\Gamma & ::= & \varnothing \mid \Gamma, x : \tau \mid \Gamma, a & \text{Context} \\
\varphi & ::= & \checkmark \mid \times \mid K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x \mid x \not\approx K & \text{Literals} \\
 & \mid & x \approx \bot \mid x \not\approx \bot \mid \mathsf{let}\ x = e & \\
\Phi & ::= & \varphi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi & \text{Formula} \\
\Theta & ::= & \langle\, \Gamma \mid \Phi \,\rangle & \text{Refinement type}
\end{array}
$$

Fig. 3.3: IR syntax

- Finally, an error-reporting pass generates comprehensible error messages (Section 3.2.3). Again there are two things to do. The function $\mathcal{R}$ processes the annotated tree produced by $\mathcal{A}$ to explicitly identify the accessible, inaccessible, or redundant clauses. The function $\mathcal{G}(\Theta)$ produces representative ***inhabitants*** of the refinement type Θ (produced by $\mathcal{U}$) that describes the uncovered values.

LYG's main contribution when compared to other coverage checkers, such as GHC's implementation of GMTM, is its incorporation of many small improvements and insights, rather than a single defining breakthrough. In particular, LYG's advantages are:

- Achieving modularity by clearly separating the source syntax (Figure 3.2) from the intermediate language (Figure 3.3).
- Correctly accounting for strictness in identifying redundant and inaccessible code (Section 3.7.5).
- Using detailed term-level reasoning (Figures 3.7, 3.8 and 3.10), which GMTM does not.
- Using *negative information* to sidestep serious performance issues in GMTM without changing the worst-case complexity (Section 3.7.4). This also enables graceful degradation (Section 3.4.3) and the ability to handle `COMPLETE` sets properly (Section 3.4.4).
- Fixing various bugs present in GMTM, both in the paper [Karachalias et al. 2015] and in GHC's implementation thereof (Section 3.5.2).

3.2.1 Desugaring to Guard Trees

The first step is to desugar the source language into the language of guard trees. The syntax of the source language is given in Figure 3.2. Definitions *defn* consist of a list of *clauses*, each of which has a list of *patterns*, and a list of *guarded right-hand sides* (GRHSs). Patterns include variables and constructor patterns, of course, but also a representative selection of extensions: wildcards, as-patterns, bang patterns, and view patterns. We explore several other extensions in Section 3.3.

The language of guard trees Gdt is much smaller; its graphical syntax is given in Figure 3.3. All of the syntactic redundancy of the source language is translated into a minimal form very similar to pattern guards. We start with an example:

$$
\begin{array}{lll}
f\ (\mathsf{Just}\ (!\,xs, \_)) & ys & = \mathsf{True} \\
f\ \mathsf{Nothing} & (g \to \mathsf{True}) & = \mathsf{False}
\end{array}
$$

This desugars to the following guard tree (where the x_i represent f's arguments):

$$
\begin{array}{l}
\dashv !x_1, \mathsf{Just}\ t_1 \leftarrow x_1, !t_1, (t_2, t_3) \leftarrow t_1, !t_2, \mathsf{let}\ xs = t_2, \mathsf{let}\ ys = x_2 \longrightarrow 1 \\
\dashv !x_1, \mathsf{Nothing} \leftarrow x_1, \mathsf{let}\ t_4 = g\ x_2, !t_4, \mathsf{True} \leftarrow t_4 \longrightarrow 2
\end{array}
$$

The first line says "evaluate x_1; then match x_1 against Just t_1; then evaluate t_1; then match t_1 against (t_2, t_3)" and so on. If any of those matches fail, we fall through into the second line. Note that I write $\dashv g_1, \ldots, g_n — t$ instead of $\dashv g_1 \dashv \cdots \dashv g_n — t$ for notational convenience.

$$\boxed{\mathcal{D}(defn) = t, \quad \mathcal{D}(\overline{x}, clause) = t, \quad \mathcal{D}(grhs) = t \\ k_{rhs} \text{ is the index of the right hand side } rhs}$$

$$\begin{array}{lcll}
\mathcal{D}(\overline{clause}^{n}) & = & \begin{array}{l}\mathcal{D}(\overline{x}, clause_1) \\ \ldots \\ \mathcal{D}(\overline{x}, clause_n)\end{array} & \overline{x}^{m} \text{ fresh; } m \text{ arity of } \overline{clause} \\
\mathcal{D}(\overline{x}, f\ \overline{pat} = rhs) & = & \dashv \overline{\mathcal{D}(x, pat)} \longrightarrow k_{rhs} & \\
\mathcal{D}(\overline{x}, f\ \overline{pat}\ \overline{grhs}^{n}) & = & \dashv \overline{\mathcal{D}(x, pat)} \begin{array}{l}\mathcal{D}(grhs_1) \\ \ldots \\ \mathcal{D}(grhs_n)\end{array} & \\
\mathcal{D}(|\ \overline{guard} = rhs) & = & \dashv \overline{\mathcal{D}(guard)} \longrightarrow k_{rhs} &
\end{array}$$

$$\boxed{\mathcal{D}(guard) = \overline{g}, \quad \mathcal{D}(x, pat) = \overline{g}}$$

$$\begin{array}{lcll}
\mathcal{D}(pat \leftarrow expr) & = & \text{let } x = expr, \mathcal{D}(x, pat) & x \text{ fresh} \\
\mathcal{D}(expr) & = & \text{let } y = expr, \mathcal{D}(y, \text{True}) & y \text{ fresh} \\
\mathcal{D}(\texttt{let } x = expr) & = & \text{let } x = expr & \\
 & & & \\
\mathcal{D}(x, y) & = & \text{let } y = x & \\
\mathcal{D}(x, _) & = & \epsilon & \\
\mathcal{D}(x, K\ \overline{pat}) & = & !x, K\ \overline{y} \leftarrow x, \overline{\mathcal{D}(y, pat)} & \overline{y} \text{ fresh } (\dagger) \\
\mathcal{D}(x, y@pat) & = & \text{let } y = x, \mathcal{D}(y, pat) & \\
\mathcal{D}(x, \ !pat) & = & !x, \mathcal{D}(x, pat) & \\
\mathcal{D}(x, expr \rightarrow pat) & = & \text{let } y = expr\ x, \mathcal{D}(y, pat) & y \text{ fresh}
\end{array}$$

Fig. 3.4: $\mathcal{D}$esugaring from source language to Gdt

Informally, matching a guard tree may *succeed* (binding the variables bound in the tree), *fail*, or *diverge*. Referring to the syntax of guard trees in Figure 3.3, matching is defined as follows:

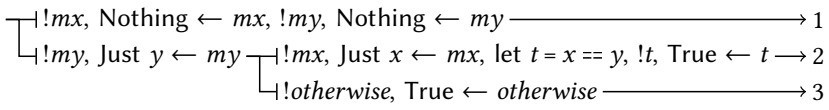
- Matching a guard tree $\longrightarrow k$ succeeds, and selects the k'th right hand side of the pattern-match group.
- Matching a guard tree $\begin{array}{|l} t_1 \\ t_2 \end{array}$ means matching against t_1; if that succeeds, the overall match succeeds; if not, match against t_2.
- Matching a guard tree $\longrightarrow\!| !x \multimap t$ evaluates x; if that diverges the match diverges; if not match t.
- Matching a guard tree $\longrightarrow\!| K\ \overline{a}\ \overline{\gamma}\ \overline{y} \leftarrow x \multimap t$ matches x against constructor K. If the match succeeds, bind $\overline{a}$ to the type components, $\overline{\gamma}$ to the constraint components and $\overline{y}$ to the term components, then match t. If the constructor match fails, then the entire match fails.
- Matching a guard tree $\longrightarrow\!| \mathsf{let}\ x = e \multimap t$ binds x (lazily) to e, and matches t.

See Section 3.6.1 for a formal account of this semantics. The desugaring algorithm, $\mathcal{D}$, is given in Figure 3.4. It is a straightforward recursive descent over the source syntax, with a little bit of bureaucracy to account for renaming. It also generates an abundance of fresh temporary variables; in practice, the implementation of $\mathcal{D}$ can be smarter than this by looking at the pattern (which might be a variable match or as-pattern) when choosing a name for a temporary variable. In that case, it is important that every binder in the source language has a unique name.

Notice that both "structural" pattern matching in the source language (e.g. the match on Nothing in the second equation), and view patterns (e.g. $g \rightarrow \mathsf{True}$) can be straightforwardly translated into a single form of matching in guard trees. The same holds for pattern guards. For example, consider this (stylistically contrived) definition of *liftEq*, which is inexhaustive:

```
liftEq Nothing Nothing = True
liftEq mx      (Just y) | Just x ← mx, x == y = True
                        | otherwise           = False
```

It desugars thus:



Notice that the pattern guard (Just $x \leftarrow mx$) and the boolean guard ($x == y$) have both turned into the same constructor-matching construct in the guard tree.

In equation (†) of Figure 3.4 we generate an explicit bang guard $!x$ to reflect the fact that pattern matching against a data constructor requires evaluation. However, Haskell's **newtype** declarations introduce data constructors that are *not* strict, so their desugaring is just like (†) but with no $!x$ (Section 3.3.7). From this point onwards, then, strictness is expressed *only* through bang guards $!x$, while constructor guards $K\ a\ b \leftarrow y$ are not considered strict.

In a way there is nothing very deep here, but it took Simon and me a surprisingly long time to come up with the language of guard trees.

3.2.2 Checking Guard Trees

The next step in Figure 3.1 is to transform the guard tree into an *annotated tree*, Ant, and an *uncovered set*, Θ. Taking the latter first, the uncovered set describes all the input values of the match that are not covered by the match. I use the language of *refinement types* to describe this set (see Figure 3.3). A refinement type $\Theta = \langle\, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \mid \Phi \,\rangle$ denotes the vector of values $x_1 \ldots x_n$ that satisfy the predicate Φ. For example (the type omitted in the last line is Maybe Bool):

$$
\begin{array}{rll}
\langle\, x{:}\mathsf{Bool} \mid \checkmark \,\rangle & \text{denotes} & \{\bot, \mathsf{True}, \mathsf{False}\} \\
\langle\, x{:}\mathsf{Bool} \mid x \not\approx \bot \,\rangle & \text{denotes} & \{\mathsf{True}, \mathsf{False}\} \\
\langle\, x{:}\mathsf{Bool} \mid x \not\approx \bot \wedge \mathsf{True} \leftarrow x \,\rangle & \text{denotes} & \{\mathsf{True}\} \\
\langle\, y{:}\ldots \mid y \not\approx \bot \wedge \mathsf{Just}\ x \leftarrow y \wedge x \not\approx \bot \,\rangle & \text{denotes} & \{\mathsf{Just\ True}, \mathsf{Just\ False}\}
\end{array}
$$

The syntax of formulas Φ is given in Figure 3.3. It consists of a collection of *literals* φ, combined with conjunction and disjunction. Unconventionally, however, a literal may bind one or more variables, and those bindings are in scope in conjunctions to the right. This can be formalised by giving a type system for Φ, and I do so in Section 3.6.1, where I define satisfiability of Φ in formal detail. The literal $\checkmark$ means "true", as illustrated above; while $\times$ means "false", so that $\langle\, \Gamma \mid \times \,\rangle$ denotes the empty set $\emptyset$.

The uncovered set function $\mathcal{U}(\Theta, t)$, defined in Figure 3.5, computes a refinement type describing the values in Θ that are not covered by the guard tree t. It

Operations on Θ

$$\begin{array}{lcl} \langle\, \Gamma \mid \Phi \,\rangle \mathbin{\dot\wedge} \varphi & = & \langle\, \Gamma \mid \Phi \wedge \varphi \,\rangle \\ \langle\, \Gamma \mid \Phi_1 \,\rangle \cup \langle\, \Gamma \mid \Phi_2 \,\rangle & = & \langle\, \Gamma \mid \Phi_1 \vee \Phi_2 \,\rangle \end{array}$$

Checking guard trees

$$\boxed{\mathcal{U}(\Theta, t) = \Theta}$$

$$\begin{array}{lcl} \mathcal{U}(\langle\, \Gamma \mid \Phi \,\rangle,\ \longrightarrow n\,) & = & \langle\, \Gamma \mid \times \,\rangle \\ \mathcal{U}(\Theta,\ \top\!\!\sqsubset^{t_1}_{t_2}) & = & \mathcal{U}(\mathcal{U}(\Theta, t_1), t_2) \\ \mathcal{U}(\Theta,\ \dashv !x \mathrel{—} t\,) & = & \mathcal{U}(\Theta \mathbin{\dot\wedge} (x \not\approx \bot), t) \\ \mathcal{U}(\Theta,\ \dashv \mathsf{let}\ x = e \mathrel{—} t\,) & = & \mathcal{U}(\Theta \mathbin{\dot\wedge} (\mathsf{let}\ x = e), t) \\ \mathcal{U}(\Theta,\ \dashv K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x \mathrel{—} t\,) & = & (\Theta \mathbin{\dot\wedge} (x \not\approx K)) \\ & & \quad \cup\, \mathcal{U}(\Theta \mathbin{\dot\wedge} (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x), t) \end{array}$$

$$\boxed{\mathcal{A}(\Theta, t) = u}$$

$$\begin{array}{lcl} \mathcal{A}(\Theta,\ \longrightarrow n\,) & = & \longrightarrow \Theta\, n \\ \mathcal{A}(\Theta,\ \top\!\!\sqsubset^{t_1}_{t_2}) & = & \top\!\!\sqsubset^{\mathcal{A}(\Theta, t_1)}_{\mathcal{A}(\mathcal{U}(\Theta, t_1), t_2)} \\ \mathcal{A}(\Theta,\ \dashv !x \mathrel{—} t\,) & = & \mathrel{—} \Theta \mathbin{\dot\wedge} (x \approx \bot) \lightning \mathrel{—} \mathcal{A}(\Theta \mathbin{\dot\wedge} (x \not\approx \bot), t) \\ \mathcal{A}(\Theta,\ \dashv \mathsf{let}\ x = e \mathrel{—} t\,) & = & \mathcal{A}(\Theta \mathbin{\dot\wedge} (\mathsf{let}\ x = e), t) \\ \mathcal{A}(\Theta,\ \dashv K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x \mathrel{—} t\,) & = & \mathcal{A}(\Theta \mathbin{\dot\wedge} (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x), t) \end{array}$$

Fig. 3.5: Coverage checking

is defined by a simple recursive descent over the guard tree, using the operation $\Theta \mathbin{\dot\wedge} \varphi$ (also defined in Figure 3.5) to extend Θ with an extra literal φ.

While $\mathcal{U}$ finds a refinement type describing values that are *not* matched by a guard tree (its set of $\mathcal{U}$ncovered values), the function $\mathcal{A}$ finds refinements describing values that *are* matched by a guard tree, or that cause matching to diverge. It does so by producing an *annotated tree* (hence $\mathcal{A}$nnotate), whose syntax is given in Figure 3.3. An annotated tree has the same general structure as the guard tree from whence it came: in particular the top-to-bottom compositions $\top\!\!\sqsubset$ are in the same places. But in an annotated tree, each $\longrightarrow \Theta\, k$ leaf is annotated with a refinement type Θ describing the input values that will lead to that right-hand side; and each $\mathrel{—} \Theta \lightning \mathrel{—}$ node is annotated with a refinement

Collect accessible ($\overline{k}$), inaccessible ($\overline{n}$) and $\mathcal{R}$edundant ($\overline{m}$) GRHSs

$$\boxed{\mathcal{R}(u) = (\overline{k}, \overline{n}, \overline{m})}$$

$$\mathcal{R}(\ \longrightarrow \Theta\, n\ ) = \begin{cases} (\epsilon, \epsilon, n), & \text{if } \mathcal{G}(\Theta) = \emptyset \\ (n, \epsilon, \epsilon), & \text{otherwise} \end{cases}$$

$$\mathcal{R}(\ \top\!\!\!\bot \begin{matrix} t \\ u \end{matrix}\ ) = (\overline{k}\,\overline{k'}, \overline{n}\,\overline{n'}, \overline{m}\,\overline{m'}) \text{ where } \begin{matrix} (\overline{k}, \overline{n}, \overline{m}) = \mathcal{R}(t) \\ (\overline{k'}, \overline{n'}, \overline{m'}) = \mathcal{R}(u) \end{matrix}$$

$$\mathcal{R}(\ \text{—}\Theta\lightning\text{—}\ t\ ) = \begin{cases} (\epsilon, m, \overline{m'}), & \text{if } \mathcal{G}(\Theta) \neq \emptyset \text{ and } \mathcal{R}(t) = (\epsilon, \epsilon, m\,\overline{m'}) \\ \mathcal{R}(t), & \text{otherwise} \end{cases}$$

Fig. 3.6: Collecting accessible, inaccessible and redundant GRHSs

type that describes the input values on which matching will diverge. Once again, $\mathcal{A}$ can be defined by a simple recursive descent over the guard tree (Figure 3.5), but note that the second equation uses $\mathcal{U}$ as an auxiliary function[2].

3.2.3 Reporting Errors

The final step in Figure 3.1 is to report errors. First, let us focus on reporting missing equations. Consider the following definition

```
data T = A | B | C
f (Just A) = True
```

If t is the guard tree obtained from f, the expression $\mathcal{U}(\langle\, x : \mathsf{Maybe\ T} \mid \checkmark \,\rangle, t)$ will produce this refinement type describing values that are not matched:

$$\Theta_f = \langle\, x{:}\mathsf{Maybe\ T} \mid x \not\approx \bot \wedge (x \not\approx \mathsf{Just} \vee (\mathsf{Just}\ y \leftarrow x \\ \wedge\ y \not\approx \bot \wedge (y \not\approx \mathsf{A} \vee (\mathsf{A} \leftarrow y \wedge \times)))) \,\rangle$$

This is not very helpful to report to the user. It would be far preferable to produce one or more concrete *inhabitants* of Θ_f to report, something like this:

[2] The implementation avoids this duplicated work – see Section 3.4.2 – but the formulation in Figure 3.5 emphasises clarity over efficiency.

```
Missing equations for function 'f':
  f Nothing  = ...
  f (Just B) = ...
  f (Just C) = ...
```

$\mathcal{G}$enerating these inhabitants is the main technical challenge in this work. It is done by $\mathcal{G}(\Theta)$ in Figure 3.7, which I discuss next in Section 3.2.4. But first notice that, by calling the very same function $\mathcal{G}$, we can readily define the function $\mathcal{R}$, which reports a triple of (accessible, inaccessible, $\mathcal{R}$edundant) GRHSs, as needed in the overall pipeline (Figure 3.1). $\mathcal{R}$ is defined in Figure 3.6:

- Having reached a leaf $\longrightarrow \Theta\, k$, if the refinement type Θ is uninhabited ($\mathcal{G}(\Theta) = \emptyset$), then no input values can cause execution to reach the right-hand side k, and it is redundant.
- Having reached a node $\mathrel{---} \Theta \lightning \mathrel{---} t$, if Θ is inhabited there is a possibility of divergence. Now suppose that all the GRHSs in t are redundant. Then we should pick the first of them and mark it as inaccessible.
- The case for $\top\!\!\sqcup \begin{smallmatrix} t \\ u \end{smallmatrix}$ follows by congruence: just combine the classifications of t and u.

To illustrate the second case, consider u' from page 18 and its annotated tree:

$$\begin{array}{lr} u'\ ()\ \mid \mathsf{False} = 1 \\ \qquad\ \mid \mathsf{False} = 2 \\ u'\ \_\qquad\quad\ = 3 \end{array} \quad\rightsquigarrow\quad \begin{array}{l} \top \\ \quad\ \Theta_1 \lightning \top \to \Theta_2\ 1 \\ \qquad\quad\ \ \sqcup \to \Theta_3\ 2 \\ \quad\ \Theta_1 \lightning \sqcup \to \Theta_4\ 3 \end{array}$$

Refinement types Θ_2 and Θ_3 are uninhabited (because of the False guards), but Θ_1 is inhabited by $\bot$. Hence we cannot delete both GRHSs as redundant, because that would make the call $u'\ \bot$ return 3 rather than diverging. Rather, we want to report the first GRHS as inaccessible, leaving the second as redundant.

3.2.4 Generating Inhabitants of a Refinement Type

Thus far, all functions have been very simple, syntax-directed transformations, but they all ultimately depend on the single function $\mathcal{G}$, which does the real work. That is our new focus. As Figure 3.7 shows, $\mathcal{G}(\Theta)$ takes a refinement type $\Theta = \langle\, \Gamma \mid \Phi \,\rangle$ and returns a (possibly-empty) set of patterns $\overline{p}$ (syntax in Figure 3.3) that give the shape of values that inhabit Θ. This is done in two steps:

Normalised refinement type syntax

$$
\begin{array}{rcll}
\nabla & ::= & \times \mid \langle \Gamma \parallel \Delta \rangle & \text{Normalised refinement type} \\
\Delta & ::= & \varnothing \mid \Delta, \delta & \text{Set of constraints} \\
\delta & ::= & \gamma \mid x \approx K\ \overline{a}\ \overline{y} \mid x \not\approx K & \text{Constraints} \\
 & \mid & x \approx \bot \mid x \not\approx \bot \mid x \approx y &
\end{array}
$$

$\mathcal{G}$enerate inhabitants of Θ $\boxed{\mathcal{G}(\Theta) = \wp(\overline{p})}$

$$\mathcal{G}(\langle\, \Gamma \mid \Phi \,\rangle) = \{\mathcal{E}(\nabla, \mathrm{dom}(\Gamma)) \mid \nabla \in \mathcal{N}(\langle \Gamma \parallel \varnothing \rangle, \Phi)\}$$

$\mathcal{N}$ormalise Φ into ∇s $\boxed{\mathcal{N}(\nabla, \Phi) = \wp(\nabla)}$

$$
\begin{array}{rcl}
\mathcal{N}(\nabla, \varphi) & = & \begin{cases} \{\langle \Gamma' \parallel \Delta' \rangle\} & \text{where } \langle \Gamma' \parallel \Delta' \rangle = \nabla \oplus_\varphi \varphi \\ \emptyset & \text{otherwise} \end{cases} \\
\mathcal{N}(\nabla, \Phi_1 \wedge \Phi_2) & = & \bigcup \{\mathcal{N}(\nabla', \Phi_2) \mid \nabla' \in \mathcal{N}(\nabla, \Phi_1)\} \\
\mathcal{N}(\nabla, \Phi_1 \vee \Phi_2) & = & \mathcal{N}(\nabla, \Phi_1) \cup \mathcal{N}(\nabla, \Phi_2)
\end{array}
$$

$\mathcal{E}$xpand variables to Pat with ∇ $\boxed{\mathcal{E}(\nabla, x) = p, \quad \mathcal{E}(\nabla, \overline{x}) = \overline{p}}$

$$
\begin{array}{rcl}
\mathcal{E}(\nabla, \overline{x}) & = & \overline{\mathcal{E}(\nabla, x)} \\
\mathcal{E}(\langle \Gamma \parallel \Delta \rangle, x) & = & \begin{cases} K\ \mathcal{E}(\langle \Gamma \parallel \Delta \rangle, \overline{y}) & \text{if } \Delta(x) \approx K\ \overline{a}\ \overline{y} \in \Delta \\ \_ & \text{otherwise} \end{cases}
\end{array}
$$

Finding the representative of a variable in Δ $\boxed{\Delta(x) = y}$

$$\Delta(x) = \begin{cases} \Delta(y) & x \approx y \in \Delta \\ x & \text{otherwise} \end{cases}$$

Fig. 3.7: Generating inhabitants of Θ via ∇

- Flatten Θ into a disjunctive union of ***normalised refinement types*** ∇, by the call $\mathcal{N}(\langle \Gamma \parallel \varnothing \rangle, \Phi)$; see Section 3.2.6.
- For each such ∇, expand Γ into a list of patterns, by the call $\mathcal{E}(\nabla, \mathrm{dom}(\Gamma))$; see Section 3.2.5.

A normalised refinement type ∇ is either empty ($\times$) or of the form $\langle \Gamma \parallel \Delta \rangle$. It is similar to a refinement type $\Theta = \langle\, \Gamma \mid \Phi \,\rangle$, but it takes a much more restricted form (Figure 3.7): Δ is simply a conjunction of literals δ; there are no disjunctions as in φ. Instead, disjunction reflects in the fact that $\mathcal{N}$ returns a *set* of normalised refinement types.

Beyond these syntactic differences, I enforce the following invariants on a $\nabla = \langle \Gamma \parallel \Delta \rangle$:

I1 *Mutual compatibility*: No two constraints in Δ should *conflict* with each other, where $x \approx \bot$ conflicts with $x \not\approx \bot$, and $x \approx K \_ \_$ conflicts with $x \not\approx K$, for all x.

I2 *Inhabitation*: If $x{:}\tau \in \Gamma$ and τ reduces to a data type under type constraints in Δ, there must be at least one constructor K (or $\bot$) which x can be instantiated to without contradicting I1; see Section 3.2.7.

I3 *Triangular form*: A $x \approx y$ constraint implies absence of any other constraint mentioning x in its left-hand side.

I4 *Single solution*: There is at most one positive constructor constraint $x \approx K\ \overline{a}\ \overline{y}$ for a given x.

Invariants I1 and I2 prevent Δ from being self-contradictory, so that the resulting ∇ (which denotes a set of values) is always inhabited. I use $\times$ to represent the canonical uninhabited refinement type. Invariants I3 and I4 require Δ to be in solved form, from which it is easy to "read off" a value that inhabits it — this reading-off step is performed by $\mathcal{E}$ (Section 3.2.5).

The setup here is directly analogous to the setup of standard unification algorithms. In unification, we start with a set of equalities between types (analogous to Θ) and, by unification, normalise it to a substitution (analogous to ∇). That substitution can itself be regarded as a set of equalities, but in a restricted form. And indeed the normalisation algorithm (described in Section 3.2.6) is a form of generalised unification.

Notice that I allow Δ to contain variable/variable equalities $x \approx y$, providing a function $\Delta(x)$ (defined in Figure 3.7) that follows these indirections to find the

Add a formula literal to ∇ $\boxed{\nabla \oplus_\varphi \varphi = \nabla}$

$$
\begin{array}{rcll}
\nabla \oplus_\varphi \times & = & \times & (1)\\
\nabla \oplus_\varphi \checkmark & = & \nabla & (2)\\
\langle \Gamma \parallel \Delta \rangle \oplus_\varphi K\ \overline{a}\ \overline{\gamma}\ \overline{y{:}\tau} \leftarrow x & = & \langle \Gamma, \overline{a}, \overline{y{:}\tau} \parallel \Delta \rangle \oplus_\delta \overline{\gamma} \oplus_\delta \overline{y' \not\approx \bot} & (3)\\
 & & \quad \oplus_\delta x \approx K\ \overline{a}\ \overline{y} & \\
 & & \quad \text{where } \{\overline{y'}\} \subseteq \{\overline{y}\} \text{ bind strict fields} & \\
\langle \Gamma \parallel \Delta \rangle \oplus_\varphi \text{let } x{:}\tau = K\ \overline{\sigma}\ \overline{\gamma}\ \overline{e} & = & \langle \Gamma, x{:}\tau, \overline{a} \parallel \Delta \rangle \oplus_\delta \overline{a \sim \sigma} \oplus_\delta x \not\approx \bot & (4)\\
 & & \quad \oplus_\delta x \approx K\ \overline{a}\ \overline{y} \oplus_\varphi \overline{\text{let } y{:}\tau' = e} & \\
 & & \quad \text{where } \overline{a}\,\overline{y} \text{ fresh, } \overline{e{:}\tau'} & \\
\langle \Gamma \parallel \Delta \rangle \oplus_\varphi \text{let } x{:}\tau = y & = & \langle \Gamma, x{:}\tau \parallel \Delta \rangle \oplus_\delta x \approx y & (5)\\
\langle \Gamma \parallel \Delta \rangle \oplus_\varphi \text{let } x{:}\tau = e & = & \langle \Gamma, x{:}\tau \parallel \Delta \rangle & (6)\\
\langle \Gamma \parallel \Delta \rangle \oplus_\varphi \varphi & = & \langle \Gamma \parallel \Delta \rangle \oplus_\delta \varphi & (7)
\end{array}
$$

Fig. 3.8: Adding a formula literal to the normalised refinement type ∇

"representative" of x in Δ. A perfectly viable alternative would be to omit such indirections from Δ and instead aggressively substitute them away.

3.2.5 Expanding a Normalised Refinement Type to a Pattern

$\mathcal{E}$xpanding a match variable x under ∇ to a pattern, by calling $\mathcal{E}$ in Figure 3.7, is straightforward and overloaded to operate similarly on multiple match variables. When there is a solution like $\Delta(x) \approx \text{Just } y$ in Δ for the match variable x of interest, recursively expand y and wrap it in a Just. Invariant I4 guarantees that there is at most one such solution and $\mathcal{E}$ is well-defined. When there is no solution for x, return _. See Section 3.4.5 for how I improve on that in the implementation by taking negative information into account.

3.2.6 Normalising a Refinement Type

$\mathcal{N}$ormalisation, carried out by $\mathcal{N}$ in Figure 3.7, is largely a matter of repeatedly adding a literal φ to a normalised type, thus $\nabla \oplus_\varphi \varphi$. This function is where all the work is done, in Figures 3.8 and 3.9. It does so by expressing a literal φ in terms of simpler constraints δ, and calling out to $\oplus_\delta$ to add the simpler constraints to ∇ (Figure 3.9). $\mathcal{N}$, $\oplus_\varphi$ and $\oplus_\delta$ all work on the principle that if

Add a constraint to ∇ $\boxed{\nabla \oplus_\delta \delta = \nabla}$

$$\times \oplus_\delta \delta = \times \tag{8}$$

$$\langle \Gamma \parallel \Delta \rangle \oplus_\delta \gamma = \begin{cases} \langle \Gamma \parallel (\Delta, \gamma) \rangle & \text{if type checker deems } \gamma \text{ compatible with } \Delta \text{ and } \forall x \in \mathsf{dom}(\Gamma) : \langle \Gamma \parallel (\Delta, \gamma) \rangle \vdash x \,\mathsf{inh} \\ \times & \text{otherwise} \end{cases} \tag{9}$$

$$\langle \Gamma \parallel \Delta \rangle \oplus_\delta x \approx K\ \overline{a}\ \overline{y} = \begin{cases} \langle \Gamma \parallel \Delta \rangle \oplus_\delta \overline{a \sim b} \oplus_\delta \overline{y \approx z} & \text{if } \Delta(x) \approx K\ \overline{b}\ \overline{z} \in \Delta \\ \times & \text{if } \Delta(x) \approx K'\ \overline{b}\ \overline{z} \in \Delta \text{ and } K \neq K' \\ \times & \text{if } \Delta(x) \not\approx K \in \Delta \\ \langle \Gamma \parallel (\Delta, \Delta(x) \approx K\ \overline{a}\ \overline{y}) \rangle & \text{otherwise} \end{cases} \tag{10}$$

$$\langle \Gamma \parallel \Delta \rangle \oplus_\delta x \not\approx K = \begin{cases} \times & \text{if } \Delta(x) \approx K\ \overline{a}\ \overline{y} \in \Delta \\ \langle \Gamma \parallel (\Delta, \Delta(x) \not\approx K) \rangle & \text{if } \langle \Gamma \parallel (\Delta, \Delta(x) \not\approx K) \rangle \vdash x \,\mathsf{inh} \\ \times & \text{otherwise} \end{cases} \tag{11}$$

$$\langle \Gamma \parallel \Delta \rangle \oplus_\delta x \approx \bot = \begin{cases} \times & \text{if } \Delta(x) \not\approx \bot \in \Delta \\ \langle \Gamma \parallel (\Delta, \Delta(x) \approx \bot) \rangle & \text{otherwise} \end{cases} \tag{12}$$

$$\langle \Gamma \parallel \Delta \rangle \oplus_\delta x \not\approx \bot = \begin{cases} \times & \text{if } \Delta(x) \approx \bot \in \Delta \\ \langle \Gamma \parallel (\Delta, \Delta(x) \not\approx \bot) \rangle & \text{if } \langle \Gamma \parallel (\Delta, \Delta(x) \not\approx \bot) \rangle \vdash x \,\mathsf{inh} \\ \times & \text{otherwise} \end{cases} \tag{13}$$

$$\langle \Gamma \parallel \Delta \rangle \oplus_\delta x \approx y = \begin{cases} \langle \Gamma \parallel \Delta \rangle & \text{if } x' = y' \\ \langle \Gamma \parallel ((\Delta \setminus x'), x' \approx y') \rangle \oplus_\delta (\Delta|_{x'}\ [y'/x']) & \text{otherwise} \end{cases} \tag{14}$$

where $x' = \Delta(x)$ and $y' = \Delta(y)$

$\boxed{\Delta \setminus x = \Delta}$

$$\begin{aligned} \varnothing \setminus x &= \varnothing \\ (\Delta, x \approx K\ \overline{a}\ \overline{y}) \setminus x &= \Delta \setminus x \\ (\Delta, x \not\approx K) \setminus x &= \Delta \setminus x \\ (\Delta, x \approx \bot) \setminus x &= \Delta \setminus x \\ (\Delta, x \not\approx \bot) \setminus x &= \Delta \setminus x \\ (\Delta, \delta) \setminus x &= (\Delta \setminus x), \delta \end{aligned}$$

$\boxed{\Delta|_x = \Delta}$

$$\begin{aligned} \varnothing|_x &= \varnothing \\ (\Delta, x \approx K\ \overline{a}\ \overline{y})|_x &= \Delta|_x,\ x \approx K\ \overline{a}\ \overline{y} \\ (\Delta, x \not\approx K)|_x &= \Delta|_x,\ x \not\approx K \\ (\Delta, x \approx \bot)|_x &= \Delta|_x,\ x \approx \bot \\ (\Delta, x \not\approx \bot)|_x &= \Delta|_x,\ x \not\approx \bot \\ (\Delta, \delta)|_x &= \Delta|_x \end{aligned}$$

Fig. 3.9: Adding a constraint to the normalised refinement type ∇

the incoming ∇ satisfies the Invariants I1 to I4 from Section 3.2.4, then either the resulting ∇ is $\times$ or it satisfies I1 to I4.

In Equation (3), a pattern guard extends the context and adds suitable type constraints and a positive constructor constraint arising from the binding. Equation (4) of $\oplus_\varphi$ performs some limited, but important reasoning about let bindings: it flattens possibly nested constructor applications, such as let $x = \mathsf{Just}\ \mathsf{True}$, and asserts that such constructor applications cannot be $\bot$. Note that Equation (6) simply discards let bindings that cannot be expressed in ∇; we will see an extension in Section 3.3.3 that avoids this information loss.

That brings us to the prime unification procedure, $\oplus_\delta$. When adding $x \approx \mathsf{Just}\ y$, Equation (10), the unification procedure will first look for a solution for x with *that same constructor*. Let's say there is $\Delta(x) \approx \mathsf{Just}\ u \in \Delta$. Then $\oplus_\delta$ operates on the transitively implied equality $\mathsf{Just}\ y \approx \mathsf{Just}\ u$ by equating type and term variables with new constraints, i.e. $y \approx u$. The original constraint, although not conflicting, is not added to the normalised refinement type because of I3.

If there is a solution involving a different constructor like $\Delta(x) \approx \mathsf{Nothing}$ or if there was a negative constructor constraint $\Delta(x) \not\approx \mathsf{Just}$, the new constraint is incompatible with the existing solution. Otherwise, the constraint is compatible and is added to Δ.

Adding a negative constructor constraint $x \not\approx \mathsf{Just}$ is quite similar (Equation (11)), except that we have to make sure that x still satisfies I2, which is checked by the $\nabla \vdash \Delta(x)\ \mathsf{inh}$ judgment (cf. Section 3.2.7) in Figure 3.10. Handling positive and negative constraints involving $\bot$ is analogous.

Adding a type constraint γ (Equation (9)) entails calling out to the type checker to assert that the constraint is consistent with existing type constraints. Afterwards, we have to ensure I2 is upheld for *all* variables in the domain of Γ, because the new type constraint could have rendered a type empty. To demonstrate why this is necessary, imagine we have $\langle x : a \parallel x \not\approx \bot \rangle$ and try to add $a \sim \mathsf{Void}$. Although the type constraint is consistent, x in $\langle x : a \parallel x \not\approx \bot, a \sim \mathsf{Void} \rangle$ is no longer inhabited. There is room for being smart about which variables we have to re-check: For example, we can exclude variables whose type is a non-GADT data type.

Equation (14) of $\oplus_\delta$ equates two variables ($x \approx y$) by merging their equivalence classes. Consider the case where x and y are not in the same equivalence class. Then $\Delta(y)$ is arbitrarily chosen to be the new representative of the merged equivalence class. To uphold I3, all constraints mentioning $\Delta(x)$ have to be removed and renamed in terms of $\Delta(y)$ and then re-added to Δ, one of which in turn might uncover a contradiction.

Test if x is inhabited considering ∇ $\boxed{\nabla \vdash x\,\mathsf{inh}}$

$$\frac{(\langle \Gamma \parallel \Delta \rangle \oplus_\delta x \approx \bot) \neq \times}{\langle \Gamma \parallel \Delta \rangle \vdash x\,\mathsf{inh}} \vdash\textsc{Bot} \qquad \frac{x : \tau \in \Gamma \quad \mathsf{Cons}(\langle \Gamma \parallel \Delta \rangle, \tau) = \bot}{\langle \Gamma \parallel \Delta \rangle \vdash x\,\mathsf{inh}} \vdash\textsc{NoCpl}$$

$$\frac{\begin{array}{c} x : \tau \in \Gamma \quad K \in \mathsf{Cons}(\langle \Gamma \parallel \Delta \rangle, \tau) \\ \mathsf{Inst}(\langle \Gamma \parallel \Delta \rangle, x, K) \neq \times \end{array}}{\langle \Gamma \parallel \Delta \rangle \vdash x\,\mathsf{inh}} \vdash\textsc{Inst}$$

Find data constructors of τ $\boxed{\mathsf{Cons}(\langle \Gamma \parallel \Delta \rangle, \tau) = \overline{K}}$

$$\mathsf{Cons}(\langle \Gamma \parallel \Delta \rangle, \tau) = \begin{cases} \overline{K} & \tau = T\ \overline{\sigma} \text{ and } T \text{ data type with constructors } \overline{K} \text{ (after} \\ & \text{normalisation according to the type constraints in } \Delta) \\ \bot & \text{otherwise} \end{cases}$$

Instantiate x to data constructor K $\boxed{\mathsf{Inst}(\nabla, x, K) = \nabla}$

$$\mathsf{Inst}(\langle \Gamma \parallel \Delta \rangle, x, K) = \langle \Gamma, \overline{a}, \overline{y : \sigma} \parallel \Delta \rangle \oplus_\delta \tau_x \sim \tau \oplus_\delta \overline{\gamma} \oplus_\delta x \approx K\ \overline{a}\ \overline{y} \oplus_\delta \overline{y' \not\approx \bot}$$
$$\text{where } K : \forall \overline{a}.\ \overline{\gamma} \Rightarrow \overline{\sigma} \to \tau,\ \overline{a}\,\overline{y} \text{ fresh},\ x : \tau_x \in \Gamma,$$
$$\{\overline{y'}\} \subseteq \{\overline{y}\} \text{ bind strict fields}$$

Fig. 3.10: Testing for inhabitation

3.2.7 Testing for Inhabitation

The process for adding a constraint to a normalised type above (which turned out to be a unification procedure in disguise) makes use of an *inhabitation test* $\nabla \vdash x\,\mathsf{inh}$, depicted in Figure 3.10. This tests whether there are any values of x that satisfy ∇. If not, ∇ does not uphold I2. For example, the conjunction $x \not\approx \mathsf{Just}, x \not\approx \mathsf{Nothing}, x \not\approx \bot$ does not satisfy I2, because no value of x satisfies all those constraints.

The $\vdash$Bot judgment of $\nabla \vdash x\,\mathsf{inh}$ tries to instantiate x to $\bot$ to conclude that x is inhabited. $\vdash$Inst instantiates x to one of its data constructors. That will only work if its type ultimately reduces to a data type under the type constraints in ∇.

Rule ⊢NoCpl will accept unconditionally when its type is not a data type, i.e. for $x : \mathsf{Int} \to \mathsf{Int}$.

Note that the outlined approach is complete in the sense that $\nabla \vdash x\,\mathsf{inh}$ is derivable if and only if x is actually inhabited in ∇, because that means we do not have any ∇s floating around in the checking process that actually are not inhabited and trigger false positive warnings. But that also means that the $\vdash\ \mathsf{inh}$ relation is undecidable! Consider the following example:

```
data T = MkT !T
f :: SMaybe T → ()
f SNothing = ()
```

This is exhaustive, because T is an uninhabited type. Upon adding the constraint $x \not\approx \mathsf{SNothing}$ on the match variable x via $\oplus_\delta$, we perform an inhabitation test, which tries to instantiate the SJust constructor via ⊢Inst. That implies adding (via $\oplus_\delta$) the constraints $x \approx \mathsf{SJust}\ y, y \not\approx \bot$, the latter of which leads to an inhabitation test on y. That leads to instantiation of the MkT constructor, which leads to constraints $y \approx \mathsf{MkT}\ z, z \not\approx \bot$, and so on for z etc.. An infinite chain of fruitless instantiation attempts!

This situation is a lot like deciding equality of equirecursive types [Pierce 2002, Chapter 21], in that we could break out of the infinite, coinductive proof chain by assuming that T is uninhabited for any recursive occurrences of T beyond the first.

Unfortunately, GADTs might still recurse endlessly through the type index. So in practice, the implementation adopts a fuel-based approach that conservatively assumes that a variable is inhabited after n such instantiations (we have $n = 100$ for list-like constructors and $n = 1$ otherwise) and we consider supplementing that with a simple termination analysis to detect simple uninhabited data types like T in the future.

3.3 Extensions

LYG is well equipped to handle the fragment of Haskell it was designed to handle. But GHC extends Haskell in non-trivial ways. This section exemplifies easy accommodation of new language features and measures to increase precision of the checking process, demonstrating the modularity and extensibility of the approach.

3.3.1 Long-Distance Information

Coverage checking should also work for **case** expressions and nested function definitions, like

$$\begin{array}{ll} f\ \mathsf{True} = 1 & \\ f\ x & = \ldots\ (\textbf{case}\ x\ \textbf{of}\ \{\mathsf{False} \to 2; \mathsf{True} \to 3\})\ \ldots \end{array}$$

GMTM and unextended LYG will not produce any warnings for this definition. But the reader can easily make the "long distance connection" that the last GRHS of the **case** expression is redundant! That follows by context-sensitive reasoning, knowing that x was already matched against True.

In terms of LYG, the input values of the second GRHS of f, described by $\Theta_2 = \langle\, x : \mathsf{Bool} \mid x \not\approx \bot, x \not\approx \mathsf{True} \,\rangle$, encode the information we are after: we just have to start checking the case expression starting from Θ_2 as the initial set of reaching values instead of $\langle\, x : \mathsf{Bool} \mid \checkmark \,\rangle$. We already need Θ_2 to determine whether the second GRHS of f is accessible, so long-distance information comes almost for free.

3.3.2 Empty Case

As can be seen in Figure 3.2, Haskell function definitions need to have at least one clause. That leads to an awkward situation when pattern matching on empty data types, like Void:

$$\begin{array}{ll} \mathit{absurd1}\ \_ = \mathit{undefined} & \mathit{absurd1}, \mathit{absurd2}, \mathit{absurd3} :: \mathsf{Void} \to a \\ \mathit{absurd2}\ !\_ = \mathit{undefined} & \mathit{absurd3}\ x = \textbf{case}\ x\ \textbf{of}\ \{\,\} \end{array}$$

absurd1 returns *undefined* when called with $\bot$, thus masking the original $\bot$ with the error thrown by *undefined*. *absurd2* would diverge alright, but LYG will report its GRHS as inaccessible! Hence GHC provides an extension, called `EmptyCase`, that allows the definition of *absurd3* above. Such a **case** expression without any alternatives evaluates its argument to WHNF and crashes when evaluation returns.

It is quite easy to see that Gdt lacks expressive power to desugar `EmptyCase` into, since all leaves in a guard tree need to have corresponding GRHSs. Therefore, we need to introduce empty alternatives $\bullet_{\mathsf{Gdt}}$ to Gdt and $\bullet_{\mathsf{Ant}}$ to Ant. This is how they affect the checking process:

$$\mathcal{U}(\Theta, \bullet_{\mathsf{Gdt}}) = \Theta \qquad \mathcal{A}(\Theta, \bullet_{\mathsf{Gdt}}) = \bullet_{\mathsf{Ant}}$$

Since `EmptyCase`, unlike regular **case**, evaluates its scrutinee to WHNF *before* matching any of the patterns, the set of reaching values is refined with a $x \not\approx \bot$ constraint *before* traversing the guard tree, thus checking starts starts with $\mathcal{U}(\langle\, \Gamma \mid x \not\approx \bot \,\rangle, \bullet_{\mathrm{Gdt}})$.

3.3.3 View Patterns

The source syntax had support for view patterns to start with (cf. Figure 3.2). And even the desugaring I gave as part of the definition of $\mathcal{D}$ in Figure 3.4 is accurate. But this desugaring alone is insufficient for the checker to conclude that *safeLast* from Section 3.1.2 is an exhaustive definition! To see why, let us look at its guard tree:

$$\begin{array}{l} \text{let } y_1 = \mathit{reverse}\ x_1, !y_1, \mathsf{Nothing} \leftarrow y_1 \longrightarrow 1 \\ \text{let } y_2 = \mathit{reverse}\ x_1, !y_2, \mathsf{Just}\ t_1 \leftarrow y_2, !t_1, (t_2, t_3) \leftarrow t_1 \longrightarrow 2 \end{array}$$

As far as LYG is concerned, the matches on both y_1 and y_2 are non-exhaustive. But that's actually too conservative: Both bind the same value! By making the connection between y_1 and y_2, the checker could infer that the match was exhaustive.

This can be fixed by maintaining equivalence classes of semantically equivalent expressions in Δ, similar to what we do for variables. We simply extend the syntax of δ and change the last **let** case of $\oplus_\varphi$. Then we can handle the new constraint in $\oplus_\delta$, as follows:

$$\delta = \ldots \mid e \approx x \qquad \langle \Gamma \,\|\, \Delta \rangle \ \oplus_\varphi\ \text{let } x : \tau = e = \langle \Gamma, x : \tau \,\|\, \Delta \rangle \ \oplus_\delta\ e \approx x$$

$$\langle \Gamma \,\|\, \Delta \rangle \ \oplus_\delta\ e \approx x = \begin{cases} \langle \Gamma \,\|\, \Delta \rangle \ \oplus_\delta\ x \approx y, & \text{if } e' \approx y \in \Delta \text{ and } e \equiv_\Delta e' \\ \langle \Gamma \,\|\, \Delta, e \approx \Delta(x) \rangle, & \text{otherwise} \end{cases}$$

Where $\equiv_\Delta$ is (an approximation to) semantic equivalence modulo substitution under Δ. A clever data structure is needed to answer queries of the form $e \approx \_ \in \Delta$, efficiently. In the implementation, I use a trie to index expressions rapidly [Peyton Jones and Graf 2023] and sacrifice reasoning modulo Δ in doing so. Plugging in an SMT solver to decide $\equiv_\Delta$ would be more precise, but certainly less efficient.

3.3.4 Pattern Synonyms

To accommodate checking of pattern synonyms P, we first have to extend the source syntax and IR syntax by adding the syntactic concept of a *ConLike*:

$$\begin{array}{rcl} cl & ::= & K \mid P \\ pat & ::= & x \mid \_ \mid \boxed{cl}\ \overline{pat} \mid x@pat \mid \ldots \end{array} \qquad \begin{array}{rcll} P \in & \mathsf{PS} & & \\ C \in & \mathsf{CL} & ::= & K \mid P \\ p \in & \mathsf{Pat} & ::= & \_ \mid \boxed{C}\ \overline{p} \mid \ldots \end{array}$$

Assuming every definition encountered so far is changed to handle ConLikes C instead of data constructors K, everything should work fine. So why introduce the new syntactic variant in the first place? Consider

```
pattern P = ()
pattern Q = ()
n = case P of Q → 1; P → 2
```

If P and Q were data constructors, the first alternative of the **case** would be redundant, because P cannot match Q. But pattern synonyms are quite different: a value produced by P might match a pattern Q, as indeed is the case in this example.

My solution is a conservative one: I weaken the test that sends ∇ to $\times$ of Equation (10) in the definition of $\oplus_\delta$ dealing with positive ConLike constraints $x \approx C\ \overline{a}\ \overline{y}$:

$$\langle \Gamma \,\|\, \Delta \rangle \oplus_\delta x \approx C\ \overline{a}\ \overline{y} \quad = \quad \begin{cases} \langle \Gamma \,\|\, \Delta \rangle \oplus_\delta \overline{a \sim b} \oplus_\delta \overline{y \approx z} & \text{if } \Delta(x) \approx C\ \overline{b}\ \overline{z} \in \Delta \\ \times & \text{if } \Delta(x) \approx C'\ \overline{b}\ \overline{z} \in \Delta \\ & \text{and } \boxed{C \cap C' = \emptyset} \\ \times & \text{if } \Delta(x) \not\approx C \in \Delta \\ \langle \Gamma \,\|\, (\Delta, \Delta(x) \approx C\ \overline{a}\ \overline{y}) \rangle & \text{otherwise} \end{cases}$$

where the suggestive notation $C \cap C' = \emptyset$ is only true iff C and C' are distinct data constructors.

Note that the slight relaxation means that the constructed ∇ might violate $I4$, specifically when $C \cap C' \neq \emptyset$. In practice that condition only matters for the well-definedness of $\mathcal{E}$, which in case of multiple solutions (i.e. $x \approx \mathsf{P}, x \approx \mathsf{Q}$) has to commit to one of them for the purposes of reporting warnings. Fixing that requires a bit of boring engineering.

Another subtle point appears in rule (†) in Figure 3.4: should I or should I not add a bang guard for pattern synonyms? There is no way to know without

breaking the abstraction offered by the synonym. In effect, its strictness or otherwise is part of its client-visible semantics. In the implementation, I have compromised by assuming that all pattern synonyms are strict for the purposes of coverage checking [GHC issue 2019m].

3.3.5 `COMPLETE` Pragmas

In a sense, every algebraic data type defines its own builtin `COMPLETE` set, consisting of all its data constructors, so the coverage checker already manages a single `COMPLETE` set.

Judgment form ⊢Inst from Figure 3.10 currently makes sure that this `COMPLETE` set is in fact inhabited. Furthermore, ⊢NoCpl handles the case when *no* `COMPLETE` set for the given type (think $x :: \mathsf{Int} \to \mathsf{Int}$) can be found. The prudent way to generalise this is by looking up all `COMPLETE` sets attached to a type and check that none of them is completely covered:

$$\frac{(\langle \Gamma \parallel \Delta \rangle \ \oplus_\delta \ x \approx \bot) \neq \times}{\langle \Gamma \parallel \Delta \rangle \vdash x \,\mathsf{inh}} \vdash\text{Bot} \qquad \frac{\mathsf{Cons}(\langle \Gamma \parallel \Delta \rangle, \tau) = \overline{C_1, ..., C_n} \quad x : \tau \in \Gamma \quad \overline{\mathsf{Inst}(\langle \Gamma \parallel \Delta \rangle, x, C_j) \neq \times}}{\langle \Gamma \parallel \Delta \rangle \vdash x \,\mathsf{inh}} \vdash\text{Inst}$$

$$\mathsf{Cons}(\langle \Gamma \parallel \Delta \rangle, \tau) = \begin{cases} \overline{C_1, ..., C_n} & \tau = T\ \overline{\sigma};\ \ T \text{ type constructor with } \texttt{COMPLETE} \\ & \text{sets } \overline{C_1, ..., C_n} \text{ (after normalisation according} \\ & \text{to the type constraints in } \Delta) \\ \epsilon & \text{otherwise} \end{cases}$$

Cons was changed to return a list of all available `COMPLETE` sets, and ⊢Inst tries to find an inhabiting ConLike C_j in each one of them in turn. Note that ⊢NoCpl is gone, because it coincides with ⊢Inst for the case where the list returned by Cons was empty. The judgment has become simpler and and more general at the same time! A worry is that checking against multiple `COMPLETE` sets so frequently is computationally intractable. We will worry about that in Section 3.4.4.

3.3.6 Literals

The source syntax in Figure 3.11 deliberately left out literal patterns l. Literals are very similar to nullary data constructors, with one caveat: they do not come with a builtin `COMPLETE` set. Before Section 3.3.5, that would have meant quite a

bit of hand waving and complication to the $\vdash$ inh judgment. Now, literals can be handled like disjoint pattern synonyms (i.e. $l_1 \cap l_2 = \emptyset$ for any two literals l_1, l_2) without a `COMPLETE` set!

Overloaded literals can be supported as well, but we will find ourselves in a similar situation as with pattern synonyms:

```
instance Num () where
  fromInteger _ = ()
n = case (0 :: ()) of 1 → 1; 0 → 2  -- returns 1
```

Considering overloaded literals to be disjoint would mean marking the first alternative as redundant, which is unsound. Hence overloaded literals are regarded as possibly overlapping, so they behave exactly like nullary pattern synonyms without a `COMPLETE` set.

3.3.7 Newtypes

In Haskell, a newtype declares a new type that is completely isomorphic to, but definitionally distinct from, an existing type. For example:

```
newtype NT a = MkNT [a]
dup :: NT a → NT a
dup (MkNT xs) = MkNT (xs ++ xs)
```

Here the type NT a is isomorphic to $[a]$. We convert to and fro using the "data constructor" MkNT, either as in a term or in a pattern.

To a first approximation, programmers interact with a newtype as if it was a data type with a single constructor with a single field. But the pattern matching semantics of newtypes are different! Here are three key examples that distinguish newtypes from data types. Functions g_1, g_2 match on a *newtype* N, while

$$cl ::= K \mid N \qquad \begin{array}{ll} N \in & \mathsf{NT} \\ C \in & K \mid N \end{array}$$

$$\mathcal{D}(x, N\ pat_1 \ldots pat_n) = N\ y_1 \ldots y_n \leftarrow x, \mathcal{D}(y_1, pat_1), \ldots, \mathcal{D}(y_n, pat_n)$$

$$\Delta_{\text{NT}}(x) \quad = \quad \begin{cases} \Delta_{\text{NT}}(y) & x \approx y \in \Delta \text{ or } x \approx N\ \overline{a}\ y \in \Delta \\ x & \text{otherwise} \end{cases}$$

$$\langle \Gamma \,\|\, \Delta \rangle \ \oplus_\varphi \text{let } x{:}\tau = K\ \overline{\sigma}\ \overline{\gamma}\ \overline{e} = \ldots \text{as before} \ldots \tag{4a}$$

$$\begin{aligned} \langle \Gamma \,\|\, \Delta \rangle \ \oplus_\varphi \text{let } x{:}\tau = N\ \overline{\sigma}\ e \quad &= \langle \Gamma, x{:}\tau, \overline{a} \,\|\, \Delta \rangle \ \oplus_\delta\ \overline{a \sim \sigma}\ \oplus_\delta\ x \approx N\ \overline{a}\ y \\ &\quad \oplus_\varphi \text{let } y{:}\tau' = e \quad \text{where } \overline{a}\, y \text{ fresh} \end{aligned} \tag{4b}$$

$$\langle \Gamma \,\|\, \Delta \rangle \ \oplus_\delta x \approx K\ \overline{a}\ \overline{y} \quad = \ldots \text{as before} \ldots \tag{10a}$$

$$\langle \Gamma \,\|\, \Delta \rangle \ \oplus_\delta\ x \approx N\ \overline{a}\ y = \begin{cases} \langle \Gamma \,\|\, \Delta \rangle \ \oplus_\delta\ \overline{a \sim b}\ \oplus_\delta\ y \approx z & \text{if } x' \approx N\ \overline{b}\ z \in \Delta \\ \langle \Gamma \,\|\, \Delta \rangle & \text{if } x' = \Delta_{\text{NT}}(y') \\ \langle \Gamma \,\|\, ((\Delta \backslash x'), x' \approx N\ \overline{a}\ y') \rangle & \\ \quad \oplus_\delta\ (\Delta|_{x'}\ [y'/x']) & \text{otherwise} \end{cases} \tag{10b}$$

where $x' = \Delta(x)$ and $y' = \Delta(y)$

$$\langle \Gamma \,\|\, \Delta \rangle \ \oplus_\delta x \not\approx K \quad = \ldots \text{as before} \ldots \tag{11a}$$

$$\langle \Gamma \,\|\, \Delta \rangle \ \oplus_\delta x \not\approx N \quad = \times \tag{11b}$$

$$\langle \Gamma \,\|\, \Delta \rangle \ \oplus_\delta x \approx \bot \quad = \begin{cases} \times & \text{if } \Delta_{\text{NT}}(x) \not\approx \bot \in \Delta \\ \langle \Gamma \,\|\, (\Delta, \Delta_{\text{NT}}(x) \approx \bot) \rangle & \text{otherwise} \end{cases} \tag{12}$$

$$\langle \Gamma \,\|\, \Delta \rangle \ \oplus_\delta x \not\approx \bot \quad = \begin{cases} \times & \text{if } \Delta_{\text{NT}}(x) \approx \bot \in \Delta \\ \nabla' & \text{if } \nabla' \vdash x\ \mathsf{inh} \\ \times & \text{otherwise} \end{cases} \tag{13}$$

where $\nabla' = \langle \Gamma \,\|\, (\Delta, \Delta_{\text{NT}}(x) \not\approx \bot) \rangle$

Fig. 3.11: Extending coverage checking to handle newtypes

functions h_1, h_2 match on a *data type* D:

```
newtype N a = MkN a
g1 :: N Void → Bool → Int          g2 :: N () → Bool → Int
g1 _          True = 1             g2 !(MkN _) True = 1
g1 (MkN _) True = 2  -- Redundant  g2 (MkN !_) True = 2  -- Redundant
g1 !_         True = 3  -- Inaccessible  g2 _      _    = 3
data D a = MkD a
h1 :: D Void → Bool → Int          h2 :: D () → Bool → Int
h1 _          True = 1             h2 !(MkD _) True = 1
h1 (MkD _) True = 2  -- Inaccessible  h2 (MkD !_) True = 2  -- Inaccessible
h1 !_         True = 3  -- Redundant  h2 _      _    = 3
```

If the first equation of h_1 fails to match (because the second argument is False), the second equation may diverge when matching against (MkD _) or may fail (because of the False), so the equation is inaccessible. The third equation is redundant. But for a newtype, the second equation of g_1 will not evaluate the argument when matching against (MkN _) and hence is redundant. The third equation will evaluate the first argument, which is surely bottom, so matching will diverge and the equation is inaccessible. A perhaps surprising consequence is that the definition of g_1 is exhaustive, because after N Void was deprived of its sole inhabitant $\bot$ = MkN $\bot$ by the third GRHS, there is nothing left to match on (similarly for h_1). Analogous subtle reasoning justifies the difference in warnings for g_2 and h_2.

Figure 3.11 outlines a solution that handles all these cases correctly:

- A newtype pattern-match $N\ pat_1 \ldots pat_n$ is lazy: it does not force evaluation. So, compared to data constructor matches, the desugaring function $\mathcal{D}$ omits the $!x$. Additionally, Equation (4) of $\oplus_\varphi$, responsible for reasoning about **let** bindings, has a special case for newtypes that omits the $x \not\approx \bot$ constraint.
- Similar in spirit to $\Delta(x)$, which chases variable equality constraints $x \approx y$, we now also occasionally need to look through positive newtype constructor constraints $x \approx N\ \overline{a}\ y$ with $\Delta_{\text{NT}}(x)$.
- The most important usage of $\Delta_{\text{NT}}(x)$ is in the changed Equations (12) and (13) of $\oplus_\delta$, where we now check $\bot$ constraints modulo $\Delta_{\text{NT}}(x)$.
- Equation (10) (previously handling $x \approx K\ \overline{a}\ \overline{y}$) and Equation (11) (previously handling $x \not\approx K$) have been split to account for newtype constructors.

- The first case of the new Equation (10*b*) handles any existing positive newtype constructor constraints in Δ, as with Equation (10). Take note that negative newtype constructor constraints may never occur in Δ because of Equation (11*b*), as explained in the next paragraph. The remaining two cases are reminiscent of Equation (14) ($x \approx y$). Provided there are neither positive nor negative newtype constructor constraints involving x, any remaining $\bot$ constraints are moved from $\Delta(x)$ to the new representative $\Delta'_{\text{NT}}(x)$, which will be $\Delta'_{\text{NT}}(y)$ in the returned Δ'.
- The new Equation (11*b*) handles negative newtype constructor constraints by immediately rejecting. The reason it does not consider $\bot$ as an inhabitant is that for $\bot$ to be an inhabitant, it must be an inhabitant of the newtype's field. For that, we must have $x \approx K\ y$ for some y, which contradicts with the very constraint we want to add!

To see how these changes facilitate correct warnings for newtype matches, first consider the changed invariant I3 which ensures $\Delta_{\text{NT}}(x)$ is a well-defined function like $\Delta(x)$:

I3 *Triangular form*: Constraints of the form $x \approx y$ and $x \approx N\ \overline{a}\ y$ imply absence of any other constraint mentioning x in its left-hand side.

We want Δ to uphold the semantic equation $\bot \equiv N\bot$. In particular, whenever we have $x \approx N\ \overline{a}\ y$, we want $x \approx \bot$ iff $y \approx \bot$ (similarly for $x \not\approx \bot$). Equations (10b), (12) and (13) facilitate just that, modulo $\Delta_{\text{NT}}(x)$. Finally, a new invariant I5 relates positive newtype constructor equalities to $\bot$ constraints:

I5 *Newtype erasure*: Whenever $x \approx N\ \overline{a}\ y \in \Delta$, we have $x \approx \bot \in \Delta$ if and only if $y \approx \bot \in \Delta$, and $x \not\approx \bot \in \Delta$ if and only if $y \not\approx \bot \in \Delta$.

An alternative design might take inspiration in the coercion semantics of GHC Core, a typed intermediate language of GHC based on System F, and compose coercions attached to $\approx$. However, that would entail deep changes to syntax as well as to the definition of $\mathcal{E}$ to recover the newtype constructor patterns visible in source syntax.

3.3.8 Strictness, Divergence and Other Side-Effects

Instead of extending the source language, let us discuss ripping out a language feature for a change! So far, we have focused on Haskell as the source language,

which is lazy by default. Although the difference in evaluation strategy of the source language becomes irrelevant after desugaring, it raises the question of how much my approach could be simplified if we targeted a source language that was strict by default, such as OCaml, Lean, Idris, Rust, Python or C#.

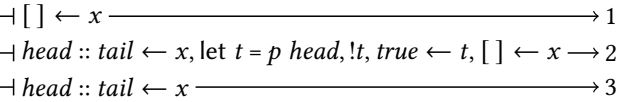
On first thought, it is tempting to simply drop all parts related to laziness from the formalism, such as $!x$ from Grd and $\multimap$ from Ant. Actually, Ant and $\mathcal{R}$ could vanish altogether and $\mathcal{A}$ could just collect the redundant GRHS directly! Since there would not be any bang guards, there is no reason to have $x \approx \bot$ and $x \not\approx \bot$ constraints either. Most importantly, the $\vdash_{\text{BOT}}$ judgment form has to go, because $\bot$ does not inhabit any types anymore.

And compiler writers for total languages such as Lean, Idris or Agda would live happily after: Talking about $x \not\approx \bot$ constraints made no sense there to begin with. Not so with OCaml or Rust, which are strict, non-total languages and allow arbitrary side-effects in expressions. Here's an example in OCaml:

```
let rec f p x =
  match x with
  | []       -> []
  | head :: _ when p head && x = [] -> [head]
  | _ :: tail -> f p tail;;
```

Is the second clause redundant? It depends on whether p performs a side-effect, such as throwing an exception, diverging, or even releasing a mutex. We may not say without knowing the definition of p, so the second clause has an inaccessible RHS but is not redundant. It's a similar situation as in a lazy language, although the fact that side-effects only matter in the guard of a match clause (where we can put arbitrary expressions) makes the issue much less prominent.

We could come up with a desugaring function for OCaml that desugars the pattern match above to the following guard tree:



Compared to Haskell, note the lack of a bang guard on the match variable x. Instead, there's now a bang guard on t, the new temporary that stands for $p\ head$. The bang guard will keep alive the second clause of the guard tree and LYG would not classify the second clause as redundant, although it will be flagged as inaccessible. Since the RHS of a **let** guard, such as $p\ head$, might have

arbitrary side-effects, equational reasoning is lost and we may no longer identify $p\ \mathit{head} \approx t$ as in Section 3.3.3.

Zooming out a bit more, desugaring of Haskell pattern-matches using bang guards $!x$ can be understood as forcing *one specific effect*, namely divergence. In this work, I have given this side-effect special treatment in the formalism in order to get accurate coverage warnings in a lazy language.

3.3.9 Or-patterns

Since this work appeared at ICFP in 2020, GHC 9.12 accumulated a new extension to the pattern language: `OrPatterns`[3]. Or-patterns are an established language feature in many other languages such as OCaml and Python, and can be used as follows:

```
data LogLevel = Debug | Info | Error
notifyAdmin :: LogLevel → Bool
notifyAdmin Error           = True
notifyAdmin (Debug; Info) = False
```

Here, the second clause matches when either Debug or Info matches the parameter. When the programmer later adds a new data constructor Warning to LogLevel, LYG should report the match in *notifyAdmin* as inexhaustive. This coverage warning prompts the programmer to make a conscious decision about which value should be returned for *notifyAdmin* Warning. That is far better than the alternative of using a wildcard match for the last clause: doing so would silently define *notifyAdmin* Warning = False.

Or-patterns were an interesting real-world benchmark to see how well LYG scales to new language features. Previously in Figure 3.4, if one part of a pattern failed to match, the whole pattern would fail. As a result, the desugaring function $\mathcal{D}$ could map a pattern into a (conjunctive) list of guards $(\overline{\mathrm{Grd}})$, which was then exploded into a nesting of $\dashv g \multimap t$ forms, suitable for a single recursive definition of $\mathcal{U}$ and $\mathcal{A}$. However, with Or-patterns, we need a way to encode (disjunctive) first-match semantics in the result of $\mathcal{D}(x, \mathit{pat})$. Such first-match semantics is currently exclusive to the $\top\!\!\sqcup_{t_2}^{t_1}$ guard tree form. So one way to desugar Or-patterns would be to desugar patterns into full guard trees instead of

[3] https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0522-or-patterns.rst

lists of guards. That would be akin to *exploding* each Or-pattern into two clauses. We would get the equality

$$\mathcal{D}(f\ (pat_a;\ pat_b)\ pat_c = rhs) \ = \ \begin{array}{l} \dashv \mathcal{D}(x_1, pat_a),\ \mathcal{D}(x_2, pat_c) \longrightarrow k_{rhs} \\ \lfloor \dashv \mathcal{D}(x_1, pat_b),\ \mathcal{D}(x_2, pat_c) \longrightarrow k_{rhs} \end{array}$$

thus duplicating the desugaring of pat_c. It is easy to see how a sequence of Or-patterns may lead to an exponential number of duplications of pat_c, leading to unacceptable checking performance. Hence I propose a different solution: ***Guard DAGs*** (directed-acyclic graphs).

Figure 3.12 defines the stucture of guard DAGs (GrdDag) inductively. Now consider the function

```
f :: Ordering → Ordering → Int
f (LT; EQ) (EQ; GT) = 1
f _        _        = 2
```

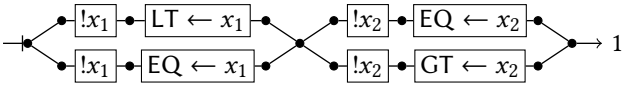
The desugaring to guard trees according to Figure 3.12 is

$$\begin{array}{l} \dashv (((!x_1, \mathsf{LT} \leftarrow x_1)\ \mathsf{or}\ (!x_1, \mathsf{EQ} \leftarrow x_1)), ((!x_2, \mathsf{EQ} \leftarrow x_2)\ \mathsf{or}\ (!x_2, \mathsf{GT} \leftarrow x_2))) \longrightarrow 1 \\ \lfloor \longrightarrow 2 \end{array}$$

Matching is defined as follows:

- Matching g means matching a single guard $g \in \mathrm{Grd}$, which was done by $\dashv g \mathrel{—} t$ previously. However, the new $\dashv d \mathrel{—} t$ form stores a guard DAG d instead of a single guard g.
- Matching a parallel composition (d_1 or d_2) means matching against d_1; if that succeeds, the overall match succeeds; if not, match against d_2.
- Matching a sequential composition (d_1, d_2) means matching against d_1; if that succeeds, match against d_2. If either match fails, the whole match fails.

Matching parallel composition (d_1 or d_2) is much like matching $\begin{array}{l}\dashv t_1 \\ \lfloor t_2\end{array}$, and matching sequential composition (d_1, d_2) is much like matching $\dashv d \mathrel{—} t$.
A clearer, non-flat visualisation of the guard DAG of the first clause could be



$$pat ::= \ldots \mid \boxed{pat_1;\ pat_2}$$

$$\begin{array}{rcl} t \in & \text{Gdt} & ::= \ldots \mid \ \longrightarrow\!\!\dashv \boxed{d} \longrightarrow t \\ d \in & \text{GrdDag} & ::= g \mid \boxed{(d_1 \text{ or } d_2)} \mid (d_1, d_2) \end{array}$$

$$\boxed{\mathcal{D}(x, pat) = \boxed{d}}$$

$$\mathcal{D}(x, (pat_1;\ pat_2)) = (\mathcal{D}(x, pat_1) \text{ or } \mathcal{D}(x, pat_2))$$

$$\boxed{C(\Theta, d) = \Theta}$$

$$\begin{array}{lcl} C(\Theta, (d_1 \text{ or } d_2)) & = & C(\Theta, d_1) \cup C(\mathcal{U}(\Theta, d_1), d_2) \\ C(\Theta, (d_1, d_2)) & = & C(C(\Theta, d_1), d_2) \\ C(\Theta, !x) & = & \Theta \mathbin{\dot\wedge} (x \not\approx \bot) \\ C(\Theta, \text{let } x = e) & = & \Theta \mathbin{\dot\wedge} (\text{let } x = e) \\ C(\Theta, K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x) & = & \Theta \mathbin{\dot\wedge} (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x) \end{array}$$

$$\boxed{\mathcal{U}(\Theta, t) = \Theta, \qquad \mathcal{U}(\Theta, d) = \Theta}$$

$$\begin{array}{lcl} \mathcal{U}(\langle\, \Gamma \mid \Phi \,\rangle,\ \longrightarrow n\,) & = & \langle\, \Gamma \mid \times \,\rangle \\ \mathcal{U}(\Theta,\ \top\!\!\!\begin{array}{l} t_1 \\ \bot\ t_2 \end{array}) & = & \mathcal{U}(\mathcal{U}(\Theta, t_1), t_2) \\ \mathcal{U}(\Theta,\ \longrightarrow\!\!\dashv d \longrightarrow t\,) & = & \mathcal{U}(\Theta, d) \cup \mathcal{U}(C(\Theta, d), t) \\ \\ \mathcal{U}(\Theta, (d_1 \text{ or } d_2)) & = & \mathcal{U}(\mathcal{U}(\Theta, d_1), d_2) \\ \mathcal{U}(\Theta, (d_1, d_2)) & = & \mathcal{U}(\Theta, d_1) \cup \mathcal{U}(C(\Theta, d_1), d_2) \\ \mathcal{U}(\langle\, \Gamma \mid \Phi \,\rangle, !x) & = & \langle\, \Gamma \mid \times \,\rangle \\ \mathcal{U}(\langle\, \Gamma \mid \Phi \,\rangle, \text{let } x = e) & = & \langle\, \Gamma \mid \times \,\rangle \\ \mathcal{U}(\Theta, K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x) & = & \Theta \mathbin{\dot\wedge} (x \not\approx K) \end{array}$$

$$\boxed{\mathcal{A}(\Theta, t) = u, \qquad \mathcal{A}(\Theta, d) = \Theta}$$

$$\begin{array}{lcl} \mathcal{A}(\Theta,\ \longrightarrow n\,) & = & \longrightarrow \Theta\, n \\ \mathcal{A}(\Theta,\ \top\!\!\!\begin{array}{l} t_1 \\ \bot\ t_2 \end{array}) & = & \top\!\!\!\begin{array}{l} \mathcal{A}(\Theta, t_1) \\ \bot\ \mathcal{A}(\mathcal{U}(\Theta, t_1), t_2) \end{array} \\ \mathcal{A}(\Theta,\ \longrightarrow\!\!\dashv d \longrightarrow t\,) & = & \longrightarrow \mathcal{A}(\Theta, d) \lightning \longrightarrow \mathcal{A}(C(\Theta, d), t) \\ \\ \mathcal{A}(\Theta, (d_1 \text{ or } d_2)) & = & \mathcal{A}(\Theta, d_1) \cup \mathcal{A}(\mathcal{U}(\Theta, d_1), d_2) \\ \mathcal{A}(\Theta, (d_1, d_2)) & = & \mathcal{A}(\Theta, d_1) \cup \mathcal{A}(C(\Theta, d_1), d_2) \\ \mathcal{A}(\Theta, !x) & = & \Theta \mathbin{\dot\wedge} (x \approx \bot) \\ \mathcal{A}(\langle\, \Gamma \mid \Phi \,\rangle, \text{let } x = e) & = & \langle\, \Gamma \mid \times \,\rangle \\ \mathcal{A}(\langle\, \Gamma \mid \Phi \,\rangle, K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x) & = & \langle\, \Gamma \mid \times \,\rangle \end{array}$$

Fig. 3.12: Extending coverage checking to handle Or-patterns

This visualisation acknowledges that GrdDag really models labelled *series-parallel graphs* [Eppstein 1992], a very specific kind of DAG with a straightforward encoding as an algebraic data type: every guard g induces a series-parallel graph with a single edge from source to sink; conjunction (d_1, d_2) corresponds to series composition of graphs for d_1 and d_2; and disjunction (d_1 or d_2) corresponds to parallel composition of graphs for d_1 and d_2.

Although the redefinition of coverage checking functions in Figure 3.12 is much more expansive in size than the original definition in Figure 3.5, we will see that the encoded logic is derivative.

There is a new function C that computes the *covered* set of a guard dag d. This function was previously inlined into the recursive call sites of $\mathcal{U}$ and $\mathcal{A}$; it computes the set of Θ reaching t in $\dashv g \dashv t$. It is no longer possible to inline it because the $\dashv d \dashv t$ form now carries a guard DAG d with nested structure; hence a separate recursive function is needed.

As expected, computing the uncovered set of parallel composition (d_1 or d_2) is much the same as for the $\top\!\!\sqcup_{t_2}^{t_1}$ form, and similarly for sequential composition (d_1, d_2) and the $\dashv d \dashv t$ form. Similarly, the uncovered set for the $\longrightarrow n$ form is the same as that of the irrefutable guards $!x$ and let $x = e$. The implementation in GHC (Section 3.4) extracts this shared code into reusable polymorphic combinators.

The changes to $\mathcal{A}$ are similar in nature. The use of $\cup$ in $\mathcal{A}(\Theta, (g_1, g_2))$ may be unexpected, since usually sequential composition leads to conjunction $\dot{\wedge}$, not disjunction $\cup$. Nevertheless, $\cup$ is the correct choice, because it follows directly from the previous definition of $\mathcal{A}(\Theta, \dashv g_1, g_2 \dashv t)$ and how the resulting $\dashv \Theta_1 \lightning \dashv \Theta_2 \lightning \dashv u$ annotations are used in $\mathcal{R}$ (Figure 3.6): u can be redundant only if there is no inhabitant in $\Theta_1 \cup \Theta_2$; otherwise it is inaccessible.

It is reassuring to know that extending coverage checking in Figure 3.5 with Or-patterns is derivative and compatible with all the other proposed extensions, although it takes a slight refactoring. On the other hand, years of maintaining LYG have shown that most of the complexity rests in the inhabitation test (Figure 3.10). I did not need to touch that to implement Or-patterns, and neither did I need to adjust Figure 3.6 or later: this is compelling evidence that the core of my approach is quite extensible and robust.

3.4 Implementation

My implementation of LYG has been part of GHC since the 8.10 release in 2020, including all extensions in Section 3.3, except for strict-by-default source syntax and Or-patterns. The implementation accumulates quite a few tricks that go beyond the pure formalism. This section is dedicated to describing these.

3.4.1 Phase Ordering

GHC runs the coverage checker between type checking and desugaring to GHC Core, a typed intermediate representation lacking the connection to source syntax. It would be unreasonable to do it later because warning messages need to reference source syntax, not GHC Core, in order to be comprehensible. At the same time, coverage checks involving GADTs need a type checked program, so coverage checking cannot happen before type-checking.

3.4.2 Interleaving $\mathcal{U}$ and $\mathcal{A}$

The set of reaching values is an argument to both $\mathcal{U}$ and $\mathcal{A}$. Given a particular set of input values and a guard tree, one can see by a simple inductive argument that both $\mathcal{U}$ and $\mathcal{A}$ are always called at the same arguments! Hence for an implementation it makes sense to compute both results together, if only for not having to recompute the results of $\mathcal{U}$ again in $\mathcal{A}$.

But there's more: Looking at the last clause of $\mathcal{U}$ in Figure 3.5, we can see that Θ is syntactically duplicated every time we check a pattern guard. In the worst case, that can amount to exponential growth of the refinement predicate and for the time to prove it empty!

What we really want is to summarise a Θ into a more compact canonical form before doing these kinds of *splits*. But that's exactly what ∇ is! Therefore, in the implementation I do not pass around and annotate refinement types, but the result of calling $\mathcal{N}$ on them directly.

We can see the resulting definition in Figure 3.13. The readability is clouded by unwrapping of pairs. $\mathcal{UA}$ requires that each ∇ individually is non-empty, i.e. not $\times$. This invariant is maintained by adding φ constraints through $\dot{\oplus}_\varphi$, which filters out any ∇ that would become empty.

$$\boxed{\overline{\nabla} \mathbin{\dot{\oplus}_\varphi} \varphi = \overline{\nabla}}$$

$$\begin{array}{rcl} \epsilon \mathbin{\dot{\oplus}_\varphi} \varphi & = & \epsilon \\ (\nabla_1 \dots \nabla_n) \mathbin{\dot{\oplus}_\varphi} \varphi & = & \begin{cases} (\langle \Gamma \parallel \Delta \rangle)\ (\nabla_2 \dots \nabla_n \mathbin{\dot{\oplus}_\varphi} \varphi) & \text{if } \langle \Gamma \parallel \Delta \rangle = \nabla \mathbin{\oplus_\varphi} \varphi \\ (\nabla_2 \dots \nabla_n) \mathbin{\dot{\oplus}_\varphi} \varphi & \text{otherwise} \end{cases} \end{array}$$

$$\boxed{\mathcal{UA}(\overline{\nabla}, t) = (\overline{\nabla}, u)}$$

$$\begin{array}{lcl} \mathcal{UA}(\overline{\nabla},\ \longrightarrow n) & = & (\epsilon,\ \longrightarrow \overline{\nabla}\, n) \\ \mathcal{UA}(\overline{\nabla},\ \top\!\!\!\bot \begin{smallmatrix} t_1 \\ t_2 \end{smallmatrix}) & = & (\overline{\nabla}_2,\ \top\!\!\!\bot \begin{smallmatrix} u_1 \\ u_2 \end{smallmatrix}) \\ & & \quad \text{where } (\overline{\nabla}_1, u_1) = \mathcal{UA}(\overline{\nabla}, t_1) \\ & & \quad\ \ (\overline{\nabla}_2, u_2) = \mathcal{UA}(\overline{\nabla}_1, t_2) \\ \mathcal{UA}(\overline{\nabla},\ \longrightarrow\!\!| !x \multimap t) & = & \multimap \overline{\nabla} \mathbin{\dot{\oplus}_\varphi} (x \approx \bot) \lightning \multimap u \\ & & \quad \text{where } (\overline{\nabla}', u) = \mathcal{UA}(\overline{\nabla} \mathbin{\dot{\oplus}_\varphi} (x \not\approx \bot), t) \\ \mathcal{UA}(\overline{\nabla},\ \longrightarrow\!\!| \mathsf{let}\ x = e \multimap t) & = & \mathcal{UA}(\overline{\nabla} \mathbin{\dot{\oplus}_\varphi} (\mathsf{let}\ x = e), t) \\ \mathcal{UA}(\overline{\nabla},\ \longrightarrow\!\!| K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x \multimap t) & = & ((\overline{\nabla} \mathbin{\dot{\oplus}_\varphi} (x \not\approx K))\ \overline{\nabla_u}, u) \\ & & \quad \text{where } \overline{\nabla_c} = \overline{\nabla} \mathbin{\dot{\oplus}_\varphi} (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x) \\ & & \quad\ \ (\overline{\nabla_u}, u) = \mathcal{UA}(\overline{\nabla_c}, t) \end{array}$$

Fig. 3.13: Fast coverage checking

3.4.3 Throttling for Graceful Degradation

Even with the tweaks from Section 3.4.2, checking certain pattern matches remains NP-hard [Sekar et al. 1995]. Naturally, there will be cases where we have to conservatively approximate in order not to slow down compilation too much. Consider the following example and its corresponding guard tree:

$$\begin{array}{l} g\ \_ \mid \mathsf{True} \leftarrow f_1\ 1,\ \mathsf{True} \leftarrow f_2\ 1\ = () \\ \quad \mid \mathsf{True} \leftarrow f_1\ 2,\ \mathsf{True} \leftarrow f_2\ 2\ = () \\ \quad \mid \dots \\ \quad \mid \mathsf{True} \leftarrow f_1\ N, \mathsf{True} \leftarrow f_2\ N = () \end{array}$$

$$\begin{array}{l} \multimap\!\!| \mathsf{let}\ a_1 = f_1\ 1, !a_1, \mathsf{True} \leftarrow a_1, \mathsf{let}\ b_1 = f_2\ 1, !b_1, \mathsf{True} \leftarrow b_1 \longrightarrow 1 \\ \multimap\!\!| \mathsf{let}\ a_2 = f_1\ 2, !a_2, \mathsf{True} \leftarrow a_2, \mathsf{let}\ b_2 = f_1\ 2, !b_2, \mathsf{True} \leftarrow b_2 \longrightarrow 2 \\ \multimap\!\!| \dots \longrightarrow \dots \\ \multimap\!\!| \mathsf{let}\ a_N = f_1\ N, !a_N, \mathsf{True} \leftarrow a_N, \mathsf{let}\ b_N = f_2\ N, !b_N, \mathsf{True} \leftarrow b_N \longrightarrow \mathrm{N} \end{array}$$

Each of the N GRHS can fall through in two distinct ways: By failure of either pattern guard involving f_1 or f_2. Initially, we start out with a single input ∇. After the first equation it will split into two sub-∇s, after the second into four, and so on. This exponential pattern repeats N times, and leads to horrible performance!

Instead of *refining* ∇ with the pattern guard, leading to a split, we could just continue with the original ∇, thus forgetting about the $a_1 \not\approx \mathsf{True}$ or $b_1 \not\approx \mathsf{True}$ constraints. In terms of the modeled refinement type, ∇ is still a superset of both refinements, and thus a sound overapproximation.

In the implementation, I call this *throttling*: limiting the number of reaching ∇s to a constant. Whenever a split would exceed this limit, we continue with the original reaching ∇s, a conservative estimate, instead. Intuitively, throttling corresponds to *forgetting* what we matched on in that particular subtree. Throttling is refreshingly easy to implement! Only the last clause of $\mathcal{UA}$, where splitting is performed, needs to change:

$$\mathcal{UA}(\overline{\nabla},\ \dashv K\ \overline{a}\ \overline{\gamma}\ \overline{y:\tau} \leftarrow x \multimap t\ ) = (\left\lfloor (\overline{\nabla} \dot{\oplus}_\varphi (x \not\approx K))\ \overline{\nabla_u} \right\rfloor_{\overline{\nabla}}, u)$$
$$\text{where } \overline{\nabla_c} = \overline{\nabla} \dot{\oplus}_\varphi (K\ \overline{a}\ \overline{\gamma}\ \overline{y:\tau} \leftarrow x)$$
$$(\overline{\nabla_u}, u) = \mathcal{UA}(\overline{\nabla_c}, t)$$

where the new throttling operator $\lfloor \_ \rfloor_\_$ is defined simply as

$$\left\lfloor \overline{\nabla} \right\rfloor_{\overline{\nabla}'} = \begin{cases} \overline{\nabla} & \text{if } |\{\overline{\nabla}\}| \leqslant U \\ \overline{\nabla}' & \text{otherwise} \end{cases}$$

with U being an arbitrary constant. GHC uses 30 as the limit in the implementation (dynamically configurable via a command-line flag) without noticing any false positives in terms of exhaustiveness warnings outside of the test suite.

It is worth noting that Or-patterns (Section 3.3.9) introduce a function C to compute the covered set of a guard DAG d, and its case for $(d_1 \text{ or } d_2)$ splits the incoming Θ, much the same as the pattern guard case splits the uncovered set; hence I throttle there as well to ensure graceful degradation.

3.4.4 Maintaining Residual `COMPLETE` Sets

The implementation tries hard to make the inhabitation test as efficient as possible. For example, Δs are represented by a mapping from variables to their

positive and negative constraints for easier indexing. But there are also asymptotic improvements. Consider the following function:

```
data T = A1 | ... | A1000      f A1      = 1
pattern P = ...                f A2      = 2
{# COMPLETE A1, P #}           ...
                               f A1000 = 1000
```

f is exhaustively defined. To see that we need to perform an inhabitation test for the match variable x after the last clause. The test will conclude that the builtin `COMPLETE` set was completely overlapped. But in order to conclude that, the algorithm tries to instantiate x (via ⊢Inst) to each of its 1000 constructors and try to add a positive constructor constraint! What a waste of time, given that we could just look at the negative constraints on x *before* trying to instantiate x. But asymptotically it should not matter much, since we are doing this only once at the end.

Except that is not true, because we also perform redundancy checking! At any point in f's definition there might be a match on P, after which all remaining clauses would be redundant by the user-supplied `COMPLETE` set. Therefore, we have to perform the expensive inhabitation test *after every clause*, involving $O(n)$ instantiations each.

Clearly, we can be smarter about that! Indeed, I cache *residual* `COMPLETE` *sets* in the implementation: Starting from the full `COMPLETE` sets, I delete ConLikes from them whenever I add a new negative constructor constraint, maintaining the invariant that each of the sets is inhabited by at least one constructor. Note how this trick never needs to check the same constructor twice (except after adding new type constraints), thus we have an amortised $O(n)$ instantiations for the whole checking process.

3.4.5 Reporting Uncovered Patterns

The expansion function $\mathcal{E}$ in Figure 3.7 exists purely for presenting uncovered patterns to the user. It does not account for negative information, however, which can lead to surprising warnings. Consider a definition like b True = (). The computed uncovered set of b is the normalised refinement type $\nabla_b = \langle x : \mathsf{Bool} \,||\, x \not\approx \bot, x \not\approx \mathsf{True} \rangle$, which crucially contains no positive information on x! As a result, $\mathcal{E}(\nabla_b) = \_$ and only the very unhelpful wildcard pattern _ will be reported as uncovered.

My implementation does better and shows that this is just a presentational matter. It splits ∇_b on all possible constructors of Bool, immediately rejecting the refinement $\nabla_b \oplus_\delta x \approx \text{True}$ due to $x \not\approx \text{True} \in \nabla_b$. What remains is the refinement $\nabla_b \oplus_\delta x \approx \text{False} = \langle x : \text{Bool} \,\|\, x \not\approx \bot, x \not\approx \text{True}, x \approx \text{False} \rangle$, which has the desired positive information for which $\mathcal{E}$ will happily report False as the uncovered pattern.

Additionally, my implementation formats negative information on opaque data types such as Int and Char, since idiomatic use would match on literals rather than on GHC-specific data constructors. For example, coverage checking $f\ 0 = ()$ will report something like this:

```
Missing equations for function 'f':
  f x = ... where 'x' is not one of {0}
```

3.4.6 Structured Guard Tree Types

Since we submitted our work to ICFP in 2020, I continued to improve and refactor the implementation of LYG in GHC. Many of the changes were incremental improvements and bug fixes that are not easy to present without a lot of context, but one particularly important innovation[4] was the introduction of syntax-specific instances of guard trees, such as

```
type SrcInfo = String  -- approximately; identifies the k in rhs_k
data PmMatch p = PmMatch { pm_pats :: p, pm_grhss :: [PmGRHS p] }
data PmGRHS p  = PmGRHS  { pg_grds :: p, pg_rhs :: SrcInfo }
```

These types are in structural correspondence to the *match* and *grhs* constructs in Figure 3.2 from whence they desugar. Prior to coverage checking, type parameter p is instantiated to lists of guards $\overline{\text{Grd}}$ (resp. GrdDag after Or-patterns were introduced, Section 3.3.9), and coverage checking elaborates this list into so-called RedSets, carrying Θs encoding covered and diverging input values.

Of course, the meaning of PmMatch and PmGRHS is in terms of the desugaring into unrestricted guard trees Gdt, as before. However, with the new encoding it became much easier to extract covered sets for long-distance information (Section 3.3.1), because the *pm_grhss* field has the same number of elements as there are *grhs* in a *match*, so a simple Data.List.*zip* suffices to bring covered sets and *grhs* together.

[4] https://gitlab.haskell.org/ghc/ghc/-/commit/1207576

3.5 Evaluation

To put the new coverage checker to the test, Ryan Scott performed a survey of real-world Haskell code using the `head.hackage` repository[5]. `head.hackage` contains a sizable collection of libraries and minimal patches necessary to make them build with a development version of GHC. Ryan identified those libraries which compiled without coverage warnings using GHC 8.8.3 (which uses GMTM as its checking algorithm) but emitted warnings when compiled using the LYG version of GHC.

Of the 361 libraries in `head.hackage`, seven of them revealed coverage issues that only LYG warned about. Two of the libraries, `pandoc` and `pandoc-types`, have cases that were flagged as redundant due to LYG's improved treatment of guards and term equalities.
One library, `geniplate-mirror`, has a case that was redundant by way of long-distance information. Another library, `generic-data`, has a case that is redundant due to bang patterns.

The last three libraries—`Cabal`, `HsYAML`, and `network`—were the most interesting. `HsYAML` in particular defines this function:

```
go' _ _ _ xs | False = error (show xs)
go' _ _ _ xs = err xs
```

The first clause is clearly unreachable, and LYG now flags it as such. However, the authors of `HsYAML` likely left in this clause because it is useful for debugging purposes. One can comment out the second clause and remove the `False` guard to quickly try out a code path that prints a more detailed error message. Moreover, leaving the first clause in the code ensures that it is typechecked and less susceptible to bitrotting over time.

In order to support this use case in `HsYAML`, we added a primitive definition *considerAccessible* = `False` in GHC 9.2, to be used instead of `False` above and signalling to GHC that the first clause should not get marked as redundant. The unreachable code in `Cabal` and `network` is of a similar caliber and could benefit from *considerAccessible* as well.

	Time (milliseconds)			Allocation (megabytes)		
Testcase	8.8.3	HEAD	Change	8.8.3	HEAD	Change
T11276	1.16	1.69	45.7%	1.86	2.39	28.6%
T11303	28.1	18.0	-36.0%	60.2	39.9	-33.8%
T11303b	1.15	0.39	-65.8%	1.65	0.47	-71.8%
T11374	4.62	3.00	-35.0%	6.16	3.20	-48.1%
T11822	1,060	16.0	-98.5%	2,010	27.9	-98.6%
T11195	2,680	22.3	-99.2%	3,080	39.5	-98.7%
T17096	7,470	16.6	-99.8%	17,300	35.4	-99.8%
PmSeriesS	44.5	2.58	-94.2%	52.9	6.19	-88.3%
PmSeriesT	48.3	6.86	-85.8%	61.4	17.6	-71.4%
PmSeriesV	131	4.54	-96.5%	139	9.53	-93.2%

Table 3.1: The relative compile-time performance of GHC 8.8.3 (which implements GMTM) and HEAD (which implements LYG) on test cases designed to stress-test coverage checking.

3.5.1 Performance Tests

To compare the efficiency of GMTM and LYG quantitatively, Ryan Scott collected a series of test cases from GHC's test suite that are designed to test the compile-time performance of coverage checking[6]. Table 3.1 lists each of these 11 test cases. Test cases with a `T` prefix are taken from user-submitted bug reports about the poor performance of GMTM. Test cases with a `PmSeries` prefix are adapted from Maranget [2007], which presents several test cases that caused GHC to exhibit exponential running times during coverage checking.

Ryan compiled each test case with GHC 8.8.3, which uses GMTM as its checking algorithm, and GHC HEAD (a pre-release of GHC 8.10), which uses LYG. He measured (1) the time spent in the desugarer, the phase of compilation in which coverage checking occurs, and (2) how many megabytes were allocated during desugaring. Table 3.1 shows these figures as well as the percent change going from 8.8.3 to HEAD. Most cases exhibit a noticeable improvement under LYG, with the exception of `T11276`. Investigating `T11276` suggested that the performance of GHC's equality constraint solver had become more expensive at

[5] https://gitlab.haskell.org/ghc/head.hackage/commit/30a310f

[6] These measurements were validated as part of the artifact evaluation process during the ICFP 2020 publication.

the time [GHC issue 2020c], and these extra costs outweighed the performance benefits of using LYG. This performance bug was fixed in GHC 9.0[7].

Note that for typical code (rather than for regression tests), time spent doing coverage checking is dwarfed by the time the rest of the desugarer takes. A very desirable property for a static analysis that is irrelevant to the compilation process!

3.5.2 GHC Issues

Implementing LYG in GHC has fixed over 30 bug reports related to coverage checking. These include:

- Better compile-time performance [GHC issue 2019b,c, 2015a, 2016e]
- More accurate warnings for empty **case** expressions [GHC issue 2019a, 2018f,h, 2015b, 2017f]
- More accurate warnings due to LYG's desugaring [GHC issue 2018a, 2017d, 2016c,d, 2020d]
- More accurate warnings due to improved term-level reasoning [GHC issue 2017a, 2018c,d,e, 2019d, 2016a, 2018i, 2019e,k]
- More accurate warnings due to tracking long-distance information [GHC issue 2020a,b, 2019j]
- Improved treatment of `COMPLETE` sets [GHC issue 2018b, 2017b,c, 2016b, 2019f,g, 2017e, 2019h, 2017g]
- Better treatment of strictness, bang patterns, and newtypes [GHC issue 2018g,j, 2019i,l]

3.6 Soundness

The evaluation in Section 3.5 yields compelling evidence that LYG is *sound*. That is, in terms of the formalism, LYG *overapproximates* — but never underapproximates — the set of reaching values passed to $\mathcal{U}$ and $\mathcal{A}$. As a result, LYG will

[7] https://gitlab.haskell.org/ghc/ghc/-/commit/fd7ea0f

never fail to report uncovered clauses (no false negatives), but it may report false positives. Similarly, LYG will never report accessible clauses as redundant (no false positives), but it may fail to report clauses which are redundant when the code involved is too close to "undecidable territory".

Remarkably, the symbolic checking process involving $\mathcal{U}$, $\mathcal{A}$ und $\mathcal{R}$ does not overapproximate at all. To my knowledge, LYG overapproximates only in these three mechanisms:

- LYG can run out of fuel for inhabitation testing (Section 3.2.7).
- Throttling (Section 3.4.3) is useful when implementing LYG efficiently.
- LYG forgoes non-trivial semantic analysis of expressions. LYG can recognize identical patterns or subexpressions, but it stops short of anything more sophisticated, such as interprocedural analysis or SMT-style reasoning (Section 3.7.2).

But what does it actually *mean* for a value to match a particular part of a guard tree, such as a right-hand side $\longrightarrow k$, mathematically? In what precise sense does LYG, or does not, overapproximate this supposed *semantics*?

Since this work appeared at ICFP 2020, Dieterichs [2021] worked out both a formal semantics as well as a mechanised correctness proof in Lean 3 for the coverage checking pass from guard trees into uncovered set and annotated trees.[8] He shows that $\mathcal{U}$, $\mathcal{A}$ and $\mathcal{R}$ preserve key semantic properties of the guard trees under analysis, provided that function $\mathcal{G}(\Theta)$ for generating inhabitants indeed overapproximates Θ. I will briefly summarise the correctness results here. For that, I need to define a plausible formal semantics for guard trees and refinement predicates.

3.6.1 Semantics

I have described the semantics of guard trees and guards informally in Section 3.2.1. Figure 3.14 formalises this intuition, describing the semantics of guard trees by a function $\mathrm{Gdt}[\![t]\!]_\rho$ that, given a guard tree t and an environment ρ describing a vector of values to match against, returns

- $\mathrm{success}(k)$ when ρ is a vector of values that will reach RHS k when matched against t.

[8] Types and type constraints are ignored; their interaction is largely a black box to LYG anyway.

Semantics of guard trees

$$
\begin{array}{rcl}
d \in \mathsf{D} & = & \bot \mid K\ \overline{d} \mid \ldots \\
\rho \in \mathsf{Env} & ::= & [\overline{x \mapsto d}] \\
r \in \mathsf{Res}[\Box] & ::= & \mathsf{success}(\Box) \mid \mathsf{fail} \mid \mathsf{diverge}
\end{array}
$$

$$\boxed{\mathsf{Expr}[\![e]\!]_\rho \in \mathsf{D}, \qquad \mathsf{Grd}[\![g]\!]_\rho \in \mathsf{Res}[\rho], \qquad \mathsf{Gdt}[\![t]\!]_\rho \in \mathsf{Res}[k]}$$

$$
\begin{array}{lcl}
\mathsf{Expr}[\![K\ \overline{e}]\!]_\rho & = & K\ \overline{\mathsf{Expr}[\![e]\!]_\rho} \\
\mathsf{Expr}[\![e]\!]_\rho & = & \ldots \\
\\
\mathsf{Grd}[\![\mathsf{let}\ x = e]\!]_\rho & = & \mathsf{success}(\rho[x \mapsto \mathsf{Expr}[\![e]\!]_\rho]) \\
\mathsf{Grd}[\![K\ \overline{y} \leftarrow x]\!]_\rho & = & \begin{cases} \mathsf{success}(\rho[\overline{y \mapsto d}]) & \text{if } \rho(x) = K\ \overline{d} \\ \mathsf{fail} & \text{otherwise} \end{cases} \\
\mathsf{Grd}[\![!x]\!]_\rho & = & \begin{cases} \mathsf{diverge} & \text{if } \rho(x) = \bot \\ \mathsf{success}(\rho) & \text{otherwise} \end{cases} \\
\\
\mathsf{Gdt}[\![\ \longrightarrow k\ ]\!]_\rho & = & \mathsf{success}(k) \\
\mathsf{Gdt}[\![\ \top\!\!\begin{array}{l} t_1 \\ \bot\ t_2 \end{array}\ ]\!]_\rho & = & \begin{cases} \mathsf{Gdt}[\![t_2]\!]_\rho & \text{if } \mathsf{Gdt}[\![t_1]\!]_\rho = \mathsf{fail} \\ \mathsf{Gdt}[\![t_1]\!]_\rho & \text{otherwise} \end{cases} \\
\mathsf{Gdt}[\![\ \longrightarrow\!| g \text{---} t\ ]\!]_\rho & = & \begin{cases} \mathsf{Gdt}[\![t]\!]_{\rho'} & \text{if } \mathsf{Grd}[\![g]\!]_\rho = \mathsf{success}(\rho') \\ \mathsf{Grd}[\![g]\!]_\rho & \text{otherwise} \end{cases}
\end{array}
$$

Semantics of refinement types

$$\boxed{\rho \vDash (\varphi, \rho), \qquad \rho \vDash \Theta}$$

$$\frac{}{\rho \vDash (\checkmark, \rho)} \qquad \frac{\rho(x) = K\ \overline{d}}{\rho \vDash (K\ \overline{y} \leftarrow x, \rho[\overline{y \mapsto d}])} \qquad \frac{\rho(x) \neq K\ \overline{d}}{\rho \vDash (x \not\approx K, \rho)}$$

$$\frac{\rho(x) = \bot}{\rho \vDash (x \approx \bot, \rho)} \qquad \frac{\rho(x) \neq \bot}{\rho \vDash (x \not\approx \bot, \rho)} \qquad \frac{}{\rho \vDash (\mathsf{let}\ x = e, \rho[x \mapsto \mathsf{Expr}[\![e]\!]_\rho])}$$

$$\frac{\Gamma_1 \vdash \rho_1 \quad \Gamma_2 \vdash \rho_2 \quad \rho_1 \vDash (\varphi, \rho_2) \quad \rho_2 \vDash \langle\, \Gamma_2 \mid \Phi \,\rangle}{\rho_1 \vDash \langle\, \Gamma_1 \mid \varphi \wedge \Phi \,\rangle}$$

$$\frac{\rho \vDash \langle\, \Gamma \mid \Phi_1 \,\rangle}{\rho \vDash \langle\, \Gamma \mid \Phi_1 \vee \Phi_2 \,\rangle} \qquad \frac{\rho \vDash \langle\, \Gamma \mid \Phi_2 \,\rangle}{\rho \vDash \langle\, \Gamma \mid \Phi_1 \vee \Phi_2 \,\rangle}$$

Fig. 3.14: Semantics of guard trees

- fail when ρ is a vector of values that is not covered by t.
- diverge when ρ is a vector of values that will lead to divergence when matched against t.

Likewise, the valuation $\mathsf{Grd}[\![g]\!]_\rho$ returns $\mathsf{success}(\rho')$ when the vector of values ρ matches guard g, extending ρ with new bindings into ρ'. The semantics of expressions $\mathsf{Expr}[\![e]\!]_\rho$ maps into the semantic domain D, just as the environment ρ. Since this work leaves open a lot of details about the expression fragment of the source language, the semantics leaves open D and most of $\mathsf{Expr}[\![e]\!]_\rho$ as well, with the exception of postulating a semantics for the data constructor application case.

Refinement types have been introduced by informal examples in Section 3.2.2, denoting refinement types Θ by sets of vectors of values ρ that satisfy the encoded refinement predicate. Figure 3.14 finally defines the satisfiability relation by an inductive predicate $\rho \vDash \Theta$. Thus, whenever a vector of values ρ is part of the set denoted by a refinement type Θ, the inductive predicate must be provable.

The definition of $\rho \vDash \Theta$ assumes that conjunction $\wedge$ is associated to the right, $\varphi \wedge \Phi$, highlighting the unusual scoping semantics briefly mentioned in Section 3.2.2. Any binding constructs in the φ to the left of $\wedge$, such as $\mathsf{let}\ x = e$ or $K\ \overline{y} \leftarrow x$, introduce names that are subsequently in scope in the Φ to the right of $\wedge$. In hindsight, I could have picked a different operator symbol to avoid this confusion, for example $(\varphi\ \mathsf{in}\ \Phi)$, such as in Dieterichs [2021]. Doing so would however complicate the $(\Theta \mathbin{\dot\wedge} \varphi)$ operator a bit. In the absence of types, the postulated judgment $\Gamma \vdash \rho$ merely becomes a scoping check, namely that Γ has the same domain as ρ.

3.6.2 Formal Soundness Statement

Having stated plausible semantics for the inputs and outputs of $\mathcal{U}$, I can formulate what it means for $\mathcal{U}$ to be correct, following Dieterichs [2021, Section 4.1] who mechanised the proof in Lean 3.

Theorem 3.1. *Let* $\mathcal{U}(\langle \Gamma \mid \checkmark \rangle, t) = \Theta$. *Then* $\mathit{Gdt}[\![t]\!]_\rho = \mathsf{fail}$ *if and only if* $\rho \vDash \Theta$.

In other words: when Θ is the set of uncovered values of guard tree t as computed by $\mathcal{U}$, then any vector of values ρ that falls through all clauses of t (i.e. $\mathsf{Gdt}[\![t]\!]_\rho = \mathsf{fail}$) is in Θ (i.e. $\rho \vDash \Theta$). In this precise sense, $\mathcal{U}$ is *sound*. Conversely, when $\mathcal{U}$ returns a non-empty refinement type Θ, there exists a vector of values

ρ in Θ, and by Theorem 3.1 we have that ρ must also fall through all clauses of t. In this precise sense, $\mathcal{U}$ is *complete*.

Of course, the judgment $\rho \vDash \Theta$ frequently compares domain values d that ultimately come from evaluating expressions $\mathsf{Expr}[\![e]\!]_\rho$, rendering the predicate undecidable for many source languages. In the language of abstract interpretation (cf. Section 2.3), each Θ defines a semantic program property $P_\Theta \triangleq \{\ \rho \mid \rho \vDash \Theta\ \}$. It is important for an implementation of $\mathcal{G}$ to be sound wrt. any such P_Θ, which means that it will *overapproximate*; roughly $P_\Theta \subseteq \mathcal{G}(\Theta)$. Dieterichs captures this in his `can_prove_empty` definition to parameterise over sound implementations of $\mathcal{G}$. In Section 4.2, he proves the following soundness theorem about $\mathcal{A}$ and $\mathcal{R}$.

Theorem 3.2. *Let $\mathcal{R}(\mathcal{A}(\langle \Gamma \mid \checkmark \rangle, t)) = (a, i, r)$ and $\mathcal{G}$ sound in the above sense.*

- *If $Gdt[\![t]\!]_\rho = \mathsf{success}(k)$, then $k \in a$, i.e. clause k is accessible according to $\mathcal{A}$ and $\mathcal{R}$.*
- *If $k \in r$ is redundant, then removing clause k from guard tree t does not change the semantics of t, i.e. $\forall \rho.\ Gdt[\![t]\!]_\rho = Gdt[\![remove(k, t)]\!]_\rho$ (where $remove(k, t)$ is the implied removal operation).*

Perhaps unsurprisingly, proving correct the transformation in the second part of Theorem 3.2 proved far more subtle than the proof for Theorem 3.1. Fortunately, the mechanisation provides confidence in the proof's correctness.

3.7 Related Work

3.7.1 Comparison with GADTs Meet Their Match

Karachalias et al. [2015] present GADTs Meet Their Match (GMTM), an algorithm which handles many of the subtleties of GADTs, guards, and laziness mentioned in Section 3.1. Despite this, the GMTM algorithm still gives incorrect warnings in many cases.

GMTM Does Not Consider Laziness in its Full Glory

The formalism in Karachalias et al. [2015] incorporates strictness constraints, but these constraints can only arise from matching against data constructors. GMTM does not consider strict matches that arise from strict fields of data constructors or bang patterns. A consequence of this is that GMTM would incorrectly warn

that *v* (page 19) is missing a case for SJust, even though such a case is unreachable. LYG, on the other hand, more thoroughly tracks strictness when desugaring Haskell programs.

GMTM's Treatment of Guards Is Shallow

GMTM can only reason about guards through an abstract term oracle. Although the algorithm is parametric over the choice of oracle, in practice the implementation of GMTM in GHC uses an extremely simple oracle that can only reason about guards in a limited fashion. More sophisticated uses of guards, such as in this variation of the *safeLast* function from Section 3.1.2, will cause GMTM to emit erroneous warnings:

```
safeLast2 xs
  | (x : _) ← reverse xs = Just x
  | [ ]     ← reverse xs = Nothing
```

While GMTM's term oracle is customisable, it is not as simple to customize as one might hope. The formalism in Karachalias et al. [2015] represents all guards as $p \leftarrow e$, where p is a pattern and e is an expression. This is a straightforward, syntactic representation, but it also makes it more difficult to analyse when e is a complicated expression. This is one of the reasons why it is difficult for GMTM to accurately give warnings for the *safeLast* function, since it would require recognizing that both clauses scrutinise the same expression in their view patterns.

LYG makes analysing term equalities simpler by first desugaring guards from the surface syntax to guard trees. The $\oplus_\varphi$ function, which is roughly a counterpart to GMTM's term oracle, can then reason about terms arising from patterns. While $\oplus_\varphi$ is already more powerful than a trivial term oracle, its real strength lies in the fact that it can easily be extended, as LYG's treatment of view patterns (Section 3.3.3) demonstrates. While GMTM's term oracle could be improved to accomplish the same thing, it is unlikely to be as straightforward of a process as extending $\oplus_\varphi$.

3.7.2 Comparison with Similar Coverage Checkers

Structural and Semantic Pattern Matching Analysis in Haskell

Kalvoda and Kerckhove [2019] implement a variation of GMTM that leverages an SMT solver to give more accurate coverage warnings for programs that use

guards. For instance, their implementation can conclude that the *signum* function from Section 3.1.1 is exhaustive. This is something that LYG cannot do out of the box, although it would be possible to extend $\oplus_\varphi$ with SMT-like reasoning about booleans and linear integer arithmetic.

Warnings for Pattern Matching

Maranget [2007] presents a coverage checking algorithm for OCaml that can identify clauses that are not *useful*, i.e. *useless*. While OCaml is a strict language, the algorithm can be adapted to handle languages with non-strict semantics such as Haskell. In a lazy setting, uselessness corresponds to unreachable clauses. Maranget does not distinguish inaccessible clauses from redundant ones; thus clauses flagged as useless (such as the first two clauses of u' in Section 3.1.3) generally cannot be deleted without changing (lazy) program semantics.

Case Trees in Dependently Typed Languages

Case trees [Augustsson 1985] are a standard way of compiling pattern-matches to efficient code. Much like LYG's guard trees, case trees present a simplified representation of pattern matching. Several compilers for dependently typed languages also use case trees as coverage checking algorithms, as a well-typed case tree can guarantee that it covers all possible cases. Case trees play an integral role in coverage checking in Agda [Cockx and Abel 2018; Norell 2007] and the Equations plugin for Rocq[9] [Sozeau 2010; Sozeau and Mangin 2019]. Oury [2007] checks for coverage in a dependently typed setting using sets of inhabitants of data types, which have similarities to case trees.

One could take inspiration from case trees should one wish to extend LYG to support dependent types. My implementation of LYG in GHC can already handle quasi-dependently typed code, such as the `singletons` library [Eisenberg and Stolarek 2014; Eisenberg and Weirich 2012], so I expect that it can be adapted to full dependent types. One key change that would be required is extending equation (9) in Figure 3.9 to reason about term constraints in addition to type constraints. GHC's constraint solver already has limited support for term-level reasoning as part of its `DataKinds` language extension [Yorgey et al. 2012], so the groundwork is present.

[9] Formerly Coq.

Refinement Type–Based Totality Checking in Liquid Haskell

In addition to LYG, Liquid Haskell uses refinement types to perform a limited form of exhaustivity checking [Vazou, Seidel, et al. 2014; Vazou, Tondwalkar, et al. 2017]. While exhaustiveness checks are optional in ordinary Haskell, they are mandatory for Liquid Haskell, as proofs written in Liquid Haskell require user-defined functions to be total (and therefore exhaustive) in order to be sound. For example, consider this non-exhaustive function:

$$\begin{aligned} &\mathit{fibPartial} :: \mathsf{Integer} \to \mathsf{Integer} \\ &\mathit{fibPartial}\ 0 = 0 \\ &\mathit{fibPartial}\ 1 = 1 \end{aligned}$$

When compiled, GHC fills out this definition by adding an extra $\mathit{fibPartial}\ \_ = \mathit{error}$ `"undefined"` clause. Liquid Haskell leverages this by giving *error* the refinement type:

$$\mathit{error} :: \{\, v : \mathsf{String} \mid \mathit{false} \,\} \to a$$

As a result, attempting to use *fibPartial* in a proof will fail to verify unless the user can prove that *fibPartial* is only ever invoked with the arguments 0 or 1.

3.7.3 Other Representations of Constraints

Leveraging Existing Constraint Solvers

LYG represents Φ constraints using logical predicates that are tailor-made for LYG's purposes. One could instead imagine encoding Φ constraints in a more standard logic and then using an "off-the-shelf" constraint solver to check them. This would render Figures 3.8 and 3.9 and the arguably rather intricate Sections 3.2.6 and 3.2.7 unnecessary, and it allows the checker to benefit from improvements to the solver without any further maintenance burden.

Encoding Φ constraints into another logic would have its downsides, however. The $\oplus_\varphi$ function is able to reason about LYG-oriented predicates rather efficiently, but other constraint solvers (e.g. STM solvers) might incur significant constant factors. Moreover, elaborating from one logic to another could inhibit programmers from forming a mental model of how coverage checking works.

Refinement Types versus Predicates

Refinement types Θ and predicates Φ are very similar. The main difference between the two is that refinement types carry a typing context Γ that is used for inhabitation testing. Predicates are quite fully featured on their own, however, as they can bind variables that scope over conjunctions. The scoping semantics of predicates allows $\mathcal{U}$ and $\mathcal{A}$ to be purely syntactic transformations, and in fact, they could be modified to take Φ as an argument rather than Θ.

Making $\mathcal{U}$ and $\mathcal{A}$ operate over Θ or Φ is ultimately a design choice. I have opted to operate over Θ mainly because I find it more intuitive to think about coverage checking as refining a vector of values as it falls from one match to the next. In my opinion, that intuition is more easily expressed with refinement types than predicates alone.

3.7.4 Positive and Negative Information

LYG's use of positive and negative constructor constraints is inspired by Sestoft [1996], which uses positive and negative information to implement a pattern-match compiler for ML. Sestoft utilises positive and negative information to generate decision trees that avoid scrutinizing the same terms repeatedly. This insight is equally applicable to coverage checking and is one of the primary reasons for LYG's efficiency.

Besides efficiency, the accuracy of redundancy warnings involving `COMPLETE` sets hinge on negative constraints. To see why this is not possible in other checkers that only track positive information, such as those of Karachalias et al. [2015] (Section 3.7.1) and Maranget [2007] (Section 3.7.2), consider the following example:

```
pattern True' = True
{-# COMPLETE True', False #-}
f False = 1
f True' = 2
f True  = 3
```

GMTM would have to commit to a particular `COMPLETE` set when encountering the match on `False`, without any semantic considerations. Choosing {`True'`, `False`} here will mark the third GRHS as redundant, while choosing {`True`, `False`} will not. GHC's implementation used to try each `COMPLETE` set in turn and would disambiguate using a complicated metric based on the number

and kinds of warnings the choice of each set would generate [GHC team 2020], which was broken still [GHC issue 2017g].

Negative constraints make LYG efficient in other places too, such as in this example:

```
data T = A1 | ... | A1000
h A1 _   = 1
h _  A1  = 2
```

In *h*, GMTM would split the value vector (which is like LYG's Δs without negative constructor constraints) into 1000 alternatives over the first match variable, and then *each* of the 999 value vectors reaching the second GRHS into another 1000 alternatives over the second match variable. Negative constraints allow LYG to compress the 999 value vectors falling through into a single one indicating that the match variable can no longer be A1.

3.7.5 Strict Fields in Inhabitation Testing

The Inst function in Figure 3.10 takes strict fields into account during inhabitation testing, which is essential to conclude that the *v* function from page 19 is exhaustive. This trick was pioneered by Oury [2007], who uses it to check for unreachable cases in the presence of dependent types. Coverage checkers for strict and total programming languages usually implement inhabitation testing, but sometimes with less-than-perfect results. As two data points, Ryan decided to see how OCaml and Idris, two call-by-value languages that check for pattern-match coverage[10], would fare when checking functions like *v*:

```
(*OCaml*)
type void = |;;
let v (None : void option) : int = 0;;
let v' (o : void option) : int =
  match o with
  | None   -> 0
  | Some _ -> 1;;
```

```
-- Idris
v : Maybe Void → Int
v Nothing = 0
v' : Maybe Void → Int
v' Nothing = 0
v' (Just _) = 1
```

[10] Idris has separate compile-time and runtime semantics, the latter of which is call-by-value.

Both OCaml 4.10.0 and Idris 1.3.2 correctly mark their respective versions of v as exhaustive. OCaml also correctly warns that the Some case in v' is unreachable, while Idris emits no warnings for v' at all.

Section 3.2.7 also contains an example of a function f that LYG will fail to recognize as exhaustive due to LYG's conservative, fuel-based approach to inhabitation testing. Porting f to OCaml and Idris reveals that both languages will also conservatively claim that f is non-exhaustive:

```
(*OCaml*)                                  -- Idris
type t = MkT of t;;                        data T : Type where
let f (None : t option) : int = 0;;          MkT : T → T
                                           f : Maybe T → Int
                                           f Nothing = 0
```

Indeed, the warning that OCaml produces will cite

```
Some (MkT (MkT (MkT (MkT (MkT _)))))
```

as a case that is not matched, which suggests that OCaml may also be using a fuel-based approach. I believe these examples show that inhabitation testing is something that programming language implementors have discovered independently, but with varying degrees of success in putting into practice. I hope that LYG can bring this knowledge into wider use.

4

Abstracting Denotational Interpreters

A *static program analysis* infers facts about a program, such as "this program is well-typed", "this higher-order function is always called with argument $\bar{\lambda}x.x + 1$" or "this program never evaluates x". In a functional-language setting, such static analyses are often defined *compositionally* on the input term: the result of analysing a term is obtained by analysing its subterms separately and combining the results. For example, consider the claim "(*even* 42) has type Bool". Type analysis separately computes *even* : Int → Bool and 42 : Int, and then combines these results to produce the result type *even* 42 : Bool, without looking at the definition of *even* again.

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 subgoals 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, Pnueli, et al. 1978], reanalysing function bodies at call sites. Hence the new analysis becomes non-modular, leading to scalability problems for a compiler.

Contributions. In this chapter, I 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. My 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 and modular soundness proofs just like for AAM or big-step definitional interpreters [Darais, Labich, et al. 2017; Keidel, Poulsen, et al. 2018]. However, the shared, compositional structure enables a wide range of summary mechanisms in static analyses that I think are beyond the reach of non-compositional reachable-states abstractions like AAM.

- I use a concrete example (absence analysis) to argue for the usefulness of compositional, summary-based analysis in Section 4.1 and I demonstrate the difficulty of conducting an ad-hoc soundness proof wrt. a non-compositional small-step operational semantics.
- Section 4.3 walks through the definition of my generic denotational interpreter and its type class algebra in Haskell. I demonstrate the ease with which different instances of my interpreter endow the object language with call-by-name, call-by-need and call-by-value evaluation strategies, each producing (abstractions of) small-step 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 4.4.2 proves that the by-name and by-need instantiations are total by embedding the generic interpreter and its instances in Guarded Cubical Agda.

- Section 4.4.1 proves that the by-need instantiation of my 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 4.5, I recover summary-based usage analysis, a generalisation of absence analysis in Section 4.1. Further examples comprise Milner's Type Analysis and 0CFA control-flow analysis, demonstrating the wide range of applicability of my framework. To put my framework to the test in the real world, I refactored the Demand Analysis of the Glasgow Haskell Compiler into an abstract denotational interpreter.
- In Section 4.6, I 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 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 a soundness proof for usage analysis wrt. the by-name and by-need semantics in half a page, building on reusable semantics-specific theorems.
- I compare to the enormous body of related approaches in Section 4.7.

Acknowledgements. The work in this chapter is an extended version of Graf, Jones, et al. [2024]. For years, I have been working with Simon Peyton Jones on GHC's optimisation passes, its Demand Analysis in particular. This chapter establishes a framework as a first step towards describing the results of our work. Naturally, Simon was involved in iteratively improving the narrative of this chapter, however the technical contributions are exclusively my own. Our third author, Sven Keidel, helpfully offered to read our work with a fresh pair of eyes, pointing out many ways in which to improve the scientific writing and helped prioritising interesting contributions. We had fruitful discussions about how to modularise the soundness proofs via parametricity.

4.1 Problem Statement

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

4.1.1 Object Language

To set the stage, I start by defining the object language of this chapter, an *untyped* lambda calculus with ***recursive*** let bindings and algebraic data types:

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

This language is very similar to that of Launchbury [1993] and Sestoft [1997]. It is factored into *A-normal form*, that is, the arguments of applications are restricted to be variables, so the difference between lazy and eager semantics is manifest in the semantics of **let**. Note that $\bar{\lambda}x.x$ (with an overline) denotes syntax, whereas $\lambda x.\ x + 1$ denotes an anonymous mathematical function. In this section, only the highlighted parts are relevant and **let** is considered non-recursive, but the interpreter definition in Section 4.3 supports data types and recursive **let** as well. Throughout this chapter it is assumed that all bound program variables are distinct.

4.1.2 Absence Analysis

In order to define and explore absence analysis in this subsection, I 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 4.1 defines an absence analysis $\mathcal{A}[\![\mathsf{e}]\!]_\rho$ for lazy program semantics that conservatively approximates semantic absence. For illustrative purposes, my analysis definition only works for the special case of non-recursive **let**, but

$$
\begin{aligned}
a \in \mathsf{Absence} &::= \mathsf{A} \mid \mathsf{U} \\
\varphi \in \mathsf{Uses} &= \mathsf{Var} \to \mathsf{Absence} \\
\pi \in \mathsf{Args} &::= a \fatsemi \pi \mid \mathsf{Rep}\ a \\
\theta \in \mathsf{AbsTy} &::= \langle \varphi, \pi \rangle \\
\mathsf{Rep}\ a &\equiv a \fatsemi \mathsf{Rep}\ a
\end{aligned}
$$

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

$$
\begin{aligned}
\mathcal{A}[\![\mathsf{x}]\!]_\rho &= \rho(\mathsf{x}) \\
\mathcal{A}[\![\bar{\lambda}\mathsf{x}.\mathsf{e}]\!]_\rho &= \mathit{fun}_\mathsf{x}(\lambda\theta.\ \mathcal{A}[\![\mathsf{e}]\!]_{\rho[\mathsf{x} \mapsto \theta]}) \\
\mathcal{A}[\![\mathsf{e}\ \mathsf{x}]\!]_\rho &= \mathit{app}(\mathcal{A}[\![\mathsf{e}]\!]_\rho)(\rho(\mathsf{x})) \\
\mathcal{A}[\![\textbf{let}\ \mathsf{x} = \mathsf{e}_1\ \textbf{in}\ \mathsf{e}_2]\!]_\rho &= \mathcal{A}[\![\mathsf{e}_2]\!]_{\rho[\mathsf{x} \mapsto \mathsf{x} \,\&\, \mathcal{A}[\![\mathsf{e}_1]\!]_\rho]} \\
\\
\mathit{fun}_\mathsf{x}(f) &= \langle \varphi[\mathsf{x} \mapsto \mathsf{A}], \varphi(\mathsf{x}) \fatsemi \pi \rangle \\
&\quad \text{where } \langle \varphi, \pi \rangle = f(\langle [\mathsf{x} \mapsto \mathsf{U}], \mathsf{Rep}\ \mathsf{U} \rangle) \\
\mathit{app}(\langle \varphi_f, a \fatsemi \pi \rangle)(\langle \varphi_a, \_ \rangle) &= \langle \varphi_f \sqcup (a * \varphi_a), \pi \rangle \\
\\
\mathsf{A} * \varphi &= [] \\
\mathsf{U} * \varphi &= \varphi \\
\mathsf{x} \,\&\, \langle \varphi, \pi \rangle &= \langle \varphi[\mathsf{x} \mapsto \mathsf{U}], \pi \rangle
\end{aligned}
$$

Fig. 4.1: Absence analysis

later sections will assume recursive let semantics.[1] It takes an environment $\rho \in \mathsf{Var} \rightharpoonup \mathsf{AbsTy}$ containing absence information about the free variables of e and returns an *absence type* $\langle \varphi, \pi \rangle \in \mathsf{AbsTy}$; an abstract representation of e. The first component $\varphi \in \mathsf{Uses}$ of the absence type captures how e uses its free variables by associating an Absence flag with each variable. When $\varphi(\mathsf{x}) = \mathsf{A}$, then x is absent in e; otherwise, $\varphi(\mathsf{x}) = \mathsf{U}$ and x might be used in e. The second component $\pi \in \mathsf{Args}$ of the absence type describes how e uses actual arguments supplied at application sites. For example, function $f \triangleq \bar{\lambda}x.y$ has absence type $\langle [y \mapsto \mathsf{U}], \mathsf{A} \fatsemi \mathsf{Rep}\ \mathsf{U} \rangle$. Mapping $[y \mapsto \mathsf{U}]$ indicates that f may use its free variable y. The literal notation $[y \mapsto \mathsf{U}]$ maps any variable other than y to A. Furthermore, $\mathsf{A} \fatsemi \mathsf{Rep}\ \mathsf{U}$ indicates that f's first argument is absent and all further

[1] Given an order that I will define in due course, the generalised definition for recursive as well as non-recursive let is $\mathcal{A}[\![\textbf{let}\ \mathsf{x} = \mathsf{e}_1\ \textbf{in}\ \mathsf{e}_2]\!]_\rho = \mathcal{A}[\![\mathsf{e}_2]\!]_{\rho[\mathsf{x} \mapsto \mathsf{lfp}(\lambda\theta.\ \mathsf{x} \& \mathcal{A}[\![\mathsf{e}_1]\!]_{\rho[\mathsf{x} \mapsto \theta]})]}$.

arguments are potentially used. The element Rep U denotes an infinite repetition of U, as expressed by the non-syntactic equality Rep U ≡ U ⨾ Rep U.

Let us trace the analysis on the example expression $\mathsf{e} \triangleq \textbf{let}\ k = \bar{\lambda} y.\bar{\lambda} z.y\ \textbf{in}\ k\ x_1\ x_2$, where the initial environment for e, $\rho_\mathsf{e}(\mathsf{x}) \triangleq \langle [\mathsf{x} \mapsto \mathsf{U}], \mathsf{Rep\ U} \rangle$, declares the free variables of e with a pessimistic argument description Rep U.

$$
\begin{aligned}
& \mathcal{A}[\![\textbf{let}\ k = \bar{\lambda} y.\bar{\lambda} z.y\ \textbf{in}\ k\ x_1\ x_2]\!]_{\rho_\mathsf{e}} && (4.1)\\
& \quad \wr\ \text{Unfold } \mathcal{A}[\![\textbf{let}\ \mathsf{x} = \mathsf{e}_1\ \textbf{in}\ \mathsf{e}_2]\!].\ \text{NB: Lazy Let!}\ \int \\
&= \mathcal{A}[\![k\ x_1\ x_2]\!]_{\rho_\mathsf{e}[k \mapsto k \& \mathcal{A}[\![\bar{\lambda} y.\bar{\lambda} z.y]\!]_{\rho_\mathsf{e}}]} && (4.2)\\
& \quad \wr\ \text{Unfold } \mathcal{A}[\![\_]\!],\ \rho_1 \triangleq \rho_\mathsf{e}[k \mapsto k \& \mathcal{A}[\![\bar{\lambda} y.\bar{\lambda} z.y]\!]_{\rho_\mathsf{e}}]\ \int \\
&= app(app(\rho_1(k))(\rho_1(x_1)))(\rho_1(x_2)) && (4.3)\\
& \quad \wr\ \text{Unfold } \rho_1(k)\ \int \\
&= app(app(k\ \&\ \mathcal{A}[\![\bar{\lambda} y.\bar{\lambda} z.y]\!]_{\rho_1})(\rho_1(x_1)))(\rho_1(x_2)) && (4.4)\\
& \quad \wr\ \text{Unfold } \mathcal{A}[\![\bar{\lambda}\mathsf{x}.\mathsf{e}]\!] \text{ twice, } \mathcal{A}[\![\mathsf{x}]\!]\ \int \\
&= app(app(k\ \&\ fun_y(\lambda\theta_y.\ fun_z(\lambda\theta_z.\ \theta_y)))(\ldots))(\ldots) && (4.5)\\
& \quad \wr\ \text{Unfold } \textit{fun} \text{ twice, simplify}\ \int \\
&= app(app(\langle [k \mapsto \mathsf{U}],\ \mathsf{U} \between \mathsf{A} \between \mathsf{Rep\ U} \rangle)(\ \rho_1(x_1)\ ))(\ldots) && (4.6)\\
& \quad \wr\ \text{Unfold } \textit{app},\ \rho_1(x_1) = \rho_\mathsf{e}(x_1),\ \text{simplify}\ \int \\
&= app(\langle [k \mapsto \mathsf{U}, x_1 \mapsto \mathsf{U}],\ \mathsf{A} \between \mathsf{Rep\ U} \rangle)(\ \rho_1(x_2)\ ) && (4.7)\\
& \quad \wr\ \text{Unfold } \textit{app},\ \text{simplify}\ \int \\
&= \langle [k \mapsto \mathsf{U}, x_1 \mapsto \mathsf{U}], \mathsf{Rep\ U} \rangle && (4.8)
\end{aligned}
$$

Let us look at the steps in a bit more detail. Step (4.1) extends the environment with an absence type for the let right-hand side of k. The steps up until (4.5) successively expose applications of the *app* and *fun* helper functions applied to environment entries for the involved variables. Step (4.5) then computes the absence type $fun_y(\lambda\theta_y.\ fun_z(\lambda\theta_z.\ \theta_y)) = \langle [], \mathsf{U} \between \mathsf{A} \between \mathsf{Rep\ U} \rangle$. The Uses component is empty because $\bar{\lambda} y.\bar{\lambda} z.y$ has no free variables, and k & ... will add $[k \mapsto \mathsf{U}]$ as the single use. The *app* steps (4.6) and (4.7) simply zip up the uses of arguments $\rho_1(x_1)$ and $\rho_1(x_2)$ with the Absence flags in U ⨾ A ⨾ Rep U as highlighted, adding the Uses from $\rho_1(x_1) = \langle [x_1 \mapsto \mathsf{U}], \mathsf{Rep\ U} \rangle$ but *not* from $\rho_1(x_2)$, because the first actual argument (x_1) is used whereas the second (x_2) is absent. The join on Uses in the definition of *app* is defined pointwise from the order $\mathsf{A} \sqsubset \mathsf{U}$, i.e. $(\varphi_1 \sqcup \varphi_2)(\mathsf{x}) \triangleq \varphi_1(\mathsf{x}) \sqcup \varphi_2(\mathsf{x})$.

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

4.1.3 Compositionality, Summaries and Modularity

The absence type $\langle [], \mathsf{U} \fatsemi \mathsf{A} \fatsemi \mathsf{Rep}\ \mathsf{U} \rangle$ above is a finite *summary* of the lambda term $\bar{\lambda}y.\bar{\lambda}z.y$. Let me define what I mean by "summary" in order for us to understand what is so appealing about a summary-based analysis such as $\mathcal{A}[\![\_]\!]$.

Just as a denotational semantics, the interpreter $\mathcal{A}[\![\_]\!]$ *denotes* a term in a *semantic domain* (AbsTy). This interpretation is *compositional*: the denotation of a term is a recombination of the denotations of its subterms.

In order for a denotational semantics to faithfully and compositionally denote lambda terms, its semantic domain must contain infinite elements. However, every element of the semantic domain AbsTy of absence analysis is *finitely representable* (data, even!), so the denotation of lambda terms must be approximate in some sense. We call such a finitely representable and thus approximate denotation a *summary*.

The approximate nature of summaries is best appreciated when analysing beta redexes such as $\mathcal{A}[\![(\bar{\lambda}\mathrm{x.e})\ \mathrm{y}]\!]$, which invokes the *summary mechanism*. In the definition of $\mathcal{A}[\![\_]\!]$, I took care to explicate this mechanism via the adjoint functions *fun* and *app*. For the summary mechanism to be sound, we must have $\mathcal{A}[\![\mathrm{e[y/x]}]\!] \sqsubseteq \mathcal{A}[\![(\bar{\lambda}\mathrm{x.e})\ \mathrm{y}]\!]$ (where $\mathrm{e[y/x]}$ substitutes y for x in e).

To support efficient separate compilation, a compiler analysis must be *modular*, and summaries are indispensable in achieving that. Let us say that the example function $k = \bar{\lambda}y.\bar{\lambda}z.y$ is defined in module A and there is a use site $(k\ x_1\ x_2)$ in module B. Then a *modular analysis* must not reanalyse A.k at its use site in B. The analysis $\mathcal{A}[\![\_]\!]$ facilitates that easily, because it can serialise the AbsTy summary for k into module A's signature file.

The same way summaries enable efficient *inter*-module compilation, they enable efficient *intra*-module compilation: Compositionality implies that $\mathcal{A}[\![\textbf{let}\ f = \bar{\lambda}x.\mathrm{e}_{big}\ \textbf{in}\ f\ f\ f]\!]$ is a function of $\mathcal{A}[\![\bar{\lambda}x.\mathrm{e}_{big}]\!]$, and finite summaries prevent having to reanalyse e_{big} repeatedly for each call of f.

This is why summary-based analyses are great: they scale.

4.1.4 Summaries vs. Abstracting Abstract Machines

Now, instead of coming up with a summary mechanism, we could simply have "inlined" k during analysis of the example above to see that x_2 is absent in a simple first-order sense. The *call strings* approach to interprocedural program analysis [Sharir, Pnueli, et al. 1978] turns this idea into a static analysis, and the AAM recipe could be used to derive an absence analysis based on call strings that is sound by construction. Alas, following this path gives up on modularity, because a call-strings-based analysis would need to invoke the function $(\lambda\theta_y.\ \lambda\theta_z.\ \theta_y) : \mathsf{AbsTy} \to \mathsf{AbsTy} \to \mathsf{AbsTy}$ that describes k's inline expansion *at every use site*, leading to scalability problems in a compiler.

4.1.5 Problem: Proving Soundness of Summary-Based Analyses

In this subsection, I demonstrate the difficulty of proving soundness for summary-based analyses. For absence analysis, I have proved (in the Appendix) the following correctness statement:

□ 195 **Theorem 4.1** ($\mathcal{A}[\![\_]\!]$ infers absence)**.** *If* $\mathcal{A}[\![\mathsf{e}]\!]_{\rho_\mathsf{e}} = \langle\varphi, \pi\rangle$ *and* $\varphi(\mathsf{x}) = \mathsf{A}$*, then* x *is absent in* e.

As the first step, we must define precisely what absence (used in the theorem statement) *means.* One plausible definition is in terms of the standard operational semantics in Section 4.2:

Definition 4.2 (Absence)**.** *A variable* x *is* used *in an expression* e *if and only if there exists a trace* $(\textbf{let}\ \mathsf{x} = \mathsf{e}'\ \textbf{in}\ \mathsf{e}, \rho, \mu, \kappa) \hookrightarrow^* \ldots \xrightarrow{\mathit{Look}(\mathsf{x})} \ldots$ *that looks up the heap entry of* x*, i.e. it evaluates* x*. Otherwise,* x *is* absent *in* e.

Absence of x means that x is not looked up *regardless of the context in which* e *is used*, to justify rewrites via contextual improvement [Moran and Sands 1999]. Furthermore, we must prove that the summary mechanism approximates beta reduction, captured syntactically in the following *substitution lemma* [Pierce 2002]:

□ 187 **Lemma 4.3** (Substitution)**.** $\mathcal{A}[\![\mathsf{e}]\!]_{\rho[\mathsf{x}\mapsto\rho(\mathsf{y})]} \sqsubseteq \mathcal{A}[\![(\bar{\lambda}\mathsf{x}.\mathsf{e})\ \mathsf{y}]\!]_\rho$.

Definition 4.2 and the substitution Lemma 4.3 will make a reappearance in Section 4.6. They are necessary components of a soundness proof. Building on

these definitions, we may finally attempt the proof for Theorem 4.1. I suggest for the reader to have a cursory look at the Appendix, either by clicking on the link next to the Theorem statement or going to page 195. The proof is exemplary of far more ambitious proofs such as in Sergey, Vytiniotis, et al. [2017] and Breitner [2016, Section 4]. Though seemingly disparate, these proofs all follow an established preservation-style proof technique at heart. The proof of Sergey, Vytiniotis, et al. [2017] for a generalisation of $\mathcal{A}[\![\_]\!]$ is roughly structured as follows:

1. Instrument a standard call-by-need semantics (a variant of our reference semantics in Section 4.2) 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 4.2 carries the let-bound variable that is looked up.

2. Give a declarative type system that characterises the results of the analysis (i.e. $\mathcal{A}[\![\_]\!]$) in a lenient (upwards closed) way. In case of Theorem 4.1, I define an analysis function on machine configurations for the proof (Figure A.1).

3. Prove that evaluation of well-typed terms in the instrumented semantics is bisimilar to evaluation of the term in the standard semantics, i.e. does not get stuck when the standard semantics would not. A classic *logical relation* [Nielson et al. 1999]. In my case, I 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, Vytiniotis, et al. [2017] becomes non-deterministic. Does this instrumentation still express the desired semantic property?

- Step (2) all but duplicates a complicated analysis definition (i.e. $\mathcal{A}[\![\_]\!]$) into a type system (in Figure 7 of Sergey, Vytiniotis, et al. [2017]) 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.[2] Another page worth of Figures; the amount of duplicated proof artifacts is staggering. In my 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 in the proof of Sergey, Vytiniotis, et al. [2017] is the *substitution lemma* A.8 to be applied during beta reduction; exactly as in my proof.

- While proving that a single step $\sigma_1 \hookrightarrow \sigma_2$ preserves analysis information in step (3), I actually got stuck in the Upd case, and would need to redo the proof using step-indexing [Appel and McAllester 2001]. This case mutates the heap and thus is notoriously difficult; I give a proper account in Theorem 4.18.

  Although the proof in Sergey, Vytiniotis, et al. [2017] is perceived as detailed and rigorous, it is quite terse in the corresponding EUpd case of the single-step safety proof in lemma A.6.

There are two main problems to address, and I believe the first causes the second.

1. Although analysis and semantics are individually simple, it is conceptually difficult to connect them, causing an explosion of formal artefacts. This is because one is compositional while the other is not.

2. Compared to analysis and semantics, the soundness proof is rather complicated because it is *entangled*: 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.

Abstract interpretation [Cousot and Cousot 1977] provides a framework to tackle problem (2), but its systematic applications seem to require a structurally matching semantics. For example, the book of Cousot [2021] starts from a *compositional*, trace-generating semantics for an imperative first-order language to derive compositional analyses.

In this chapter, I present the ***denotational interpreter*** design pattern to solve both problems above. Inspired by Cousot, I define a ***compositional semantics***

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

that exhibits operational detail for higher-order languages; one with which it is possible to express *operational properties* such as *usage cardinality*, i.e. "e evaluates x at most u times", as required in Sergey, Vytiniotis, et al. [2017].[3]

The example of usage analysis in Section 4.5 (generalising $\mathcal{A}[\![\_]\!]$) demonstrates that we can ***define summary-based analyses*** as denotational interpreters.

Since both semantics and analysis are ***derived from the same generic interpreter***, solving problem (1), I can prove usage analysis to be an *abstract interpretation* of call-by-need semantics. Doing so disentangles the preservation proof such that the proof for usage analysis in Corollary 4.21 takes no more than a semantic substitution lemma and a bit of plumbing, solving problem (2). Intriguingly, Corollary 4.21 can be proved without referring to the shared interpreter definition or the Galois connection at all, by appealing to parametricity [Reynolds 1983] to prove Lemma 4.19. This suggests that my approach scales to large interpreters such as for WebAssembly [Brandl et al. 2023].

4.2 Reference Semantics: Lazy Krivine Machine

Before I introduce my novel denotational interpreters, let us recall the semantic ground truth of this work and others [Breitner 2016; Sergey, Vytiniotis, et al. 2017]: The Mark II machine of Sestoft [1997] given in Figure 4.2, 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.

The configurations σ in this transition system resemble abstract machine states, consisting of a control expression e, an environment ρ mapping lexically-scoped variables to their current heap address, a heap μ listing a closure for each address, and a stack of continuation frames κ. There is one harmless non-standard extension: For Look transitions, I 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 Let_1, Look and Upd rules.

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

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

$$\boxed{\sigma_1 \hookrightarrow \sigma_2}$$

$$\textsc{Let}_1 \frac{\mathsf{a} \notin \mathrm{dom}(\mu) \qquad \rho' = \rho[\mathsf{x} \mapsto \mathsf{a}]}{(\mathbf{let}\ \mathsf{x} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2, \rho, \mu, \kappa) \hookrightarrow (\mathsf{e}_2, \rho', \mu[\mathsf{a} \mapsto (\mathsf{x}, \rho', \mathsf{e}_1)], \kappa)}$$

$$\textsc{App}_1 \frac{\mathsf{a} = \rho(\mathsf{x})}{(\mathsf{e}\ \mathsf{x}, \rho, \mu, \kappa) \hookrightarrow (\mathsf{e}, \rho, \mu, \mathbf{ap}(\mathsf{a}) \cdot \kappa)}$$

$$\textsc{App}_2 \frac{}{(\bar{\lambda}\mathsf{x}.\mathsf{e}, \rho, \mu, \mathbf{ap}(\mathsf{a}) \cdot \kappa) \hookrightarrow (\mathsf{e}, \rho[\mathsf{x} \mapsto \mathsf{a}], \mu, \kappa)}$$

$$\textsc{Look}(\mathsf{y}) \frac{\mathsf{a} = \rho(\mathsf{x}) \qquad (\mathsf{y}, \rho', \mathsf{e}) = \mu(\mathsf{a})}{(\mathsf{x}, \rho, \mu, \kappa) \hookrightarrow (\mathsf{e}, \rho', \mu, \mathbf{upd}(\mathsf{a}) \cdot \kappa)}$$

$$\textsc{Upd} \frac{\mu(\mathsf{a}) = (\mathsf{x}, \_, \_)}{(\mathsf{v}, \rho, \mu, \mathbf{upd}(\mathsf{a}) \cdot \kappa) \hookrightarrow (\mathsf{v}, \rho, \mu[\mathsf{a} \mapsto (\mathsf{x}, \rho, \mathsf{v})], \kappa)}$$

$$\textsc{Case}_1 \frac{}{(\mathbf{case}\ \mathsf{e}_s\ \mathbf{of}\ \overline{K\ \overline{\mathsf{x}} \to \mathsf{e}_r}, \rho, \mu, \kappa) \hookrightarrow (\mathsf{e}_s, \rho, \mu, \mathbf{sel}(\rho, \overline{K\ \overline{\mathsf{x}} \to \mathsf{e}_r}) \cdot \kappa)}$$

$$\textsc{Case}_2 \frac{K_i = K',\ \overline{\mathsf{a} = \rho(\mathsf{y})}}{(K'\ \overline{y}, \rho, \mu, \mathbf{sel}(\rho', \overline{K\ \overline{\mathsf{x}} \to \mathsf{e}}) \cdot \kappa) \hookrightarrow (\mathsf{e}_i, \rho'[\overline{\mathsf{x}_i \mapsto \mathsf{a}}], \mu, \kappa)}$$

Fig. 4.2: Lazy Krivine transition semantics $\hookrightarrow$

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

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

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

$$
\begin{array}{lll}
(\mathbf{let}\ i = \bar{\lambda}x.x\ \mathbf{in}\ i\ i, [], [], \mathbf{stop}) & & \text{where} \\
\xhookrightarrow{\textsc{Let}_1} & (i\ i, \rho_1, \mu, \mathbf{stop}) & \rho_1 = [i \mapsto \mathsf{a}_1] \\
\xhookrightarrow{\textsc{App}_1} & (i, \rho_1, \mu, \kappa) & \mu = [\mathsf{a}_1 \mapsto (i, \rho_1, \bar{\lambda}x.x)] \\
\xhookrightarrow{\textsc{Look}(i)} & (\bar{\lambda}x.x, \rho_1, \mu, \mathbf{upd}(\mathsf{a}_1) \cdot \kappa) & \kappa = \mathbf{ap}(\mathsf{a}_1) \cdot \mathbf{stop} \\
\xhookrightarrow{\textsc{Upd}} & (\bar{\lambda}x.x, \rho_1, \mu, \kappa) & \\
\xhookrightarrow{\textsc{App}_2} & (x, \rho_2, \mu, \mathbf{stop}) & \rho_2 = [i \mapsto \mathsf{a}_1, x \mapsto \mathsf{a}_1] \\
\xhookrightarrow{\textsc{Look}(i)} & (\bar{\lambda}x.x, \rho_1, \mu, \mathbf{upd}(\mathsf{a}_1) \cdot \mathbf{stop}) & \\
\xhookrightarrow{\textsc{Upd}} & (\bar{\lambda}x.x, \rho_1, \mu, \mathbf{stop}) &
\end{array}
\tag{4.1}
$$

The corresponding by-name trace simply omits the highlighted update steps. The last $\textsc{Look}(i)$ transition exemplifies that the lambda-bound variable in control differs from the let-bound variable used to identify the heap entry.

The second example evaluates $\mathrm{e} \triangleq \textbf{let } i = (\bar{\lambda} y.\bar{\lambda} x.x)\ i \textbf{ in } i\ i$, demonstrating recursive **let** semantics and memoisation of i:

$$
\begin{array}{lll}
(\mathrm{e}, [], [], \textbf{stop}) & & \text{where} \\
\xhookrightarrow{\textsc{Let}_1} & (i\ i, \rho_1, \mu_1, \textbf{stop}) & \rho_1 = [i \mapsto \mathsf{a}_1] \\
\xhookrightarrow{\textsc{App}_1} & (i, \rho_1, \mu_1, \textbf{ap}(\mathsf{a}_1) \cdot \textbf{stop}) & \mu_1 = [\mathsf{a}_1 \mapsto (i, \rho_1, (\bar{\lambda} y.\bar{\lambda} x.x)\ i)] \\
\xhookrightarrow{\textsc{Look}(i)} & ((\bar{\lambda} y.\bar{\lambda} x.x)\ i, \rho_1, \mu_1, \kappa) & \kappa = \textbf{upd}(\mathsf{a}_1) \cdot \textbf{ap}(\mathsf{a}_1) \cdot \textbf{stop} \\
\xhookrightarrow{\textsc{App}_1} & (\bar{\lambda} y.\bar{\lambda} x.x, \rho_1, \mu_1, \textbf{ap}(\mathsf{a}_1) \cdot \kappa) & \\
\xhookrightarrow{\textsc{App}_2} & (\bar{\lambda} x.x, \rho_2, \mu_1, \kappa) & \rho_2 = [i \mapsto \mathsf{a}_1, y \mapsto \mathsf{a}_1] \\
\xhookrightarrow{\textsc{Upd}} & (\bar{\lambda} x.x, \rho_2, \mu_2, \textbf{ap}(\mathsf{a}_1) \cdot \textbf{stop}) & \mu_2 = [\mathsf{a}_1 \mapsto (i, \rho_2, \bar{\lambda} x.x)] \\
\xhookrightarrow{\textsc{App}_2} & (x, \rho_3, \mu_2, \textbf{stop}) & \rho_3 = [i \mapsto \mathsf{a}_1, y \mapsto \mathsf{a}_1, x \mapsto \mathsf{a}_1] \\
\xhookrightarrow{\textsc{Look}(i)} & (\bar{\lambda} x.x, \rho_2, \mu_2, \textbf{upd}(\mathsf{a}_1) \cdot \textbf{stop}) & \\
\xhookrightarrow{\textsc{Upd}} & (\bar{\lambda} x.x, \rho_2, \mu_2, \textbf{stop}) &
\end{array}
\tag{4.2}
$$

4.3 A Denotational Interpreter

In this section, I present the main contribution of this chapter, namely a generic *denotational interpreter*[4] for a functional language which I 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. Other semantic domains give rise to useful *summary-based* static analyses such as usage analysis in Section 4.5. The major contribution of my framework is that the derived summary-based analyses may observe operational detail in an intuitive and semantically meaningful way. Adhering to my design pattern pays off in that it enables sharing of soundness proofs, thus drastically simplifying the soundness proof obligation per derived analysis (Section 4.6).

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

```
data Exp = Var Name | Lam Name Exp | App Exp Name
         | Let Name Exp Exp | ConApp Tag [Name] | Case Exp Alts
type Name = String
type Alts = Tag :⇀ ([Name], Exp)
data Tag  = ...; conArity :: Tag → Int
```

Fig. 4.3: Syntax

```
type (:⇀) = Map
ε          :: Ord k ⇒ k :⇀ v
_[_ ↦ _]   :: Ord k ⇒ (k :⇀ v) → k → v → (k :⇀ v)
_[_ ↦ _]   :: Ord k ⇒ (k :⇀ v) → [k] → [v] → (k :⇀ v)
(!)        :: Ord k ⇒ (k :⇀ v) → k → v
dom        :: Ord k ⇒ (k :⇀ v) → Set k
(∈)        :: Ord k ⇒ k → Set k → Bool
(◁)        :: (b → c) → (a :⇀ b) → (a :⇀ c)
assocs     :: (k :⇀ v) → [(k, v)]
```

Fig. 4.4: Environments

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

I extract from this document runnable Haskell files. Furthermore, the (terminating) interpreter outputs are directly generated from this extract. For completenes, I repeat the full definitions in Appendix D.

4.3.1 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 → D. One of the main features of these semantic domains is that they lack *operational*, or, *intensional detail* that is unnecessary to assigning each observationally distinct expression a distinct meaning. For example, it is not possible to observe evaluation cardinality, which is the whole point of analyses such as usage analysis (Section 4.5).

A distinctive feature of my work is that the semantic domains are instead *traces* that describe the *steps* taken by an abstract machine, and that *end* in semantic values. It is possible to describe usage cardinality as a property of the traces thus generated, as required for a soundness proof of usage analysis. I choose $\mathsf{D_{na}}$, defined below, as the first example of such a semantic domain, because it is simple and illustrative of the approach. Instantiated at $\mathsf{D_{na}}$, the generic interpreter will produce precisely the traces of the by-name variant of the Krivine machine in Figure 4.2.

```
type Dna = D T
type D τ = τ (Value τ);
data T v = Step Event (T v) | Ret v
data Event = Look Name | Upd | App1 | App2 | Let1 | Case1 | Case2
data Value τ = Stuck | Fun (D τ → D τ) | Con Tag [D τ]

instance Monad T where
  return v = Ret v
  Ret v >>= k = k v
  Step e τ >>= k = Step e (τ >>= k)
```

A trace T either returns a value (Ret) or makes a small-step transition (Step). Each step Step *ev rest* is decorated with an event *ev*, which describes what happens in that step. For example, event Look *x* describes the lookup of variable *x* :: Name in the environment. Note that the choice of Event is use-case (i.e. analysis) specific and suggests a spectrum of intensionality, with **data** Event = Unit on the more abstract end of the spectrum and arbitrary syntactic detail attached to each of Event's constructors at the intensional end of the spectrum.[5]

A trace in $\mathsf{D_{na}}$ = T (Value T) eventually terminates with a Value that is either stuck (Stuck), a function waiting to be applied to a domain value (Fun), or a constructor application giving the denotations of its fields (Con). Let me postpone worries about well-definedness and totality of this encoding to Section 4.4.2.

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

4.3.2 The Interpreter

Traditionally, a denotational semantics is expressed as a mathematical function, often written $[\![e]\!]_\rho$, to give an expression $e :: \mathsf{Exp}$ a meaning, or *denotation*, in terms of some semantic domain D. The environment $\rho :: \mathsf{Name} :\rightharpoonup \mathsf{D}$ gives meaning to the free variables of e, by mapping each free variable to its denotation in D. I sketch the Haskell encoding of Exp in Figure 4.3 and the API of environments and sets in Figure 4.4. For concise notation, I will use a small number of infix operators: $(:\rightharpoonup)$ as a synonym for finite Maps, with $m\,!\,x$ for looking up x in m, ε for the empty map, $m[x \mapsto d]$ for updates, *assocs* m for a list of key-value pairs in m, $f \lhd m$ for mapping f over every value in m, *dom* m for the set of keys present in the map, and $(\in)$ for membership tests in that set.

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

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

I have parameterised the semantic domain d over three type classes — Trace, Domain and HasBind — whose signatures are given in Figure 4.6.[6] Each of the three type classes offer knobs that I will tweak to derive different evaluation strategies as well as static analyses.

Figure 4.5 gives the complete definition of $\mathcal{S}[\![\_]\!]\_$, with type class instances for the domain $\mathsf{D_{na}}$ of Section 4.3.1 in Figure 4.7. Together this is enough to actually run the denotational interpreter to produce traces. I use *read* $:: \mathsf{String} \to \mathsf{Exp}$ as a parsing function, and a Show instance for $\mathsf{D}\ \tau$ that displays traces. For example, we can evaluate the expression **let** $i = \bar{\lambda} x.x$ **in** $i\ i$ like this:

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

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

where $\langle\lambda\rangle$ means that the trace ends in a Fun value. We cannot generally print $\mathsf{D_{na}}$ or Functions thereof, but in this case the result would be the value $\bar{\lambda} x.x$.

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

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

Fig. 4.5: Denotational Interpreter

This is in direct correspondence to the earlier call-by-name small-step trace (4.1) on page 81.

The definition of $\mathcal{S}[\![\_]\!]\_$, given in Figure 4.5, is by structural recursion over the input expression. For example, to get the denotation of Lam *x body*, we must recursively invoke $\mathcal{S}[\![\_]\!]\_$ on *body*, extending the environment to bind *x* to its denotation. We wrap that body denotation in *step* App$_2$, to prefix the trace of *body* with an App$_2$ event whenever the function is invoked, where *step* is a method of class Trace. Finally, we use *fun* to build the returned denotation; the details necessarily depend on the Domain, so *fun* is a method of class Domain. While the lambda-bound $x :: \text{Name}$ passed to *fun* is ignored in the Domain $\text{D}_{\mathbf{na}}$ instance of the concrete by-name semantics, it is useful for abstract domains such as that of usage analysis (Section 4.5). (I refrain from passing field binders or other syntactic tokens in *select* and let binders in *bind* as well, because the analyses considered do not need them.) The other cases follow a similar pattern; they each do some work, before handing off to type class methods to do the domain-specific work.

The HasBind type class defines a particular *evaluation strategy*, as we shall see in Section 4.3.3. The *bind* method of HasBind is used to give meaning to recursive let bindings: it takes two functionals for building the denotation of the

```
class Trace d where
  step :: Event → d → d

class Domain d where
  stuck :: d
  fun :: Name → (d → d) → d
  apply :: d → d → d

  con :: Tag → [d] → d
  select :: d → (Tag :⇀ ([d] → d)) → d
```

Fig. 4.6: Interface of the semantic domain

```
instance Trace (T v) where
  step = Step

instance Monad τ ⇒ Domain (D τ) where
  stuck = return Stuck
  fun _ f = return (Fun f)
  apply d a = d >>= λv → case v of
    Fun f → f a
    _     → stuck
  con k ds = return (Con k ds)
  select dv alts = dv >>= λv → case v of
    Con k ds | k ∈ dom alts → (alts ! k) ds
    _                       → stuck

class HasBind d where
  bind :: (d → d) → (d → d) → d

instance HasBind Dna where
  bind rhs body = let d = rhs d in body d
```

Fig. 4.7: By-name instance for $\mathsf{D}_{\mathbf{na}}$

right-hand side and that of the let body, given a denotation for the right-hand side. The concrete implementation for *bind* given in Figure 4.7 computes a d such that $d = rhs\ d$ and passes the recursively-defined d to *body*.[7] 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. The Appendix contains examples of eager evaluation strategies that will yield from d inside *bind* instead of calling *body* immediately.

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

```
λ> takeT 5 $ 𝒮⟦read "let x = x in x"⟧ε :: T (Maybe (Value T))
```

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

```
λ> takeT 7 $ 𝒮⟦read "let w = λy. y y in w w"⟧ε :: T (Maybe (Value T))
```

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

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

The reason $\mathcal{S}[\![\_]\!]\_$ is productive is due to the coinductive nature of T's definition in Haskell.[8] Productivity requires that the monadic bind operator ($\gg=$) for T guards the recursion, as in the delay monad of Capretta [2005].

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

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

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

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

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

4.3.3 More Evaluation Strategies

By varying the HasBind instance of the 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 without changing the interpreter definition, supporting the claim that the denotational interpreter design pattern is equally suited to model lazy as well as strict semantics. More importantly, in order to prove usage analysis sound wrt. by-need evaluation in Section 4.6, we need to define a semantic domain for call-by-need! It turns out that the interpreter thus derived is the — to my knowledge — first provably adequate denotational semantics for call-by-need (Section 4.4.1).

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

Call-by-name

Let us redefine by-name semantics via the ByName *trace transformer* in Figure 4.8, so called because ByName τ inherits its Monad and Trace instance from τ and in reminiscence of Darais, Might, et al. [2015]. The old $\mathsf{D}_{\mathbf{na}}$ can be recovered as D (ByName T) and I refer to its interpreter instance as $\mathcal{S}_{\mathbf{name}}[\![e]\!]_\rho$.

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

```
𝒮_name⟦e⟧ρ = 𝒮⟦e⟧ρ :: D (ByName T)
newtype ByName τ v = ByName {unByName :: τ v}
instance Monad τ ⇒ Monad (ByName τ) where
  return = ByName ∘ return
  ByName m >>= f = ByName (m >>= unByName ∘ f)
instance Trace (τ v) ⇒ Trace (ByName τ v) where
  step e = ByName ∘ step e ∘ unByName
instance HasBind (D (ByName τ)) where
  bind rhs body = let d = rhs d in body d
```

Fig. 4.8: Redefinition of call-by-name semantics from Figure 4.7

Call-by-need

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

My trace transformer ByNeed in Figure 4.9 solves this type equation via $\theta \triangleq$ ByNeed T. It embeds a standard state transformer monad, whose key operations *get* and *put* are given in Figure 4.9.

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

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

Fig. 4.9: Call-by-need

The key part is again the implementation of HasBind for D (ByNeed τ), because that is the only place where thunks are allocated. The implementation of *bind* designates a fresh heap address a to hold the denotation of the right-hand side. Both *rhs* and *body* are called with *fetch a*, a denotation that looks up a in the heap and runs it. If I were to omit the *memo a* action explained next, I would thus have recovered another form of call-by-name semantics based on mutable state instead of guarded fixpoints such as in ByName and ByValue. The whole purpose of the *memo a d* combinator then is to *memoise* the computation of d the first time we run the computation, via *fetch a* in the Var case of $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_$. So *memo a d* yields from d until it has reached a value, and then *upd*ates the heap after an additional Upd step. Repeated access to the same variable will run the replacement *memo a* (*return v*), which immediately yields v after performing a *step* Upd that does nothing.[9]

Although the code is carefully written, it is worth stressing how compact and expressive it is. I was able to move from traces to stateful traces just by wrapping traces T in a state transformer ByNeed, without modifying the main $\mathcal{S}[\![\_]\!]_\_$ function at all. In doing so, I provide the simplest encoding of a denotational by-need semantics that I know of.

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

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

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

This trace is in clear correspondence to the earlier by-need LK trace (4.2). 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 $\textsc{App}_1 \hookrightarrow \textsc{App}_2$ before producing a value. This work is cached, so that the second Look bracket does not do any beta reduction.

The examples so far suggest that $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_$ agrees with the LK machine on *many* programs. The next section proves that $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_$ agrees with the LK machine on *all* programs, including ones that diverge. However, I will first

[9] More serious semantics would omit updates after the first evaluation as an *optimisation*, i.e. update with $\mu[a \mapsto \mathit{return}\ v]$, but doing so complicates relating the semantics to Figure 4.2, where omission of update frames for values behaves differently. For now, my goal is not to formalise this optimisation, but rather to show adequacy wrt. an established semantics.

```
𝒮value⟦e⟧ρ = 𝒮⟦e⟧ρ :: D (ByValue T)
newtype ByValue τ v = ByValue {unByValue :: τ v}
instance Monad τ ⇒ Monad (ByValue τ) where ...
instance Trace (τ v) ⇒ Trace (ByValue τ v) where ...

class Extract τ where getValue :: τ v → v
instance Extract T where
  getValue (Ret v)    = v
  getValue (Step _ τ) = getValue τ

instance (Trace (D (ByValue τ)), Monad τ, Extract τ)
         ⇒ HasBind (D (ByValue τ)) where
  bind rhs body = step Let0 $ do
    let d = rhs (return v)              :: D (ByValue τ)
        v = getValue (unByValue d) :: Value (ByValue τ)
    v1 ← d
    body (return v1)
```

Fig. 4.10: Call-by-value

demonstrate that my interpreter can be instantiated to different call-by-value semantics as well.

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.

To show that my denotational interpreter pattern equally well applies to such by-value evaluation strategies, I will introduce three more concrete semantic domain instances for call-by-value in the following subsections. The first of these semantics corresponds to the straightforward intuition for by-value evaluation. Unfortunately, this instance turns out to be partial (i.e. it may loop for some inputs). I will fix this by adopting higher-order state such as for call-by-need, yielding the second by-value semantics. The third and final by-value semantics is the clairvoyant semantics of Hackett and Hutton [2019], who provide a clever semantics that is cost equivalent to call-by-need, yet abstains from higher-order mutable state. It turns out that the clairvoyant interpreter is partial as well, and it is unclear how to fix it.

The first call-by-value evaluation strategy is implemented with the ByValue trace transformer shown in Figure 4.10. Function *bind* defines a denotation $d :: \mathsf{D}\ (\mathsf{ByValue}\ \tau)$ of the right-hand side by mutual recursion with its returned value $v :: \mathsf{Value}\ (\mathsf{ByValue}\ \tau)$ that I will discuss shortly.

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

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

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 4.5.

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

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

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

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

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

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

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

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

Lazy Initialisation and Black-holing

Recall that my simple ByValue transformer above yields a potentially looping interpreter. Typical strict languages work around this issue in either of two

```
𝒮vinit⟦e⟧ρ(μ) = unByVInit (𝒮⟦e⟧ρ :: D (ByVInit T)) μ
newtype ByVInit τ v
  = ByVInit { unByVInit :: Heap (ByVInit τ) → τ (v, Heap (ByVInit τ)) }
instance (Monad τ, ∀v. Trace (τ v)) ⇒ HasBind (D (ByVInit τ)) where
  bind rhs body = do μ ← get
                     let a = nextFree μ
                     put μ[a ↦ stuck]
                     step Let0 (memo a (rhs (fetch a))) >>= body ∘ return
```

Fig. 4.11: Call-by-value with lazy initialisation

```
𝒮clair⟦e⟧ρ = runClair $ 𝒮⟦e⟧ρ :: T (Value (Clairvoyant T))
data Fork f a = Empty | Single a | Fork (f a) (f a)
data ParT m a = ParT (m (Fork (ParT m) a))
instance Monad τ ⇒ Alternative (ParT τ) where
  empty = ParT (pure Empty); l <|> r = ParT (pure (Fork l r))
newtype Clairvoyant τ a = Clairvoyant (ParT τ a)
runClair :: D (Clairvoyant T) → T (Value (Clairvoyant T))
instance (Extract τ, Monad τ, ∀v. Trace (τ v))
         ⇒ HasBind (D (Clairvoyant τ)) where
  bind rhs body = Clairvoyant (skip <|> let') >>= body
    where skip = return (Clairvoyant empty)
          let' = fmap return $ step Let0 $ ... fix ... rhs ... getValue ...
```

Fig. 4.12: Clairvoyant Call-by-value

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 can recover a total interpreter using the semantics of Nakata [2010], building on the same encoding as ByNeed and initialising the heap with a *black hole* [Peyton Jones 1992] *stuck* in *bind* as in Figure 4.11.

```
λ> 𝒮vinit⟦read "let x = x in x x"⟧ε(ε) :: T (Value _, Heap _)
```

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

Clairvoyant Call-by-value

Clairvoyant call-by-value [Hackett and Hutton 2019] is an alternative to call-by-need semantics that exploits non-determinism and a cost model to absolve of the heap. We can instantiate my interpreter to generate the shortest clairvoyant call-by-value trace as well, as sketched out in Figure 4.12. Doing so yields an evaluation strategy that either skips or speculates let bindings, depending on whether or not the binding is going to be needed in the future:

$$\lambda\text{>}\ \mathcal{S}_{\textbf{clair}}[\![\mathit{read}\ \texttt{"let f = λx.x in let g = λy.f in g"}]\!]_{\varepsilon}$$

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

$$\lambda\text{>}\ \mathcal{S}_{\textbf{clair}}[\![\mathit{read}\ \texttt{"let f = λx.x in let g = λy.f in g g"}]\!]_{\varepsilon}$$

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

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

$$\lambda\text{>}\ \mathcal{S}_{\textbf{clair}}[\![\mathit{read}\ \texttt{"let i = Z() in let w = λy.y y in w w"}]\!]_{\varepsilon}$$

```
^CInterrupted
```

The program diverges without producing even a prefix of a trace because the binding for i might be needed at an unknown point in the future (a *liveness property* [Lamport 1977] and hence impossible to verify at runtime). This renders Clairvoyant call-by-value inadequate for verifying properties of infinite executions.

4.4 Totality and Semantic Adequacy

In this section, I prove that $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_$ produces small-step traces of the lazy Krivine machine and is indeed a *denotational semantics*.[10] Excitingly, to my knowledge, $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_$ is the first denotational call-by-need semantics that was proven so!

Specifically, denotational semantics must be total and adequate. *Totality* says that the interpreter is well-defined for every input expression and *adequacy* says that the interpreter produces similar traces as the reference semantics. This is an important result because it allows us to switch between operational reference semantics and denotational interpreter as needed, thus guaranteeing compatibility of definitions such as absence in Definition 4.2.

I will start with an informal overview of the results in Sections 4.4.1 and 4.4.2 before giving a formal account culminating in Sections 4.4.5 and 4.4.6. As before, all (pen-and-paper) proofs omitted in the main body can be found in the Appendix.

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

For proving adequacy of $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_$, I will give a function α from small-step traces in the lazy Krivine machine (Figure 4.2) to denotational traces T, with Events and all, such that

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

where $\mathit{init}(\mathrm{e}) \hookrightarrow \ldots$ denotes the *maximal* (i.e. longest possible) LK trace evaluating the closed expression e. For example, for the LK trace (4.2) on page 82, α produces the trace at the end of Section 4.3.3 on page 91.

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

□ 197 **Theorem 4.4** (Strong Adequacy). *Let e be a closed expression, $\tau \triangleq \mathcal{S}_{\mathbf{need}}[\![e]\!]_\varepsilon(\varepsilon)$ the denotational by-need trace and $\mathit{init}(\mathrm{e}) \hookrightarrow \ldots$ the maximal LK trace. Then*

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

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

- *every* $ev :: \mathsf{Event}$ *in* $\tau = \overline{\mathsf{Step}\ ev\ \ldots}$ *refers to a transition rule in* $init(\mathrm{e}) \hookrightarrow \ldots$.

Proof sketch. We formally define $\alpha((\sigma_i)_{i \in \overline{n}}) \triangleq \alpha_{\mathbb{S}^\infty}((\sigma_i)_{i \in \overline{n}}, \mathbf{stop})$, where $\alpha_{\mathbb{S}^\infty}$ is defined in Figure 4.13 on page 106, and prove $\alpha(init(\mathrm{e}) \hookrightarrow \ldots) = \mathcal{S}_{\mathbf{need}}[\![e]\!]_\varepsilon(\varepsilon)$. Then it suffices to prove that α preserves the observable properties of interest.

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

Theorem 4.5 (Totality)**.** *The interpreters* $\mathcal{S}_{\mathbf{name}}[\![e]\!]_\rho$ *and* $\mathcal{S}_{\mathbf{need}}[\![e]\!]_\rho(\mu)$ *are defined for every* e, ρ, μ.

Proof sketch. In the Appendix on page 213, I provide an implementation of the generic interpreter $\mathcal{S}[\![\_]\!]\_$ and its instances at ByName and ByNeed in Guarded Cubical Agda, which offers a total type theory with *guarded recursive types* [Møgelberg and Veltri 2019]. Agda enforces that all encodable functions are total, therefore $\mathcal{S}_{\mathbf{name}}[\![\_]\!]\_$ and $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$ must be total as well.

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

This concludes the high-level, informal discussion of adequacy and totality results for $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$. We will now take a more in-depth tour to justify these claims. This tour will start by recalling the limitations of inductive and coinductive definitions when it comes to formalising programming language semantics in Section 4.4.3. Section 4.4.4 then explains how guarded recursive types address these limitations and hence are a good fit to model denotational semantics. Finally, Section 4.4.5 will describe how $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$ can be encoded in Guarded Cubical Agda. While the previous three subsections motivate and introduce definitions by guarded recursion in some detail, the adequacy proof in Section 4.4.6 showcases associated proofs by Löb induction. I will use both guarded recursion and Löb induction extensively in many proofs of Section 4.6.

4.4.3 Limitations of Induction and Coinduction

Let us first recall what problem coinductive types solve and why their use in formalisation is comparatively rare compared to inductive types.

It is common in functional programming languages and theorem provers to define functions by recursion on an inductive data type argument. Such functions are automatically total, because inductive data is always of finite depth, admitting a termination proof by *well-founded induction* on that depth.

But many total functions prevalent in lazy programming languages, such as *map* (+2) :: [Int] → [Int] in Haskell, *are not recursive in this sense*! The reason for that is that lazy input data such as the infinite list [1..] which evaluates to [1, 2, 3, ...] can be of infinite depth, hence violating the finite depth precondition. That is, a direct proof by induction that *list* ≜ *map* (+2) [1..] is a total definition is not possible! It *is* total by *coinduction*, though, because the definition of *map* (+2) is *productive*: To evaluate *head list*, only a finite computation (1 + 2) needs to be carried out, and similarly for *list* !! 10, or any finite prefix of *list*. This is because the recursive call in the definition of *map* is *guarded* by the list constructor (:):

$$map\ f\ (x : xs) = f\ x : map\ f\ xs$$

While induction defines total objects (e.g. functions, proofs) by *destructing* an inductive *input* value, coinduction defines total objects by *constructing* a coinductive *output* value. Induction permits recursive calls only on the *input's parts* at *any position* in the function body, while coinduction permits recursive calls only in *guarded position*, such as in recursive fields of a coinductive data constructor, but with *any input* whatsoever.

It is fairly simple for a human or a computer to check what constitutes a part of an inductive value and thus what is a valid inductive definition, but it is far more complicated to check what constitutes a valid coinductive definition. Hence, although most theorem provers *do* admit definitions by coinduction such as *map* that satisfy simple syntactic productivity checks [Coquand 1994], syntactic productivity is easily defeated by refactorings such as extracting local bindings:

$$\begin{aligned} &map2\ f\ [\,] &&= [\,] \\ &map2\ f\ (x : xs) &&= f\ x : rest \\ &\quad \textbf{where}\ rest = map2\ f\ xs \end{aligned}$$

Here, *rest* occurs in guarded position and hence the recursive call to *map2* occurs in guarded position as well, but guardedness of the recursive call is no longer

syntactically evident and hence rejected by theorem provers such as Rocq[11] or Agda.

That is in constrast to inductive definitions, where it is simple to anticipate the arguments of recursive calls. The recursive call to *map2* is still decreasing in the input list, and it is often simple enough to leave the code in a form where the decreasing recursive call is evident. So it is fair to say that ***syntactic productivity checks are a severe limitation*** of current implementations of coinduction and render coinductive definitions far less useful than inductive definitions.

This is particularly embarassing for expressing dynamic processes such as program semantics, because their natural implementation is in terms of potentially infinite program traces which are best expressed coinductively.

In fact, our data type T is exactly such a coinductive data type, and hence we would like $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$ to be a coinductive definition as well. It is however impossible to show that $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$ syntactically guards all its recursive calls, because we do not even see the Step constructor syntactically in the definition of $\mathcal{S}[\![\_]\!]\_$, only calls to the type class method *step*! Thus, to prove $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$ total by coinduction in Rocq or Agda, we would need to manually specialise and inline many type class methods into $\mathcal{S}[\![\_]\!]\_$. Alas, any such manual transformation diminishes the confidence in the proof method!

There is another limitation why $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$ cannot easily be proven total by coinduction: Recall that D τ = τ (Value τ). Finite traces in the semantic domain $\mathsf{D}_{\mathbf{ne}}$ end in a $\mathsf{Value}_{\mathbf{ne}}$, and the data constructor Fun :: (D τ → D τ) → Value τ has a ***negative recursive occurrence*** of Value τ! This constructor is disallowed in inductive as well as traditional coinductive data type definitions, which is one reason why denotational semantics traditionally made use of algebraic domain theory [Scott 1970], sized types [Hughes et al. 1996] or other fuel-based encodings to prove totality.

4.4.4 Guarded Type Theory

Fortunately, *guarded type theories* both lift the syntactic productivity restriction as well as allow restricted forms of negative recursion in data types.

[11] Formerly Coq.

Guarded type theories postpone the productivity check to the type system, where it becomes a *semantic* instead of a *syntactic* property. This enables compositional reasoning about productivity, and of course stability under type-preserving refactorings such as extraction of the *rest* auxiliary definition in the revised implementation *map2* above.

The fundamental innovation of guarded recursive type theory is the integration of the *later* modality $\triangleright$ [Nakano 2000] which allows to define coinductive data types with negative recursive occurrences such as in the data constructor Fun that we have identified as problematic above.

The way that is achieved is roughly as follows: The type $\triangleright T$ represents data of type T that will become available after a finite amount of computation, such as unrolling one layer of a fixpoint definition or one (:) constructor of an infinite stream such as *map* (2+) [1..]. While peeling off one layer is a finite computation, there may be an infinite number of such layers in turn. Consuming the entirety of such an infinite layering is impossible, but it is possible to observe any finite prefix in a total manner.

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

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

where $ones : Str$ is the constant stream of 1. In particular, Str is the fixpoint of a locally contractive functor $F(X) = \mathbb{N} \times \triangleright X$. According to Birkedal and Mogelberg [2013], any type expression in simply typed lambda calculus defines a locally contractive functor as long as any occurrence of X is under a $\triangleright$. The most exciting consequence is that changing the Fun data constructor to $\mathsf{Fun} :: (\triangleright (\mathsf{D}\ \tau) \to \mathsf{D}\ \tau) \to \mathsf{Value}\ \tau$ makes $\mathsf{Value}\ \tau$ a well-defined type,[12] whereas traditional coinductive theories reject *any* negative recursive occurrence.

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

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

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

allowing us to apply a familiar framework of reasoning around ▸. In order not to obscure my work with pointless symbol pushing that is hard to verify without a machine, I will often omit the idiom brackets [McBride and Paterson 2008] $\{\!|\_|\!\}$ to indicate where the ▸ "effects" happen.

4.4.5 Total Encoding in Guarded Cubical Agda

The purpose of this subsection is to understand how $\mathcal{S}_{\mathbf{name}}[\![\_]\!]\_$ and $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$ can be proved total by encoding them in Guarded Cubical Agda, which implements Ticked Cubical Type Theory [Møgelberg and Veltri 2019]. The Agda code that documents this proof can be found in the Appendix on page 213.

To understand the Agda code, let me outline the changes necessary to encode $\mathcal{S}[\![\_]\!]\_$ as well as the concrete instances $\mathsf{D}_{\mathbf{na}}$ and $\mathsf{D}_{\mathbf{ne}}$ from Figures 4.7 and 4.9.

- 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* (Look *x*) *d* for some *x* :: Name and a delayed *d* :: ▸(D *τ*). This is enforced as follows:
  1. The Domain *d* 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* (Look *x*) *d* for some *x* :: Name, *d* :: ▸(D *τ*).
  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.
  3. The guarded recursive data type Value has a constructor 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* (Look *x*) *d*. This is in contrast to the original definition Fun :: (D *τ* → D *τ*) → Value *τ* which would *not* type-check. The concrete Domain implementation then translates between Σ D *p* and Name × ▸(D *τ*), essentially *defunctionalising* [Reynolds 1972] the latter into the former.
- Expectedly, HasBind becomes more complicated because it encodes the fixpoint combinator. I settled on *bind* :: ▸(▸D → D) → (▸D → D) → D. (I

tried rolling up *step* (Look x) _ in the definition of $\mathcal{S}[\![\_]\!]\_$ to get a simpler type *bind* :: (Σ D $p \to$ D) $\to$ (Σ D $p \to$ D) $\to$ 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 $\mathcal{S}[\![\_]\!]\_$ in a way that the type checker is satisfied; an exercise that is a bit more involved than one might expect, see the Appendix. Nevertheless, I find it remarkable how non-invasive these adjustment are! I had to conduct almost no proof external to the domain definition.

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

4.4.6 Proof of Adequacy For $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$

Building on the totality result for $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$, I will prove in this subsection that $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$ produces an abstraction of the small-step trace of the lazy Krivine (LK) machine from Section 4.2. The main result is Theorem 4.17, from which the earlier Theorem 4.4 in Section 4.4.1 follows.

To formalise the main result, I must characterise the maximal traces in the LK transition system and relate them to the traces produced by $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$ via function $\alpha_{\mathbb{S}^\infty}$ in Figure 4.13.

Maximal Lazy Krivine Traces

Formally, an LK trace is a trace in ($\hookrightarrow$) from Figure 4.2, i.e. a non-empty and potentially infinite sequence of LK states $(\sigma_i)_{i \in \overline{n}}$, such that $\sigma_i \hookrightarrow \sigma_{i+1}$ for $i, (i+1) \in \overline{n}$. The *length* of $(\sigma_i)_{i \in \overline{n}}$ is the potentially infinite number of $\hookrightarrow$ transitions n, where infinity is expressed by the first limit ordinal ω. The notation $\overline{n}$ means $\{m \in \mathbb{N} \mid m \leqslant n\}$ when $n \in \mathbb{N}$ is finite (note that $0 \in \mathbb{N}$), and $\mathbb{N}$ when $n = \omega$ is infinite.

The *source* state σ_0 exists for finite and infinite traces, while the *target* state σ_n is only defined when $n \neq \omega$ is finite. When the control expression of a state σ (selected via $ctrl(\sigma)$) is a value v, we call σ a *reduction* state, in which case the continuation (selected via $cont(\sigma)$) determines the next transition. Otherwise, σ is a *search* state and $ctrl(\sigma)$ determines the next transition.

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

Definition 4.6 (Deep)**.** *An LK trace $(\sigma_i)_{i \in \overline{n}}$ is* κ-deep *if every intermediate continuation $\kappa_i \triangleq cont(\sigma_i)$ extends κ (so $\kappa_i = \kappa$ or $\kappa_i = \_ \cdot \kappa$, abbreviated $\kappa_i = \ldots\kappa$).*

Definition 4.7 (Interior)**.** *A trace $(\sigma_i)_{i \in \overline{n}}$ is called* interior *(notated as $(\sigma_i)_{i \in \overline{n}}$ inter) if it is $cont(\sigma_0)$-deep.*

A *balanced trace* [Sestoft 1997] is an interior trace that is about to return from the initial evaluation context; it corresponds to a big-step evaluation of the initial control expression:

Definition 4.8 (Balanced)**.** *An interior trace $(\sigma_i)_{i \in \overline{n}}$ is* balanced *if the target state exists and is a reduction state with continuation $cont(\sigma_0)$.*

In the following I give an example for interior and balanced traces. I will omit the first component of heap entries in the examples because they bear no semantic significance apart from instrumenting Look transitions, and it is confusing when the heap-bound expression is a variable x, e.g. (y, ρ, x). Of course, function $\alpha_{\mathbb{S}^\infty}$ in Figure 4.13 will need to look at the first component.

Example 4.9. *Let $\rho = [x \mapsto \mathsf{a}_1], \mu = [\mathsf{a}_1 \mapsto (\_, [], \bar{\lambda} y.y)]$ and κ an arbitrary continuation. The trace*

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

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

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

is neither interior nor balanced.

As shown by Sestoft [1997], a balanced trace starting at a control expression e and ending with v corresponds to a derivation of $\mathsf{e} \Downarrow \mathsf{v}$ in a big-step semantics or a non-$\bot$ result in a Scott-style denotational semantics. It is when a derivation in a big-step semantics does *not* exist that a small-step semantics shows finesse. In this case, a small-step semantics differentiates two different kinds of *maximally interior* (or, just *maximal*) traces, namely *diverging* and *stuck* traces:

Definition 4.10 (Maximal)**.** *An LK trace* $(\sigma_i)_{i\in\overline{n}}$ *is* maximal *if and only if it is interior and there is no* σ_{n+1} *such that* $(\sigma_i)_{i\in\overline{n+1}}$ *is interior. More formally,*

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

Definition 4.11 (Diverging)**.** *An infinite and interior trace is called* diverging.

Definition 4.12 (Stuck)**.** *A finite, maximal and unbalanced trace is called* stuck.

Usually, stuckness is associated with a state of a transition system rather than a trace. That is not possible in my framework; the following example clarifies.

Example 4.13 (Stuck and diverging traces)**.** *Consider the interior trace*

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

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

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

Note that balanced traces are maximal traces as well. In fact, balanced, diverging and stuck traces are the *only* three kinds of maximal traces, as the following lemma formalises:

□ 197 **Lemma 4.14** (Characterisation of maximal traces)**.** *An LK trace* $(\sigma_i)_{i\in\overline{n}}$ *is maximal if and only if it is balanced, diverging or stuck.*

Interiority guarantees that the particular initial stack $\mathit{cont}(\sigma_0)$ of a maximal trace is irrelevant to execution, so maximal traces that differ only in the initial stack are bisimilar. This is a consequence of the fact that the semantics of a called function may not depend on the contents of the call stack; this fact is encoded implicitly in big-step derivations.

Abstraction preserves Termination Observable

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

In order to show that $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_$ preserves the termination observable of e,

- I define a family of representation functions α from LK traces to by-need traces, formally in $\mathsf{T}\ (\mathsf{Value}_{\mathbf{ne}}, \mathsf{Heap}_{\mathbf{ne}})$ (Figure 4.13),
- I show that the main function $\alpha_{\mathbb{S}^\infty}$ preserves the termination observable of a given LK trace $\mathit{init}(\mathrm{e}) \hookrightarrow \ldots$ (Lemma 4.16), and
- I show that running $\mathcal{S}_{\mathbf{need}}[\![e]\!]_\varepsilon(\varepsilon)$ is the same as mapping $\alpha_{\mathbb{S}^\infty}$ over the LK trace $\mathit{init}(\mathrm{e}) \hookrightarrow \ldots$, hence the termination behavior is observable (Theorem 4.17).

In the following, I will omit repeated wrapping and unwrapping of the ByNeed newtype wrapper when constructing and taking apart elements in $\mathsf{D}_{\mathbf{ne}}$. Furthermore, I will sometimes need to disambiguate the clashing definitions from Section 4.3 and Section 4.1 by adorning "Haskell objects" with a tilde, in which case $\tilde{\mu} \triangleq \alpha_{\mathbb{H}}(\mu) :: \mathsf{Heap}_{\mathbf{ne}}$ denotes a semantic by-need heap, defined as an abstraction of a syntactic LK heap $\mu \in \mathbb{H}$.

Now consider the trace abstraction function $\alpha_{\mathbb{S}^\infty}$ from Figure 4.13. It maps syntactic entities in the transition system to the definable entities in the denotational by-need trace domain $\mathsf{T}\ (\mathsf{Value}_{\mathbf{ne}}, \mathsf{Heap}_{\mathbf{ne}})$, henceforth abbreviated as T because it is the only use of the type constructor T in this subsection.

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

As such, the expression $\{\!|\alpha_{\mathbb{S}^\infty}((\sigma_{i+1})_{i \in \overline{n-1}}, \kappa_0)|\!\}$ has type $\triangleright \mathsf{T}$, where the $\triangleright$ in the type of $(\sigma_{i+1})_{i \in \overline{n-1}}$ maps through $\alpha_{\mathbb{S}^\infty}$ via the idiom brackets $\{\!|\_|\!\}$. Definitional equality $=$ on T is defined in the obvious structural way by guarded recursion (as it would be if it was a finite, inductive type).

Function $\alpha_{\mathbb{S}^\infty}$ expects an LK trace as well as a continuation parameter κ_0 that remains constant; it is initialised to the continuation of the source state $\mathit{cont}(\sigma_0)$ in order to tell apart stuck from balanced traces when $\alpha_{\mathbb{V}}$ is ultimately called on the target state. To that end, the first two equations of $\alpha_{\mathbb{V}}$ will not match unless the final continuation is the same as the initial continuation $\mathit{cont}(\sigma_0)$, indicated by the $(\kappa = \kappa_0)$ test at the end of the line.

The event abstraction function $\alpha_{\mathbb{E}_\mathbb{V}}(\sigma)$ encodes how intensional information from small-step transitions is retained as Events. Its implementation does not influence the adequacy result, and I imagine that this function is tweaked on

$$\boxed{\begin{array}{c}\alpha_{\mathbb{E}_\mathbb{V}} : \mathbb{S} \to \mathsf{Event} \qquad \alpha_{\mathbb{E}} : \mathbb{E} \times \mathbb{H} \to (\mathsf{Name} :\rightharpoonup \mathsf{D}_{\mathbf{ne}}) \\ \alpha_{\mathbb{H}} : \mathbb{H} \to \mathsf{Heap}_{\mathbf{ne}} \qquad \alpha_{\mathbb{V}} : \mathbb{S} \times \mathbb{K} \to \mathsf{Value}_{\mathbf{ne}} \\ \alpha_{\mathbb{S}^\infty} : \mathbb{S}^\infty \times \mathbb{K} \to \mathsf{T}\,(\mathsf{Value}_{\mathbf{ne}}, \mathsf{Heap}_{\mathbf{ne}})\end{array}}$$

$$\alpha_{\mathbb{E}_\mathbb{V}}(\sigma) = \begin{cases} \mathsf{Let}_1 & \text{if } \sigma = (\mathbf{let}\ \mathsf{x} = \_\ \mathbf{in}\ \_,\_,\_,\_) \\ \mathsf{App}_1 & \text{if } \sigma = (\mathsf{e}\ \mathsf{x}, \_, \_, \_) \\ \mathsf{Case}_1 & \text{if } \sigma = (\mathbf{case}\ \_\ \mathbf{of}\ \_,\_,\_,\_) \\ \mathsf{Look}\ y & \text{if } \sigma = (\mathsf{x}, \rho, \mu, \_), \mu(\rho(\mathsf{x})) = (\mathsf{y}, \_, \_) \\ \mathsf{App}_2 & \text{if } \sigma = (\bar{\lambda}\_.\_, \_, \_, \mathbf{ap}(\_) \cdot \_) \\ \mathsf{Case}_2 & \text{if } \sigma = (K\ \_,\_,\_, \mathbf{sel}(\_, \_) \cdot \_) \\ \mathsf{Upd} & \text{if } \sigma = (\mathsf{v}, \_, \_, \mathbf{upd}(\_) \cdot \_) \end{cases}$$

$$\alpha_{\mathbb{E}}([\overline{\mathsf{x} \mapsto \mathsf{a}}], \mu) = [\overline{x \mapsto \mathsf{Step}\ (\mathsf{Look}\ y)\ (\mathit{fetch}\ a) \mid \mu(\mathsf{a}) = (\mathsf{y}, \_, \_)}]$$

$$\alpha_{\mathbb{H}}([\overline{\mathsf{a} \mapsto (\_, \rho, \mathsf{e})}]) = [\overline{a \mapsto \mathit{memo}\ a\ (\mathcal{S}[\![e]\!]_{\alpha_{\mathbb{E}}(\rho,\mu)})}]$$

$$\alpha_{\mathbb{V}}((\bar{\lambda}\mathsf{x}.\mathsf{e}, \rho, \mu, \kappa), \kappa_0) = \mathsf{Fun}\ (\lambda d \to \mathsf{Step}\ \mathsf{App}_2\ (\mathcal{S}[\![e]\!]_{(\alpha_{\mathbb{E}}(\rho,\mu))[x \mapsto d]}))(\kappa = \kappa_0)$$

$$\alpha_{\mathbb{V}}((K\ \overline{\mathsf{x}}, \rho, \mu, \kappa), \kappa_0) = \mathsf{Con}\ k\ (\mathit{map}\ (\alpha_{\mathbb{E}}(\rho, \mu)\ !)\ \mathit{xs}) \qquad (\kappa = \kappa_0)$$

$$\alpha_{\mathbb{V}}(\_, \_, \mu, \_), \_) = \mathsf{Stuck}$$

$$\alpha_{\mathbb{S}^\infty}((\sigma_i)_{i \in \overline{n}}, \kappa_0) = \begin{cases} \mathsf{Step}\ (\alpha_{\mathbb{E}_\mathbb{V}}(\sigma_0))\ \{\!|\alpha_{\mathbb{S}^\infty}((\sigma_{i+1})_{i \in \overline{n-1}}, \kappa_0)|\!\} & \text{if } n > 0 \\ \mathsf{Ret}\ (\alpha_{\mathbb{V}}(\sigma_0, \kappa_0), \alpha_{\mathbb{H}}(\mu)) & \text{otherwise} \end{cases}$$
$$\text{where } (\_, \_, \mu, \_) = \sigma_0$$

Fig. 4.13: Abstraction functions for $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$

an as-needed basis to retain more or less intensional detail, depending on the particular trace property one is interested in observing. In our example, we focus on Look y events that carry with them the $y :: \mathsf{Name}$ of the let binding that allocated the heap entry. This event corresponds precisely to a Look(y) transition, so $\alpha_{\mathbb{E}_\mathbb{V}}(\sigma)$ maps σ to Look y when σ is about to make a Look(y) transition. In that case, the control expression must be x, and y is the first component of the heap entry $\mu(\rho(\mathsf{x}))$. The other cases are similar.

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

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

The length is an important property of LK traces that is preserved by α.

Lemma 4.15 (Abstraction preserves length)**.** *Let $(\sigma_i)_{i\in\overline{n}}$ be a trace. Then*

$$\mathit{len}\ (\alpha_{\mathbb{S}^\infty}((\sigma_i)_{i\in\overline{n}}, \mathit{cont}(\sigma_0))) = n.$$

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

$$\text{löb} = \text{fix} : \forall P.\ (\blacktriangleright P \Longrightarrow P) \Longrightarrow P$$

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

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

and use IH to prove P_{len}.

To that end, let $(\sigma_i)_{i\in\overline{n}}$ be an LK trace and define $\tau \triangleq \alpha_{\mathbb{S}^\infty}((\sigma_i)_{i\in\overline{n}}, \mathit{cont}(\sigma_0))$. Now proceed by case analysis on n:

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

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

  We use this fact and congruence to prove

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

□

It is rewarding to study the use of Löb induction in the proof above in detail, because many proofs in this subsection as well as Section 4.6 will make good use of it.

The next step is to prove that $\alpha_{\mathbb{S}^\infty}$ preserves the termination observable; then all that is left to do is to show that $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$ abstracts LK traces via $\alpha_{\mathbb{S}^\infty}$. The preservation property is formally expressed as follows:

Lemma 4.16 (Abstraction preserves termination observable). *Let $(\sigma_i)_{i\in\overline{n}}$ be a maximal trace. Then $\alpha_{\mathbb{S}^\infty}((\sigma_i)_{i\in\overline{n}}, cont(\sigma_0))$ ...* □ 198

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

The previous lemma allows us to apply the classifying terminology of maximal LK traces to a $\tau :: \mathsf{T}$ in the range of $\alpha_{\mathbb{S}^\infty}$. For such a maximal τ we will say that it is balanced when it ends with Ret (v, μ) for a $v \not= \mathsf{Stuck}$, stuck if it is ending in Ret (Stuck, μ) and diverging if it is infinite.

The final remaining step is to prove that $\mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_$ produces an abstraction of traces in the LK machine:

Theorem 4.17 ($\mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_$ abstracts LK machine). *Let $(\sigma_i)_{i\in\overline{n}}$ be a maximal LK trace with source state* $(\mathsf{e}, \rho, \mu, \kappa)$. *Then $\alpha_{\mathbb{S}^\infty}((\sigma_i)_{i\in\overline{n}}, \kappa) = \mathcal{S}_{\mathbf{need}}[\![e]\!]_{\alpha_{\mathbb{E}}(\rho,\mu)}(\alpha_{\mathbb{H}}(\mu))$, where $\alpha_{\mathbb{S}^\infty}$ is the function defined in Figure 4.13.*

Proof. Let us abbreviate the proposed correctness relation as

$$P_\alpha((\sigma_i)_{i\in\overline{n}}) \triangleq (\sigma_i)_{i\in\overline{n}}\,\mathsf{max} \Longrightarrow \alpha_{\mathbb{S}^\infty}((\sigma_i)_{i\in\overline{n}}, \kappa) = \mathcal{S}_{\mathbf{need}}[\![e]\!]_{\alpha_{\mathbb{E}}(\rho,\mu)}(\alpha_{\mathbb{H}}(\mu)) \quad \text{where } (\mathsf{e}, \rho, \mu, \kappa) = \sigma_0$$

We prove it by Löb induction, with $IH \in \blacktriangleright P_\alpha$ as the inductive hypothesis.

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

- **Case** x: Note first that Look is the only applicable transition rule according to rule inversion on $ctrl(\sigma_0) = \mathsf{x}$.

  In case that $n = 0$, $(\sigma_i)_{i\in\overline{n}}$ is stuck because $ctrl(\sigma_0)$ is not a value, hence $\alpha_{\mathbb{S}^\infty}$ returns Ret (Stuck, _). Since Look does not apply (otherwise $n > 0$), we must have $\mathsf{x} \notin \mathrm{dom}(\rho)$, and hence $\tau =$ Ret (Stuck, _) by calculation as well.

  Otherwise, $\sigma_1 \triangleq (\mathsf{e}', \rho', \mu, \mathbf{upd}(\mathsf{a}) \cdot \kappa)$, $\sigma_0 \hookrightarrow \sigma_1$ via Look(y), and $\rho(\mathsf{x}) = \mathsf{a}$, $\mu(\mathsf{a}) = (\mathsf{y}, \rho', \mathsf{e}')$. This matches $\tilde{\rho}\,!\,x = step\ (\mathsf{Look}\ y)\ (fetch\ a)$ in the interpreter.

  It suffices to show that the tails equate *later*.

We can infer that $\tilde{\mu}\,!\,a = \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e']\!]_{\tilde{\rho}'})$ by definition of $\alpha_{\mathbb{H}}$, so

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

Let us define $\tau^{\blacktriangleright} \triangleq \{\!|\mathcal{S}_{\mathbf{need}}[\![e']\!]_{\tilde{\rho}'}(\tilde{\mu})|\!\}$ and apply the induction hypothesis IH to the maximal trace starting at σ_1. This trace has length $m-1$ for some $m > 0$. The induction hypothesis yields the equality

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

Any Steps in $\tau^{\blacktriangleright}$ match the transitions of $(\sigma_{i+1})_{i \in \overline{m-1}}$ per IH, and $\gg\!=$ simply forwards these Steps. What remains to be shown is that the continuation passed to $\gg\!=$ operates correctly.

If $\tau^{\blacktriangleright}$ is infinite, we are done, because the continuation is never called. If $\tau^{\blacktriangleright}$ ends in $\mathsf{Ret}\ (\mathsf{Stuck}, \tilde{\mu}_m)$, then $(\sigma_{i+1})_{i \in \overline{m-1}}$ is stuck as well by Lemma 4.16. Then so is $(\sigma_i)_{i \in \overline{m}}$ and it turns out that $n = m$. Likewise, the continuation of $\tau^{\blacktriangleright}$ will return $\mathsf{Ret}\ (\mathsf{Stuck}, \tilde{\mu}_m)$ unchanged, where $\tilde{\mu}_m$ corresponds to the final heap in σ_m via $\alpha_{\mathbb{H}}$.

Otherwise $\tau^{\blacktriangleright}$ ends with $\mathsf{Ret}\ (\mathit{val}, \tilde{\mu}_m)$ and by Lemma 4.16 $(\sigma_{i+1})_{i \in \overline{m-1}}$ is balanced; hence $\mathit{cont}(\sigma_m) = \mathbf{upd}(\mathsf{a}) \cdot \kappa$ and $\mathit{ctrl}(\sigma_m)$ is a value. So $\sigma_m = (\mathsf{v}, \rho_m, \mu_m, \mathbf{upd}(\mathsf{a}) \cdot \kappa)$ and the $\textsc{Upd}$ transition fires, reaching $\sigma_{m+1} \triangleq (\mathsf{v}, \rho_m, \mu_m[\mathsf{a} \mapsto (\mathsf{y}, \rho_m, \mathsf{v})], \kappa)$. It turns out that $n = m+1$ and σ_{m+1} is the target state σ_n. That is because σ_{m+1} remains a reduction state and has continuation κ, so $(\sigma_i)_{i \in \overline{m+1}}$ is balanced. Likewise, the continuation argument of $\gg\!=$ does a $\mathsf{Step}\ \mathsf{Upd}$ on $\mathsf{Ret}\ (\mathit{val}, \tilde{\mu}_m)$, updating the heap. By cases on v and the $\mathsf{Domain}\ \mathsf{D}_{\mathbf{ne}}$ instance we can see that

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

and this equality concludes the proof, because the heap in σ_n is exactly $\mu_m[\mathsf{a} \mapsto (\mathsf{y}, \rho_m, \mathsf{v})]$.

- **Case** e x: The cases where τ gets stuck or diverges before finishing evaluation of e are similar to the variable case. So let us focus on the situation

when $\tau^{\blacktriangleright} \triangleq \{\!|\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\tilde{\rho}}(\tilde{\mu})|\!\}$ returns and let σ_m be LK state at the end of the balanced trace $(\sigma_{i+1})_{i \in \overline{m-1}}$ through e starting in stack $\mathbf{ap}(\mathsf{a}) \cdot \kappa$.

Now, either there exists a transition $\sigma_m \hookrightarrow \sigma_{m+1}$, or it does not. When the transition exists, it must must leave the stack $\mathbf{ap}(\mathsf{a}) \cdot \kappa$ due to maximality, necessarily by an $\textsc{App}_2$ transition. That in turn means that the value in $\mathit{ctrl}(\sigma_m)$ must be a lambda $\bar{\lambda}\mathsf{y}.\mathsf{e}'$, and $\sigma_{m+1} = (\mathsf{e}', \rho_m[\mathsf{y} \mapsto \rho(\mathsf{x})], \mu_m, \kappa)$.

Likewise, $\tau^{\blacktriangleright}$ ends in

$$\alpha_{\mathbb{V}}(\sigma_m, \mathbf{ap}(\mathsf{a}) \cdot \kappa) = \mathsf{Fun}\ (\lambda d \to \mathit{step}\ \mathsf{App}_2\ (\mathcal{S}_{\mathbf{need}}[\![e']\!]_{\tilde{\rho}_m[y \mapsto d]}))$$

(where $\tilde{\rho}_m$ corresponds to the environment in σ_m in the usual way, similarly for $\tilde{\mu}_m$). The *apply* implementation of Domain $\mathsf{D}_{\mathbf{ne}}$ applies the Fun value to the argument denotation $\tilde{\rho}\,!\,x$, hence it remains to be shown that $\mathcal{S}_{\mathbf{need}}[\![e']\!]_{\tilde{\rho}_m[y \mapsto \tilde{\rho}\,!\,x]}(\tilde{\mu}_m)$ is equal to $\alpha_{\mathbb{S}^\infty}((\sigma_{i+m+1})_{i \in \overline{k}}, \kappa)$ *later*, where $(\sigma_{i+m+1})_{i \in \overline{k}}$ is the maximal trace starting at σ_{m+1}.

We can once again apply the induction hypothesis to this situation. From this and our earlier equalities, we get $\alpha_{\mathbb{S}^\infty}((\sigma_i)_{i \in \overline{n}}, \kappa) = \tau$, concluding the proof of the case where there exists a transition $\sigma_m \hookrightarrow \sigma_{m+1}$.

When $\sigma_m \not\hookrightarrow$, then $\mathit{ctrl}(\sigma_m)$ is not a lambda, otherwise $\textsc{App}_2$ would apply. In this case, *apply* gets to see a Stuck or Con and returns Stuck as well.

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

□

Thus, my denotational by-need interpreter is built on firm semantic ground.

4.5 Static Analysis

So far, the semantic domains I have proposed have all been *infinite*, simply because the dynamic traces they express are potentially infinite as well. However, by instantiating the generic denotational interpreter on page 86 with a semantic domain in which every element is *finite data*, 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 by choosing appropriate semantic domains. For example, I have successfully realised the following analyses as denotational interpreters:

- Section 4.5.1 defines a summary-based *usage analysis*, the running example of this work. I prove that usage analysis correctly infers absence in Section 4.6[13].
- Section 4.5.2 defines a *type analysis* with let generalisation that implements Milner's Algorithm J, inferring polytypes such as $\forall\alpha_3.\ \texttt{option}\ (\alpha_3 \to \alpha_3)$ that act as summaries.
- Section 4.5.3 defines 0CFA *control-flow analysis* [Shivers 1991], a non-modular analysis lacking a finite summary mechanism, simply as a proof of concept.
- To demonstrate that my framework scales to real-world compilers, I have refactored relevant parts of *Demand Analysis* in the Glasgow Haskell Compiler into an instance of a denotational interpreter for GHC Core as an artefact. The code for this generic denotational interpreter can be found in Appendix C. The resulting compiler bootstraps and passes the testsuite.[14] Demand Analysis is the real-world implementation of the cardinality analysis work of Sergey, Vytiniotis, et al. [2017], generalising usage analysis from Section 4.5.1 and implementing strictness analysis as well. Section 4.5.5 contains a report of this case study.

[13] Recall that the main body omits code for presentation purposes, but the full code can be looked up in Appendix D.

[14] There is a small caveat: I did not try to optimise for compiler performance in my 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.

4.5.1 Usage Analysis

In this subsection, I give a detailed account of *usage analysis* as an instance of the denotational interpreter. Usage analysis generalises the summary-based absence analysis from Section 4.1. It is a compelling example because it illustrates that my framework is suitable to infer *operational properties*, such as an upper bound on the number of variable lookups.

Trace Abstraction in Trace T_U

In order to recover usage analysis as an instance of my generic interpreter, we must define its finitely represented semantic domain D_U. Often, the first step in doing so is to replace the potentially infinite traces T in dynamic semantic domains such as $\mathsf{D}_\mathbf{na}$ with finite data such as T_U in Figure 4.14. A *usage trace* $\langle \varphi, \mathit{val} \rangle :: \mathsf{T}_\mathsf{U}\ v$ is a pair of a value $\mathit{val} :: v$ and a finite map $\varphi :: \mathsf{Uses}$, mapping variables to a *usage* U. The usage $\varphi\ !?\ x$ assigned to x is meant to approximate the number of $\mathsf{Look}\ x$ events; U_0 means "at most 0 times", U_1 means "at most 1 times", and U_ω means "an unknown number of times". In this way, T_U is an *abstraction* of T: it squashes all $\mathsf{Look}\ x$ events into a single entry $\varphi\ !?\ x :: \mathsf{U}$ and discards all other events.

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

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

We would like to abstract this trace into $\langle [\, i \mapsto \mathsf{U}_1, j \mapsto \mathsf{U}_\omega \,], \ldots \rangle$. One plausible way to achieve this is to replace every $\mathsf{Step}\ (\mathsf{Look}\ x)\ \ldots$ in the by-name trace with a call to $\mathit{step}\ (\mathsf{Look}\ x)\ \ldots$ from the $\mathsf{Trace}\ \mathsf{T}_\mathsf{U}$ instance in Figure 4.14, essentially folding over the trace. The *step* implementation increments the usage of x whenever a $\mathsf{Look}\ x$ event occurs. The addition operation used to carry out incrementation is defined in type class instances $\mathsf{UVec}\ \mathsf{U}$ and $\mathsf{UVec}\ \mathsf{Uses}$, together with scalar multiplication[15]. For example, $\mathsf{U}_0 + u = u$ and $\mathsf{U}_1 + \mathsf{U}_1 = \mathsf{U}_\omega$ in U, as well as $\mathsf{U}_0 * u = \mathsf{U}_0$, $\mathsf{U}_\omega * \mathsf{U}_1 = \mathsf{U}_\omega$. These operations lift to Uses pointwise, e.g. $[\, i \mapsto \mathsf{U}_1 \,] + (\mathsf{U}_\omega * [\, j \mapsto \mathsf{U}_1 \,]) = [\, i \mapsto \mathsf{U}_1, j \mapsto \mathsf{U}_\omega \,]$.

Abstracting T into T_U but keeping the concrete semantic Value definition amounts to what Darais, Labich, et al. [2017] call a *collecting semantics*. To

[15] UVec models U-modules. It is not a vector space because U lacks inverses, but the intuition is close enough and the term "vector" more familiar.

```
data U = U0 | U1 | Uω                 data TU v = ⟨Uses, v⟩
type Uses = Name :⇀ U                 instance Trace (TU v) where
class UVec a where                      step (Look x) ⟨φ, v⟩ = ⟨[x ↦ U1] + φ, v⟩
  (+) :: a → a → a                      step _        τ      = τ
  (∗) :: U → a → a                    instance Monad TU where
instance UVec U where ...               return a = ⟨ε, a⟩
instance UVec Uses where ...            ⟨φ1, a⟩ ≫= k = let ⟨φ2, b⟩ = k a in ⟨φ1 + φ2, b⟩
```

Fig. 4.14: Usage U and usage trace T_U

```
S_usage⟦e⟧ρ = S⟦e⟧ρ :: DU
data ValueU = U :∗: ValueU | Rep U
type DU = TU ValueU
instance Domain DU where
  stuck          = ⊥
  fun x f        = case f ⟨[x ↦ U1], Rep Uω⟩ of
    ⟨φ, v⟩ → ⟨φ[x ↦ U0], φ !? x :∗: v⟩
  apply ⟨φ1, v1⟩ ⟨φ2, _⟩ = case peel v1 of
    (u, v2) → ⟨φ1 + u ∗ φ2, v2⟩
  con _ ds       = foldl apply ⟨ε, Rep Uω⟩ ds
  select d fs    = d ≫ lub [ f (replicate (conArity k) ⟨ε, Rep Uω⟩)
                           | (k, f) ← assocs fs]

peel :: ValueU → (U, ValueU)          instance Lat U where ...
peel (Rep u) = (u, Rep u)             instance Lat Uses where ...
peel (u :∗: v) = (u, v)               instance Lat ValueU where ...
(!?) :: Uses → Name → U               instance Lat DU where ...
φ !? x | x ∈ dom φ = φ ! x            instance HasBind DU where
       | otherwise = U0                 bind rhs body = body (kleeneFix rhs)
```

Fig. 4.15: Summary-based usage analysis

recover such an analysis-specific collecting semantics, it is sufficient to define a Monad instance on $\mathsf{T_U}$ mirroring trace concatenation and then running our interpreter at, e.g. $\mathsf{D}\ (\mathsf{ByName}\ \mathsf{T_U}) \cong \mathsf{T_U}\ (\mathsf{Value}\ \mathsf{T_U})$ on expression e from earlier:

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

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 finitely represented $\mathsf{Value_U}$ to replace $\mathsf{Value}\ \mathsf{T_U}$.

Value Abstraction $\mathsf{Value_U}$ and Summarisation in Domain $\mathsf{D_U}$

In this subsection, we complement the trace type $\mathsf{T_U}$ from the previous subsection with an abstract value type $\mathsf{Value_U}$ to get the finitely represented semantic domain $\mathsf{D_U} = \mathsf{T_U}\ \mathsf{Value_U}$ in Figure 4.15, and thus a *static usage analysis* $\mathcal{S}_{\textbf{usage}}[\![\_]\!]_\_$ when we instantiate $\mathcal{S}[\![\_]\!]_\_$ at $\mathsf{D_U}$.

The definition of $\mathsf{Value_U}$ is just a copy of $\pi \in \mathrm{Args}$ in Figure 4.1 that lists argument usage U instead of Absence flags; the entire intuition transfers. For example, the $\mathsf{Value_U}$ abstracting $\bar{\lambda}y.\bar{\lambda}z.y$ is $\mathsf{U}_1 : \mathsf{U}_0 : \mathsf{Rep}\ \mathsf{U}_\omega$, because the first argument is used once while the second is used 0 times. What we previously called absence types $\theta \in \mathrm{AbsTy}$ in Figure 4.1 is now the abstract semantic domain $\mathsf{D_U}$. It is now evident that usage analysis is a modest generalisation of absence analysis in Figure 4.1: a variable is absent (A) when it has usage U_0, otherwise it is used (U).

Consider the analysis result for the example expression from Section 4.1,

$$\mathcal{S}_{\textbf{usage}}[\![(\textbf{let } k = \bar{\lambda}y.\bar{\lambda}z.y \textbf{ in } k\ x_1\ x_2)]\!]_{\rho_e} = \langle [k \mapsto \mathsf{U}_1, x_1 \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle.$$

Usage analysis successfully infers that x_1 is used at most once and that x_2 is absent, because it does not occur in the reported Uses.

On the other hand,

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

demonstrates the limitations of the first-order summary mechanism. While the program trace would only have one lookup for j, the analysis is unable to reason through the indirect call and conservatively reports that j may be used many times.

The Domain instance is responsible for implementing the summary mechanism. While *stuck* expressions do not evaluate anything and hence are denoted by $\bot = \langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_0 \rangle$, the *fun* and *apply* functions play exactly the same roles as fun_x and *app* in Figure 4.1. Let us briefly review how the summary for the right-hand side $\bar{\lambda} x.x$ of i in the previous example is computed:

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

The definition of *fun* x applies the lambda body to a *proxy* $\langle [x \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle$ to summarise how the body uses its argument by way of looking at how it uses x.[16] Every use of x's proxy will contribute a usage of U_1 on x, and multiple uses in the lambda body would accumulate to a usage of U_ω. In this case there is only a single use of x and the final usage $\varphi\ !?\ x = \mathsf{U}_1$ from the lambda body will be prepended to the value abstraction. Occurrences of x unleash the uninformative top value ($\mathsf{Rep}\ \mathsf{U}_\omega$) from x's proxy for lack of knowing the actual argument at call sites.

The definition of *apply* to apply such summaries to an argument is nearly the same as in Figure 4.1, except for the use of $+$ instead of $\sqcup$ to carry over $\mathsf{U}_1 + \mathsf{U}_1 = \mathsf{U}_\omega$, and an explicit *peel* to view a $\mathsf{Value}_\mathsf{U}$ in terms of $\fatsemi$ (it is $\mathsf{Rep}\ u \equiv u \fatsemi \mathsf{Rep}\ u$). The usage u thus pelt from the value determines how often the actual argument was evaluated, and multiplying the uses of the argument φ_2 with u accounts for that.

The following example illustrates the summary mechanism for data types:

$$
\mathcal{S}_{\mathbf{usage}}[\![(\mathbf{let}\ z = Z()\ \mathbf{in}\ \mathbf{case}\ S(z)\ \mathbf{of}\ S(n) \to n)]\!]_\varepsilon = \langle [z \mapsto \mathsf{U}_\omega], \mathsf{Rep}\ \mathsf{U}_\omega \rangle.
$$

Our analysis imprecisely infers that z might be used many times when it is only used once.[17] This is achieved in *con* by repeatedly *apply*ing to the top value ($\mathsf{Rep}\ \mathsf{U}_\omega$), as if a data constructor was a lambda-bound variable. Dually, *select* does not need to track how fields are used and can pass $\langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_\omega \rangle$ as proxies

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

[17] Following Sergey, Vytiniotis, et al. [2017] we could model *demand* as a property of evaluation contexts and propagate uses of field binders to the scrutinee's fields to do better.

```
class Eq a ⇒ Lat a where ⊥ :: a; (⊔) :: a → a → a;
kleeneFix :: Lat a ⇒ (a → a) → a;        lub :: Lat a ⇒ [a] → a
kleeneFix f = go ⊥ where go x = let x' = f x in if x' ⊑ x then x' else go x'
```

Fig. 4.16: Order theory and Kleene iteration

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.

Note that the finite representation of the type $\mathsf{D_U}$ rules out injective implementations of *fun x* :: $(\mathsf{D_U} \to \mathsf{D_U}) \to \mathsf{D_U}$ and thus requires the aforementioned *approximate* summary mechanism. There is another potential source of approximation: the HasBind instance discussed next.

Finite Fixpoint Strategy in HasBind $\mathsf{D_U}$ and Totality

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

For the dynamic semantics in Section 4.3 we made liberal use of *guarded fixpoints*, that is, recursively defined values such as **let** *d* = *rhs d* **in** *body d* in HasBind $\mathsf{D_{na}}$ (Figure 4.7). At least for $\mathcal{S}_{\mathbf{name}}[\![\_]\!]\_$ and $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$, we have proved in Section 4.4.1 that these fixpoints always exist by a coinductive argument. Alas, among other things this argument relies on the Step constructor — and thus the *step* method — of the trace type T being *lazy* in the tail of the trace!

When we replaced T in favour of the finite data type $\mathsf{T_U}$ in Section 4.5.1 to get a collecting semantics D (ByName $\mathsf{T_U}$), we got a partial interpreter. That was because the *step* implementation of $\mathsf{T_U}$ is *not* lazy, and hence the guarded fixpoint **let** *d* = *rhs d* **in** *body d* is not guaranteed to exist.

In general, finite data trace types cannot have a lazy *step* implementation, so finite data domains such as $\mathsf{D_U}$ require a different fixpoint strategy to ensure termination. Depending on the abstract domain, different fixpoint strategies can be employed. For an unusual example, in my type analysis (Section 4.5.2), we generate and solve a constraint system via unification to define fixpoints. In case of $\mathsf{D_U}$, we compute least fixpoints by Kleene iteration *kleeneFix* in Figure 4.16. *kleeneFix* requires us to define an order on $\mathsf{D_U}$, which is induced by $\mathsf{U}_0 \sqsubset \mathsf{U}_1 \sqsubset \mathsf{U}_\omega$ in the same way that the order on AbsTy in Section 4.1.2 was induced from the order A ⊏ U on Absence flags.

The iteration procedure terminates whenever the type class instances of $\mathsf{D_U}$ are monotone and there are no infinite ascending chains in $\mathsf{D_U}$. Alas, our $\mathsf{Value_U}$ indeed contains such infinite chains, for example, $\mathsf{U_1} \fatsemi \mathsf{U_1} \fatsemi \ldots \fatsemi \mathsf{Rep}\ \mathsf{U_0}$! This is easily worked around in practice by employing appropriate monotone widening measures such as trimming any $\mathsf{Value_U}$ at depth 10 to flat $\mathsf{Rep}\ \mathsf{U}_\omega$. The resulting definition of HasBind is safe for by-name and by-need semantics.

4.5.2 Type Analysis: Algorithm J

Computing least fixpoints is common practice in static program analysis. However, some abstract domains employ quite different fixpoint strategies. The abstract domain of the type analysis I define in this subsection is an interesting example: Type analysis — specifically, Milner's Algorithm J — computes fixpoints by generating and solving a constraint system via unification. Furthermore, since the domain is familiar, it is a good one to study in the context of denotational interpreters.

Figure 4.17 outlines the abstract domain J Type at which the generic denotational interpreter can be instantiated to perform Type analysis. I omit implementational details that are derivative of Milner's description of Algorithm J. The full implementation can be found in Appendix D, but the provided code is sufficiently exemplary of the approach.

Type analysis $\mathcal{S}_{\mathbf{type}}[\![\_]\!]$ infers the most general type of an expression, e.g.

$$\mathcal{S}_{\mathbf{type}}[\![(\mathbf{let}\ f = \bar{\lambda} g.\bar{\lambda} x.g\ x\ \mathbf{in}\ f)]\!] = \forall \alpha_4, \alpha_5.\ (\alpha_4 \to \alpha_5) \to \alpha_4 \to \alpha_5.$$

The most general type can be *polymorphic* when it universally quantifies over *generic* type variables such as α_4 and α_5 above. In general, such a PolyType is of the form $\forall \overline{\alpha}.\ \theta$, where θ ranges over a monomorphic Type that can be either a type variable $\mathsf{TyVar}\ \alpha$ (I will use θ_α as meta variable for this form), a function type $\theta_1 :\!\to\! \theta_2$, or a type constructor application TyConApp, where $\mathsf{TyConApp}\ \mathsf{OptionTyCon}\ [\theta_1]$ is printed as `option` θ_1. The Wrong type indicates a type error and is printed as **wrong**.

Key to the analysis is its abstract trace type J, offering means to invoke unification (*unify*), fresh name generation (*freshTyVar*, *instantiatePolyTy*) and let generalisation (*generaliseTy*). My type J implements these effects by maintaining two pieces of state via the standard monad transformer StateT:

1. a consistent set of type constraints as a unifying substitution Subst.

```
data TyCon = BoolTyCon | NatTyCon | OptionTyCon | PairTyCon
data Type = Type :→: Type | TyConApp TyCon [Type] | TyVar Name | Wrong
data PolyType = PT [Name] Type

type Subst = Name :⇀ Type
type Constraint = (Type, Type)
newtype J a = J (StateT (Set Name, Subst) Maybe a)
unify           :: Constraint → J ()
freshTyVar      :: J Type
instantiatePolyTy :: PolyType → J Type
generaliseTy    :: J Type → J PolyType
closedType      :: J Type → PolyType

𝒮type⟦e⟧ = closedType (𝒮⟦e⟧ε) :: PolyType

instance Trace (J v) where step _ = id
instance Domain (J Type) where
  stuck = return Wrong
  fun _ f = do
    θα ← freshTyVar
    θ  ← f (return θα)
    return (θα :→: θ)
  con k ds = ...
  apply v a = do
    θ1 ← v
    θ2 ← a
    θα ← freshTyVar
    unify (θ1, θ2 :→: θα)
    return θα
  select dv fs = ...
uniFix :: (J Type → J Type) → J Type
uniFix rhs = do
  θα ← freshTyVar
  θ  ← rhs (return θα)
  unify (θα, θ)
  return θα
instance HasBind (J Type) where
  bind rhs body = do
    σ ← generaliseTy (uniFix rhs)
    body (instantiatePolyTy σ)
```


Fig. 4.17: Type analysis with let generalisation (Algorithm J)

2. the set of used names as a Set Name. This is to supply fresh names in *freshTyVar* and to instantiate a polytype $\forall\alpha.\alpha \to \alpha$ to a monotype $\alpha_1 \to \alpha_1$ for fresh α_1 as done by *instantiatePolyTy*, but also to identify the type variables which are *generic* [Milner 1978] in the ambient type context and hence may be generalised by *generaliseTy*.

Unification failure is signalled by returning Nothing in the base monad Maybe, and function *closedType* for handling J effects will return Wrong when that happens:

$$\mathcal{S}_{\mathbf{type}}[\![(\mathbf{let}\ x = \mathit{None}()\ \mathbf{in}\ x\ x)]\!] = \mathbf{wrong}$$

The operational detail offered by Trace is ignored by J, but the Domain and HasBind instances for the abstract semantic domain J Type are quite interesting. Throughout the analysis, the invariant is maintained that the J Type denotations of let-bound variables in the interpreter environment ρ are of the form *instantiatePolyTy* σ for a polytype σ, while lambda- and field-bound variables are denoted by *return* θ, yielding the same monotype θ at all use sites. Thus, let-bound denotations instantiate polytypes on-the-fly at occurrence sites, just as in Algorithm J.

Both *return* θ as well as *instantiatePolyTy* σ are summaries in the sense of Section 4.1.3, where the latter kind is more useful than the former. While stateful computations of type J Type are not generally finitely representable, both σ and θ are just data, and hence the expressions *return* θ and *instantiatePolyTy* σ could be defunctionalised into bespoke data constructors of J Type as well; hence they are finitely representable denotations and thus summaries.

As expected, *stuck* terms are denoted by the monotype Wrong. The definition of *fun* resembles the abstraction rule of Algorithm J, in that it draws a fresh variable type θ_α :: Type (of the form TyVar α) to stand for the type of the argument. This type is passed as a monotype *return* θ_α to the body denotation f, where it will be added to the environment (cf. Figure 4.5) in order to compute the result type θ of the function body. The type for the whole function is then $\theta_\alpha :\!\!\to: \theta$. The definition for *apply* is a literal translation of Algorithm J as well. The cases for *con* and *select* are omitted as their implementation follows a similar routine.

The generalisation and instantiation machinery comes to bear in the implementation of *bind*, which implements a combination of the *fix* and *let* cases in Algorithm J, computing fixpoints by unification (*uniFix*). It is best understood by tracing the right-hand side of o in the following example:

$$\mathcal{S}_{\mathbf{type}}[\![(\mathbf{let}\ i = \bar{\lambda}x.x\ \mathbf{in}\ \mathbf{let}\ o = \mathit{Some}(i)\ \mathbf{in}\ o)]\!] = \forall\alpha_6.\ \texttt{option}\ (\alpha_6 \to \alpha_6)$$

The implementation of *bind* ties the recursive knot by calling *uniFix*. It works by calling the iteratee *rhs* (corresponding to $Some(i)$) with a fresh unification variable type θ_α, for example α_1. The result of the call to *rhs* in turn is a monotype θ, for example $\texttt{option}\ (\alpha_3 \to \alpha_3)$ for *generic* α_3, meaning that α_3 is a fresh name introduced in the right-hand side while instantiating the polymorphic identity function i. Then θ_α is unified with θ, substituting α_1 with $\texttt{option}\ (\alpha_3 \to \alpha_3)$. This concludes the implementation of Milner's *fix* case.

For Milner's *let* case, the type θ_α returned by the call to *uniFix* is generalised to $\forall\alpha_3.\ \texttt{option}\ (\alpha_3 \to \alpha_3)$ by universally quantifying the generic variable α_3. It is easy for *generaliseTy* to deduce that α_3 must be generic wrt. the right-hand side, because α_3 was freshly drawn in *uniFix* and thus does not occur in the set of used Names prior to the call to *generaliseTy*. The generalised polytype σ is then instantiated afresh via *instantiatePolyTy* σ at every use site of o in the let body, implementing polymorphic instantiation.

Thus we shall conclude this short excursion into type analysis and continue with a classic, call-strings-based interprocedural analysis: control-flow analysis.

4.5.3 Control-flow Analysis

Traditionally, control-flow analysis (CFA) [Shivers 1991] is an important instance of higher-order abstract interpreters [Darais, Labich, et al. 2017; Van Horn and Might 2010]. Although one of the main advantages of denotational interpreters is that summary-based analyses can be derived as instances, this subsection demonstrates that a call-strings-based CFA can be derived as an instance from the generic denotational interpreter in Figure 4.5 as well.

CFA overapproximates the set of syntactic values an expression evaluates 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 or constant propagation in an interprocedural setting.

Figure 4.18 implements the 0CFA variant of control-flow analysis. For a given expression, it reports a set of *program labels* — textual representations of positions in the program — that the expression might evaluate to:

$$\mathcal{S}_{\mathbf{cfa}}[\![(\mathbf{let}\ i = \bar{\lambda}x.x\ \mathbf{in}\ \mathbf{let}\ j = \bar{\lambda}y.y\ \mathbf{in}\ i\ j\ j)]\!] = \{\lambda y..\} \tag{4.1}$$

Here, 0CFA infers that the example expression will evaluate to the lambda expression bound to j. This lambda is uniquely identified by the reported label

```
𝒮cfa⟦e⟧ = runCFA (𝒮⟦e⟧ε); runCFA :: DC → Labels
newtype Labels = Lbls (Set Label)
type DC = State Cache Labels
data Cache = Cache (Label :⇀ ConCache) (Label :⇀ FunCache)
type ConCache = (Tag, [Labels])
data FunCache = FC (Maybe (Labels, Labels)) (DC → DC)

updConCache :: Label → Tag → [Labels] → State Cache ()
updFunCache :: Label → (DC → DC) → State Cache ()
cachedCall :: Labels → Labels → DC
cachedCons :: Labels → State Cache (Tag :⇀ [Labels])

instance HasBind DC where ...
instance Trace DC where step _ = id
instance Domain DC where
  stuck = return ⊥
  fun _ ℓ f = do
    updFunCache ℓ f
    return (Lbls (Set.singleton ℓ))
  apply dv da = do
    v ← dv
    a ← da
    cachedCall v a
  con ℓ k ds = do
    lbls ← sequence ds
    updConCache ℓ k lbls
    return (Lbls (Set.singleton ℓ))
  select dv fs = do
    v ← dv
    tag2flds ← cachedCons v
    lub <$> sequence [ f (map return (tag2flds ! k))
                     | (k, f) ← Map.assocs fs, k ∈ dom tag2flds]
```

Fig. 4.18: Domain D_C for 0CFA control-flow analysis

$\lambda y..$ per the unique binder assumption in Section 4.1.1. Furthermore, the analysis determined that the expression cannot evaluate to the lambda expression bound to i, hence its label $\lambda x..$ is *not* included in the set.

By contrast, when i is dynamically called both with i and with j, the result becomes approximate because 0CFA joins together the information from the two call sites:

$$\mathcal{S}_{\mathbf{cfa}}[\![(\mathbf{let}\ i = \bar{\lambda} x.x\ \mathbf{in}\ \mathbf{let}\ j = \bar{\lambda} y.y\ \mathbf{in}\ i\ \ i\ \ j)]\!] = \{\lambda x.., \lambda y..\}$$

Labels for constructor applications simply print their syntax, e.g.

$$\mathcal{S}_{\mathbf{cfa}}[\![(\mathbf{let}\ x = \mathbf{let}\ y = S(x)\ \mathbf{in}\ S(y)\ \mathbf{in\ case}\ x\ \mathbf{of}\ \{Z() \to x; S(z) \to z\})]\!] = \{S(x)\}. \tag{4.2}$$

Note that in this example, 0CFA discovers that x evaluates to $S(y)$ and hence is able to conclude that the $Z()$ branch of the case expression is dead. In doing so, 0CFA rules out that the expression evaluates to $S(y)$, reporting $S(x)$ as the only value of the expression.

In general, the label (i.e. string) $S(y)$ does not uniquely determine a position in the program because the expression may occur multiple times. However, eliminating such common subexpressions is semantics-preserving, so I argue that this poor man's notion of program labels is good enough for the purpose of this demonstration.

To facilitate 0CFA as an instance of the generic denotational interpreter, I need to slightly revise the Domain class to pass the syntactic label to *fun* and *con*:

```
type Label = String
class Domain d where
  con :: Label → Tag → [d] → d
  fun :: Name → Label → (d → d) → d
```

Constructing and forwarding labels appropriately in $\mathcal{S}[\![\_]\!]\_$ and adjusting previous Domain instances is routine; the curious reader may consult Appendix D.

Figure 4.18 represents sets of labels with the type Labels, the type of abstract values of the analysis. The abstract domain $\mathsf{D_C}$ of 0CFA is simply a stateful computation returning Labels. To this end, I define $\mathsf{D_C}$ in terms of the standard State monad to mutate a Cache, an abstraction of the heap discussed next.

Recall that each Label determines a syntactic value in the program. The Cache maintains, for every labelled value encountered thus far, an approximation of its action on Labels.

For example, the interpreter denotes the syntactic constructor application $S(y)$ in the right-hand side of x in (4.2) by calling the Domain method *con*. This method is implemented by updating the ConCache field under the label $S(y)$ so that it carries the constructor tag S as well as the Labels that its field y denotes. In our example, y denotes to the set $\{S(x)\}$, so the ConCache entry at label $S(y)$ is updated to $(S, [\{S(x)\}])$. This information is then available when evaluating the **case** expression in (4.2) with *select*, where the scrutinee x returns $v \triangleq \{S(y)\}$. Function *cachedCons* looks up for each label in v the respective ConCache entry and merges these entries into an environment *tag2flds* :: Tag :⇀ [Labels], representing all the possible shapes the scrutinee can take. In our case, *tag2flds* is just a singleton environment $[S \mapsto [\{S(x)\}]]$, because for the single scrutinee label $S(y)$, the ConCache only contains the entry $(S, [\{S(x)\}])$. This environment is subsequently joined with the alternatives of the case expression. The only alternative that matches is $S(z) \to z$, where z is bound to $\{S(x)\}$ from the information in the ConCache. The alternative $Z() \to x$ is dead because v does not contain a label $Z()$.

For an example involving the FunCache, consider the example (4.1), repeated here for easy reference:

$$\mathcal{S}_{\mathbf{cfa}}[\![(\mathbf{let}\ i = \bar{\lambda}x.x\ \mathbf{in}\ \mathbf{let}\ j = \bar{\lambda}y.y\ \mathbf{in}\ i\ j\ j)]\!] = \{\lambda y..\}$$

When the lambda expression $\bar{\lambda}x.x$ in the right-hand side of i is denoted by *fun*, it updates the FunCache at label $\lambda x..$ with the corresponding abstract transformer $(\lambda x \to x) :: \mathsf{D_C} \to \mathsf{D_C}$, registering it for call sites. Later, the application site $i\ j$ is denoted by *apply* with the denotations of i and j. The denotation for i is bound to *dv* and returns a set $v \triangleq \{\lambda x..\}$, while the denotation for j is bound to *da* and returns a set $a \triangleq \{\lambda y..\}$. These sets are passed to *cachedCall* which iterates over the labels in the callee v. For each such label, it looks up the abstract transformer in FunCache, applies it to the set of labels a (taking approximative measures described below) and joins together the labels returned from each call. In our example, there is just a single callee label $\lambda x..$, the abstract transformer of which is the identity function $(\lambda x \to x) :: \mathsf{D_C} \to \mathsf{D_C}$. Applying the identity transformer to the set of labels $\{\lambda y..\}$ from the denotation of the argument j returns this same set; the result of the application $i\ j$.

The above description calls a function label's abstract transformer anew at every call site. This yields the exact control-flow semantics of the original control-flow analysis work [Shivers 1991, Section 3.4], which is potentially diverging. The way 0CFA (and my implementation of it) becomes finite is by maintaining only

a single point approximation of each abstract transformer's graph (k-CFA would maintain one point per contour). Maintaining the single point approximation Maybe (Labels, Labels) in a field of the FunCache is a standard, if somewhat delicate hassle in control-flow analyses.

This single point approximation could be seen as the transformer's summary, but this summary is *call-site sensitive* and thus potentially infinite, violating my definition of "summary" in Section 4.1.3: Since the single point must be applicable at all call sites, the function body must be reanalysed as the inputs from call sites increase; thus the abstract transformer must be part of the denotation. The denotation is thus not finitely representable, and not even properly inductive! After all, $\mathsf{D_C}$ transitively (through Cache) recurses into $\mathsf{D_C} \to \mathsf{D_C}$, and the negative occurrence of $\mathsf{D_C}$ is problematic, as discussed in Section 4.4.3. This highlights a common challenge with instances of CFA: the obligation to prove that the analysis actually terminates on all inputs; an obligation that I will gloss over in this short demonstration.

Note that the given formulation of 0CFA is not modular because it lacks finite summaries; that is, the single point approximation for a function $\mathsf{A}.f$ is not generally applicable at a call site in module B such as $\mathsf{A}.f\ \mathsf{B}.x$ because the labels that $\mathsf{B}.x$ evaluate to might not be known when compiling module A. Shivers [1991, Section 3.8.2] proposes a solution to this problem that I chose not to implement for the simple proof of concept here.

4.5.4 Stateful Analysis and Annotations

Thus far, the static analyses derived from the generic denotational interpreter produce a single abstract denotation $d \triangleq \mathcal{S}[\![e]\!]_\epsilon$ for the program expression e. However, in practice static compiler analyses such as GHC's Demand Analysis usually drive a subsequent optimisation, for which a single denotation for the entire program is insufficient. Rather, we need one for every sub-expression, or at least every binding.

If we are interested in analysis results for variables *bound* in e, then either the analysis must collect these results in d, or we must redundantly re-run the analysis for subexpressions.

In this subsection, I will demonstrate how to lift a pure, *single-result analysis* — such as the usage analysis we have seen — into a stateful analysis that gives results for every binder, such that

- it collects analysis results for bound variables in a separate, global map, and
- it caches fixpoints in yet another global map, so that nested fixpoint iteration can be sped up by starting from a previous approximation.

It is a common pattern for analyses to be stateful in this way [Sergey, Vytiniotis, et al. 2017]; GHC's Demand Analysis is a good real-world example. The following demonstration targets usage analysis, but the technique should be easy to adapt for other least-fixpoint-based analyses or for type analysis (omitting the caching of fixpoints).

The Need for Isolating Bound Variable Usage

For a concrete example, let us compare the results of usage analysis from Section 4.5.1 on the expression $\mathrm{e}_1 \triangleq \textbf{let}\ i = \bar{\lambda} x.(\textbf{let}\ j = \bar{\lambda} y.y\ \textbf{in}\ j)\ \textbf{in}\ i\ i\ i$ and its subexpression $\mathrm{e}_2 \triangleq \textbf{let}\ j = \bar{\lambda} y.y\ \textbf{in}\ j$:

$$\begin{array}{lcl} \mathcal{S}_{\mathbf{usage}}[\![(\textbf{let}\ i = \bar{\lambda} x.(\textbf{let}\ j = \bar{\lambda} y.y\ \textbf{in}\ j)\ \textbf{in}\ i\ i\ i)]\!]_{\varepsilon} & = & \langle [i \mapsto \mathsf{U}_\omega, j \mapsto \mathsf{U}_\omega], \mathsf{Rep}\ \mathsf{U}_\omega \rangle \\ \mathcal{S}_{\mathbf{usage}}[\![(\textbf{let}\ j = \bar{\lambda} y.y\ \textbf{in}\ j)]\!]_{\varepsilon} & = & \langle [j \mapsto \mathsf{U}_1], \mathsf{U}_1 \fatsemi \mathsf{Rep}\ \mathsf{U}_\omega \rangle \end{array}$$

The analysis reports a different usage U_1 for the bound variable j in the subexpression e_2 versus U_ω in the containing expression e_1. This is because in order for single-result usage analysis to report information on *bound* variable j at all, it treats j like a *free* variable of i, adding a use on j for every call of i. While this treatment reflects that multiple $\textsc{Look}(j)$ events will be observed when evaluating e_1, each event associates to a *different* activation (i.e. heap entry) of the let binding j. The result U_1 reported for j in subexpression e_2 is more useful because it reflects that every *activation* of the binding j is looked up at most once over its lifetime, which is indeed the formal property of interest in Section 4.6.

Rather than to re-run the analysis for every let binding such as j, I will now demonstrate a way to write out an *annotation* for j, just before analysis leaves the **let** that binds j. Annotations for bound variables constitute analysis state that will be maintained separately from information on free variables.

Maintaining Annotations by Implementing StaticDomain

Figure 4.19 lifts the existing definition for single-result usage analysis $\mathcal{S}_{\mathbf{usage}}[\![\_]\!]_\_$ into a stateful analysis $\mathcal{S}_{\mathbf{usage}\rightsquigarrow}[\![\_]\!]_\_$ that writes out usage information on bound variables into a separate map. Consider the result on the example expression

```
class Domain d ⇒ StaticDomain d where
  type Ann d :: *
  extractAnn :: Name → d → (d, Ann d)
  funS    :: Monad m ⇒ Name → (m d → m d) → m d
  selectS :: Monad m ⇒ m d → (Tag :⇀ ([m d] → m d)) → m d
  bindS   :: Monad m ⇒ Name → d → (d → m d) → (d → m d) → m d
instance StaticDomain D_U where
  type Ann D_U = U
  extractAnn x ⟨φ, v⟩ = (⟨Map.delete x φ, v⟩, φ !? x)
  funS x f = do
    ⟨φ, v⟩ ← f (return ⟨[x ↦ U_1], Rep U_ω⟩)
    return ⟨φ[x ↦ U_0], φ !? x ⨾ v⟩
  selectS md mfs = do
    d ← md
    alts ← sequence [ f (replicate (conArity k) (return ⟨ε, Rep U_ω⟩))
                    | (k, f) ← Map.assocs mfs]
    return (d ≫ lub alts)
  bindS _ init rhs body = kleeneFixAboveM init rhs >=> body
kleeneFixAboveM :: (Monad m, Lat a) ⇒ a → (a → m a) → m a
S_usage⇝[[e]]_ρ = runAnn (S[[e]]_return◁ρ) :: (D_U, Name :⇀ U)
data Refs s d = Refs (STRef s (Name :⇀ d)) (STRef s (Name :⇀ Ann d))
newtype AnnT s d a = AnnT (Refs s d → ST s a)
type AnnD s d = AnnT s d d
runAnn :: (∀s. AnnD s d) → (d, Name :⇀ Ann d)
instance Monad (AnnT s d) where ...
instance Trace d ⇒ Trace (AnnD s d) where
  step ev (AnnT f) = AnnT (λrefs → step ev <$> f refs)
instance StaticDomain d ⇒ Domain (AnnD s d) where ...
readCache  :: Lat d ⇒ Name → AnnD s d
writeCache :: Name → d → AnnT s d ()
annotate   :: StaticDomain d ⇒ Name → AnnD s d → AnnD s d
instance (Lat d, StaticDomain d) ⇒ HasBind (AnnD s d) where
  bind x rhs body = do
    init ← readCache x
    let rhs' d_1 = do d_2 ← rhs (return d_1); writeCache x d_2; return d_2
    annotate x (bindS x init rhs' (body ∘ return))
```

Fig. 4.19: Trace transformer AnnT for recording annotations and caching of fixpoints

e_1 from above, where the pair $(d, anns)$ returned by $\mathcal{S}_{\mathbf{usage}\rightsquigarrow}[\![\_]\!]_\_$ is printed as $d \rightsquigarrow anns$:

$$\mathcal{S}_{\mathbf{usage}\rightsquigarrow}[\![(\mathbf{let}\ i = \bar{\lambda}x.\mathbf{let}\ j = \bar{\lambda}y.y\ \mathbf{in}\ j\ \mathbf{in}\ i\ i\ i)]\!]_\varepsilon = \langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_\omega \rangle \rightsquigarrow [i \mapsto \mathsf{U}_\omega, j \mapsto \mathsf{U}_1]$$

The annotations for both bound variables i and j are returned in an annotation environment separate from the empty abstract free variable environment $\varepsilon :: \mathsf{Uses}$ of the expression. Furthermore, the use U_1 reported for j is exactly as when analysing the subexpression e_2 in isolation, as required.

Lifting the single-result analysis $\mathcal{S}_{\mathbf{usage}}[\![\_]\!]_\_$ defined on $\mathsf{D_U}$ to a stateful analysis $\mathcal{S}_{\mathbf{usage}\rightsquigarrow}[\![\_]\!]_\_$ requires implementing a type class instance $\mathsf{StaticDomain}\ \mathsf{D_U}$. Before going into detail about how this lifting is implemented in terms of type AnnT, let us review its type class interface. The type class $\mathsf{StaticDomain}$ defines the associated type Ann of annotations in the particular static domain, along with a function $extractAnn\ x\ d$ for extracting information on a let-bound x from the denotation d. The instance for $\mathsf{D_U}$ instantiates $\mathsf{Ann}\ \mathsf{D_U}$ to bound variable use U, and $extractAnn\ x\ \langle \varphi, v \rangle$ isolates the free variable use $\varphi\,!\,x$ as annotation. The remaining type class methods $funS$, $selectS$ and $bindS$ are simple monadic generalisations of their counterparts in Domain and $\mathsf{HasBind}$. As can be seen, I again needed to slightly revise the $\mathsf{HasBind}$ type class in order to pass the name x of the let-bound variable to $bind$ and $bindS$, similar as for fun.

The implementation of $\mathsf{StaticDomain}$ requires very little extra code to maintain, because the original definitions of fun, $select$ and $bind$ can be recovered in terms of the generalised definitions via the standard $\mathsf{Identity}$ monad as follows, where $coerce$ denotes a safe zero-cost coercion function provided by GHC [Breitner et al. 2014]:

```
newtype Identity a = Identity { runIdentity :: a }
fun' :: StaticDomain d ⇒ Name → Label → (d → d) → d
fun' x f = runIdentity (funS x (coerce f))
select' :: StaticDomain d ⇒ d → (Tag :⇀ ([d] → d)) → d
select' d fs = runIdentity (selectS (Identity d) (coerce fs))
bind' :: (Lat d, StaticDomain d) ⇒ Name → (d → d) → (d → d) → d
bind' x rhs body = runIdentity (bindS x ⊥ (coerce rhs) (coerce body))
```

Any reasonable instance of $\mathsf{StaticDomain}$ must satisfy the laws $fun = fun'$, $select = select'$ and $bind = bind'$.

Let us now look at how AnnT extends the pure, single-result usage analysis into a stateful one that maintains annotations.

Trace Transformer AnnT for Stateful Analysis

Every instance StaticDomain *d* induces an instance Domain (AnnD *s d*), where the type AnnD *s d* is another example of a *trace transformer*: It transforms the Trace instance on type *d* into a Trace instance for AnnD *s d*. The abstract domain AnnD is defined in terms of the abstract trace type AnnT, which is a standard ST monad utilising efficient and pure mutable state threads [Launchbury and Peyton Jones 1994], stacked below a Refs environment that carries the mutable ref cells. A stateful analysis computation $\forall s.$ AnnD *s* $\mathsf{D}_\cup$ is then run with *runAnn*, initialising Refs with ref cells pointing to empty environments. (The universal quantification over *s* in the type of *runAnn* ensures that no mutable STRef from Refs escapes the functional state thread of the underlying ST computation [Launchbury and Peyton Jones 1994].)

The induced instance Domain (AnnD *s d*) is implemented by lifting operations *stuck*, *apply* and *con* into monadic AnnT *s d* context and by calling *funS* and *selectS*. Finally, the stateful nature of the domain AnnD *s d* is exploited in the HasBind (AnnD *s d*) instance, in two ways:

- The call to *annotate* writes out the annotation on the let-bound variable *x* that is extracted from the denotation returned by the call to *bindS*. The omitted definition of *annotate* is just a thin wrapper around *extractAnn* to store the extracted annotation in the Name :⇀ Ann *d* ref cell of Refs, the contents of which are returned from *runAnn*.

- The calls to *readCache* and *writeCache* read from and write to the Name:⇀ *d* ref cell of Refs in order to provide the initial value *init* for fixpoint iteration. To this end, *kleeneFix* is generalised to *kleeneFixAboveM init f* which iterates the monadic function *f* starting from *init* until a reductive point of *f* is found (i.e. a *d* such that *f d* $\sqsubseteq$ *return d*). When fixpoint iteration is first started, there is no cached value, in which case *readCache* returns $\bot$ to be used as the initial value, just as for the single-result analysis. However, after every iteration of *rhs*, the call to *writeCache* persists the current iterate, which will be the initial value of the fixpoint iteration for any future calls to *bind* for the same let binding *x*.

The caching technique is important because naïve fixpoint iteration in single-result analysis can be exponentially slow for nested let bindings, such as in

$$\bar{\lambda} z.\mathbf{let}\ x_1 = (\mathbf{let}\ x_2 = \ldots(\mathbf{let}\ x_n = z\ \mathbf{in}\ x_n)\ldots\ \mathbf{in}\ x_2)\ \mathbf{in}\ x_1.$$

Naïvely, every let binding needs two iterations per one iteration of its enclosing binding: the first iteration assuming $\bot$ as the initial value for x_i and the next assuming the fixpoint $\langle [z \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle$. Ultimately, z is used in the denotation of x_n, ..., x_1, totalling to 2^n iterations for x_n during stateless analysis.

Stateful caching of the previous fixpoint improves this drastically. The right-hand side of $x_n = z$ is only iterated $n+1$ times in total: once with $\bot$ as the initial value for x_n, once more to confirm the fixpoint $\langle [z \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle$ and then $n-1$ more times to confirm the fixpoints of $x_{n-1}, ..., x_1$.

It is possible to improve the number of iterations for x_n to a constant, by employing classic chaotic iteration and worklist techniques. These techniques require a decoupling of iteration order from the lexical nesting imposed by the syntax tree, instead choosing the next iteratee by examining the graph of data flow dependencies, as in Graf [2017]. Crucially, such sophisticated and stateful data-flow frameworks can be developed and improved without complicating the analysis domain, which is often very complicated in its own right.

4.5.5 Case Study: GHC's Demand Analyser

To test how well my denotational interpreter framework scales to real-world applications, I applied the design pattern to GHC's existing Demand Analyser and will reproduce the salient points here. GHC's Demand Analyser infers nested usage [Sergey, Vytiniotis, et al. 2017], strictness [Peyton Jones, Sestoft, et al. 2006] and boxity information. These analysis results thus fuel a number of optimisations, such as dead code elimination and unboxing through the worker/wrapper transformation [Gill and Hutton 2009], update avoidance [Gustavsson 1998], eta expansion and eta reduction, and inlining under lambdas, to name a few.

Concretely, my refactoring entailed

- identifying which parts of the analyser need to be part of the Domain interface,
- writing an abstract denotational interpreter for GHC Core, the typed intermediate representation of GHC,
- validating the usefulness of this interpreter by instantiating it at the GHC Core-specific analogue of the concrete by-need domain $\mathsf{D}_{\mathbf{ne}}$, and finally
- defining the abstract Domain instance for Demand Analysis, to replace its compositional analysis function on expressions by a call to the denotational interpreter.

```
data Expr
  = Var       Id
  | Lit       Literal
  | App       Expr Expr
  | Lam       Var Expr
  | Let       Bind Expr
  | Case      Expr Id Type Alt
  | Cast      Expr Coercion
  | Tick      Tickish Expr
  | Type      Type
  | Coercion Coercion
data Var = Id ... | TyVar ... | CoVar ...
type Id = Var  -- always a term-level Id
data Literal = LitNumber ... | LitFloat ... | LitString ...
type Alt = (AltCon, [Var], Expr)
data AltCon = LitAlt Literal | DataAlt DataCon | DEFAULT
data Bind = NonRec Id Expr | Rec [(Id, Expr)]
data Type       = ...
data Coercion   = ...
```

Fig. 4.20: GHC Core

The resulting compiler bootstraps and passes the testsuite.

GHC Core

GHC Core implements a variant of the polymorphic lambda calculus System F_ω called System F_C [Sulzmann et al. 2007]. Its definition in GHC is given in Figure 4.20 and includes explicit type applications as well as witnesses of type equality constraints called *coercions*.

GHC Core's Expr has a lot in common with the untyped object language Exp introduced in Section 4.1.1. For example, there are constructors for Var, App, Lam, Let and Case. There are a number of differences, however:

- GHC Core allows non-variable arguments in applications. This has implications on the denotational interpreter, which must let-bind non-variable arguments to establish A-normal form on-the-fly.

- There is no distinguished ConApp form. That is because data constructors are just special kinds of Ids and may be unsaturated; the interpreter must eta-expand such data constructor applications on-the-fly.
- Case alternatives allow matching on literals (LitAlt) as well as data constructors (DataAlt), and include a default alternative (DEFAULT) that matches any case not matched by other alternatives. Furthermore, after Case evaluates the scrutinee, its value is bound to a designated Id called the *case binder* that scopes over all case alternatives.
- Beyond data constructors, there are other distinguished Ids without a local binding, such as "global" identifiers imported from a different module, class method selectors and primitive operations defined by the runtime system.
- Let bindings are either explicitly non-recursive (NonRec) or a mutually recursive group with potentially many bindings (Rec).
- Not shown in Figure 4.20 is GHC's support for *inline unfoldings* attached to let-bound Ids as well as *rewrite rules* declared by `RULES` pragmas. Each give rise to additional right-hand sides which must be handled with conservative care. Mistreatment of these subtle constructs in the Demand Analyser has caused numerous bugs over the years.

Beyond these differences, GHC Core includes forms for embedding Literals, Types and Coercions in select expression forms. Type abstraction and application use regular Lam and App constructors, whereas rewriting an expression's type along a Coercion happens through Casts. The constructor Tick annotates debugging and profiling information and can be ignored.

A Semantic Domain for GHC Core

Figure 4.21 defines the semantic domain abstraction for which I implemented both a concrete ByNeed instance as well as an abstract instance for Demand Analysis. Its design was inspired by the domain definition in Figure 4.6, but ultimately driven by the hands-on desire to accommodate both ByNeed and Demand Analysis as instances.

The *stuck*, *con*, *fun*, *apply* and *select* methods serve the exact same purpose as in prior sections, generalised to deal with the Core expressions they are modelled

```
data Event = Look Id | LookArg CoreExpr | Update
           | App1 | App2 | Case1 | Case2 | Let1
class Trace d where step :: Event → d → d

class Domain d where
  stuck :: d
  lit :: Literal → d
  global :: Id → d
  classOp :: Id → Class → d
  primOp :: Id → PrimOp → d
  fun :: Id → (d → d) → d
  con :: DataCon → [d] → d
  apply :: d → (Bool, d) → d
  select :: d → CoreExpr → Id → [DAlt d] → d
  erased :: d
  keepAlive :: [d] → d → d
type DAlt d = (AltCon, [Id], d → [d] → d)
data BindHint = BindArg Id | BindLet Bind
class HasBind d where
  bind :: BindHint → [[d] → d] → ([d] → d) → d
```

Fig. 4.21: A Domain interface for GHC Core

after. Method *apply* receives an additional Bool to tell whether it is a runtime-irrelevant type application. Unsurprisingly, there is a method *lit* for embedding Literals, similar to *con*. Demand Analysis assigns special meaning to primitive operations (*primOp*), class method selectors (*classOp*) and imported Ids (*global*), so each get their own Domain method.

Types and coercions are erased at runtime, represented by method *erased*. Coercion expressions, inline unfoldings and rewrite RULES keep alive their free variables (*keepAlive*).

The HasBind type class accommodates both non-recursive as well as mutually recursive let bindings. The BindHint is used to communicate whether such a binding comes from the on-the-fly ANF-isation pass of the interpreter (BindArg) or whether it was a manifest let binding in the Core program (BindLet).

```
type D d = (Trace d, Domain d, HasBind d)
anfise        :: D d ⇒ [Expr] → (Name :⇀ d) → ([d] → d) → d
evalConApp    :: D d ⇒ DataCon → [d] → d
𝒮⟦_⟧_         :: D d ⇒ Expr → (Name :⇀ d) → d
𝒮⟦Type _⟧ρ                  = erased
𝒮⟦Lit l⟧ρ                   = lit l
𝒮⟦Var x⟧ρ | not special     = ρ ! x
           | otherwise      = ...
𝒮⟦Lam x e⟧ρ                 = fun x (λd → step App₂ (𝒮⟦e⟧ρ[x↦d]))
𝒮⟦e@App { }⟧ρ
  | Var v ← f, Just dc ← isDataConWorkId_maybe v
  = anfise as ρ (evalConApp dc)
  | otherwise
  = anfise (f : as) ρ $ λ(df : das) →
      go df (zipWith (λd a → (d, isTypeArg a)) das as)
  where
    (f, as) = collectArgs e
    go df [ ] = df
    go df ((d, is_ty) : ds) = go (step App₁ $ apply df (is_ty, d)) ds
𝒮⟦Let b@(NonRec x rhs) body⟧ρ =
  bind (BindLet b)
    [λds → keepAliveUnfRules x ρ (𝒮⟦rhs⟧ρ)]
    (λds → step Let₁ (𝒮⟦body⟧ρ[x↦step (Lookup x) (only ds)]))
...
```

Fig. 4.22: A glimpse of the Glasgow Haskell Denotational Interpreter (GHDi)

The Glasgow Haskell Denotational Interpreter (GHDi)

Figure 4.22 shows a slightly adjusted and abridged version of the denotational interpreter. The actual definition takes around 100 lines of Haskell; its full definition can be looked up in Appendix C. Its highlights include erasure of types, a new case for literals, on-the-fly ANF-isation in the application case and picking out data constructor application from regular function application in order to eta expand accordingly in *evalConApp*. Whenever an ANF-ised argument is looked up, a LookArg event is emitted; this is simply for a lack of a globally unique Id. In the Let case, the call to *keepAliveUnfRules* makes sure to keep alive the free variables of inline unfoldings and rewrite rules attached to x.

The Domain and HasBind instance for the concrete semantics $\mathsf{D}_{\mathbf{ne}}$ is routine. The resulting denotational interpreter can execute GHC Core expressions. To demonstrate this, I wrote a small REPL around it that takes Haskell expressions, optimises them using the GHC middle-end and then executes the resulting GHC Core:

```
$ ./ghdi $(ghc --print-libdir)
prompt> let f x = x*42 :: Int; {-# NOINLINE f #-} in even $ f 3
Above expression as (optimised) Core:
  join {
    f_sZe [InlPrag=NOINLINE, Dmd=1C(1,L)] :: Int -> Bool
    [LclId[JoinId(1)(Just [!])], Arity=1, Str=<1L>]
    f_sZe (x_aYj [OS=OneShot] :: Int)
      = case x_aYj of { I# x1_aHU ->
        case remInt# (*# x1_aHU 42#) 2# of {
          __DEFAULT -> False;
          0# -> True
        }
        } } in
  jump f_sZe (I# 3#)
Trace of denotational interpreter:
Let1->App1->Lookup(f_sZe)->Update->App2->Case1->
  LookupArg(I# 3#)->Update->Case2->Case1->App1->
  App1->App2->App2->LookupArg(*# x1_aHU 42#)->App1->App1->
  App2->App2->Update->Case2-><(True, [0↦_, 1↦_, 2↦_])>
```

```
data Card = C_00 | C_01 | C_0N | C_10 | C_11 | C_1N
data SubDemand = ...
data Demand = Card :* SubDemand
type DmdT s v = AnalEnv → SubDemand → AnalM s (v, Uses)
type DmdVal = [Demand]
type DmdD s = DmdT s DmdVal
instance Trace (DmdD s) where
  step (Look x) d = λenv sd → do
    (v, φ) ← d env sd
    if isBoundAtTopLvl env x then ... else pure (v, φ + [x ↦ C_11 :* sd])
  step _ d = d
instance Domain (DmdD s) where ...
instance HasBind (DmdD s) where ...
```

Fig. 4.23: A rough outline of the semantic domain of Demand Analysis

Demand Analysis as Denotational Interpreter

Figure 4.23 gives a rough sketch of the semantic domain definition for Demand Analysis.

The overall goal is to infer a Demand for each variable binding, where a demand $n :* sd$ describes how often ($n ::$ Card) and how deep ($sd ::$ SubDemand) a variable is evaluated. A Card is a generalisation of usage cardinality U, describing an *interval* of evaluation cardinality; for example, C_1N means "evaluated at least once, but potentially many times". The cardinalities with lower bound 0 (C_00, C_01, C_0N) correspond to the usage cardinalities U_0, U_1, U_ω; the lower bound encodes whether or not a variable was evaluated strictly. A SubDemand is best understood as an abstraction of evaluation contexts. An in-depth description with examples can be found in Sergey, Vytiniotis, et al. [2017].

The abstract trace type DmdT produces some value v as well as a Uses, just as for T_U in Section 4.5.1. However, it does so in a rather deep nest of types:

- AnalM s plays the role of AnnT s in Section 4.5.4, maintaining annotations and speeding up fixpoint iteration.
- The analysis result is *indexed* by a SubDemand; a description of how deep the expression is to be evaluated. The more precise the SubDemand

describes the actual evaluation contexts the expression occurs in at runtime, the more accurate are the Uses returned for that expression.

- Furthermore, an AnalEnv carries global state such as optimisation flags, means for reducing types and further syntactic information about bindings, such as whether a variable is bound at the top-level.

An abstract domain defined as a function sounds antithetical to the mantra in Section 4.1.3 that abstract domains are finitely represented. However, Demand Analysis only ever maintains one particular point of the indexed domain, that is, every expression is analysed under one particular SubDemand. This SubDemand may increase during fixpoint iteration, though, causing another round of analysis, but there is always a top element in SubDemand to default to (for example for exported functions). We apply the typical widening measures in HasBind, so in practice Demand Analysis has not run into infinite loops for a couple of years.

Type DmdVal is similar to $\mathsf{Value}_{\mathsf{U}}$, except that it lists full Demands instead of flat usage cardinalities U.

The Trace instance is very similar to the one for D_{U}, it is just a little bit more complicated because of special code for top-level bindings and the fact that bindings get annotated with demands instead of simple usage cardinalities. The demand $\mathsf{C\_11} :* sd$ describes a single, strict use of the variable in the evaluation context described by sub-demand sd.

The resulting analysis is sufficient to bootstrap the compiler and passes the testsuite. However, the compiler performance takes a serious hit due to the implementation of $bind :: \mathsf{BindHint} \to [[d]] \to d] \to ([d] \to d) \to d$. The way fixpoint iteration updates one binding d in mutually recursive groups $[d]$ at a time is very inefficient for the linked list representation, also because every $[d]$ ultimately turns into as many updates of the $\mathsf{Name} :\rightharpoonup d$ mapping. It would be far preferable to operate on the $\mathsf{Name} :\rightharpoonup d$ environment directly. Finding a good abstraction that achieves this without exposing the whole environment is left for future work.

It is nice that different static analyses fit into the same framework as the call-by-need semantics. Another important benefit is that correctness proofs become simpler, as we will see next.

4.6 Generic Abstract By-Name and By-Need Interpretation

In this section I prove and apply an abstract interpretation theorem of the form

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

This statement can be read as follows: For an expression e, the *static analysis* $\mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]$ on the right-hand side *overapproximates* ($\sqsupseteq$) a property of the by-need *semantics* $\mathcal{S}_{\mathbf{need}}[\![e]\!]$ on the left-hand side. The abstraction function $\alpha_{\mathcal{S}}$, given in Figure 4.24, defines the semantic property of interest in terms of the abstract domain $\widehat{\mathsf{D}}$ of $\mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_\rho$, which is short for $\mathcal{S}[\![e]\!]_\rho :: \widehat{\mathsf{D}}$. That is: the type class instances on $\widehat{\mathsf{D}}$ determine $\alpha_{\mathcal{S}}$, and hence the semantic property that is soundly abstracted by $\mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_\rho$, as per the relationship between semantic properties and Galois connections outlined in Section 2.3.

I will instantiate the theorem at $\mathsf{D_U}$ in order to prove that usage analysis $\mathcal{S}_{\mathbf{usage}}[\![e]\!]_\rho = \mathcal{S}_{\mathsf{D_U}}[\![e]\!]_\rho$ infers absence, just as absence analysis in Section 4.1. This proof will be much simpler than the proof for Theorem 4.1, because the complicated preservation proof is reusably contained in the abstract interpretation theorem.

Similar to the storyline in Section 4.4, I will show how to *use* Theorem 4.18 before developing the theoretical framework to *prove* it, which I do in Sections 4.6.6 to 4.6.9.

4.6.1 A Reusable Abstract By-Need Interpretation Theorem

In this subsection, I introduce Theorem 4.18 for abstract by-need interpretation, which we will apply to prove usage analysis sound in Section 4.6.3. The theorem corresponds to the following derived inference rule, referring to the *abstraction laws* in Figure 4.25 by name:

$$\frac{\textsc{Mono}\quad \textsc{Step-App}\quad \textsc{Step-Sel}\quad \textsc{Stuck-App}\quad \textsc{Stuck-Sel}\quad \textsc{Beta-App}\quad \textsc{Beta-Sel}\quad \textsc{ByName-Bind}\quad \textsc{Step-Inc}\quad \textsc{Update}}{\alpha_{\mathcal{S}}(\mathcal{S}_{\mathbf{need}}[\![e]\!]) \sqsubseteq \mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]}$$

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

$$\alpha_{\mathcal{S}} : ((\mathsf{Name} :\rightharpoonup \mathsf{D_{ne}}) \to \mathsf{D_{ne}}) \to ((\mathsf{Name} :\rightharpoonup \widehat{\mathsf{D}}) \to \widehat{\mathsf{D}})$$
$$\alpha_{\mathbb{E}} : \wp(\mathsf{Heap_{ne}} \times (\mathsf{Name} :\rightharpoonup \mathsf{D_{ne}})) \rightleftarrows (\mathsf{Name} :\rightharpoonup \widehat{\mathsf{D}}) : \gamma_{\mathbb{E}}$$
$$\alpha_{\mathbb{D}} : \mathsf{Heap_{ne}} \to (\wp(\mathsf{D_{ne}}) \rightleftarrows \widehat{\mathsf{D}}) : \gamma_{\mathbb{D}}$$
$$\alpha_{\mathbb{T}} : \wp(\mathsf{T}\ (\mathsf{Value_{ne}}, \mathsf{Heap_{ne}})) \rightleftarrows \widehat{\mathsf{D}} : \gamma_{\mathbb{T}} \qquad \beta_{\mathbb{T}} : \mathsf{T}\ (\mathsf{Value_{ne}}, \mathsf{Heap_{ne}}) \to \widehat{\mathsf{D}}$$

$$\begin{aligned}
\alpha_{\mathcal{S}}(S)(\widehat{\rho}) &= \alpha_{\mathbb{T}}(\{\ S(\rho)(\mu) \mid (\mu,\rho) \in \gamma_{\mathbb{E}}(\widehat{\rho})\ \}) \\
\alpha_{\mathbb{E}}(E)(x) &= \alpha_{\mathbb{T}}(\{\ \rho(x)(\mu) \mid (\mu,\rho) \in E\ \}) \\
\alpha_{\mathbb{D}}(\mu)(D) &= \alpha_{\mathbb{T}}(\{\ d(\mu) \mid d \in D\ \}) \\
\alpha_{\mathbb{T}}(T) &= \bigsqcup\{\ \beta_{\mathbb{T}}(\tau) \mid \tau \in T\ \}
\end{aligned}$$

$$\beta_{\mathbb{T}}(\tau) = \begin{cases} \mathit{step}\ e\ (\beta_{\mathbb{T}}(\tau')) & \text{if } \tau = \mathsf{Step}\ e\ \tau' \\ \mathit{stuck} & \text{if } \tau = \mathsf{Ret}\ (\mathsf{Stuck}, \mu) \\ \mathit{fun}\ (\alpha_{\mathbb{D}}(\mu) \circ f^* \circ \gamma_{\mathbb{D}}(\mu)) & \text{if } \tau = \mathsf{Ret}\ (\mathsf{Fun}\ f, \mu) \\ \mathit{con}\ k\ (\mathit{map}\ (\alpha_{\mathbb{D}}(\mu) \circ \{\_\})\ \mathit{ds}) & \text{if } \tau = \mathsf{Ret}\ (\mathsf{Con}\ k\ \mathit{ds}, \mu) \end{cases}$$

Fig. 4.24: Galois connection $\alpha_{\mathcal{S}}$ for by-need abstraction derived from Trace, Domain and Lat instances on $\widehat{\mathsf{D}}$

Note that I get to determine the abstraction function $\alpha_{\mathcal{S}}$ based on the Trace, Domain and Lat instance on *your* $\widehat{\mathsf{D}}$. Figure 4.24 defines how $\alpha_{\mathcal{S}}$ is thus derived.

Let us calculate $\alpha_{\mathcal{S}}$ for the closed example expression $\mathrm{e} \triangleq \mathbf{let}\ i = (\bar{\lambda} y.\bar{\lambda} x.x)\ i\ \mathbf{in}\ i$:

$$\begin{aligned}
&\alpha_{\mathcal{S}}(\mathcal{S}_{\mathbf{need}}[\![(\mathbf{let}\ i = (\bar{\lambda} y.\bar{\lambda} x.x)\ i\ \mathbf{in}\ i)]\!])(\varepsilon) \\
&= \beta_{\mathbb{T}}(\mathcal{S}_{\mathbf{need}}[\![(\mathbf{let}\ i = (\bar{\lambda} y.\bar{\lambda} x.x)\ i\ \mathbf{in}\ i)]\!]_{\varepsilon}(\varepsilon)) && (4.1) \\
&= \beta_{\mathbb{T}}(\textsc{Let}_1 \hookrightarrow \textsc{Look}(i) \hookrightarrow \textsc{App}_1 \hookrightarrow \textsc{App}_2 \hookrightarrow \textsc{Upd} \hookrightarrow \langle(\lambda, [0 \mapsto _])\rangle) && (4.2) \\
&= \mathit{step}\ \mathsf{Let}_1\ \$\ \mathit{step}\ (\mathsf{Look}\ \texttt{"i"})\ \$\ \ldots\ \$ && (4.3) \\
&\quad \mathit{fun}\ (\lambda \widehat{d} \to \bigsqcup\{\beta_{\mathbb{T}}(\textsc{App}_2 \hookrightarrow d([0 \mapsto _])) \mid d \in \gamma_{\mathbb{D}}([0 \mapsto _])(\widehat{d})\}) \\
&\sqsubseteq \mathit{step}\ \mathsf{Let}_1\ \$\ \mathit{step}\ (\mathsf{Look}\ \texttt{"i"})\ \$\ \ldots\ \$\ \mathit{fun}\ (\lambda \widehat{d} \to \mathit{step}\ \mathsf{App}_2\ \widehat{d}) && (4.4) \\
&= \langle [i \mapsto \mathsf{U}_1], \mathsf{U}_1 \wr \mathsf{Rep}\ \mathsf{U}_\omega \rangle :: \mathsf{D_U} && (4.5) \\
&= \mathcal{S}_{\mathbf{usage}}[\![(\mathbf{let}\ i = (\bar{\lambda} y.\bar{\lambda} x.x)\ i\ \mathbf{in}\ i)]\!]_{\varepsilon}
\end{aligned}$$

In (4.1), $\alpha_{\mathcal{S}}(\mathcal{S}_{\mathbf{need}}[\![e]\!])(\varepsilon)$ simplifies to $\beta_{\mathbb{T}}(\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\varepsilon}(\varepsilon))$. Function $\beta_{\mathbb{T}}$ then folds the by-need trace (4.2) into an abstract domain element in $\widehat{\mathsf{D}}$. It does so by eliminating every Step *ev* in the trace with a call to *step ev*, and every concrete Value at the end of the trace with a call to the corresponding Domain method,

Mono
step, *stuck*, *fun*, *apply*, *con*, *select*, *bind* monotone

Step-App
$step\ ev\ (apply\ d\ a) \sqsubseteq apply\ (step\ ev\ d)\ a$

Step-Sel
$step\ ev\ (select\ d\ alts) \sqsubseteq select\ (step\ ev\ d)\ alts$

Stuck-App
$$\frac{d \in \{stuck, con\ k\ ds\}}{stuck \sqsubseteq apply\ d\ a}$$

Stuck-Sel
$$\frac{d \in \{stuck, fun\ x\ f\} \cup \{con\ k\ ds \mid k \notin dom\ alts\}}{stuck \sqsubseteq select\ d\ alts}$$

Beta-App
$$\frac{f \text{ polymorphic} \qquad x \text{ fresh}}{f\ a \sqsubseteq apply\ (fun\ x\ f)\ a}$$

Beta-Sel
$$\frac{alts \text{ polymorphic} \qquad k \in dom\ alts}{(alts\,!\,k)\ ds \sqsubseteq select\ (con\ k\ ds)\ alts}$$

ByName-Bind
$$\frac{rhs, body \text{ polymorphic}}{body\ (lfp\ rhs) \sqsubseteq bind\ rhs\ body}$$

Step-Inc
$d \sqsubseteq step\ ev\ d$

Update
$step\ \mathsf{Upd}\ d = d$

Fig. 4.25: By-name and by-need abstraction laws for type class instances of abstract domain $\widehat{\mathsf{D}}$

following the structure of types as in Backhouse and Backhouse [2004]. Since $\widehat{\mathsf{D}}$ has a Lat instance, $\beta_{\mathbb{T}}$ is a *representation function* [Nielson et al. 1999, Section 4.3], giving rise to Galois connections $\alpha_{\mathbb{T}} \rightleftarrows \gamma_{\mathbb{T}}$ and $\alpha_{\mathbb{D}}(\mu) \rightleftarrows \gamma_{\mathbb{D}}(\mu)$. This implies that $\alpha_{\mathbb{D}}(\mu) \circ \gamma_{\mathbb{D}}(\mu) \sqsubseteq id$, justifying the approximation step ($\sqsubseteq$) in (4.4). For the concrete example, we instantiate $\widehat{\mathsf{D}}$ to D_{U} in step (4.5) to assert that the resulting usage type indeed coincides with the result of $\mathcal{S}_{\mathbf{usage}}[\![\_]\!]$, as predicted by the abstract interpretation theorem.

The abstraction function $\alpha_{\mathbb{D}}$ for by-need elements d is a bit unusual because it is *indexed* by a heap to give meaning to addresses referenced by d. My framework is carefully set up in a way that $\alpha_{\mathbb{D}}(\mu)$ is preserved when μ is modified by memoisation "in the future", reminiscent of Kripke's possible worlds. For similar reasons, the abstraction function for environments pairs up definable by-need environments ρ, the entries of which are of the form $step\ (\mathsf{Look}\ y)\ (fetch\ a)$, with definable heaps μ, the entries of which are of the form $memo\ a\ d$.

Thanks to fixing $\alpha_{\mathcal{S}}$, we can prove the following abstraction theorem, corresponding to the inference rule at the begin of this subsection:

□ 163 **Theorem 4.18** (Abstract By-need Interpretation)**.** *Let e be an expression, $\widehat{\mathsf{D}}$ a domain with instances for* Trace, Domain, HasBind *and* Lat*, and let $\alpha_{\mathcal{S}}$ be the abstraction function from Figure 4.24. If the abstraction laws in Figure 4.25 hold, then $\mathcal{S}_{\widehat{\mathsf{D}}}[\![\_]\!]$ is an abstract interpreter that is sound wrt. $\alpha_{\mathcal{S}}$, that is,*

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

Let us unpack law Beta-App to see how the abstraction laws in Figure 4.25 are to be understood. To prove Beta-App, one has to show that $\forall f\ a\ x.\ f\ a \sqsubseteq \mathit{apply}\ (\mathit{fun}\ x\ f)\ a$ in the abstract domain $\widehat{\mathsf{D}}$. This states that summarising f through *fun*, then *apply*ing the summary to a must approximate a direct call to f; it amounts to proving correct the summary mechanism. In Section 4.1, I have proved a substitution Lemma 4.3, which is a syntactic form of this statement. The "f polymorphic" premise asserts that f is definable at polymorphic type $\forall d.\ (\mathsf{Trace}\ d, \mathsf{Domain}\ d, \mathsf{HasBind}\ d) \Rightarrow d \to d$, which is important to prove Beta-App (in Section 4.6.2).

Law Beta-Sel states a similar property for data constructor redexes. Law ByName-Bind 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 laws are congruence rules involving *step* and *stuck* as well as a monotonicity requirement for all involved operations. These laws follow the mantra "evaluation improves approximation"; for example, law Stuck-App expresses that applying a stuck term or constructor evaluates to (and thus approximates) a stuck term, and Stuck-Sel expresses the same for *select* stack frames. In Section 4.6.8, I show a result similar to Theorem 4.18 for by-name evaluation which does not require the by-need specific laws Step-Inc and Update.

Note that none of the laws mention the concrete semantics or the abstraction function $\alpha_{\mathcal{S}}$. This is how fixing the concrete semantics and $\alpha_{\mathcal{S}}$ pays off; the usual abstraction laws such as $\alpha_{\mathbb{D}}(\mu)(\mathit{apply}\ d\ a) \sqsubseteq \widehat{\mathit{apply}}\ (\alpha_{\mathbb{D}}(\mu)(d))\ (\alpha_{\mathbb{D}}(\mu)(a))$ further decompose into Beta-App. I think this is a nice advantage to my approach, because the author of the analysis does not need to reason about by-need heaps in order to soundly approximate a semantic trace property expressed via Trace instance!

4.6.2 A Modular Proof for Beta-App: A Simpler Substitution Lemma

In order to instantiate Theorem 4.18 for usage analysis in Section 4.6.3, I need to prove in particular that $\mathsf{D_U}$ satisfies the abstraction law Beta-App in Figure 4.25. Beta-App corresponds to the syntactic substitution Lemma 4.3 of Section 4.1, and this subsection presents its proof.

Before we discuss this proof, note that the proof for Lemma 4.3 has a serious drawback: It relies on knowing the complete definition of $\mathcal{A}[\![\_]\!]$ and thus is *non-modular*. As a result, the proof complexity scales in the size of the interpreter, and whenever the definition of $\mathcal{A}[\![\_]\!]$ changes, Lemma 4.3 must be updated. The complexity of such non-modular proofs would become unmanageable for large denotational interpreters such as for WebAssembly [Brandl et al. 2023].

For Beta-App, dubbed *semantic substitution*, the proof fares much better:

Lemma 4.19 (Beta-App, Semantic substitution). *Let* $x :: \mathsf{Name}$ *be fresh,* $a :: \mathsf{D_U}$ *and* $f :: (\mathsf{Trace}\ d, \mathsf{Domain}\ d, \mathsf{HasBind}\ d) \Rightarrow d \to d$. *Then* $f\ a \sqsubseteq \mathit{apply}\ (\mathit{fun}\ x\ f)\ a$ *in* $\mathsf{D_U}$. □ 198

As can be seen, its statement does not refer to the interpreter definition $\mathcal{S}[\![\_]\!]\_$ *at all.* Instead, the complexity of its proof scales with the number of *abstract operations* supported in the semantic domain of the interpreter for a much more *modular* proof. This modular proof appeals to parametricity [Reynolds 1983] of f's polymorphic type $\forall d.\ (\mathsf{Trace}\ d, \mathsf{Domain}\ d, \mathsf{HasBind}\ d) \Rightarrow d \to d$. Of course, any function defined by the generic interpreter satisfies this requirement.

Without the premise of Beta-App, the law cannot be proved for usage analysis; the following monotone, but non-polymorphic definition for f is a counter-example:

Example 4.20. *Let* $z \neq x \neq y$. *The monotone function* $f :: \mathsf{D_U} \to \mathsf{D_U}$ *defined as*

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

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

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

To prove Lemma 4.19 via parametricity, we encode f's type in System F as $f : \forall X.\, \mathsf{Dict}(X) \to X \to X$ (where $\mathsf{Dict}(d)$ encodes the type class dictionaries of ($\mathsf{Trace}\ d, \mathsf{Domain}\ d, \mathsf{HasBind}\ d$)) and derive the following free theorem:

$$\frac{R \subseteq A \times B \qquad (\mathit{inst}_1, \mathit{inst}_2) \in \mathsf{Dict}(R) \qquad (d_1, d_2) \in R}{(f_A(\mathit{inst}_1)(d_1), f_B(\mathit{inst}_2)(d_2)) \in R}$$

The key to making use of parametricity is to find a useful instantiation of this theorem, of relation R in particular. I successfully proved Beta-App with the following instantiation:

$$\begin{array}{c} A \triangleq B \triangleq \mathsf{D_U}, \qquad \mathit{inst}_1 \triangleq \mathit{inst}_2 \triangleq \mathit{inst}, \qquad d_1 \triangleq a, \qquad d_2 \triangleq \mathit{pre}(x) \\ R_{x,a}(d_1, d_2) \triangleq \forall g.\, d_1 = g(a) \wedge d_2 = g(\mathit{pre}(x)) \implies g(a) \sqsubseteq \mathit{apply}(\mathit{fun}(x, g), a) \end{array}$$

where $\mathit{pre}(x) \triangleq \langle [x \mapsto \mathsf{U_1}], \mathsf{Rep}\ \mathsf{U}_\omega \rangle$ is the argument that the implementation of $\mathit{fun}\ x\ f$ passes to f and inst is the canonical instance dictionary at $\mathsf{D_U}$. This yields the following inference rule:

$$\frac{(\mathit{inst}, \mathit{inst}) \in \mathsf{Dict}(R_{x,a}) \qquad a \sqsubseteq \mathit{apply}(\mathit{fun}(x, \mathit{id}), a)}{f_{\mathsf{D_U}}(\mathit{inst})(a) \sqsubseteq \mathit{apply}(\mathit{fun}(x, f_{\mathsf{D_U}}(\mathit{inst})), a)}$$

where $(\mathit{inst}, \mathit{inst}) \in \mathsf{Dict}(R_{x,a})$ entails showing one lemma per type class method, such as

$$\frac{(\forall d_1, d_2.\, R_{x,a}(d_1, d_2) \implies R_{x,a}(f_1(d_1), f_2(d_2)))}{R_{x,a}(\mathit{fun}(y, f_1), \mathit{fun}(y, f_2))}$$

Discharging each of these 7+1 subgoals concludes the proof of Lemma 4.19. Next, we will use Lemma 4.19 to instantiate Theorem 4.18 for usage analysis.

4.6.3 A Simpler Proof That Usage Analysis Infers Absence

Equipped with the generic abstract interpretation Theorem 4.18, I will prove in this subsection that usage analysis from Section 4.5 infers absence in the same sense as absence analysis from Section 4.1. The reason I do so is to evaluate the proof complexity of my approach against the preservation-style proof framework in Section 4.1.

Specifically, Theorem 4.18 makes it very simple to relate by-need semantics with usage analysis, taking the place of the absence-analysis-specific preservation lemma. I give the full proof inline:

Corollary 4.21 ($\mathcal{S}_{\mathbf{usage}}[\![\_]\!]$ abstracts $\mathcal{S}_{\mathbf{need}}[\![\_]\!]$)**.** *Let e be an expression and $\alpha_{\mathcal{S}}$ the abstraction function from Figure 4.24. Then $\alpha_{\mathcal{S}}(\mathcal{S}_{\mathbf{need}}[\![e]\!]) \sqsubseteq \mathcal{S}_{\mathbf{usage}}[\![e]\!]$.*

Proof. By Theorem 4.18, it suffices to show the abstraction laws in Figure 4.25.

- Mono: Always immediate, since $\sqcup$ and $+$ are the only functions matching on U, and these are monotonic.
- Stuck-App, Stuck-Sel: Trivial, since $\mathit{stuck} = \bot$.
- Step-App, Step-Sel, Step-Inc, Update: Follows by unfolding *step*, *apply*, *select* and associativity of $+$.
- Beta-App: Follows from Lemma 4.19.
- Beta-Sel: Very similar to Lemma 4.19. Note that *con* is quite like an n-ary function application (*apply*) to an unknown function (hence value $\mathsf{Rep}\ \mathsf{U}_\omega$), whereas *select* is like the matching n-ary abstraction (*fun*), except that use of the field binders is not recorded because *con* assumes it to be U_ω anyway.
- ByName-Bind: *kleeneFix* approximates the least fixpoint *lfp* since the iteratee *rhs* is monotone. (As I said in Section 4.5.1, I omit a widening operator for *rhs* that guarantees that *kleeneFix* terminates.)

□

The next step is to leave behind the definition of absence in terms of the LK machine in favour of one using $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$. That is a welcome simplification because it leaves us with a single semantic artefact — the denotational interpreter — instead of an operational semantics and a separate static analysis as in Section 4.1. Thanks to adequacy (Theorem 4.4), this new notion is not a redefinition but provably equivalent to Definition 4.2:

Lemma 4.22 (Denotational absence)**.** *Variable x is used in e if and only if there exists a by-need evaluation context E and expression e' such that the trace $\mathcal{S}_{\mathbf{need}}[\![E[\mathsf{Let}\ x\ e'\ e]]\!]_\varepsilon(\varepsilon)$ contains a $\mathsf{Look}\ x$ event. Otherwise, x is absent in e.* □ 205

I define the by-need evaluation contexts for our language in the Appendix, in Figure A.2 on page page 204. Thus insulated from the LK machine, we may restate and prove Theorem 4.1 for usage analysis.

Theorem 4.23 ($\mathcal{S}_{\mathbf{usage}}[\![\_]\!]_\_$ infers absence). *Let* $\rho_e \triangleq \overline{[y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle]}$ *be the initial environment with an entry for every free variable* y *of an expression* e. *If* $\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_e} = \langle \varphi, v \rangle$ *and* $\varphi\ !?\ x = \mathsf{U}_0$, *then* x *is absent in* e. □ 210

Proof sketch. If x is used in e, there is a trace $\mathcal{S}_{\mathbf{need}}[\![E[\mathsf{Let}\ x\ e'\ e]]\!]_\varepsilon(\varepsilon)$ containing a $\mathsf{Look}\ x$ event. The abstraction function $\alpha_{\mathcal{S}}$ induced by D_U aggregates lookups in the trace into a $\varphi' :: \mathsf{Uses}$, e.g. $\beta_{\mathbb{T}}(\textsc{Look}(i) \hookrightarrow \textsc{Look}(x) \hookrightarrow \textsc{Look}(i) \hookrightarrow \langle ... \rangle) = \langle [i \mapsto \mathsf{U}_\omega, x \mapsto \mathsf{U}_1], ... \rangle$. Clearly, it is $\varphi'\ !?\ x \sqsupseteq \mathsf{U}_1$, because there is at least one $\mathsf{Look}\ x$. Corollary 4.21 and a context invariance Lemma A.15 prove that the computed φ approximates φ', so $\varphi\ !?\ x \sqsupseteq \varphi'\ !?\ x \sqsupseteq \mathsf{U}_1 \neq \mathsf{U}_0$. □

4.6.4 Comparison to Ad-hoc Preservation Proof

Although I have not disclosed the proof for Theorem 4.18 yet, we have seen enough to draw a comparison to the preservation-style proof framework in Section 4.1.

- Where there were multiple separate *semantic artefacts* in Section 4.1, such as a small-step semantics and an extension of the absence analysis to machine configurations σ in order to state preservation (Lemma A.9), my 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 Corollary 4.21 in half a page replaces a tedious, error-prone and incomplete *proof for the preservation lemma* of Section 4.1 (Lemma A.9). Of course, in this section I lean on Theorem 4.18 to prove what amounts to a preservation lemma; the difference is that my proof will properly account for heap update and can be shared with other analyses that are sound wrt. by-name and by-need, such as type analysis. Thus, I achieve the goal of disentangling semantic details from the proof.
- Furthermore, the proof for Corollary 4.21 by parametricity in this section is *modular*, in contrast to Lemma 4.3 which is proven by cases over the interpreter definition. More work needs to be done to achieve a modular proof of the underlying Theorem 4.18, however. We will see that the pendant for abstract by-**name** interpretation (Theorem 4.28) already has a modular proof.

4.6.5 Interlude

So far, we have seen how to *use* the abstract interpretation Theorem 4.18, but not proved it. Proving this theorem correct is the overarching goal of the remaining subsections of this section.

In order to properly define the representation function $\beta_{\mathbb{T}}$ and thus $\alpha_{\mathcal{S}}$ from Figure 4.24 on infinite traces, I will need to define the concept of a *safety extension* in Section 4.6.7. Before I prove Theorem 4.18 in Section 4.6.9, it is very illuminating to first prove the corresponding by-**name** abstract interpretation Theorem 4.28 in Section 4.6.8. This proof is modular because it appeals to parametricity. After the non-modular proof for Theorem 4.18, I will discuss in Section 4.6.10 why it cannot be proved modularly by parametricity, conjecturing that defining and proving a Kripke logical relation on type structure could provide a modular proof.

4.6.6 Abstracting Guarded Fixpoints

In this subsection, I show that least fixpoints abstract guarded fixpoints, an important property in later proofs.

Suppose that we were only interested in the trace component of our semantic domain, thus effectively restricting ourselves to $\mathbb{T} \triangleq \mathsf{T}\ ()$, and that we were to approximate properties $P \in \wp(\mathbb{T})$ about such traces by a Galois connection $\alpha : (\wp(\mathbb{T}), \subseteq) \rightleftarrows (\widehat{\mathsf{D}}, \sqsubseteq) : \gamma$. Alas, although the abstraction function α is well-defined as a mathematical function, it most certainly is *not* computable at infinite inputs (in $\mathbb{T}^\infty$), for example at the guarded fixpoint (recall that $\mathit{fix}\ f = f\ (\mathit{fix}\ f)$)

$$\mathit{fix}\ (\mathsf{Step}\ (\mathsf{Look}\ x)) = \mathsf{Step}\ (\mathsf{Look}\ x)\ (\mathsf{Step}\ (\mathsf{Look}\ x)\ \ldots).$$

The whole point about *static* analyses is that they approximate program behavior in finite data. As we have discussed in Section 4.5.1, this rules out use of *guarded* fixpoints fix for usage analysis, so it computes the *least* fixpoint lfp instead. Concretely, static analyses often approximate the abstraction of the guarded fixpoint by the least fixpoint of the abstracted iteratee, assuming the following approximation relationship, where $f^* :: \wp(\mathbb{T}) \to \wp(\mathbb{T})$ denotes the lifting of f to powersets:

$$\alpha\ (\{\mathit{fix}\ (\mathsf{Step}\ (\mathsf{Look}\ x))\}) \sqsubseteq \mathit{lfp}\ (\alpha \circ (\mathsf{Step}\ (\mathsf{Look}\ x))^* \circ \gamma).$$

I will now formally prove this property:

Lemma 4.24 (Guarded fixpoint abstraction)**.** *Let* $\widehat{\mathsf{D}}$ *be a domain with an instance for* Lat*, and let* $\alpha : (\wp(\mathbb{T}), \subseteq) \rightleftarrows (\widehat{\mathsf{D}}, \sqsubseteq) : \gamma$ *a Galois connection. Then, for any iteratee* $f :: \mathbb{T} \to \mathbb{T}$*,*

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

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

Proof. I should note that I will be sloppy in the treatment of the *later* modality $\triangleright$ here. Since I have proven totality of all expressions worth considering in Section 4.4.2, the utility of being explicit is rather low (much more so since a pen and paper proof is not type checked) and I will admit myself this kind of sloppiness from now on.

Let us proceed by Löb induction.

$$\begin{aligned}
&\alpha(\{\mathit{fix}\ f\}) \sqsubseteq \mathit{lfp}\ (\alpha \circ f^* \circ \gamma) \\
=\ &\langle\!|\ \mathit{fix}\ f = f\ (\mathit{fix}\ f)\ |\!\rangle \\
&\alpha(\{f\ (\mathit{fix}\ f)\}) \\
=\ &\langle\!|\ \text{Commute } f \text{ and } \{_\}\ |\!\rangle \\
&\alpha(f^*\ (\{\mathit{fix}\ f\})) \\
\sqsubseteq\ &\langle\!|\ \mathit{id} \sqsubseteq \gamma \circ \alpha\ |\!\rangle \\
&\alpha(f^*\ (\gamma(\alpha(\{\mathit{fix}\ f\})))) \\
\sqsubseteq\ &\langle\!|\ \text{Induction hypothesis}\ |\!\rangle \\
&\alpha(f^*\ (\gamma(\mathit{lfp}\ (\alpha \circ f^* \circ \gamma)))) \\
=\ &\langle\!|\ \mathit{lfp}\ \widehat{f} = \widehat{f}\ (\mathit{lfp}\ \widehat{f})\ |\!\rangle \\
&\mathit{lfp}\ (\alpha \circ f^* \circ \gamma)
\end{aligned}$$

□

4.6.7 Safety Properties and Safety Extension of a Galois Connection

Figure 4.24 describes a semantic trace property as a "fold", in terms of a Trace instance. Of course such a fold (an inductive elimination procedure) has no meaning when the trace is infinite! Yet it is always clear what I mean: When the trace is infinite and described by a guarded fixpoint, I consider the meaning of the fold as the limit (i.e. least fixpoint) of folding over its finite prefixes. In this subsection, I will prove that this intuition is sound when abstracting *safety properties* [Lamport 1977].

Suppose again that $\mathbb{T} \triangleq \mathsf{T}\ ()$, that $\mathbb{T}^\infty \subseteq \mathbb{T}$ denotes the subset of infinite traces and that $\mathbb{T}^* \subseteq \mathbb{T}$ denotes the subset of finite traces. Then a safety trace property $P \subseteq \mathbb{T}$ is defined as follows:

Definition 4.25 (Safety property)**.** *A trace property $P \subseteq \mathbb{T}$ is a* safety property *iff, whenever $\tau_1 \in \mathbb{T}^\infty$ violates P (so $\tau_1 \notin P$), then there exists some proper prefix $\tau_2 \in \mathbb{T}^*$ (written $\tau_2 \lessdot \tau_1$) that already violates P ($\tau_2 \notin P$).*

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

The contraposition of the above definition is

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

and we can exploit safety to extend a finitary Galois connection, such as α_S in Figure 4.24 defined by a fold over the trace, to infinite inputs:

Lemma 4.26 (Safety extension)**.** *Let $\widehat{\mathsf{D}}$ be a domain with an instance for* Lat*, $\alpha : (\wp(\mathbb{T}^*), \subseteq) \rightleftarrows (\widehat{\mathsf{D}}, \sqsubseteq) : \gamma$ a Galois connection and $P \in \wp(\mathbb{T})$ a safety property. Then any domain element $\widehat{d}$ that soundly approximates P via γ on finite traces soundly approximates P on infinite traces as well:*

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

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

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

Proof. First note that α^∞ uniquely determines the Galois connection via the representation function

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

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

$$\begin{array}{ll} & \tau \in P \\ \implies & \langle\!\langle\ P \text{ safety property}\ \rangle\!\rangle \\ & (\forall \tau_2.\ \tau_2 \lessdot \tau \implies \tau_2 \in P \cap \mathbb{T}^*) \end{array}$$

$$
\begin{aligned}
&\implies && \wr \text{ Assumption } P \cap \mathbb{T}^* \subseteq \gamma(\widehat{d}) \int \\
&&& (\forall \tau_2.\ \tau_2 \lessdot \tau \implies \tau_2 \in \gamma(\widehat{d})) \\
&\iff && \wr \text{ Definition of Union } \int \\
&&& \bigcup \{\tau_2 \mid \tau_2 \lessdot \tau\} \subseteq \gamma(\widehat{d}) \\
&\iff && \wr \text{ Galois } \int \\
&&& \alpha(\bigcup \{\tau_2 \mid \tau_2 \lessdot \tau\}) \sqsubseteq \widehat{d} \\
&\iff && \wr \text{ Definition of } \beta^\infty \int \\
&&& \beta^\infty(\tau) \sqsubseteq \widehat{d} \\
&\iff && \wr \text{ Galois } \int \\
&&& \tau \in \gamma^\infty(\widehat{d})
\end{aligned}
$$

□

From now on, we tacitly assume that all trace properties of interest are safety properties, and that any Galois connection defined in Haskell via fold, such as in Figure 4.24, has been extended to infinite traces via Lemma 4.26.

4.6.8 Abstract By-name Interpretation, in Detail

I will now prove that the by-name abstraction laws in Figure 4.25 induce an abstract interpretation of by-name semantics via $\alpha_{\mathcal{S}}$ defined in Figure 4.26.

Compared to the by-need trace abstraction in Figure 4.24, the by-name trace abstraction function in Figure 4.26 is similar yet somewhat simpler because no heap is involved. We will see that the same is true for the soundness proofs: the proof for by-need is structurally similar to by-name, however accounting for heap update leads to considerable complication.

Note that I omit ByName newtype wrappers, as in many other preceding sections, as well as the Name passed to *fun* as a poor man's De Bruijn level. The meaning of the Galois connection in Figure 4.26 on infinite traces is determined by Lemma 4.26.

I will now prove sound by-name interpretation by appealing to parametricity [Reynolds 1983]. Specifically, I will apply the abstraction theorem to the System F encoding of the type of $\mathcal{S}[\![\_]\!]\_$,

$$\mathcal{S}[\![\_]\!]\_ : \forall X.\ \mathsf{Dict}(X) \to \mathsf{Exp} \to (\mathsf{Name} \rightharpoonup X) \to X,$$

$$\begin{array}{c} \alpha_{\mathcal{S}} : ((\mathsf{Name} :\rightharpoonup \mathsf{D_{na}}) \to \mathsf{D_{na}}) \rightleftarrows ((\mathsf{Name} :\rightharpoonup \widehat{\mathsf{D}}) \to \widehat{\mathsf{D}}) : \gamma_{\mathcal{S}} \\ \alpha_{\mathbb{E}} : \wp(\mathsf{Name} :\rightharpoonup \mathsf{D_{na}}) \rightleftarrows (\mathsf{Name} :\rightharpoonup \widehat{\mathsf{D}}) : \gamma_{\mathbb{E}} \\ \alpha_{\mathbb{D}} : \wp(\mathsf{D_{na}}) \rightleftarrows \widehat{\mathsf{D}} : \gamma_{\mathbb{D}} \qquad \beta_{\mathbb{T}} : \mathsf{T}\ (\mathsf{Value}\ (\mathsf{ByName}\ \mathsf{T})) \to \widehat{\mathsf{D}} \end{array}$$

$$\begin{array}{lcl} \alpha_{\mathcal{S}}(S)(\widehat{\rho}) & = & \alpha_{\mathbb{T}}(\{\ S(\rho) \mid \rho \in \gamma_{\mathbb{E}}(\widehat{\rho})\ \}) \\ \alpha_{\mathbb{E}}(E)(x) & = & \alpha_{\mathbb{T}}(\{\ \rho(x) \mid \rho \in E\ \}) \\ \alpha_{\mathbb{D}}(D) & = & \bigsqcup\{\ \beta_{\mathbb{T}}(d) \mid d \in D\ \} \end{array}$$

$$\beta_{\mathbb{T}}(\tau) = \begin{cases} \mathit{step}\ e\ (\beta_{\mathbb{T}}(\tau)) & \text{if } \tau = \mathsf{Step}\ e\ \tau' \\ \mathit{stuck} & \text{if } \tau = \mathsf{Ret}\ \mathsf{Stuck} \\ \mathit{fun}\ (\alpha_{\mathbb{D}} \circ f^* \circ \gamma_{\mathbb{D}}) & \text{if } \tau = \mathsf{Ret}\ (\mathsf{Fun}\ f) \\ \mathit{con}\ k\ (\mathit{map}\ (\alpha_{\mathbb{D}} \circ \{\_\})\ \mathit{ds}) & \text{if } \tau = \mathsf{Ret}\ (\mathsf{Con}\ k\ \mathit{ds}) \end{cases}$$

Fig. 4.26: Galois connection $\alpha_{\mathcal{S}}$ for by-name abstraction derived from Trace, Domain and Lat instances on $\widehat{\mathsf{D}}$

where $\mathsf{Dict}(d)$ encodes $(\mathsf{Trace}\ d, \mathsf{Domain}\ d, \mathsf{HasBind}\ d)$. The abstraction theorem yields the following free theorem about relations R between base values

$$\frac{R \subseteq A \times B \qquad (\mathit{inst}_1, \mathit{inst}_2) \in \mathsf{Dict}(R) \qquad (\rho_1, \rho_2) \in \mathsf{Name} \rightharpoonup R}{(\mathcal{S}_A[\![e]\!](\mathit{inst}_1)(\rho_1), \mathcal{S}_B[\![e]\!](\mathit{inst}_2)(\rho_2)) \in R} \tag{4.6}$$

In the following proof, I will instantiate R at $R(d, \widehat{d}) \triangleq \alpha_{\mathbb{D}}(\{d\}) \sqsubseteq \widehat{d}$ to show the abstraction relationship.

I will need the following auxiliary lemma for the *apply* and *select* cases:

Lemma 4.27 (By-name bind). *It is* $\beta_{\mathbb{T}}(d \gg\!= f) \sqsubseteq \widehat{f}\ \widehat{d}$ *if*

1. $\beta_{\mathbb{T}}(d) \sqsubseteq \widehat{d}$, *and*
2. *for all events* ev *and elements* $\widehat{d'}$, $\widehat{\mathit{step}}\ \mathit{ev}\ (\widehat{f}\ \widehat{d'}) \sqsubseteq \widehat{f}\ (\widehat{\mathit{step}}\ \mathit{ev}\ \widehat{d'})$, *and*
3. *for all values* v, $\beta_{\mathbb{T}}(f\ v) \sqsubseteq \widehat{f}\ (\beta_{\mathbb{T}}(\mathsf{Ret}\ v))$.

Proof. By Löb induction.

If $d = \mathsf{Step}\ \mathit{ev}\ d'$, define $\widehat{d'} \triangleq \beta_{\mathbb{T}}(d')$. We get

$$\begin{array}{l} \beta_{\mathbb{T}}(d \gg\!= f) = \beta_{\mathbb{T}}(\mathsf{Step}\ \mathit{ev}\ d' \gg\!= f) = \widehat{\mathit{step}}\ \mathit{ev}\ (\beta_{\mathbb{T}}(d' \gg\!= f)) \\ \sqsubseteq \quad \wr\ \text{Induction hypothesis at } \beta_{\mathbb{T}}(d') = \widehat{d'}, \text{Monotonicity of } \widehat{\mathit{step}}\ \int \end{array}$$

$$
\begin{aligned}
&\widehat{step}\ ev\ (\widehat{f}\ (\beta_{\mathbb{T}}(d'))) \\
\sqsubseteq\ &\{\!|\ \text{Assumption (2)}\ |\!\} \\
&\widehat{f}\ (\widehat{step}\ ev\ (\beta_{\mathbb{T}}(d'))) = \widehat{f}\ (\beta_{\mathbb{T}}(d)) \\
\sqsubseteq\ &\{\!|\ \text{Assumption (1)}\ |\!\} \\
&\widehat{f}\ \widehat{d}
\end{aligned}
$$

Otherwise, $d = \mathsf{Ret}\ v$ for some $v :: \mathsf{Value}$.

$$
\begin{aligned}
&\beta_{\mathbb{T}}(\mathsf{Ret}\ v \gg\!= f) = \beta_{\mathbb{T}}(f\ v) \\
\sqsubseteq\ &\{\!|\ \text{Assumption (3)}\ |\!\} \\
&\widehat{f}\ (\beta_{\mathbb{T}}(\mathsf{Ret}\ v)) = \widehat{f}\ (\beta_{\mathbb{T}}(d)) \\
\sqsubseteq\ &\{\!|\ \text{Assumption (1)}\ |\!\} \\
&\widehat{f}\ \widehat{d}
\end{aligned}
$$

□

What follows is the sound abstraction proof that instantiates the free theorem. Note that its statement fixes the interpreter definition to $\mathcal{S}[\![\_]\!]\_$, however the proof would still work if generalised to *any* definition with the same type as $\mathcal{S}[\![\_]\!]\_$! Hence the proof is automatically modular in the sense of Section 4.6.2.

Theorem 4.28 (Abstract By-name Interpretation)**.** *Let e be an expression, $\widehat{\mathsf{D}}$ a domain with instances for* Trace, Domain, HasBind *and* Lat*, and let $\alpha_{\mathcal{S}}$ be the abstraction function from Figure 4.26. If the by-name abstraction laws in Figure 4.25 hold, then $\mathcal{S}_{\widehat{\mathsf{D}}}[\![\_]\!]$ is an abstract interpreter that is sound wrt. $\alpha_{\mathcal{S}}$,*

$$\alpha_{\mathcal{S}}(\mathcal{S}_{\mathbf{name}}[\![e]\!]) \sqsubseteq \mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!].$$

Proof. Let $inst : \mathsf{Dict}(\mathsf{D}_{\mathbf{na}})$, $\widehat{inst} : \mathsf{Dict}(\widehat{\mathsf{D}})$ the canonical dictionaries from the type class instance definitions. Instantiate the free theorem (4.6) above as follows:

$$A \triangleq \mathsf{D}_{\mathbf{na}}, B \triangleq \widehat{\mathsf{D}}, R(d, \widehat{d}) \triangleq \alpha_{\mathbb{D}}(\{d\}) \sqsubseteq \widehat{d}, inst_1 \triangleq inst, inst_2 \triangleq \widehat{inst}, e \triangleq e$$

Note that $(\rho, \widehat{\rho}) \in (\mathsf{Name} \rightharpoonup R) \iff \alpha_{\mathbb{E}}(\{\rho\}) \sqsubseteq \widehat{\rho} \iff \rho \in \gamma_{\mathbb{E}}(\widehat{\rho})$ by simple calculation.

The above instantiation yields, in Haskell,

$$\frac{(inst, \widehat{inst}) \in \mathsf{Dict}(R) \qquad \rho \in \gamma_{\mathbb{E}}(\widehat{\rho})}{\alpha_{\mathbb{D}}(\{\mathcal{S}_{\mathbf{name}}[\![e]\!]_{\rho}\}) \sqsubseteq \mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_{\widehat{\rho}}}$$

and since ρ and $\widehat{\rho}$ can be chosen arbitrarily and $\alpha_{\mathbb{D}}$ is defined elementwise by $\beta_{\mathbb{T}}$, we get

$$\frac{(\mathit{inst}, \widehat{\mathit{inst}}) \in \mathsf{Dict}(R)}{\alpha_{\mathbb{D}}(\{\mathcal{S}_{\mathbf{name}}[\![e]\!]_\rho \mid \rho \in \gamma_{\mathbb{E}}(\widehat{\rho})\}) \sqsubseteq \mathcal{S}_{\widehat{\mathbb{D}}}[\![e]\!]_{\widehat{\rho}}}$$

We refold the definition of $\alpha_{\mathcal{S}}$ to get

$$\frac{(\mathit{inst}, \widehat{\mathit{inst}}) \in \mathsf{Dict}(R)}{\alpha_{\mathcal{S}}(\mathcal{S}_{\mathbf{name}}[\![e]\!]) \sqsubseteq \mathcal{S}_{\widehat{\mathbb{D}}}[\![e]\!]}$$

Hence, in order to show the goal, it suffices to prove $(\mathit{inst}, \widehat{\mathit{inst}}) \in \mathsf{Dict}(R)$. By the relational interpretation of products, we get one subgoal per instance method. Note that $R(d, \widehat{d}) \iff \beta_{\mathbb{T}}(d) \sqsubseteq \widehat{d}$ by unfolding the definition of $\alpha_{\mathbb{D}}$, and it is more direct to argue in terms of the latter.

- **Case** *step*. Goal: $\dfrac{(d, \widehat{d}) \in R}{(\mathit{step}\ \mathit{ev}\ d, \widehat{\mathit{step}}\ \mathit{ev}\ \widehat{d}) \in R}$.
  Then $\beta_{\mathbb{T}}(\mathsf{Step}\ \mathit{ev}\ d) = \widehat{\mathit{step}}\ \mathit{ev}\ (\beta_{\mathbb{T}}(d)) \sqsubseteq \widehat{\mathit{step}}\ \mathit{ev}\ \widehat{d}$ by assumption and monotonicity.

- **Case** *stuck*. Goal: $(\mathit{stuck}, \widehat{\mathit{stuck}}) \in R$.
  Then $\beta_{\mathbb{T}}(\mathit{stuck}) = \beta_{\mathbb{T}}(\mathsf{Ret}\ \mathsf{Stuck}) = \widehat{\mathit{stuck}}$.

- **Case** *fun*. Goal: $\dfrac{\forall (d, \widehat{d}) \in R.\ (f\ d, \widehat{f}\ \widehat{d}) \in R}{(\mathit{fun}\ f, \widehat{\mathit{fun}}\ \widehat{f}) \in R}$.
  Then $\beta_{\mathbb{T}}(\mathit{fun}\ f) = \beta_{\mathbb{T}}(\mathsf{Ret}\ (\mathsf{Fun}\ f)) = \widehat{\mathit{fun}}\ (\alpha_{\mathbb{D}} \circ f^* \circ \gamma_{\mathbb{D}})$ and it suffices to show that $\alpha_{\mathbb{D}} \circ f^* \circ \gamma_{\mathbb{D}} \sqsubseteq \widehat{f}$ by monotonicity of $\widehat{\mathit{fun}}$.

  $$\begin{aligned}
  &(\alpha_{\mathbb{D}} \circ f^* \circ \gamma_{\mathbb{D}})\ \widehat{d} \\
  &= \ \wr\ \text{Unfold } _^*, \alpha_{\mathbb{D}}, \text{simplify}\ \wr \\
  &\quad \bigsqcup\{\beta_{\mathbb{T}}(f\ d) \mid d \in \gamma_{\mathbb{D}}(\widehat{d})\} \\
  &\sqsubseteq \ \wr\ \text{Apply premise to } \beta_{\mathbb{T}}(d) \sqsubseteq \widehat{d}\ \wr \\
  &\quad \widehat{f}\ \widehat{d}
  \end{aligned}$$

- **Case** *apply*. Goal: $\dfrac{(d, \widehat{d}) \in R \qquad (a, \widehat{a}) \in R}{(\mathit{apply}\ d\ a, \widehat{\mathit{apply}}\ \widehat{d}\ \widehat{a}) \in R}$.
  $\mathit{apply}\ d\ a$ is defined as $d \gg\!= \lambda v \to \mathbf{case}\ v\ \mathbf{of}\ \mathsf{Fun}\ g \to g\ a; \_ \to \mathit{stuck}$. In

order to show the goal, we need to apply Lemma 4.27 at $\widehat{f}\ \widehat{d} \triangleq \widehat{apply}\ \widehat{d}\ \widehat{a}$. We get three subgoals for the premises of Lemma 4.27:

- $\beta_{\mathbb{T}}(d) \sqsubseteq \widehat{d}$: By assumption $(d, \widehat{d}) \in R$.
- $\forall ev\ \widehat{d'}.\ \widehat{step}\ ev\ (\widehat{apply}\ \widehat{d'}\ \widehat{a}) \sqsubseteq \widehat{apply}\ (\widehat{step}\ ev\ \widehat{d'})\ \widehat{a}$: By assumption Step-App.
- $\forall v.\ \beta_{\mathbb{T}}(\mathbf{case}\ v\ \mathbf{of}\ \mathsf{Fun}\ g \to g\ a; \_ \to stuck) \sqsubseteq \widehat{apply}\ (\beta_{\mathbb{T}}(\mathsf{Ret}\ v))\ \widehat{a}$: By cases over v.
  - **Case** $v = \mathsf{Stuck}$: Then $\beta_{\mathbb{T}}(stuck) = \widehat{stuck} \sqsubseteq \widehat{apply}\ \widehat{stuck}\ \widehat{a}$ by assumption Stuck-App.
  - **Case** $v = \mathsf{Con}\ k\ ds$: Then $\beta_{\mathbb{T}}(stuck) = \widehat{stuck} \sqsubseteq \widehat{apply}\ (\widehat{con}\ k\ \widehat{ds})\ \widehat{a}$ by assumption Stuck-App, for the suitable $\widehat{ds}$.
  - **Case** $v = \mathsf{Fun}\ g$: Then

$$
\begin{array}{ll}
& \beta_{\mathbb{T}}(g\ a) \\
\sqsubseteq & \{\!|\ id \sqsubseteq \gamma_{\mathbb{D}} \circ \alpha_{\mathbb{D}},\ \text{rearrange}\ |\!\} \\
& (\alpha_{\mathbb{D}} \circ g^* \circ \gamma_{\mathbb{D}})\ (\alpha_{\mathbb{D}}\ a) \\
\sqsubseteq & \{\!|\ \text{Assumption}\ \beta_{\mathbb{T}}(a) \sqsubseteq \widehat{a}\ |\!\} \\
& (\alpha_{\mathbb{D}} \circ g^* \circ \gamma_{\mathbb{D}})\ \widehat{a} \\
\sqsubseteq & \{\!|\ \text{Assumption Beta-App}\ |\!\} \\
& \widehat{apply}\ (\widehat{fun}\ (\alpha_{\mathbb{D}} \circ g^* \circ \gamma_{\mathbb{D}}))\ \widehat{a} \\
= & \{\!|\ \text{Definition of}\ \beta_{\mathbb{T}}, v\ |\!\} \\
& \widehat{apply}\ (\beta_{\mathbb{T}}(\mathsf{Ret}\ v))\ \widehat{a}
\end{array}
$$

- **Case** *con*. Goal: $\dfrac{(ds, \widehat{ds}) \in [R]}{(con\ k\ ds, \widehat{con}\ k\ \widehat{ds}) \in R}$.
  Then $\beta_{\mathbb{T}}(con\ k\ ds) = \beta_{\mathbb{T}}(\mathsf{Ret}\ (\mathsf{Con}\ k\ ds)) = \widehat{con}\ k\ (map\ (\alpha_{\mathbb{D}} \circ \{\_\})\ ds)$. The assumption $(ds, \widehat{ds}) \in [R]$ implies $map\ (\alpha_{\mathbb{D}} \circ \{\_\})\ ds \sqsubseteq \widehat{ds}$ and the goal follows by monotonicity of $\widehat{con}$.
- **Case** *select*. Goal: $\dfrac{(d, \widehat{d}) \in R \qquad (alts, \widehat{alts}) \in \mathsf{Tag} :\rightharpoonup ([R] \to R)}{(select\ d\ alts, \widehat{select}\ \widehat{d}\ \widehat{alts}) \in R}$.
  select d fs is defined as $d \gg\!= \lambda v \to \mathbf{case}\ v\ \mathbf{of}\ \mathsf{Con}\ k\ ds \mid k \in dom\ alts \to (alts\,!\,k)\ ds; \_ \to stuck$. In order to show the goal, we need to apply

Lemma 4.27 at $\widehat{f}\ \widehat{d} \triangleq \widehat{\mathit{select}}\ \widehat{d}\ \widehat{\mathit{alts}}$. We get three subgoals for the premises of Lemma 4.27:

- $\beta_{\mathbb{T}}(d) \sqsubseteq \widehat{d}$: By assumption $(d, \widehat{d}) \in R$.
- $\forall \mathit{ev}\ \widehat{d'}.\ \widehat{\mathit{step}}\ \mathit{ev}\ (\widehat{\mathit{select}}\ \widehat{d'}\ \widehat{\mathit{alts}}) \sqsubseteq \widehat{\mathit{select}}\ (\widehat{\mathit{step}}\ \mathit{ev}\ \widehat{d'})\ \widehat{\mathit{alts}}$: By assumption Step-Sel.
- $\forall v.\ \beta_{\mathbb{T}}(\mathbf{case}\ v\ \mathbf{of}\ \mathsf{Con}\ k\ ds \mid k \in \mathit{dom}\ \mathit{alts} \to (\mathit{alts}\,!\,k)\ ds; \_ \to \mathit{stuck}) \sqsubseteq \widehat{\mathit{select}}\ (\beta_{\mathbb{T}}(\mathsf{Ret}\ v))\ \widehat{\mathit{alts}}$:
  By cases over v. The first three all correspond to when the continuation of *select* gets stuck.
  - **Case** $v = \mathsf{Stuck}$: Then $\beta_{\mathbb{T}}(\mathit{stuck}) = \widehat{\mathit{stuck}} \sqsubseteq \widehat{\mathit{select}}\ \widehat{\mathit{stuck}}\ \widehat{\mathit{alts}}$ by assumption Stuck-Sel.
  - **Case** $v = \mathsf{Fun}\ f$: Then $\beta_{\mathbb{T}}(\mathit{stuck}) = \widehat{\mathit{stuck}} \sqsubseteq \widehat{\mathit{select}}\ (\widehat{\mathit{fun}}\ \widehat{f})\ \widehat{\mathit{alts}}$ by assumption Stuck-Sel, for the suitable $\widehat{f}$.
  - **Case** $v = \mathsf{Con}\ k\ ds$, $k \notin \mathit{dom}\ \mathit{alts}$: Then $\beta_{\mathbb{T}}(\mathit{stuck}) = \widehat{\mathit{stuck}} \sqsubseteq \widehat{\mathit{select}}\ (\widehat{\mathit{con}}\ k\ \widehat{ds})\ \widehat{\mathit{alts}}$ by assumption Stuck-Sel, for the suitable $\widehat{ds}$.
  - **Case** $v = \mathsf{Con}\ k\ ds$, $k \in \mathit{dom}\ \mathit{alts}$: Then

$$
\begin{aligned}
&\beta_{\mathbb{T}}((\mathit{alts}\,!\,k)\ ds) \\
\sqsubseteq\ &\wr\ \mathit{id} \sqsubseteq \gamma_{\mathbb{D}} \circ \alpha_{\mathbb{D}},\ \text{rearrange}\ \int \\
&(\alpha_{\mathbb{D}} \circ (\mathit{alts}\,!\,k)^* \circ \mathit{map}\ \gamma_{\mathbb{D}})\ (\mathit{map}\ (\alpha_{\mathbb{D}} \circ \{\_\})\ ds) \\
\sqsubseteq\ &\wr\ \text{Assumption}\ (\mathit{alts}, \widehat{\mathit{alts}}) \in \mathsf{Tag} :\rightarrowtail ([R] \to R)\ \int \\
&(\widehat{\mathit{alts}}\,!\,k)\ (\mathit{map}\ (\alpha_{\mathbb{D}} \circ \{\_\})\ ds) \\
\sqsubseteq\ &\wr\ \text{Assumption Beta-Sel}\ \int \\
&\widehat{\mathit{select}}\ (\widehat{\mathit{con}}\ k\ (\mathit{map}\ (\alpha_{\mathbb{D}} \circ \{\_\})\ ds))\ \widehat{\mathit{alts}} \\
=\ &\wr\ \text{Definition of}\ \beta_{\mathbb{T}},\ v\ \int \\
&\widehat{\mathit{select}}\ (\beta_{\mathbb{T}}(\mathsf{Ret}\ v))\ \widehat{\mathit{alts}}
\end{aligned}
$$

- **Case** *bind*. Goal: $\dfrac{(\forall (d, \widehat{d}) \in R.\ (\mathit{rhs}\ d, \widehat{\mathit{rhs}}\ \widehat{d}) \in R) \quad (\forall (d, \widehat{d}) \in R.\ (\mathit{body}\ d, \widehat{\mathit{body}}\ \widehat{d}) \in R)}{(\mathit{bind}\ \mathit{rhs}\ \mathit{body}, \widehat{\mathit{bind}}\ \widehat{\mathit{rhs}}\ \widehat{\mathit{body}}) \in R}$.

  It is $\mathit{bind}\ \mathit{rhs}\ \mathit{body} = \mathit{body}\ (\mathit{fix}\ \mathit{rhs})$ and $\widehat{\mathit{body}}\ (\mathit{lfp}\ \widehat{\mathit{rhs}}) \sqsubseteq \widehat{\mathit{bind}}\ \widehat{\mathit{rhs}}\ \widehat{\mathit{body}}$ by

Assumption ByName-Bind. Let us first establish that $(\mathit{fix}\ \mathit{rhs}, \widehat{\mathit{lfp}\ \mathit{rhs}}) \in R$, leaning on our theory about guarded fixpoint abstraction in Section 4.6.6:

$$
\begin{aligned}
& \alpha_{\mathbb{D}}(\{\mathit{fix}\ \mathit{rhs}\}) \\
\sqsubseteq\ & \text{⁅ By Lemma 4.24 ⁆} \\
& \mathit{lfp}\ (\alpha_{\mathbb{D}} \circ \mathit{rhs}^* \circ \gamma_{\mathbb{D}}) \\
=\ & \text{⁅ Unfolding } \_^*, \alpha_{\mathbb{D}} \text{ ⁆} \\
& \mathit{lfp}\ (\lambda \widehat{d} \to \bigsqcup \{\beta_{\mathbb{T}}(\mathit{rhs}\ d) \mid d \in \gamma_{\mathbb{D}}(\widehat{d})\}) \\
\sqsubseteq\ & \text{⁅ Apply premise about } \mathit{rhs} \text{ to } \beta_{\mathbb{T}}(d) \sqsubseteq \widehat{d} \text{ ⁆} \\
& \widehat{\mathit{lfp}\ \mathit{rhs}}
\end{aligned}
$$

Applying this fact to the second assumption proves

$$(\mathit{body}\ (\mathit{fix}\ \mathit{rhs}), \widehat{\mathit{body}\ (\mathit{lfp}\ \mathit{rhs})}) \in R$$

and thus the goal.

□

4.6.9 Abstract By-need Soundness, in Detail

The goal of this section is to prove Theorem 4.18 correct, 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 Section 4.5.2 (I have however not tried to prove the latter).

The setup in Section 4.6.8 with its Galois connection in Figure 4.26 is somewhat similar to the Galois connection in Figure 4.24, however for by-need abstraction the Galois connection for domain elements $d :: \mathsf{D}_{\mathbf{ne}}$ is indexed by a heap wrt. to which the element is abstracted.

We will later see how this indexing yields a Kripke logical relation as the soundness condition, and that, sadly, such a relation cannot easily be proven by appealing to Reynolds' parametricity. As a consequence, the following development hardcodes the definition of $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$ and often proceeds by cases on syntax of the object language instead of a more elegant and modular proof strategy concerning just the semantic domain.

The reason we need to index correctness relations by a heap is as follows: Although in Section 4.3.3 I considered an element d as an atomic denotation, such a denotation actually only carries meaning when it travels together with a heap μ that ties the addresses that d references to actual meaning.

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

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

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

Fig. 4.27: Heap progression relation

There are *many* elements (functions!) $d :: \mathrm{D}_{\mathbf{ne}}$, many with very surprising behavior, but we are only interested in elements *definable* by the interpreter:

Definition 4.29 (Definable by-need entities)**.** *We write $\vdash_{\mathbb{D}} d$, $\vdash_{\mathbb{E}} \rho$ or $\vdash_{\mathbb{H}} \mu$ to say that the by-need element d, environment ρ or heap μ is* definable*, defined as*

- $\vdash_{\mathbb{E}} \rho \triangleq \forall x \in dom\ \rho.\ \exists y\ a.\ \rho ! x = step\ (\mathsf{Look}\ y)\ (fetch\ a)$.
- $adom\ \rho \triangleq \{a \mid \rho ! x = step\ (\mathsf{Look}\ y)\ (fetch\ a)\}$.
- $\vdash_{\mathbb{D}} d \triangleq \exists e\ \rho.\ \vdash_{\mathbb{E}} \rho \wedge d = \mathcal{S}_{\mathbf{need}}[\![e]\!]\rho$.
- $adom\ d \triangleq adom\ \rho$ *where* $d = \mathcal{S}_{\mathbf{need}}[\![e]\!]\rho$.
- $\vdash_{\mathbb{H}} \mu \triangleq \forall a.\ \exists d.\ \mu ! a = memo\ a\ d \wedge \blacktriangleright(\vdash_{\mathbb{D}} d \wedge adom\ d \subseteq dom\ \mu)$.

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

Henceforth, I assume that all elements d, environments ρ and heaps μ of interest are definable in this sense. It is easy to see that definability is preserved by $\mathcal{S}_{\mathbf{need}}[\![\_]\!]\_$ whenever the environment or heap is extended; the important case is the implementation of *bind*.

The indexed family of abstraction functions improves whenever the heap with which we index is "more evaluated" — the binary relation ($\rightsquigarrow$) ("progresses to") on heaps in Figure 4.27 captures this progression. It is defined as the smallest pre-order (rules $\rightsquigarrow$-Refl, $\rightsquigarrow$-Trans) that contains rules $\rightsquigarrow$-Ext and $\rightsquigarrow$-Memo. The former corresponds to extending the heap in the Let case. The latter corresponds to memoising a heap entry after it was evaluated in the Var case.

Heap progression is the primary mechanism by which we can reason about the meaning of programs: If μ_1 progresses to μ_2 (i.e. $\mu_1 \rightsquigarrow \mu_2$), and $\mathit{adom}\ d \subseteq \mathit{dom}\ \mu_1$, then $d\ \mu_1$ has the same by-name semantics as $d\ \mu_2$, with the latter potentially terminating in fewer steps. We will exploit this observation in the abstract in Lemma 4.33, and now work towards a proof.

To that end, it is important to build witnesses of $\mu_1 \rightsquigarrow \mu_2$ in the first place:

Lemma 4.30 (Evaluation progresses the heap)**.**
If $\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}(\mu_1) = \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))$, *then* $\mu_1 \rightsquigarrow \mu_2$.

Proof. By Löb induction and cases on e.

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

  $$
  \begin{aligned}
  & (\rho_1 ! x)\ \mu_1 \\
  = \quad & \wr\ \vdash_{\mathbb{E}} \rho_1, \text{some } y, a\ \wr \\
  & \mathsf{Step}\ (\mathsf{Look}\ y)\ (\mathit{fetch}\ a\ \mu_1) \\
  = \quad & \wr\ \text{Unfold } \mathit{fetch}\ \wr \\
  & \mathsf{Step}\ (\mathsf{Look}\ y)\ ((\mu_1 ! a)\ \mu_1) \\
  = \quad & \wr\ \vdash_{\mathbb{H}} \mu, \text{some } e, \rho_3\ \wr \\
  & \mathsf{Step}\ (\mathsf{Look}\ y)\ (\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_3})\ \mu_1) \\
  = \quad & \wr\ \text{Unfold } \mathit{memo}\ \wr \\
  & \mathsf{Step}\ (\mathsf{Look}\ y)\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_3}(\mu_1) \gg\!= \mathit{upd}) \\
  = \quad & \wr\ \mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_3}(\mu_1) = \overline{\mathsf{Step}\ ev_1}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_3)) \text{ for some } \mu_3\ \wr \\
  & \mathsf{Step}\ (\mathsf{Look}\ y)\ (\overline{\mathsf{Step}\ ev_1}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_3) \gg\!= \mathit{upd})) \\
  = \quad & \wr\ \text{Unfold} \gg\!=, \mathit{upd}\ \wr \\
  & \mathsf{Step}\ (\mathsf{Look}\ y)\ (\overline{\mathsf{Step}\ ev_1}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_3) \gg\!= \lambda v\ \mu_3 \rightarrow \\
  & \quad \mathsf{Step}\ \mathsf{Upd}\ (\mathsf{Ret}\ (v, \mu_3[a \mapsto \mathit{memo}\ a\ (\mathit{return}\ v)]))))
  \end{aligned}
  $$

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

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

We have

$$\mu_1 \,!\, a = \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]\rho_3) \tag{4.7}$$

$$\blacktriangleright(\mathcal{S}_{\mathbf{need}}[\![e]\!]\rho_3(\mu_1)) = \overline{\mathsf{Step}\ ev_1}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]\rho_2(\mu_3))) \tag{4.8}$$

$$\mu_2 = \mu_3[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]\rho_2)] \tag{4.9}$$

We can apply rule $\rightsquigarrow$-Memo to Equation (4.7) and Equation (4.8) to get $\mu_1 \rightsquigarrow \mu_3[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]\rho_2)]$, and rewriting along Equation (4.9) proves the goal.

- **Case** $\mathsf{Lam}\ x\ \mathit{body}$, $\mathsf{ConApp}\ k\ \mathit{xs}$: Then $\mu_1 = \mu_2$ and the goal follows by $\rightsquigarrow$-Refl.

- **Case** $\mathsf{App}\ e_1\ x$: Let us assume that

$$\mathcal{S}_{\mathbf{need}}[\![e_1]\!]\rho_1(\mu_1) = \overline{\mathsf{Step}\ ev_1}\ (\mathcal{S}_{\mathbf{need}}[\![\mathsf{Lam}\ y\ e_2]\!]\rho_3(\mu_3))$$
$$\mathcal{S}_{\mathbf{need}}[\![e_2]\!]\rho_3[y \mapsto \rho\,!\,x](\mu_3) = \overline{\mathsf{Step}\ ev_2}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]\rho_2(\mu_2))$$

  so that $\mu_1 \rightsquigarrow \mu_3$, $\mu_3 \rightsquigarrow \mu_2$ by the induction hypothesis. The goal follows by $\rightsquigarrow$-Trans, because $\overline{ev} = [\mathsf{App}_1] +\!\!+ \overline{ev_1} +\!\!+ [\mathsf{App}_2] +\!\!+ \overline{ev_2}$.

- **Case** $\mathsf{Case}\ e_1\ \mathit{alts}$: Similar to $\mathsf{App}\ e_1\ x$.

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

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

  Let me abbreviate $d_1 \triangleq \mathcal{S}_{\mathbf{need}}[\![e_1]\!]\rho_1[x \mapsto \mathit{step}\ (\mathsf{Look}\ x)\ (\mathit{fetch}\ a)]$. We apply the induction hypothesis at $\mathcal{S}_{\mathbf{need}}[\![e_2]\!]\rho_1[x \mapsto \mathit{step}\ (\mathsf{Look}\ x)\ (\mathit{fetch}\ a)]$ to conclude that $\mu_1[a \mapsto \mathit{memo}\ a\ d_1] \rightsquigarrow \mu_2$.

  On the other hand, we have $\mu_1 \rightsquigarrow \mu_1[a \mapsto \mathit{memo}\ a\ d_1]$ by rule $\rightsquigarrow$-Ext (note that $a \notin \mathit{dom}\ \mu$), so the goal follows by $\rightsquigarrow$-Trans.

□

It is often necessary, but non-trivial to cope with equality of elements d modulo readdressing. Fortunately, we only need to consider equality in the abstract, that is, modulo $\beta_{\mathbb{D}}$, where $\beta_{\mathbb{D}}(\mu)(d) \triangleq \alpha_{\mathbb{D}}(\mu)(\{d\})$ is the representation function of $\alpha_{\mathbb{D}}$.

Lemma 4.31 (Abstract readdressing)**.** *If $a_1 \in dom\ \mu_1$, but $a_2 \notin dom\ \mu_1$, then $\beta_{\mathbb{D}}(\mu_1)(\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}) = \beta_{\mathbb{D}}(\mu_2)(\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_2})$, where ρ_2 and μ_2 are renamings of ρ_1 and μ_1 defined as follows:*

- $\rho_2 ! x = \begin{cases} step\ (\mathsf{Look}\ y)\ (fetch\ a_2) & \text{if } \rho_1 ! x = step\ (\mathsf{Look}\ y)\ (fetch\ a_1) \\ \rho_1 ! x & \text{otherwise} \end{cases}$
- $a_1 \notin dom\ \mu_2$
- $\mu_2 ! a = \begin{cases} memo\ a_2\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_4}) & \text{if } a = a_2,\ \rho_4 \text{ renaming of } \rho_3, \\ & \mu_1 ! a_1 = memo\ a_1\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_3}) \\ memo\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_4}) & \text{where } \rho_4 \text{ renaming of } \rho_3, \\ & \mu_1 ! a = memo\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_3}) \end{cases}$

Proof. Simple proof by Löb induction and cases on e. □

Readdressing allows us to prove an abstract pendant of the venerable *frame rule* of separation logic [Reynolds 2002]:

Lemma 4.32 (Abstract frame rule)**.** *If $adom\ \rho \subseteq dom\ \mu$ and $a \notin dom\ \mu$, then*

$$\beta_{\mathbb{D}}(\mu)(\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho}) = \beta_{\mathbb{D}}(\mu[a \mapsto memo\ a\ d])(\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho}).$$

Proof. By Löb induction and cases on e. Only the cases that access the heap are interesting.

- **Case** $\mathsf{Var}\ x$: We never fetch a, because $a \notin adom\ \rho$. Furthermore, the environment ρ_1 of the heap entry $\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_1}$ thus fetched satisfies $adom\ \rho_1 \subseteq dom\ \mu$ so that we may apply the induction hypothesis.
- **Case** $\mathsf{Let}\ x\ e_1\ e_2$: Follows by the induction hypothesis after readdressing the extended heap (Lemma 4.31) so that the induction hypothesis can be applied.

□

The frame rule in turn is important to show that heap progression preserves the results of the abstraction function:

Lemma 4.33 (Heap progression preserves abstraction)**.** *Let* $\widehat{\mathsf{D}}$ *be a domain with instances for* Trace, Domain, HasBind *and* Lat*, satisfying* Beta-App*,* Beta-Sel*,* ByName-Bind *and* Step-Inc *from Figure 4.25.*

$$\mu_1 \rightsquigarrow \mu_2 \wedge \mathit{adom}\ d \subseteq \mathit{dom}\ \mu_1 \implies \beta_{\mathbb{D}}(\mu_2)(d) \sqsubseteq \beta_{\mathbb{D}}(\mu_1)(d).$$

Proof. By Löb induction. Element d is definable of the form $d = \mathcal{S}_{\mathbf{need}}[\![e]\!]_\rho$ for definable ρ. Proceed by cases on e. Only the Var and Let cases are interesting.

- **Case** Let x e_1 e_2: We need to readdress the extended heaps with Lemma 4.31 so that $\mu_1[a_1 \mapsto \mathit{memo}\ a_1\ d_1] \rightsquigarrow \mu_2[a_1 \mapsto \mathit{memo}\ a_1\ d_1]$ is preserved, in which case the goal follows by applying the induction hypothesis.

- **Case** Var x: Let us assume that $\mu_1 \rightsquigarrow \mu_2$ and $\mathit{adom}\ d \subseteq \mathit{dom}\ \mu_1$. We get $d = \mathit{step}\ (\mathsf{Look}\ y)\ (\mathit{fetch}\ a)$ for some y and a. Furthermore, let us abbreviate $\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_i]\!]_{\rho_i}) \triangleq \mu_i\,!\,a$. The goal is to show

  $$\begin{aligned} &\mathit{step}\ (\mathsf{Look}\ y)\ (\beta_{\mathbb{D}}(\mu_2)(\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_2]\!]_{\rho_2}))) \\ \sqsubseteq\ &\mathit{step}\ (\mathsf{Look}\ y)\ (\beta_{\mathbb{D}}(\mu_1)(\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_1}))) \end{aligned}$$

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

  $$\blacktriangleright(\beta_{\mathbb{D}}(\mu_2)(\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_2]\!]_{\rho_2})) \sqsubseteq \beta_{\mathbb{D}}(\mu_1)(\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_1}))).$$

  Now we proceed by induction on $\mu_1 \rightsquigarrow \mu_2$, which we only use to prove correct the reflexive and transitive closure in $\rightsquigarrow$-Refl and $\rightsquigarrow$-Trans.

  - **Case** $\rightsquigarrow$-Refl: Then $\mu_1 = \mu_2$ and hence $\beta_{\mathbb{D}}(\mu_1) = \beta_{\mathbb{D}}(\mu_2)$.
  - **Case** $\rightsquigarrow$-Trans: Apply the induction hypothesis to the sub-derivations and apply transitivity of $\sqsubseteq$.
  - **Case** $\rightsquigarrow$-Ext $\dfrac{a_1 \notin \mathit{dom}\ \mu_1 \quad \mathit{adom}\ \rho \subseteq \mathit{dom}\ \mu_1 \cup \{a_1\}}{\mu_1 \rightsquigarrow \mu_1[a_1 \mapsto \mathit{memo}\ a_1\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_\rho)]}$:
    We get to refine $\mu_2 = \mu_1[a_1 \mapsto \mathit{memo}\ a_1\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]_\rho)]$. Since $a \in \mathit{dom}\ \mu_1$, we have $a_1 \neq a$ and thus $\mu_1\,!\,a = \mu_2\,!\,a$, thus $e_1 = e_2$, $\rho_1 = \rho_2$. We can exploit monotonicity of $\blacktriangleright$ and simplify the goal to

$$\begin{aligned}&\beta_{\mathbb{D}}(\mu_1[a_1 \mapsto \mathit{memo}\ a_1\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]\rho)])(\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]\rho_1))\\ \sqsubseteq\ &\beta_{\mathbb{D}}(\mu_1)(\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]\rho_1))\end{aligned}$$

This follows by applying the abstract frame rule (Lemma 4.32), because $\mathit{adom}\ \rho_1 \subseteq \mathit{dom}\ \mu_1$.

– **Case** $\rightsquigarrow$-Memo $\dfrac{\begin{array}{c}\mu_1\,!\,a_1 = \mathit{memo}\ a_1\ (\mathcal{S}_{\mathbf{need}}[\![e]\!]\rho_3)\\ \blacktriangleright(\mathcal{S}_{\mathbf{need}}[\![e]\!]\rho_3(\mu_1) = \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]\rho_4(\mu_3)))\end{array}}{\mu_1 \rightsquigarrow \mu_3[a_1 \mapsto \mathit{memo}\ a_1\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]\rho_4)]}$:

We get to refine $\mu_2 = \mu_3[a_1 \mapsto \mathit{memo}\ a_1\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]\rho_4)]$. When $a_1 \neq a$, we have $\mu_1\,!\,a = \mu_2\,!\,a$ and the goal follows as in the $\rightsquigarrow$-Ext case. Otherwise, $a = a_1$, $e_1 = e$, $\rho_3 = \rho_1$, $\rho_4 = \rho_2$, $e_2 = v$.

The goal can be simplified to

$$\blacktriangleright(\beta_{\mathbb{D}}(\mu_2)(\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]\rho_2)) \sqsubseteq \beta_{\mathbb{D}}(\mu_1)(\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]\rho_1)))$$

We reason under $\blacktriangleright$ as follows

$$\begin{aligned}&\beta_{\mathbb{D}}(\mu_2)(\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]\rho_2))\\ =\ &\wr\ \mu_2\,!\,a = \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]\rho_2)\ \text{already; update no-op}\ \int\\ &\beta_{\mathbb{T}}(\mathsf{Step\ Update}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]\rho_2(\mu_2)))\\ =\ &\wr\ \mu_2 = \mu_3[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]\rho_2)]\ \int\\ &\beta_{\mathbb{D}}(\mu_3)(\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]\rho_2))\\ \sqsubseteq\ &\wr\ \text{Assumption Step-Inc}\ \int\\ &\overline{\mathit{step}\ ev}\ (\beta_{\mathbb{D}}(\mu_3)\ (\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]\rho_2)))\\ =\ &\wr\ \text{Unfold}\ \mathit{memo}, \beta_{\mathbb{D}}\ \int\\ &\overline{\mathit{step}\ ev}\ (\beta_{\mathbb{T}}(\mathcal{S}_{\mathbf{need}}[\![v]\!]\rho_2(\mu_3) \gg\!= \mathit{upd}))\\ =\ &\wr\ \text{Refold}\ \beta_{\mathbb{T}}, \gg\!=\ \int\\ &\beta_{\mathbb{T}}(\overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]\rho_2(\mu_3)) \gg\!= \mathit{upd})\\ =\ &\wr\ \mathcal{S}_{\mathbf{need}}[\![e_1]\!]\rho_1(\mu_1) = \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]\rho_2(\mu_3))\ \int\\ &\beta_{\mathbb{T}}(\mathcal{S}_{\mathbf{need}}[\![e_1]\!]\rho_1(\mu_1) \gg\!= \mathit{upd})\\ =\ &\wr\ \text{Refold}\ \mathit{memo}, \beta_{\mathbb{D}}\ \int\\ &\beta_{\mathbb{D}}(\mu_1)(\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]\rho_1))\end{aligned}$$

□

The preceding lemma corresponds to the Upd step of the preservation Lemma A.9 from Section 4.1 where we (and Sergey, Vytiniotis, et al. [2017]) resorted to hand-waving. Here, we hand-wave no more!

In order to prove the main soundness Theorem 4.18, we need two auxiliary lemmas. Similar to Lemma 4.27, the first is about $(\gg\!=)$, while the other is about environment access.

Lemma 4.34 (By-need bind)**.** *It is* $\beta_{\mathbb{T}}((d \gg\!= f)\ \mu_1) \sqsubseteq \widehat{f}\ \widehat{d}$ *if*

1. $\beta_{\mathbb{T}}(d\ \mu_1) \sqsubseteq \widehat{d}$*, and*
2. *for all events* ev *and elements* $\widehat{d'}$*,* $\widehat{step}\ ev\ (\widehat{f}\ \widehat{d'}) \sqsubseteq \widehat{f}\ (\widehat{step}\ ev\ \widehat{d'})$*, and*
3. *for all values* v *and heaps* μ_2 *such that* $\mu_1 \rightsquigarrow \mu_2$*,* $\beta_{\mathbb{T}}(f\ \ v\ \ \mu_2) \sqsubseteq \widehat{f}\ (\beta_{\mathbb{T}}(\mathsf{Ret}\ (v, \mu_2)))$.

Proof. By assumption (1), it suffices to show $\beta_{\mathbb{T}}((d \gg\!= f)\ \mu_1) \sqsubseteq \widehat{f}\ (\beta_{\mathbb{T}}(d\ \mu_1))$. Let us first consider the case where the trace $\tau \triangleq d\ \mu_1$ is infinite; then $\tau = (d \gg\!= f)\ \mu_1$ and hence $\beta_{\mathbb{T}}((d \gg\!= f)\ \mu_1) = \beta_{\mathbb{T}}(\tau)$. By Löb induction.

$$
\begin{aligned}
&\beta_{\mathbb{T}}((d \gg\!= f)\ \mu_1) = \beta_{\mathbb{T}}(\tau) = \beta_{\mathbb{T}}(\mathsf{Step}\ ev\ \tau') = \widehat{step}\ ev\ (\beta_{\mathbb{T}}(\tau')) \\
\sqsubseteq\ &\wr \text{ Induction hypothesis at } \beta_{\mathbb{T}}(\tau'), \text{ Monotonicity of } \widehat{step} \wr \\
&\widehat{step}\ ev\ (\widehat{f}\ (\beta_{\mathbb{T}}(\tau'))) \\
\sqsubseteq\ &\wr \text{ Assumption (2) } \wr \\
&\widehat{f}\ (\widehat{step}\ ev\ (\beta_{\mathbb{T}}(\tau'))) = \widehat{f}\ (\beta_{\mathbb{T}}(\tau))
\end{aligned}
$$

Otherwise, $d\ \mu_1$ is finite and $d = \mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}$ for some e,ρ_1 since d is definable. Then $\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}(\mu_1) = \overline{\mathsf{Step}\ ev}\ (\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2))$ for some number of events $\overline{ev}$, v, ρ_2 and μ_2. By Lemma 4.30, we have $\mu_1 \rightsquigarrow \mu_2$. We proceed by induction on $\overline{ev}$.

The induction step is the same as in the infinite case above; we shift the Step transition out of the argument to $\beta_{\mathbb{T}}$, apply the induction hypothesis and apply assumption (2).

The interesting case is the base case, when $\overline{ev}$ is empty and $\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho_1}(\mu_1) = \mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2)$. Then we get, defining sv as $return\ sv \triangleq \mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}$,

$$
\begin{aligned}
&\beta_{\mathbb{T}}((return\ sv \gg\!= f)\ \mu_2) = \beta_{\mathbb{T}}(f\ sv\ \mu_2) \\
\sqsubseteq\ &\wr \text{ Assumption (3) at } \mu_1 \rightsquigarrow \mu_2 \wr \\
&\widehat{f}\ (\beta_{\mathbb{T}}(\mathsf{Ret}\ sv, \mu_2)) = \widehat{f}\ (\beta_{\mathbb{T}}(\mathcal{S}_{\mathbf{need}}[\![v]\!]_{\rho_2}(\mu_2)))
\end{aligned}
$$

Note that in order to apply assumption (3) at μ_2 above, we need that $\mu_1 \rightsquigarrow \mu_2$. This would not be possible without generalising for any such μ_2 in the first place. □

Lemma 4.35 (By-need environment unrolling). *Let* $\widehat{\mathsf{D}}$ *be a domain with instances for* Trace, Domain, HasBind *and* Lat, *satisfying* Update *from Figure 4.25, and let* $\mu ! a = \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]\rho_1)$ *and* $\rho ! x = \mathit{step}\ (\mathsf{Look}\ y)\ (\mathit{fetch}\ a)$.
If $\triangleright\ (\forall e\ \rho\ \mu.\ \beta_{\mathbb{T}}(\mathcal{S}_{\mathbf{need}}[\![e]\!]\rho(\mu)) \sqsubseteq (\mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_{\beta_{\mathbb{D}}(\mu)\lhd\rho}))$,
then $\beta_{\mathbb{D}}(\mu)(\rho ! x) \sqsubseteq \mathit{step}\ (\mathsf{Look}\ x)\ (\mathcal{S}_{\widehat{\mathsf{D}}}[\![e_1]\!]_{\beta_{\mathbb{D}}(\mu)\lhd\rho_1})$.

Proof. Note that the antecedent is exactly the Löb induction hypothesis of Theorem 4.18.

$$
\begin{aligned}
& \beta_{\mathbb{D}}(\mu)(\rho ! x) \\
= \ & \wr \text{ Unfold } \rho ! x, \mu ! a, \beta_{\mathbb{D}} \text{ and } \mathit{fetch}\ a \ \wr \\
& \mathit{step}\ (\mathsf{Look}\ x)\ (\beta_{\mathbb{T}}(\mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]\rho_1)\ \mu)) \\
= \ & \wr \text{ Unfold } \mathit{memo}\ a \ \wr \\
& \mathit{step}\ (\mathsf{Look}\ x)\ (\beta_{\mathbb{T}}((\mathcal{S}_{\mathbf{need}}[\![e_1]\!]\rho_1 \gg\!= \mathit{upd})\ \mu)) \\
\sqsubseteq \ & \wr \text{ Apply Lemma 4.34; see below } \wr \\
& \mathit{step}\ (\mathsf{Look}\ x)\ (\mathcal{S}_{\widehat{\mathsf{D}}}[\![e_1]\!]_{\beta_{\mathbb{D}}(\mu)\lhd\rho_1})
\end{aligned}
$$

In the last step, we apply Lemma 4.34 under $\mathit{step}\ (\mathsf{Look}\ x)$ at $f \triangleq \mathit{upd}$, $\widehat{f} = \mathit{id}$, $d \triangleq \mathcal{S}_{\mathbf{need}}[\![e_1]\!]\rho_1$, $\widehat{d} \triangleq \mathcal{S}_{\widehat{\mathsf{D}}}[\![e_1]\!]_{\beta_{\mathbb{D}}(\mu)\lhd\rho_1}$, which yields three subgoals (under $\triangleright$):

- $\beta_{\mathbb{T}}(\mathcal{S}_{\mathbf{need}}[\![e_1]\!]\rho_1(\mu)) \sqsubseteq \mathcal{S}_{\widehat{\mathsf{D}}}[\![e_1]\!]_{\beta_{\mathbb{D}}(\mu)\lhd\rho_1}$: By assumption.
- $\forall ev\ \widehat{d'}.\ \widehat{\mathit{step}}\ ev\ (\mathit{id}\ \widehat{d'}) \sqsubseteq \mathit{id}\ (\widehat{\mathit{step}}\ ev\ \widehat{d'})$: By reflexivity.
- $\forall v\ \mu_2.\ \mu \rightsquigarrow \mu_2 \implies \beta_{\mathbb{T}}(\mathit{upd}\ v\ \mu_2) \sqsubseteq \mathit{id}\ (\beta_{\mathbb{T}}(\mathsf{Ret}\ (v, \mu_2)))$:
  - **Case** $v = \mathsf{Stuck}$: Then $\mathit{upd}\ v\ \mu_2 = \mathsf{Ret}\ (v, \mu_2)$ and the goal follows by reflexivity.
  - **Case** $v = \mathsf{Fun}\ f$, $v = \mathsf{Con}\ k\ ds$:
    Then $\mathit{upd}\ v\ \mu_2 = \mathsf{Step}\ \mathsf{Update}\ (\mathsf{Ret}\ (v, \mu_2[a \mapsto \mathit{memo}\ a\ (\mathit{return}\ v)]))$.
    By law Update, it suffices to show
    $\beta_{\mathbb{T}}(\mathsf{Ret}\ (v, \mu_2[a \mapsto \mathit{memo}\ a\ (\mathit{return}\ v)])) \sqsubseteq \beta_{\mathbb{T}}(\mathsf{Ret}\ (v, \mu_2))$. It is $\mu_2 \rightsquigarrow \mu_2[a \mapsto \mathit{memo}\ a\ (\mathit{return}\ v)]$ by $\rightsquigarrow$-Memo and the goal follows by Lemma 4.33.

□

Finally, we can prove Theorem 4.18.

Theorem 4.18 (Abstract By-need Interpretation). *Let e be an expression, $\widehat{\mathsf{D}}$ a* ↻ 139 *domain with instances for* Trace, Domain, HasBind *and* Lat, *and let $\alpha_{\mathcal{S}}$ be the abstraction function from Figure 4.24. If the abstraction laws in Figure 4.25 hold, then $\mathcal{S}_{\widehat{\mathsf{D}}}[\![\_]\!]$ is an abstract interpreter that is sound wrt. $\alpha_{\mathcal{S}}$, that is,*

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

Proof. We simplify our proof obligation to the single-trace case.

$$\begin{array}{ll}
& \forall e.\ \alpha_{\mathcal{S}}(\mathcal{S}_{\mathbf{need}}[\![e]\!]) \sqsubseteq \mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!] \\
\iff & \wr \text{ Unfold } \alpha_{\mathcal{S}}, \alpha_{\mathbb{T}} \ \int \\
& \forall e\ \widehat{\rho}.\ \bigsqcup\{\beta_{\mathbb{T}}(\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho}(\mu)) \mid (\rho,\mu) \in \gamma_{\mathbb{E}}(\widehat{\rho})\} \sqsubseteq \mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_{\widehat{\rho}} \\
\iff & \wr \ (\rho,\mu) \in \gamma_{\mathbb{E}}(\widehat{\rho}) \iff \alpha_{\mathbb{E}}(\{(\rho,\mu)\}) \sqsubseteq \widehat{\rho}, \text{ unfold } \alpha_{\mathbb{E}}, \text{ refold } \beta_{\mathbb{D}} \ \int \\
& \forall e\ \rho\ \mu.\ \beta_{\mathbb{T}}(\mathcal{S}_{\mathbf{need}}[\![e]\!]_{\rho}(\mu)) \sqsubseteq \mathcal{S}_{\widehat{\mathsf{D}}}[\![e]\!]_{\beta_{\mathbb{D}}(\mu) \lhd \rho}
\end{array}$$

where $\beta_{\mathbb{T}} \triangleq \alpha_{\mathbb{T}} \circ \{\_\}$ and $\beta_{\mathbb{D}}(\mu) \triangleq \alpha_{\mathbb{D}}(\mu) \circ \{\_\}$ are the representation functions corresponding to $\alpha_{\mathbb{T}}$ and $\alpha_{\mathbb{D}}$, and the operation $f \lhd \rho$ from Figure 4.4 maps f over every entry in ρ. We proceed by Löb induction and cases over e.

- **Case** Var x: The case $x \notin \mathit{dom}\ \rho$ follows by reflexivity. Otherwise,

$$\beta_{\mathbb{T}}(\mathcal{S}_{\mathbf{need}}[\![\mathsf{Var}\ x]\!]_{\rho}(\mu)) = \beta_{\mathbb{T}}((\rho\,!\,x)\ \mu) = \mathcal{S}_{\widehat{\mathsf{D}}}[\![\mathsf{Var}\ x]\!]_{\beta_{\mathbb{D}}(\mu) \lhd \rho}.$$

- **Case** Lam x *body*:

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

- **Case** ConApp k xs:

$$\begin{array}{ll}
& \beta_{\mathbb{T}}(\mathcal{S}_{\mathbf{need}}[\![\mathsf{ConApp}\ k\ xs]\!]_{\rho}(\mu)) \\
= & \wr \text{ Unfold } \mathcal{S}_{\mathbf{need}}[\![_]\!]_, \beta_{\mathbb{T}} \ \int
\end{array}$$

$$
\begin{aligned}
& \mathit{con}\ k\ (\mathit{map}\ ((\beta_{\mathbb{D}}(\mu) \lhd \rho)\,!)\ \mathit{xs}) \\
= \ & \wr \text{ Refold } \mathcal{S}_{\widehat{\mathbb{D}}}[\![\_]\!]_\_ \ \wr \\
& \mathcal{S}_{\widehat{\mathbb{D}}}[\![\mathsf{Lam}\ x\ \mathit{body}]\!]_{\beta_{\mathbb{D}}(\mu) \lhd \rho}
\end{aligned}
$$

- **Case** $\mathsf{App}\ e\ x$: Very similar to the *apply* case in Theorem 4.28, except that we apply the by-need variant of the bind Lemma 4.34.

  The *stuck* case is simple. Otherwise, we have

$$
\begin{aligned}
& \beta_{\mathbb{T}}(\mathcal{S}_{\mathbf{need}}[\![\mathsf{App}\ e\ x]\!]_\rho(\mu)) \\
= \ & \wr \text{ Unfold } \mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_, \beta_{\mathbb{T}}, \mathit{apply} \ \wr \\
& \mathit{step}\ \mathsf{App}_1\ ((\mathcal{S}_{\mathbf{need}}[\![e]\!]_\rho \gg\!= \lambda v \to \\
& \quad \mathbf{case}\ v\ \mathbf{of}\ \mathsf{Fun}\ f \to f\ (\rho\,!\,x); \_ \to \mathit{stuck})\ \mu) \\
\sqsubseteq \ & \wr \text{ Apply Lemma 4.34; see below } \wr \\
& \mathit{step}\ \mathsf{App}_1\ (\widehat{\mathit{apply}}\ (\mathcal{S}_{\widehat{\mathbb{D}}}[\![e]\!]_{\beta_{\mathbb{D}}(\mu) \lhd \rho})\ (\beta_{\mathbb{D}}(\mu)(\rho\,!\,x))) \\
= \ & \wr \text{ Refold } \mathcal{S}_{\widehat{\mathbb{D}}}[\![\_]\!]_\_ \ \wr \\
& \mathcal{S}_{\widehat{\mathbb{D}}}[\![e]\!]_{\beta_{\mathbb{D}}(\mu) \lhd \rho}
\end{aligned}
$$

  In the $\sqsubseteq$ step, we apply Lemma 4.34 under $\mathit{step}\ \mathsf{App}_1$, which yields three subgoals (under $\blacktriangleright$, and abbreviating $\widehat{a} \triangleq \beta_{\mathbb{D}}(\mu)(\rho\,!\,x)$):

  - $\beta_{\mathbb{T}}(\mathcal{S}_{\mathbf{need}}[\![e]\!]_\rho(\mu)) \sqsubseteq \mathcal{S}_{\widehat{\mathbb{D}}}[\![e]\!]_{\beta_{\mathbb{D}}(\mu) \lhd \rho}$: By induction hypothesis.
  - $\forall ev\ \widehat{d'}.\ \widehat{\mathit{step}}\ ev\ (\widehat{\mathit{apply}}\ \widehat{d'}\ \widehat{a}) \sqsubseteq \widehat{\mathit{apply}}\ (\widehat{\mathit{step}}\ ev\ \widehat{d'})\ \widehat{a}$: By assumption Step-App.
  - $\forall\, v\ \mu_2.\ \mu \rightsquigarrow \mu_2 \implies \beta_{\mathbb{T}}((\mathbf{case}\ v\ \mathbf{of}\ \mathsf{Fun}\ g \to g\ (\rho\,!\,x); \_ \to \mathit{stuck})\ \mu_2) \sqsubseteq \widehat{\mathit{apply}}\ (\beta_{\mathbb{T}}(\mathsf{Ret}\ (v, \mu_2)))\ \widehat{a}$: By cases over v.
    - **Case** $v = \mathsf{Stuck}$: Then $\beta_{\mathbb{T}}(\mathit{stuck}\ \mu_2) = \widehat{\mathit{stuck}} \sqsubseteq \widehat{\mathit{apply}}\ \widehat{\mathit{stuck}}\ \widehat{a}$ by assumption Stuck-App.
    - **Case** $v = \mathsf{Con}\ k\ ds$: Then $\beta_{\mathbb{T}}(\mathit{stuck}\ \mu_2) = \widehat{\mathit{stuck}} \sqsubseteq \widehat{\mathit{apply}}\ (\widehat{\mathit{con}}\ k\ \widehat{ds})\ \widehat{a}$ by assumption Stuck-App, for the suitable $\widehat{ds}$.
    - **Case** $v = \mathsf{Fun}\ g$: Note that g has a parametric definition, of the form $(\lambda d \to \mathit{step}\ \mathsf{App}_2\ (\mathcal{S}[\![e_1]\!]_{\rho[x \mapsto d]}))$. This is important to apply Beta-App below.

$$
\begin{aligned}
& \beta_{\mathbb{T}}(g\ (\rho\,!\,x)\ \mu_2) \\
\sqsubseteq \ & \wr\ \mathit{id} \sqsubseteq \gamma_{\mathbb{D}}(\mu_2) \circ \alpha_{\mathbb{D}}(\mu_2), \text{ rearrange } \wr
\end{aligned}
$$

$$
\begin{array}{ll}
& (\alpha_{\mathbb{D}}(\mu_2) \circ g^* \circ \gamma_{\mathbb{D}}(\mu_2))\ (\beta_{\mathbb{D}}(\mu_2)(\rho\,!\,x)) \\
\sqsubseteq & \wr\ \beta_{\mathbb{D}}(\mu_2)(\rho\,!\,x) \sqsubseteq \beta_{\mathbb{D}}(\mu)(\rho\,!\,x) = \widehat{a}\ \text{by Lemma 4.33}\ \int \\
& (\alpha_{\mathbb{D}}(\mu_2) \circ g^* \circ \gamma_{\mathbb{D}}(\mu_2))\ \widehat{a} \\
\sqsubseteq & \wr\ \text{Assumption Beta-App}\ \int \\
& \widehat{apply}\ (\widehat{fun}\ (\alpha_{\mathbb{D}}(\mu_2) \circ g^* \circ \gamma_{\mathbb{D}}(\mu_2)))\ \widehat{a} \\
= & \wr\ \text{Definition of } \beta_{\mathbb{T}},\ v\ \int \\
& \widehat{apply}\ (\beta_{\mathbb{T}}(\mathsf{Ret}\ v, \mu_2))\ \widehat{a}
\end{array}
$$

- **Case** Case *e alts*: Very similar to the *select* case in Theorem 4.28.

  The cases where the interpreter returns *stuck* follow by parametricity. Otherwise, we have (for the suitable definition of $\widehat{alts}$, which satisfies $\alpha_{\mathbb{D}}(\mu_2) \circ (alts\,!\,k)^* \circ map\ (\gamma_{\mathbb{D}}(\mu_2)) \sqsubseteq \widehat{alts}\,!\,k$ by induction)

$$
\begin{array}{ll}
& \beta_{\mathbb{T}}(\mathcal{S}_{\mathbf{need}}[\![\mathsf{Case}\ e\ alts]\!]_\rho(\mu)) \\
= & \wr\ \text{Unfold } \mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_,\ \beta_{\mathbb{T}},\ select\ \int \\
& step\ \mathsf{Case}_1\ ((\mathcal{S}_{\mathbf{need}}[\![e]\!]_\rho \gg\!= \lambda v \rightarrow \\
& \quad \mathbf{case}\ v\ \mathbf{of}\ \mathsf{Con}\ k\ ds \mid k \in dom\ alts \rightarrow (alts\,!\,k)\ ds; \_ \rightarrow stuck)\ \mu) \\
\sqsubseteq & \wr\ \text{Apply Lemma 4.34; see below}\ \int \\
& step\ \mathsf{Case}_1\ (\widehat{select}\ (\mathcal{S}_{\widehat{\mathbb{D}}}[\![e]\!]_{\beta_{\mathbb{D}}(\mu) \lhd \rho})\ \widehat{alts}) \\
= & \wr\ \text{Refold } \mathcal{S}_{\widehat{\mathbb{D}}}[\![\_]\!]_\_\ \int \\
& \mathcal{S}_{\widehat{\mathbb{D}}}[\![e]\!]_{\beta_{\mathbb{D}}(\mu) \lhd \rho}
\end{array}
$$

  In the $\sqsubseteq$ step, we apply Lemma 4.34 under $step\ \mathsf{Case}_1$, which yields three subgoals (under $\blacktriangleright$):

  - $\beta_{\mathbb{T}}(\mathcal{S}_{\mathbf{need}}[\![e]\!]_\rho(\mu)) \sqsubseteq \mathcal{S}_{\widehat{\mathbb{D}}}[\![e]\!]_{\beta_{\mathbb{D}}(\mu) \lhd \rho}$: By induction hypothesis.
  - $\forall ev\ \widehat{d'}.\ \widehat{step}\ ev\ (\widehat{select}\ \widehat{d'}\ \widehat{alts}) \sqsubseteq \widehat{select}\ (\widehat{step}\ ev\ \widehat{d'})\ \widehat{alts}$: By assumption Step-Select.
  - $\forall v\ \mu_2.\ \mu \rightsquigarrow \mu_2 \implies \beta_{\mathbb{T}}((\mathbf{case}\ v\ \mathbf{of}\ \mathsf{Con}\ k\ ds \mid k \in dom\ alts \rightarrow (alts\,!\,k)\ ds; \_ \rightarrow stuck)\ \mu_2) \sqsubseteq \widehat{select}\ (\beta_{\mathbb{T}}(\mathsf{Ret}\ (v, \mu_2)))\ \widehat{alts}$: By cases over v.
    - **Case** $v = \mathsf{Stuck}$: Then $\beta_{\mathbb{T}}(stuck\ \mu_2) = \widehat{stuck} \sqsubseteq \widehat{select}\ \widehat{stuck}\ \widehat{alts}$ by assumption Stuck-Sel.
    - **Case** $v = \mathsf{Fun}\ f$: Then $\beta_{\mathbb{T}}(stuck\ \mu_2) = \widehat{stuck} \sqsubseteq \widehat{select}\ (\widehat{fun}\ \widehat{f})\ \widehat{alts}$ by assumption Stuck-Sel, for the suitable $\widehat{f}$.

* **Case** $v = \mathsf{Con}\ k\ ds$, $k \notin \mathit{dom}\ \mathit{alts}$: Then $\beta_{\mathbb{T}}(\mathit{stuck}\ \mu_2) = \widehat{\mathit{stuck}} \sqsubseteq \widehat{\mathit{select}}\ (\widehat{\mathit{con}}\ k\ \widehat{ds})\ \widehat{\mathit{alts}}$ by assumption Stuck-Sel, for the suitable $\widehat{ds}$.
* **Case** $v = \mathsf{Con}\ k\ ds$, $k \in \mathit{dom}\ \mathit{alts}$: Note that alts has a parametric definition. This is important to apply Beta-Sel below.

$$
\begin{array}{ll}
& \beta_{\mathbb{T}}((\mathit{alts}\,!\,k)\ ds\ \mu_2) \\
\sqsubseteq & \wr\ \mathit{id} \sqsubseteq \gamma_{\mathbb{D}}(\mu_2) \circ \alpha_{\mathbb{D}}(\mu_2),\ \text{rearrange}\ \int \\
& (\alpha_{\mathbb{D}}(\mu_2) \circ (\mathit{alts}\,!\,k)^* \circ \mathit{map}\ (\gamma_{\mathbb{D}}(\mu_2)))\ (\mathit{map}\ (\alpha_{\mathbb{D}}(\mu_2) \circ \{\_\})\ ds) \\
\sqsubseteq & \wr\ \text{Abstraction property of } \widehat{\mathit{alts}}\ \int \\
& (\widehat{\mathit{alts}}\,!\,k)\ (\mathit{map}\ (\alpha_{\mathbb{D}}(\mu_2) \circ \{\_\})\ ds) \\
\sqsubseteq & \wr\ \text{Assumption Beta-Sel}\ \int \\
& \widehat{\mathit{select}}\ (\widehat{\mathit{con}}\ k\ (\mathit{map}\ (\alpha_{\mathbb{D}}(\mu_2) \circ \{\_\})\ ds))\ \widehat{\mathit{alts}} \\
= & \wr\ \text{Definition of } \beta_{\mathbb{T}},\ v\ \int \\
& \widehat{\mathit{select}}\ (\beta_{\mathbb{T}}(\mathsf{Ret}\ v))\ \widehat{\mathit{alts}}
\end{array}
$$

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

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

where $\rho_1 \triangleq \rho[x \mapsto \mathit{step}\ (\mathsf{Look}\ x)\ (\mathit{fetch}\ a)]$, $a \triangleq \mathit{nextFree}\ \mu$, $\mu_1 \triangleq \mu[a \mapsto \mathit{memo}\ a\ (\mathcal{S}_{\mathbf{need}}[\![e_1]\!]_{\rho_1})]$, and $\mu \rightsquigarrow \mu_1$ by $\rightsquigarrow$-Ext.

Then $(\beta_{\mathbb{D}}(\mu_1) \lhd \rho_1)\,!\,y \sqsubseteq (\beta_{\mathbb{D}}(\mu) \lhd \rho)\,!\,y$ whenever $x \neq y$ by Lemma 4.33, and $(\beta_{\mathbb{D}}(\mu_1) \lhd \rho_1)\,!\,x \sqsubseteq \mathit{step}\ (\mathsf{Look}\ x)\ (\mathcal{S}_{\widehat{\mathbb{D}}}[\![e_1]\!]_{(\beta_{\mathbb{D}}(\mu_1) \lhd \rho)[x \mapsto \beta_{\mathbb{D}}(\mu_1)(\rho_1\,!\,x)]})$ by Lemma 4.35. From the latter fact, we gather that $(\beta_{\mathbb{D}}(\mu_1) \lhd \rho_1)\,!\,x$ must be smaller than the least fixpoint of the functional $\lambda \widehat{d_1} \to \mathit{step}\ (\mathsf{Look}\ x)\ (\mathcal{S}_{\widehat{\mathbb{D}}}[\![e_1]\!]_{(\beta_{\mathbb{D}}(\mu_1) \lhd \rho)[x \mapsto \widehat{d_1}]})$. In conclusion:

$$
\begin{array}{ll}
& \beta_{\mathbb{T}}(\mathcal{S}_{\mathbf{need}}[\![\mathsf{Let}\ x\ e_1\ e_2]\!]_\rho(\mu)) \\
= & \wr\ \text{Unfold } \mathcal{S}_{\mathbf{need}}[\![\_]\!]_\_,\ \mathit{bind},\ \beta_{\mathbb{T}}\ \int \\
& \mathit{step}\ \mathsf{Let}_1\ (\beta_{\mathbb{T}}(\mathcal{S}_{\mathbf{need}}[\![e_2]\!]_{\rho_1}(\mu_1))) \\
\sqsubseteq & \wr\ \text{Induction hypothesis}\ \int \\
& \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}_{\widehat{\mathbb{D}}}[\![e_2]\!]_{\beta_{\mathbb{D}}(\mu_1) \lhd \rho_1}) \\
\sqsubseteq & \wr\ \text{By Lemma 4.35, unfolding } \rho_1\ \int \\
& \mathbf{let}\ \widehat{d} = \mathit{step}\ (\mathsf{Look}\ x)\ (\mathcal{S}_{\widehat{\mathbb{D}}}[\![e_1]\!]_{(\beta_{\mathbb{D}}(\mu_1) \lhd \rho)[x \mapsto \beta_{\mathbb{D}}(\mu_1)(\rho_1\,!\,x)]})\ \mathbf{in} \\
& \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}_{\widehat{\mathbb{D}}}[\![e_2]\!]_{(\beta_{\mathbb{D}}(\mu_1) \lhd \rho)[x \mapsto \widehat{d}]})
\end{array}
$$

$$
\begin{aligned}
&\sqsubseteq \quad \wr \text{ Least fixpoint, rolling out } \mathit{step}\ (\mathsf{Look}\ x) \ \wr \\
&\quad \mathbf{let}\ \widehat{\mathit{rhs}}\ \widehat{d_1} = \mathcal{S}_{\widehat{\mathsf{D}}}[\![e_1]\!]_{(\beta_{\mathbb{D}}(\mu_1) \lhd \rho)[x \mapsto \mathit{step}\ (\mathsf{Look}\ x)\ \widehat{d_1}]}\ \mathbf{in} \\
&\quad \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}_{\widehat{\mathsf{D}}}[\![e_2]\!]_{(\beta_{\mathbb{D}}(\mu_1) \lhd \rho)[x \mapsto \mathit{step}\ (\mathsf{Look}\ x)\ (\mathit{lfp}\ \widehat{\mathit{rhs}})]}) \\
&\sqsubseteq \quad \wr\ \beta_{\mathbb{D}}(\mu_1)(\rho\,!\,y) \sqsubseteq \beta_{\mathbb{D}}(\mu)(\rho\,!\,y) \text{ by Lemma 4.33} \ \wr \\
&\quad \mathbf{let}\ \widehat{\mathit{rhs}}\ \widehat{d_1} = \mathcal{S}_{\widehat{\mathsf{D}}}[\![e_1]\!]_{(\beta_{\mathbb{D}}(\mu) \lhd \rho)[x \mapsto \mathit{step}\ (\mathsf{Look}\ x)\ \widehat{d_1}]}\ \mathbf{in} \\
&\quad \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}_{\widehat{\mathsf{D}}}[\![e_2]\!]_{(\beta_{\mathbb{D}}(\mu) \lhd \rho)[x \mapsto \mathit{step}\ (\mathsf{Look}\ x)\ (\mathit{lfp}\ \widehat{\mathit{rhs}})]} \\
&\sqsubseteq \quad \wr \text{ Assumption ByName-Bind} \ \wr \\
&\quad \mathbf{let}\ \widehat{\mathit{rhs}}\ \widehat{d_1} = \mathcal{S}_{\widehat{\mathsf{D}}}[\![e_1]\!]_{(\beta_{\mathbb{D}}(\mu) \lhd \rho)[x \mapsto \mathit{step}\ (\mathsf{Look}\ x)\ \widehat{d_1}]}\ \mathbf{in} \\
&\quad \widehat{\mathit{bind}}\ \widehat{\mathit{rhs}}\ (\lambda \widehat{d_1} \to \mathit{step}\ \mathsf{Let}_1\ (\mathcal{S}_{\widehat{\mathsf{D}}}[\![e_2]\!]_{(\beta_{\mathbb{D}}(\mu) \lhd \rho)[x \mapsto \mathit{step}\ (\mathsf{Look}\ x)\ \widehat{d_1}]})) \\
&= \quad \wr \text{ Refold } \mathcal{S}_{\widehat{\mathsf{D}}}[\![\mathsf{Let}\ x\ e_1\ e_2]\!]_{\beta_{\mathbb{D}}(\mu) \lhd \rho} \ \wr \\
&\quad \mathcal{S}_{\widehat{\mathsf{D}}}[\![\mathsf{Let}\ x\ e_1\ e_2]\!]_{\beta_{\mathbb{D}}(\mu) \lhd \rho}
\end{aligned}
$$

□

4.6.10 Parametricity and Relationship to Kripke Logical Relations

I remarked right at the begin of the previous subsection that the Galois connection in Figure 4.24 is really a family of definitions indexed by a heap μ. It is not possible to regard the "abstraction of a d" in isolation; rather, Lemma 4.33 expresses that once an "abstraction of a d" holds for a particular heap μ_1, this abstraction will hold for any heap μ_2 that the semantics may progress to.

Unfortunately, this indexing also means that I cannot apply parametricity to prove the abstract by-need interpretation Theorem 4.18, as I did for abstract by-name interpretation (Theorem 4.28). Such a proof would fail when the heap is extended (in *bind*), because then the index μ of the soundness relation

$$R_\mu(d, \widehat{d}) \triangleq \beta_{\mathbb{D}}(\mu)(d) \sqsubseteq \widehat{d}$$

for Theorem 4.18 must change as well. Concretely, we would roughly need the following free theorem

$$\frac{(\mathit{rhs}, \widehat{\mathit{rhs}}) \in R_{\mu[a \mapsto d]} \to R_{\mu[a \mapsto d]} \qquad (\mathit{body}, \widehat{\mathit{body}}) \in R_{\mu[a \mapsto d]} \to R_{\mu[a \mapsto d]}}{(\mathit{bind}\ \mathit{rhs}\ \mathit{body}, \widehat{\mathit{bind}}\ \widehat{\mathit{rhs}}\ \widehat{\mathit{body}}) \in R_\mu}$$

However, parametricity only yields

$$\frac{(\mathit{rhs}, \widehat{\mathit{rhs}}) \in R_\mu \to R_\mu \qquad (\mathit{body}, \widehat{\mathit{body}}) \in R_\mu \to R_\mu}{(\mathit{bind}\ \mathit{rhs}\ \mathit{body}, \widehat{\mathit{bind}}\ \widehat{\mathit{rhs}}\ \widehat{\mathit{body}}) \in R_\mu}$$

and it is impossible to prove $(\mathit{fetch}\ a, \mathit{fetch}\ a) \in R_\mu$ for the fresh a not bound in μ: the expression $\beta_{\mathbb{D}}(\mu)(\mathit{fetch}\ a)$ would not be defined.

But all is not lost: In the past 20 years the research community made immense progress in understanding so-called *Kripke logical relations* [Ahmed 2004], and I think that R_μ could well be used to define one.

To see the connection to Kripke logical relations (which I will explain momentarily), note that parametricity is essentially a procedure for constructing and proving *logical relations* by following polymorphic type structure. A logical relation is a relation that is preserved by evaluation [Nielson et al. 1999]. Parametricity lifts the logical relation R_μ on base values to arbitrary polymorphic System F types such as that of $\mathcal{S}[\![\_]\!]\_$. Furthermore, it provides a reusable proof (by induction on the *typing derivation*) that any System F term must preserve *any* logical relation constructed for its type. Reynolds [1983] called this property the "Abstraction Theorem"; contemporary work refers to it as a "Fundamental Property" of a logical relation.

When the object language System F accumulates features such as mutable state or recursive types and thus non-termination, it is necessary to *index* the logical relation with a step-index, or even the mutable heap μ in which the logical relation applies. This index describes a *possible world* of Kripke [1963], equipped with a preorder describing what worlds are *accessible* from the current world. In my work, the possible worlds are the definable heaps, and heap progression ($\rightsquigarrow$) is the accessibility relation. Lemma 4.33 states that the soundness relation R_μ is monotonic wrt. ($\rightsquigarrow$), so it should be possible to construct a Kripke logical relation for R_μ.

The construction of the Kripke logical relation would be different from that of parametricity. In particular, for the type of *bind* we would get the free theorem

$$\frac{\begin{array}{c}\forall \mu_1.\ \mu \rightsquigarrow \mu_1 \implies (\mathit{rhs}, \widehat{\mathit{rhs}}) \in R_{\mu_1} \to R_{\mu_1} \\ \forall \mu_2.\ \mu \rightsquigarrow \mu_2 \implies (\mathit{body}, \widehat{\mathit{body}}) \in R_{\mu_2} \to R_{\mu_2}\end{array}}{(\mathit{bind}\ \mathit{rhs}\ \mathit{body}, \widehat{\mathit{bind}}\ \widehat{\mathit{rhs}}\ \widehat{\mathit{body}}) \in R_\mu}$$

Notably, both ocurrences of μ_1 vary in tandem and admit an extended heap, so that $(\mathit{fetch}\ a, \mathit{lfp}\ \widehat{\mathit{rhs}}) \in R_{\mu[a \mapsto d]}$ can be proved. We instantiate the premises as

follows

$$\begin{array}{rcl} (\mathit{rhs}\ (\mathit{fetch}\ a), \widehat{\mathit{rhs}}\ (\mathit{lfp}\ \widehat{\mathit{rhs}})) & \in & R_{\mu[a \mapsto d]} \\ (\mathit{body}\ (\mathit{fetch}\ a), \widehat{\mathit{body}}\ (\mathit{lfp}\ \widehat{\mathit{rhs}})) & \in & R_{\mu[a \mapsto d]} \end{array}$$

and that is enough to show the goal $(\mathit{bind}\ \mathit{rhs}\ \mathit{body}, \widehat{\mathit{bind}}\ \widehat{\mathit{rhs}}\ \widehat{\mathit{body}}) \in R_\mu$.

I have not investigated this avenue any deeper, but it is a promising one to achieve a modular proof for Theorem 4.18. A modular proof would enable my proof framework to scale up to arbitrarily complex denotational interpreters, such as one for a by-need semantics of Haskell, for example. However, defining such a Kripke logical relation framework on pen and paper seems quite complex and I would first focus efforts on full mechanisation of denotational interpreters in an interactive theorem prover; the work of Sterling et al. [2023] and Aagaard et al. [2023] seems promising in that regard.

4.7 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]. I improve on Josephs's work in that my encoding is simpler, rigorously defined (Section 4.4.2) and proven adequate wrt. Sestoft's by-need semantics (Section 4.4.1). The clairvoyant semantics of Hackett and Hutton [2019] is a denotational cost semantics for call-by-need, but unfortunately I fail to see how their approach can be extended to totally generate detailed by-need small-step traces, cf. Section 4.3.3.

Sestoft [1997] related the derivations of Launchbury's big-step natural semantics for the language in Section 4.1.1 to the subset of *balanced* small-step LK traces. Balanced traces are a proper subset of maximal LK traces that — by nature of big-step semantics — excludes stuck and diverging traces.

My 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 my approach generates small-step traces, totally and adequately, observable by abstract interpreters.

Definitional Interpreters

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

Ager et al. [2004] pick up on Reynold's idea and successively transform a partial denotational interpreter into a variant of the LK machine, going the reverse route of Section 4.4.1.

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 I did for a denotational semantics. The work of Atkey and McBride [2013] and Møgelberg and Veltri [2019] had big influence on my use of the later modality and Löb induction.

The Trace type class is appropriate for tracking "pure" transition events, but it is not up to the task of modelling user input, for example. A redesign of Trace inspired (and instantiated) by guarded interaction trees [Frumin et al. 2023; Xia et al. 2019] would help with that.

Contextual Improvement

Abstract interpretation is useful to prove that an analysis approximates the right trace property, but it does not help to prove an *optimisation* conditional on some trace property sound, yet alone an *improvement* [Moran and Sands 1999]. If I were to prove dead code elimination sound based on my notion of absence, would I use my denotational interpreter to do so? Probably not; I would try to conduct as much of the proof as possible in the equational theory, i.e. on syntax. If need be, I could always switch to denotational interpreters via Theorem 4.4, just as in Lemma 4.22. 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 me to attempt the same. However, while Cousot and Cousot [1994] and Cousot and Cousot [2002] work with denotational semantics for a higher-order language, it was unclear to me 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. My solution delegates this complexity to the underlying theory of guarded dependent type theory [Bizjak et al. 2016; Møgelberg and Veltri 2019].

I 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 I expect relational abstractions to work just as well; however it is questionable whether the proofs would still be able to appeal to parametricity. My generic denotational interpreter is a higher-order generalisation of the generic abstract interpreter in Cousot [2021, Chapter 21]. The abstraction laws in Figure 4.25 correspond to Definition 27.1 and Theorem 4.18 to Theorem 27.4.

Abstractions of Reachable States

CFA [Shivers 1991] computes a useful control-flow graph abstraction for higher-order programs, thus lifting classic intraprocedural analyses such as constant propagation to the interprocedural setting. The contour depth parameter k allows to trade precision for performance, although in practice it is often $k \leqslant 1$.

Montagu and Jensen [2021] derive CFA from small-step traces. I think that a variant of my denotational interpreter would be a good fit for their collecting semantics. Specifically, the semantic inclusions of Lemma 2.10 that govern the transition to a big-step style interpreter follow simply by adequacy of my interpreter, Theorem 4.4.

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. Darais, Labich, et al. [2017] and others apply the AAM recipe to big-step interpreters in the style of Reynolds.

Whenever AAM is involved, abstraction follows some monadic structure inherent to dynamic semantics [Darais, Labich, et al. 2017; Sergey, Devriese, et al. 2013]. In my work, this is apparent in the Domain (D τ) instance depending on Monad τ. Decomposing such structure into a layer of reusable monad transformers has been the subject of Darais, Might, et al. [2015] and Keidel

and Erdweg [2019]. The trace transformers of Section 4.3 enable reuse along a different dimension.

A big advantage of the big-step framework of Keidel, Poulsen, et al. [2018] is that soundness proofs are modular in the sense of Section 4.6.2. In the future, I hope to modularise the proof for Theorem 4.18 by defining a Kripke logical relation as outlined in Section 4.6.10.

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, Pnueli, 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; 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. I am however doubtful that the powerset-centric AAM approach could integrate summary mechanisms; the whole recipe rests on the fact that the set of expressions and thus evaluation contexts is finite.

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

Cardinality Analysis

More interesting cardinality analyses involve the inference of summaries called *demand transformers* [Sergey, Vytiniotis, et al. 2017], such as implemented in the Demand Analysis of the Glasgow Haskell Compiler. I intend to use my framework to describe improvements to Demand Analysis in the future. A soundness proof would require a slightly different Galois connection than Figure 4.24, because Demand Analysis is not sound wrt. by-name evaluation; a testament to its precision.

5

Conclusion and Future Work

In this thesis, I presented two projects that advance the state of the art in static program analysis of programs written in a functional language.

First, I described Lower Your Guards, a coverage checking algorithm that distills rich pattern-matching into simple guard trees. Guard trees are amenable to analyses that are not easily expressible in coverage checkers that work over structural pattern-matches.

The last four years of continued maintenance of GHC's implementation offer a compelling retrospective: the approach scales well to new language features, causes very few functional bug reports in practice, and offers robust performance. Its implementation fits nicely into one of GHC's many well-engineered architecture and should be applicable to its cousins in the ML tradition.

As outlined in the Introduction, the second project is the result of breaking free from a struggle for words by defining a new language. In my case, the language is that of denotational interpreters, in terms of which I hope to describe the many static higher-order analyses of GHC.

I showed that this language is indeed useful, because both standard dynamic semantics for functional programming languages as well as useful summary-based static analyses can be succinctly and comprehensibly expressed as denotational interpreters. Moreover, denotational interpreters lend themselves well to formalisation and soundness proofs by abstract interpretation.

In brief: Functional programming is an attractive paradigm for its succinctness, maintainability and tendency to prevent programmer errors, while static program analyses detect programmer errors and inform compiler optimisations. My work concerns the static analysis of functional programs: I presented (i) a static analysis for detecting incomplete pattern-matches, as well as (ii) the framework of *denotational interpreters* to formalise higher-order program analyses and prove them sound wrt. a dynamic semantics.

5.1 Future Work

My work on denotational interpreters opens up new research strands, and I shall conclude this thesis with a discussion.

Formalising GHC's Demand Analysis

The ultimate purpose of my work on denotational interpreters has always been to describe GHC's Demand Analysis, in order settle a research effort that has been ongoing for three decades [Sergey, Peyton Jones, et al. 2014]. However, after conceiving and properly describing denotational interpreters, I sadly find myself for a lack of time to close this parenthesis that I opened in Graf [2017].

There are a number of other analyses in GHC that could well be described as denotational interpreters. Examples include the boxity analysis (in the sense of Henglein and Jørgensen [1994]) I contributed to GHC, a tentative termination analysis and a forward arity analysis.

Machine-checked Formalisation

Although I conducted most proofs using pen-and-paper only, I tried to maintain as much formal rigour as possible in order to encode the proofs in an interactive theorem prover. It might seem baffling that I did not use Guarded Cubical Agda to formalise all proofs in the first place. The reason is that I found the ergonomics in need of improvement and the lack of proof tactics frustrating. In the future, I would like to try again using an axiomatisation of *impredicative Guarded Dependent Type Theory* [Sterling et al. 2023] in Lean 4 or Rocq, hoping that these languages will ultimately provide a first class executable treatment of guarded dependent type theory. The next step would be to define a parametric Kripke-style logical relation as outlined in Section 4.6.10 in order to derive a modular proof for Theorem 4.18, so that the necessary proofs of my framework scale effortlessly to denotational interpreters of large surface languages such as Haskell or WebAssembly.

Bibliography

Frederik Lerbjerg Aagaard, Jonathan Sterling, and Lars Birkedal. Nov. 2023. "A denotationally-based program logic for higher-order store". *Electronic Notes in Theoretical Informatics and Computer Science*, Volume 3 - Proceedings of MFPS XXXIX, (Nov. 2023). DOI: 10.46298/entics.12232.

Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. 2004. "A functional correspondence between call-by-need evaluators and lazy abstract machines". *Information Processing Letters*, 90, 5, 223–232. DOI: https://doi.org/10.1016/j.ipl.2004.02.012.

Amal Jamil Ahmed. 2004. "Semantics of types for mutable state". PhD thesis. USA. AAI3136691.

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

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

Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. 1995. "A Call-by-Need Lambda Calculus". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '95). Association for Computing Machinery, San Francisco, California, USA, 233–246. ISBN: 0897916921. DOI: 10.1145/199448.199507.

Robert Atkey and Conor McBride. 2013. "Productive coprogramming with guarded recursion". In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (ICFP '13). Association for Computing Machinery, Boston, Massachusetts, USA, 197–208. ISBN: 9781450323260. DOI: 10.1145/2500365.2500597.

Lennart Augustsson. 1985. "Compiling pattern matching". In: *Functional Programming Languages and Computer Architecture*. Ed. by Jean-Pierre Jouannaud. Springer Berlin Heidelberg, Berlin, Heidelberg, 368–381. ISBN: 978-3-540-39677-2.

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

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

Lars Birkedal and Rasmus Ejlers Mogelberg. 2013. "Intensional Type Theory with Guarded Recursive Types qua Fixed Points on Universes". In: *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science* (LICS '13). IEEE Computer Society, USA, 213–222. ISBN: 9780769550206. DOI: 10.1109/LICS.2013.27.

Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E. Møgelberg, and Lars Birkedal. 2016. "Guarded Dependent Type Theory with Coinductive Types". In: *Foundations of Software Science and Computation Structures*. Ed. by Bart Jacobs and Christof Löding. Springer Berlin Heidelberg, Berlin, Heidelberg, 20–35. ISBN: 978-3-662-49630-5.

Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. "Quantified class constraints". In: *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell* (Haskell 2017). Association for Computing Machinery, Oxford, UK, 148–161. ISBN: 9781450351829. DOI: 10.1145/3122955.3122967.

Katharina Brandl, Sebastian Erdweg, Sven Keidel, and Nils Hansen. 2023. "Modular Abstract Definitional Interpreters for WebAssembly". In: *37th European Conference on Object-Oriented Programming (ECOOP 2023)* (Leibniz International Proceedings in Informatics (LIPIcs)). Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:28. ISBN: 978-3-95977-281-5. DOI: 10.4230/LIPIcs.ECOOP.2023.5.

Joachim Breitner. Apr. 2016. "Lazy Evaluation: From natural semantics to a machine-checked compiler transformation". PhD thesis. Karlsruher Institut für Technologie, Fakultät für Informatik, (Apr. 2016). DOI: 10.5445/IR/1000054251.

Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. 2014. "Safe zero-cost coercions for Haskell". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (ICFP '14). Association for Computing Machinery, Gothenburg, Sweden, 189–202. ISBN: 9781450328739. DOI: 10.1145/2628136.2628141.

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

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2007. "Finally Tagless, Partially Evaluated". In: *Programming Languages and Systems*. Ed. by Zhong Shao. Springer Berlin Heidelberg, Berlin, Heidelberg, 222–238. ISBN: 978-3-540-76637-7.

Jesper Cockx and Andreas Abel. July 2018. "Elaborating Dependent (Co)Pattern Matching". *Proc. ACM Program. Lang.*, 2, ICFP, (July 2018). DOI: 10.1145/3236770.

Thierry Coquand. 1994. "Infinite objects in type theory". In: *Types for Proofs and Programs*. Ed. by Henk Barendregt and Tobias Nipkow. Springer Berlin Heidelberg, Berlin, Heidelberg, 62–78. ISBN: 978-3-540-48440-0.

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

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

Patrick Cousot and Radhia Cousot. 1977. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (POPL '77). Association for Computing Machinery, Los Angeles, California, 238–252. ISBN: 9781450373500. DOI: 10.1145/512950.512973.

Patrick Cousot and Radhia Cousot. 2002. "Modular Static Program Analysis". In: *Compiler Construction*. Ed. by R. Nigel Horspool. Springer Berlin Heidelberg, Berlin, Heidelberg, 159–179. ISBN: 978-3-540-45937-8.

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

David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. Aug. 2017. "Abstracting Definitional Interpreters (Functional Pearl)". *Proc. ACM Program. Lang.*, 1, ICFP, (Aug. 2017). DOI: 10.1145/3110256.

David Darais, Matthew Might, and David Van Horn. 2015. “Galois transformers and modular abstract interpreters: reusable metatheory for program analysis”. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA 2015). Association for Computing Machinery, Pittsburgh, PA, USA, 552–571. ISBN: 9781450336895. DOI: 10.1145/2814270.2814308.

Henning Dieterichs. Apr. 2021. *Formal Verification of Pattern Matching Analyses*. Supervised by Sebastian Graf and Sebastian Ullrich. (Apr. 2021).

Joshua Dunfield. Aug. 2007. “A Unified System of Type Refinements”. PhD thesis. Carnegie Mellon University, (Aug. 2007). CMU-CS-07-129.

Richard A. Eisenberg and Jan Stolarek. 2014. “Promoting Functions to Type Families in Haskell”. In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell* (Haskell ’14). Association for Computing Machinery, Gothenburg, Sweden, 95–106. ISBN: 9781450330411. DOI: 10.1145/2633357.2633361.

Richard A. Eisenberg and Stephanie Weirich. 2012. “Dependently Typed Programming with Singletons”. In: *Proceedings of the 2012 Haskell Symposium* (Haskell ’12). ACM, Copenhagen, Denmark, 117–130. DOI: 10.1145/2364506.2364522.

David Eppstein. 1992. “Parallel recognition of series-parallel graphs”. *Information and Computation*, 98, 1, 41–55. DOI: https://doi.org/10.1016/0890-5401(92)90041-D.

Mattias Felleisen and D. P. Friedman. 1987. “A Calculus for Assignments in Higher-Order Languages”. In: *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (POPL ’87). Association for Computing Machinery, Munich, West Germany, 314. ISBN: 0897912152. DOI: 10.1145/41625.41654.

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

Jacques Garrigue and Jacques Le Normand. 2011. “Adding GADTs to OCaml: the direct approach”. In: *Workshop on ML*.

GHC issue. 2018a. *-Wincomplete-patterns gets confused when combining GADTs and pattern guards*. (2018). https://gitlab.haskell.org/ghc/ghc/issues/15385.

GHC issue. 2020a. *-Wincomplete-record-updates ignores context*. (2020). https://gitlab.haskell.org/ghc/ghc/issues/17783.

GHC issue. 2017a. *-Woverlapping-patterns warns on wrong patterns for Int*. (2017). https://gitlab.haskell.org/ghc/ghc/issues/14546.

GHC issue. 2019a. *`case (x :: Void) of _ -> ()` should be flagged as redundant*. (2019). https://gitlab.haskell.org/ghc/ghc/issues/17376.

GHC issue. 2018b. *“Pattern match has inaccessible right hand side” with TypeRep*. (2018). https://gitlab.haskell.org/ghc/ghc/issues/14851.

GHC issue. 2019b. *67-pattern COMPLETE pragma overwhelms the pattern match checker*. (2019). https://gitlab.haskell.org/ghc/ghc/issues/17096.

GHC issue. 2019c. *Add Luke Maranget’s series in “Warnings for Pattern Matching”*. (2019). https://gitlab.haskell.org/ghc/ghc/issues/17264.

GHC issue. 2018c. *Bogus -Woverlapping-patterns warning with OverloadedStrings*. (2018). https://gitlab.haskell.org/ghc/ghc/issues/15713.

GHC issue. 2018d. *Compiling a function with a lot of alternatives bottlenecks on insertIntHeap*. (2018). https://gitlab.haskell.org/ghc/ghc/issues/14667.

GHC issue. 2017b. *COMPLETE sets don’t work at all with data family instances*. (2017). https://gitlab.haskell.org/ghc/ghc/issues/14059.

GHC issue. 2017c. *COMPLETE sets nerf redundant pattern-match warnings*. (2017). https://gitlab.haskell.org/ghc/ghc/issues/13965.
GHC issue. 2018e. *Completeness of View Patterns With a Complete Set of Output Patterns*. (2018). https://gitlab.haskell.org/ghc/ghc/issues/15884.
GHC issue. 2018f. *EmptyCase thinks pattern match involving type family is not exhaustive, when it actually is*. (2018). https://gitlab.haskell.org/ghc/ghc/issues/14813.
GHC issue. 2018g. *Erroneous "non-exhaustive pattern match" using nested GADT with strictness annotation*. (2018). https://gitlab.haskell.org/ghc/ghc/issues/15305.
GHC issue. 2019d. *GHC thinks pattern match is exhaustive*. (2019). https://gitlab.haskell.org/ghc/ghc/issues/16289.
GHC issue. 2016a. *In a record-update construct:ghc-stage2: panic! (the 'impossible' happened)*. (2016). https://gitlab.haskell.org/ghc/ghc/issues/12957.
GHC issue. 2016b. *Inaccessible RHS warning is confusing for users*. (2016). https://gitlab.haskell.org/ghc/ghc/issues/13021.
GHC issue. 2018h. *Inconsistency w.r.t. coverage checking warnings for EmptyCase under unsatisfiable constraints*. (2018). https://gitlab.haskell.org/ghc/ghc/issues/15450.
GHC issue. 2018i. *Inconsistent pattern-match warnings when using guards versus case expressions*. (2018). https://gitlab.haskell.org/ghc/ghc/issues/15753.
GHC issue. 2019e. *Incorrect non-exhaustive pattern warning with PatternSynonyms*. (2019). https://gitlab.haskell.org/ghc/ghc/issues/16129.
GHC issue. 2017d. *Incorrect pattern match warning on nested GADTs*. (2017). https://gitlab.haskell.org/ghc/ghc/issues/14098.
GHC issue. 2019f. *Minimality of missing pattern set depends on constructor declaration order*. (2019). https://gitlab.haskell.org/ghc/ghc/issues/17386.
GHC issue. 2015a. *New pattern-match check can be non-performant*. (2015). https://gitlab.haskell.org/ghc/ghc/issues/11195.
GHC issue. 2015b. *No non-exhaustive pattern match warning given for empty case analysis*. (2015). https://gitlab.haskell.org/ghc/ghc/issues/10746.
GHC issue. 2018j. *nonVoid is too conservative w.r.t. strict argument types*. (2018). https://gitlab.haskell.org/ghc/ghc/issues/15584.
GHC issue. 2019g. *Panic during tyConAppArgs*. (2019). https://gitlab.haskell.org/ghc/ghc/issues/17112.
GHC issue. 2016c. *Pattern coverage checker ignores dictionary arguments*. (2016). https://gitlab.haskell.org/ghc/ghc/issues/12949.
GHC issue. 2017e. *Pattern match checker mistakenly concludes pattern match on pattern synonym is unreachable*. (2017). https://gitlab.haskell.org/ghc/ghc/issues/14253.
GHC issue. 2020b. *Pattern match checker stumbles over reasonably tricky pattern-match*. (2020). https://gitlab.haskell.org/ghc/ghc/issues/17703.
GHC issue. 2019h. *Pattern match checking open unions*. (2019). https://gitlab.haskell.org/ghc/ghc/issues/17149.
GHC issue. 2020c. *Pattern match coverage checker allocates twice as much for trivial program with instance constraint vs. without*. (2020). https://gitlab.haskell.org/ghc/ghc/issues/17891.
GHC issue. 2016d. *Pattern match incompleteness / inaccessibility discrepancy*. (2016). https://gitlab.haskell.org/ghc/ghc/issues/11984.
GHC issue. 2019i. *Pattern match overlap checking doesn't consider -XBangPatterns*. (2019). https://gitlab.haskell.org/ghc/ghc/issues/17234.
GHC issue. 2020d. *Pattern match warning emitted twice*. (2020). https://gitlab.haskell.org/ghc/ghc/issues/17646.

GHC issue. 2019j. *Pattern match warnings are per Match, not per GRHS*. (2019). https://gitlab.haskell.org/ghc/ghc/issues/17465.
GHC issue. 2017f. *Pattern synonym exhaustiveness checks don't play well with EmptyCase*. (2017). https://gitlab.haskell.org/ghc/ghc/issues/13717.
GHC issue. 2019k. *Pattern-match checker: True /= False*. (2019). https://gitlab.haskell.org/ghc/ghc/issues/17251.
GHC issue. 2019l. *PmCheck treats Newtype patterns the same as constructors*. (2019). https://gitlab.haskell.org/ghc/ghc/issues/17248.
GHC issue. 2016e. *Representation of value set abstractions as trees causes performance issues*. (2016). https://gitlab.haskell.org/ghc/ghc/issues/11528.
GHC issue. 2019m. *Strictness of pattern synonym matches and pattern-match checking*. (2019). https://gitlab.haskell.org/ghc/ghc/issues/17357.
GHC issue. 2017g. *Wildcard patterns and COMPLETE sets can lead to misleading redundant pattern-match warnings*. (2017). https://gitlab.haskell.org/ghc/ghc/issues/13363.
GHC team. 2020. *COMPLETE pragmas*. (2020). https://downloads.haskell.org/~ghc/8.8.3/docs/html/users_guide/glasgow_exts.html#pragma-COMPLETE.
Andy Gill and Graham Hutton. 2009. "The worker/wrapper transformation". *Journal of Functional Programming*, 19, 2, 227–251. DOI: 10.1017/S0956796809007175.
Sebastian Graf. Aug. 2017. *Call Arity vs. Demand Analysis*. (Aug. 2017).
Sebastian Graf, Simon Peyton Jones, and Sven Keidel. 2024. *Abstracting Denotational Interpreters*. (2024). arXiv: 2403.02778 [cs.PL].
Sebastian Graf, Simon Peyton Jones, and Ryan G. Scott. Aug. 2020. "Lower your guards: a compositional pattern-match coverage checker". *Proc. ACM Program. Lang.*, 4, ICFP, (Aug. 2020). DOI: 10.1145/3408989.
Jörgen Gustavsson. 1998. "A Type Based Sharing Analysis for Update Avoidance and Optimisation". In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (ICFP '98). Association for Computing Machinery, Baltimore, Maryland, USA, 39–50. ISBN: 1581130244. DOI: 10.1145/289423.289427.
Jennifer Hackett and Graham Hutton. July 2019. "Call-by-Need is Clairvoyant Call-by-Value". *Proc. ACM Program. Lang.*, 3, ICFP, (July 2019). DOI: 10.1145/3341718.
Fritz Henglein and Jesper Jørgensen. 1994. "Formally optimal boxing". In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '94). Association for Computing Machinery, Portland, Oregon, USA, 213–226. ISBN: 0897916360. DOI: 10.1145/174675.177874.
John Hughes, Lars Pareto, and Amr Sabry. 1996. "Proving the Correctness of Reactive Systems Using Sized Types". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '96). Association for Computing Machinery, St. Petersburg Beach, Florida, USA, 410–423. ISBN: 0897917693. DOI: 10.1145/237721.240882.
Thomas Johnsson. 1984. "Efficient Compilation of Lazy Evaluation". In: *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction* (SIGPLAN '84). Association for Computing Machinery, Montreal, Canada, 58–69. ISBN: 0897911393. DOI: 10.1145/502874.502880.
Mark B. Josephs. 1989. "The semantics of lazy functional languages". *Theoretical Computer Science*, 68, 1, 105–111. DOI: https://doi.org/10.1016/0304-3975(89)90122-9.
Pavel Kalvoda and Tom Sydney Kerckhove. 2019. *Structural and semantic pattern matching analysis in Haskell*. (2019). arXiv: 1909.04160 [cs.PL].

Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones. 2015. *GADTs meet their match (extended version).* Tech. rep. KU Leuven. https://people.cs.kuleuven.be/~tom.schrijvers/Research/papers/icfp2015.pdf.

Sven Keidel and Sebastian Erdweg. Oct. 2019. "Sound and reusable components for abstract interpretation". *Proc. ACM Program. Lang.*, 3, OOPSLA, (Oct. 2019). DOI: 10.1145/3360602.

Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. July 2018. "Compositional Soundness Proofs of Abstract Interpreters". *Proc. ACM Program. Lang.*, 2, ICFP, (July 2018). DOI: 10.1145/3236767.

Saul Kripke. 1963. "Semantical Considerations on Modal Logic". *Acta Philosophica Fennica*, 16, 83–94.

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

John Launchbury. 1993. "A Natural Semantics for Lazy Evaluation". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '93). Association for Computing Machinery, Charleston, South Carolina, USA, 144–154. ISBN: 0897915607. DOI: 10.1145/158511.158618.

John Launchbury and Simon Peyton Jones. 1994. "Lazy functional state threads". In: *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (PLDI '94). Association for Computing Machinery, Orlando, Florida, USA, 24–35. ISBN: 089791662X. DOI: 10.1145/178243.178246.

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

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

Ravi Mangal, Mayur Naik, and Hongseok Yang. 2014. "A Correspondence between Two Approaches to Interprocedural Analysis in the Presence of Join". In: *Programming Languages and Systems*. Ed. by Zhong Shao. Springer Berlin Heidelberg, Berlin, Heidelberg, 513–533. ISBN: 978-3-642-54833-8.

Luc Maranget. 2007. "Warnings for pattern matching". *Journal of Functional Programming*, 17, 387–421, 3.

Simon Marlow et al.. 2010. *Haskell 2010 Language Report.* https://web.archive.org/web/20100710201151/https://www.haskell.org/onlinereport/haskell2010/haskellch3.html#x8-230003.1. Accessed: 2010-07-10. (2010).

Conor McBride and Ross Paterson. 2008. "Applicative programming with effects". *Journal of Functional Programming*, 18, 1, 1–13. DOI: 10.1017/S0956796807006326.

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

Robin Milner. 1978. "A theory of type polymorphism in programming". *Journal of Computer and System Sciences*, 17, 3, 348–375. DOI: https://doi.org/10.1016/0022-0000(78)90014-4.

Rasmus Ejlers Møgelberg and Niccolò Veltri. Jan. 2019. "Bisimulation as Path Type for Guarded Recursive Types". *Proc. ACM Program. Lang.*, 3, POPL, (Jan. 2019). DOI: 10.1145/3290317.

Benoît Montagu and Thomas Jensen. 2021. "Trace-Based Control-Flow Analysis". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (PLDI 2021). Association for Computing Machinery, Virtual, Canada, 482–496. ISBN: 9781450383912. DOI: 10.1145/3453483.3454057.

Andrew Moran and David Sands. 1999. "Improvement in a Lazy Context: An Operational Theory for Call-by-Need". In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages (POPL '99). Association for Computing Machinery, San Antonio, Texas, USA, 43–56. ISBN: 1581130953. DOI: 10.1145/292540.292547.

Hiroshi Nakano. 2000. "A Modality for Recursion". In: *Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science* (LICS '00). IEEE Computer Society, USA, 255. ISBN: 0769507255.

Keiko Nakata. 2010. "Denotational Semantics for Lazy Initialization of letrec". In: *7th Workshop on Fixed Points in Computer Science, FICS 2010, Brno, Czech Republic, August 21-22, 2010*. Ed. by Luigi Santocanale. Laboratoire d'Informatique Fondamentale de Marseille, 61–67. https://hal.archives-ouvertes.fr/hal-00512377/document%5C#page=62.

Keiko Nakata and Jacques Garrigue. Sept. 2006. "Recursive Modules for Programming". *SIGPLAN Not.*, 41, 9, (Sept. 2006), 74–86. DOI: 10.1145/1160074.1159813.

Sebastian Nanz and Carlo A. Furia. 2015. "A Comparative Study of Programming Languages in Rosetta Code". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1, 778–788. DOI: 10.1109/ICSE.2015.90.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of program analysis*. Springer. ISBN: 978-3-540-65410-0. DOI: 10.1007/978-3-662-03811-6.

Ulf Norell. Sept. 2007. "Towards a practical programming language based on dependent type theory". PhD thesis. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, (Sept. 2007).

Nicolas Oury. 2007. "Pattern Matching Coverage Checking with Dependent Types Using Set Approximations". In: *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification* (PLPV '07). Association for Computing Machinery, Freiburg, Germany, 47–56. ISBN: 9781595936776. DOI: 10.1145/1292597.1292606.

Simon Peyton Jones. 1992. "Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine". *Journal of Functional Programming*. DOI: 10.1017/S0956796800000319.

Simon Peyton Jones and Sebastian Graf. 2023. *Triemaps that match*. (2023). arXiv: 2302.08775 [cs.PL].

Ed. by John T. O'Donnell and Kevin Hammond. "Measuring the effectiveness of a simple strictness analyser". *Functional Programming, Glasgow 1993: Proceedings of the 1993 Glasgow Workshop on Functional Programming, Ayr, Scotland, 5–7 July 1993*. Springer London, London, 201–221. ISBN: 978-1-4471-3236-3. DOI: 10.1007/978-1-4471-3236-3_17.

Simon Peyton Jones, Peter Sestoft, and John Hughes. July 2006. *Demand Analysis*. (July 2006). https://www.microsoft.com/en-us/research/publication/demand-analysis/.

F. Pfenning and C. Elliott. 1988. "Higher-order abstract syntax". In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (PLDI '88). Association for Computing Machinery, Atlanta, Georgia, USA, 199–208. ISBN: 0897912691. DOI: 10.1145/53990.54010.

Matthew Pickering, Gergő Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. "Pattern Synonyms". In: *Proceedings of the 9th International Symposium on Haskell* (Haskell 2016). Association for Computing Machinery, Nara, Japan, 80–91. ISBN: 9781450344340. DOI: 10.1145/2976002.2976013.

Benjamin C. Pierce. 2002. *Types and Programming Languages*. (1st ed.). The MIT Press. ISBN: 0262162091.

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

Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. Sept. 2017. "A large-scale study of programming languages and code quality in GitHub". *Commun. ACM*, 60, 10, (Sept. 2017), 91–100. DOI: 10.1145/3126905.

John C. Reynolds. 1972. "Definitional Interpreters for Higher-Order Programming Languages". In: *Proceedings of the ACM Annual Conference - Volume 2* (ACM '72). Association for Computing

Machinery, Boston, Massachusetts, USA, 717–740. ISBN: 9781450374927. DOI: 10.1145/800194.805852.

John C. Reynolds. 2002. "Separation logic: a logic for shared mutable data structures". In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 55–74. DOI: 10.1109/LICS.2002.1029817.

John C. Reynolds. 1983. "Types, Abstraction and Parametric Polymorphism". In: *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*. Ed. by R. E. A. Mason. North-Holland/IFIP, 513–523.

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

John Rushby, Sam Owre, and Natarajan Shankar. 1998. "Subtypes for specifications: Predicate subtyping in PVS". *IEEE Transactions on Software Engineering*, 24, 9, 709–720.

Dana Scott. Nov. 1970. *Outline of a Mathematical Theory of Computation*. Tech. rep. PRG02. OUCL, (Nov. 1970), 30.

Dana Scott and Christopher Strachey. Aug. 1971. *Toward a Mathematical Semantics for Computer Languages*. Tech. rep. PRG06. OUCL, (Aug. 1971), 49.

R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Dec. 1995. "Adaptive Pattern Matching". *SIAM J. Comput.*, 24, 6, (Dec. 1995), 1207–1234. DOI: 10.1137/S0097539793246252.

Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. 2013. "Monadic abstract interpreters". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '13). Association for Computing Machinery, Seattle, Washington, USA, 399–410. ISBN: 9781450320146. DOI: 10.1145/2491956.2491979.

Ilya Sergey, Simon Peyton Jones, and Dimitrios Vytiniotis. June 2014. "Theory and practice of demand analysis in Haskell". Unpublished draft. (June 2014). https://www.microsoft.com/en-us/research/publication/theory-practice-demand-analysis-haskell/.

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

Peter Sestoft. 1997. "Deriving a lazy abstract machine". *Journal of Functional Programming*, 7, 3, 231–264. DOI: 10.1017/S0956796897002712.

Peter Sestoft. 1996. "ML pattern match compilation and partial evaluation". In: *Partial Evaluation*. Springer, 446–464.

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

Olin Grigsby Shivers. May 1991. "Control-Flow Analysis of Higher-Order Languages or Taming Lambda". In: (May 1991).

Matthieu Sozeau. 2010. "Equations: A Dependent Pattern-Matching Compiler". In: *Interactive Theorem Proving*. Ed. by Matt Kaufmann and Lawrence C. Paulson. Springer Berlin Heidelberg, Berlin, Heidelberg, 419–434. ISBN: 978-3-642-14052-5.

Matthieu Sozeau and Cyprien Mangin. July 2019. "Equations Reloaded: High-Level Dependently-Typed Functional Programming and Proving in Coq". *Proc. ACM Program. Lang.*, 3, ICFP, (July 2019). DOI: 10.1145/3341690.

Simon Spies, Lennard Gäher, Daniel Gratzer, Joseph Tassarotti, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2021. "Transfinite Iris: Resolving an Existential Dilemma of Step-Indexed Separation Logic". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (PLDI 2021). Association for Computing Machinery, Virtual, Canada, 80–95. ISBN: 9781450383912. DOI: 10.1145/3453483.3454031.

Jonathan Sterling, Daniel Gratzer, and Lars Birkedal. 2023. *Denotational semantics of general store and polymorphism*. (2023). https://arxiv.org/abs/2210.02169 arXiv: 2210.02169 [cs.PL].

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. "System F with type equality coercions". In: *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (TLDI '07). Association for Computing Machinery, Nice, Nice, France, 53–66. ISBN: 159593393X. DOI: 10.1145/1190315.1190324.

Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. "Toward understanding compiler bugs in GCC and LLVM". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis* (ISSTA 2016). Association for Computing Machinery, Saarbrücken, Germany, 294–305. ISBN: 9781450343909. DOI: 10.1145/2931037.2931074.

David N. Turner, Philip Wadler, and Christian Mossin. 1995. "Once upon a type". In: *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture* (FPCA '95). Association for Computing Machinery, La Jolla, California, USA, 1–11. ISBN: 0897917197. DOI: 10.1145/224164.224168.

David Van Horn and Matthew Might. 2010. "Abstracting Abstract Machines". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (ICFP '10). Association for Computing Machinery, Baltimore, Maryland, USA, 51–62. ISBN: 9781605587943. DOI: 10.1145/1863543.1863553.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. 2014. "Refinement Types for Haskell". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (ICFP '14). ACM, Gothenburg, Sweden, 269–282. DOI: 10.1145/2628136.2628161.

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Dec. 2017. "Refinement Reflection: Complete Verification with SMT". *Proc. ACM Program. Lang.*, 2, POPL, (Dec. 2017). DOI: 10.1145/3158141.

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. Sept. 2011. "OutsideIn(X): Modular Type Inference with Local Assumptions". *J. Funct. Program.*, 21, 4-5, (Sept. 2011), 333–412. DOI: 10.1017/S0956796811000098.

Hongwei Xi. 1998a. "Dead Code Elimination Through Dependent Types". In: *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages* (PADL '99). Springer-Verlag, London, UK, 228–242.

Hongwei Xi. Sept. 1998b. "Dependent Types in Practical Programming". PhD thesis. Carnegie Mellon University, (Sept. 1998).

Hongwei Xi. 2003. "Dependently typed pattern matching". *Journal of Universal Computer Science*, 9, 851–872.

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. "Guarded Recursive Datatype Constructors". In: *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (POPL '03). ACM, New Orleans, Louisiana, USA, 224–235. DOI: 10.1145/604131.604150.

Hongwei Xi and Frank Pfenning. 1998. "Eliminating Array Bound Checking through Dependent Types". In: *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (PLDI '98). Association for Computing Machinery, Montreal, Quebec, Canada, 249–257. ISBN: 0897919874. DOI: 10.1145/277650.277732.

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Dec. 2019. "Interaction Trees: Representing Recursive and Impure Programs in Coq". *Proc. ACM Program. Lang.*, 4, POPL, (Dec. 2019). DOI: 10.1145/3371119.

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. "Giving Haskell a Promotion". In: *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (TLDI '12). ACM, Philadelphia, Pennsylvania, USA, 53–66. DOI: 10.1145/2103786.2103795.

A

Proofs for Chapter 4

A.1 Proofs for Section 4.1

Definition 4.2 (Absence)**.** *A variable* x *is* used *in an expression* e *if and only if there exists a trace* $(\mathbf{let}\ \mathsf{x} = \mathsf{e}'\ \mathbf{in}\ \mathsf{e}, \rho, \mu, \kappa) \hookrightarrow^* \ldots \xrightarrow{\mathit{Look}(\mathsf{x})} \ldots$ *that looks up the heap entry of* x*, i.e. it evaluates* x*. Otherwise,* x *is* absent *in* e.

Note that for the proofs I assume the recursive let definition

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

The partial order on AbsTy necessary for computing the least fixpoint lfp follows structurally from $\mathsf{A} \sqsubset \mathsf{U}$, as described in Section 4.1.

Abbreviation A.1. *The syntax* $\theta.\varphi$ *for an* AbsTy $\theta = \langle \varphi, \pi \rangle$ *returns the* φ *component of* θ*. The syntax* $\theta.\pi$ *returns the* π *component of* θ*.*

Definition A.2 (Abstract substitution)**.** *The operation*

$$\varphi[\mathsf{x} \Rrightarrow \varphi'] \triangleq \varphi[\mathsf{x} \mapsto \mathsf{A}] \sqcup (\varphi(\mathsf{x}) * \varphi')$$

is called the abstract substitution *operation on* Uses *and it is overloaded for* AbsTy*, so that* $(\langle \varphi, \pi \rangle)[\mathsf{x} \Rrightarrow \varphi_\mathsf{y}] \triangleq \langle \varphi[\mathsf{x} \Rrightarrow \varphi_\mathsf{y}], \pi \rangle$.

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

Lemma A.3. $\mathcal{A}[\![(\bar{\lambda}\mathsf{x}.\mathsf{e})\ \mathsf{y}]\!]_\rho = (\mathcal{A}[\![\mathsf{e}]\!]_{\rho[\mathsf{x}\mapsto\langle[\mathsf{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]})[\mathsf{x} \Rrightarrow \rho(\mathsf{y}).\varphi]$.

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

Lemma A.4. *Lambda-bound uses do not escape their scope. That is, when* x *is lambda-bound in* e*, it is* $(\mathcal{A}[\![\mathsf{e}]\!]_\rho).\varphi(\mathsf{x}) = \mathsf{A}$.

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

Lemma A.5. $\mathcal{A}[\![(\bar{\lambda}\mathsf{x}.\bar{\lambda}\mathsf{y}.\mathsf{e})\ \mathsf{z}]\!]_\rho = \mathcal{A}[\![\bar{\lambda}\mathsf{y}.((\bar{\lambda}\mathsf{x}.\mathsf{e})\ \mathsf{z})]\!]_\rho$.

Proof.
$$
\begin{aligned}
&\mathcal{A}[\![(\bar{\lambda}\mathsf{x}.\bar{\lambda}\mathsf{y}.\mathsf{e})\ \mathsf{z}]\!]_\rho \\
&\quad \wr \text{ Unfold } \mathcal{A}[\![\_]\!], \text{ Lemma A.3 } \int \\
&= (\mathit{fun}_{\mathsf{y}}(\lambda\theta_{\mathsf{y}}.\ \mathcal{A}[\![\mathsf{e}]\!]_{\rho[\mathsf{x}\mapsto\langle[\mathsf{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle,\mathsf{y}\mapsto\theta_{\mathsf{y}}]}))[\mathsf{x} \Longmapsto \rho(\mathsf{z}).\varphi] \\
&\quad \wr\ \rho(\mathsf{z})(\mathsf{y}) = \mathsf{A} \text{ by Lemma A.4, } \mathsf{x} \neq \mathsf{y} \neq \mathsf{z} \int \\
&= \mathit{fun}_{\mathsf{y}}(\lambda\theta_{\mathsf{y}}.\ (\mathcal{A}[\![\mathsf{e}]\!]_{\rho[\mathsf{x}\mapsto\langle[\mathsf{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle,\mathsf{y}\mapsto\theta_{\mathsf{y}}]})[\mathsf{x} \Longmapsto \rho(\mathsf{z}).\varphi]) \\
&\quad \wr \text{ Refold } \mathcal{A}[\![\_]\!] \int \\
&= \mathcal{A}[\![\bar{\lambda}\mathsf{y}.((\bar{\lambda}\mathsf{x}.\mathsf{e})\ \mathsf{z})]\!]_\rho
\end{aligned}
$$

□

Lemma A.6. $\mathcal{A}[\![(\bar{\lambda}\mathsf{x}.\mathsf{e})\ \mathsf{y}\ \mathsf{z}]\!]_\rho = \mathcal{A}[\![(\bar{\lambda}\mathsf{x}.\mathsf{e}\ \mathsf{z})\ \mathsf{y}]\!]_\rho$.

Proof.
$$
\begin{aligned}
&\mathcal{A}[\![(\bar{\lambda}\mathsf{x}.\mathsf{e})\ \mathsf{y}\ \mathsf{z}]\!]_\rho \\
&\quad \wr \text{ Unfold } \mathcal{A}[\![\_]\!], \text{ Lemma A.3 } \int \\
&= \mathit{app}((\mathcal{A}[\![\mathsf{e}]\!]_{\rho[\langle[\mathsf{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]})[\mathsf{x} \Longmapsto \rho(\mathsf{y}).\varphi])(\rho(\mathsf{z})) \\
&\quad \wr\ \rho(\mathsf{z})(\mathsf{x}) = \mathsf{A} \text{ by Lemma A.4, } \mathsf{y} \neq \mathsf{x} \neq \mathsf{z} \int \\
&= \mathit{app}(\mathcal{A}[\![\mathsf{e}]\!]_{\rho[\langle[\mathsf{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]})(\rho(\mathsf{z}))[\mathsf{x} \Longmapsto \rho(\mathsf{y}).\varphi] \\
&\quad \wr \text{ Refold } \mathcal{A}[\![\_]\!] \int \\
&= \mathcal{A}[\![(\bar{\lambda}\mathsf{x}.\mathsf{e}\ \mathsf{z})\ \mathsf{y}]\!]_\rho
\end{aligned}
$$

□

Lemma A.7. $\mathcal{A}[\![\textbf{let } z = (\bar{\lambda}x.e_1)\ y \textbf{ in } (\bar{\lambda}x.e_2)\ y]\!]_\rho = \mathcal{A}[\![(\bar{\lambda}x.\textbf{let } z = e_1 \textbf{ in } e_2)\ y]\!]_\rho$.

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

$$
\begin{aligned}
&\mathcal{A}[\![\textbf{let } z = (\bar{\lambda}x.e_1)\ y \textbf{ in } (\bar{\lambda}x.e_2)\ y]\!]_\rho \\
&\quad \wr \text{ Unfold } \mathcal{A}[\![\_]\!] \text{ } \wr \\
&= \mathcal{A}[\![(\bar{\lambda}x.e_2)\ y]\!]_{\rho[z \mapsto \mathsf{lfp}(\lambda\theta.\ z \& \mathcal{A}[\![(\bar{\lambda}x.e_1)\ y]\!]_{\rho[z \mapsto \theta]})]} \\
&\quad \wr \text{ Lemma A.3 } \wr \\
&= (\mathcal{A}[\![e_2]\!]_{\rho[x \mapsto \langle [x \mapsto U], \mathsf{Rep}\ U \rangle, z \mapsto \theta_z]})[x \Longmapsto \rho(y).\varphi] \\
&\quad \text{where } \theta_z = \mathsf{lfp}(\lambda\theta.\ z \ \&\ (\mathcal{A}[\![e_1]\!]_{\rho[x \mapsto \langle [x \mapsto U], \mathsf{Rep}\ U \rangle, z \mapsto \theta]})[x \Longmapsto \rho(y).\varphi]) \\
&\quad \wr \text{ Hand-waving above } \wr \\
&= (\mathcal{A}[\![e_2]\!]_{\rho[x \mapsto \langle [x \mapsto U], \mathsf{Rep}\ U \rangle, z \mapsto \theta'_z]})[x \Longmapsto \rho(y).\varphi] \\
&\quad \text{where } \theta'_z = \mathsf{lfp}(\lambda\theta.\ z \ \&\ \mathcal{A}[\![e_1]\!]_{\rho[x \mapsto \langle [x \mapsto U], \mathsf{Rep}\ U \rangle, z \mapsto \theta]}) \\
&\quad \wr \text{ Refold } \mathcal{A}[\![\_]\!] \text{ } \wr \\
&= (\mathcal{A}[\![\textbf{let } z = e_1 \textbf{ in } e_2]\!]_{\rho[x \mapsto \langle [x \mapsto U], \mathsf{Rep}\ U \rangle]})[x \Longmapsto \rho(y).\varphi] \\
&\quad \wr \text{ Lemma A.3 } \wr \\
&= \mathcal{A}[\![(\bar{\lambda}x.\textbf{let } z = e_1 \textbf{ in } e_2)\ y]\!]_\rho
\end{aligned}
$$

□

Lemma 4.3 (Substitution). $\mathcal{A}[\![e]\!]_{\rho[x \mapsto \rho(y)]} \sqsubseteq \mathcal{A}[\![(\bar{\lambda}x.e)\ y]\!]_\rho$. ↺ 76

Proof. By induction on e.

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

$$
\begin{aligned}
&\mathcal{A}[\![z]\!]_{\rho[x \mapsto \rho(y)]} \\
&\quad \wr\ x \neq z\ \wr \\
&= \rho(z) \\
&\quad \wr \text{ Refold } \mathcal{A}[\![\_]\!] \text{ } \wr
\end{aligned}
$$

$$
\begin{aligned}
&= \mathcal{A}[\![\mathsf{z}]\!]_{\rho[\mathsf{x}\mapsto\langle[\mathsf{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]} \\
&\quad \wr\ \rho(\mathsf{z}).\varphi(\mathsf{x}) = \mathsf{A}\ \int \\
&= (\mathcal{A}[\![\mathsf{z}]\!]_{\rho[\mathsf{x}\mapsto\langle[\mathsf{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]})[\mathsf{x} \Rrightarrow \rho(\mathsf{y}).\varphi] \\
&\quad \wr\ \text{Lemma A.3}\ \int \\
&= \mathcal{A}[\![(\bar{\lambda}\mathsf{x}.\mathsf{z})\ \mathsf{y}]\!]_{\rho}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathcal{A}[\![\mathsf{z}]\!]_{\rho[\mathsf{x}\mapsto\rho(\mathsf{y})]} \\
&\quad \wr\ \mathsf{x} = \mathsf{z}\ \int \\
&= \rho(\mathsf{y}) \\
&\quad \wr\ \pi \sqsubseteq \mathsf{Rep}\ \mathsf{U}\ \int \\
&\sqsubseteq \langle\rho(\mathsf{y}).\varphi, \mathsf{Rep}\ \mathsf{U}\rangle \\
&\quad \wr\ \text{Definition of } \_[\_ \Rrightarrow \_]\ \int \\
&= (\langle[\mathsf{x} \mapsto \mathsf{U}], \mathsf{Rep}\ \mathsf{U}\rangle)[\mathsf{x} \mapsto \rho(\mathsf{y}).\varphi] \\
&\quad \wr\ \text{Refold } \mathcal{A}[\![\_]\!]\ \int \\
&= (\mathcal{A}[\![\mathsf{z}]\!]_{\rho[\mathsf{x}\mapsto\langle[\mathsf{x}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]})[\mathsf{x} \Rrightarrow \rho(\mathsf{y}).\varphi] \\
&\quad \wr\ \text{Lemma A.3}\ \int \\
&= \mathcal{A}[\![(\bar{\lambda}\mathsf{x}.\mathsf{z})\ \mathsf{y}]\!]_{\rho}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathcal{A}[\![\bar{\lambda}\mathsf{z}.\mathsf{e}']\!]_{\rho[\mathsf{x}\mapsto\rho(\mathsf{y})]} \\
&\quad \wr\ \text{Unfold } \mathcal{A}[\![\_]\!]\ \int \\
&= \mathit{fun}_{\mathsf{z}}(\lambda\theta_{\mathsf{z}}.\ \mathcal{A}[\![\mathsf{e}']\!]_{\rho[\mathsf{z}\mapsto\theta_{\mathsf{z}},\mathsf{x}\mapsto\rho(\mathsf{y})]}) \\
&\quad \wr\ \text{Induction hypothesis, monotonicity}\ \int \\
&\sqsubseteq \mathit{fun}_{\mathsf{z}}(\lambda\theta_{\mathsf{z}}.\ \mathcal{A}[\![(\bar{\lambda}\mathsf{x}.\mathsf{e}')\ \mathsf{y}]\!]_{\rho[\mathsf{z}\mapsto\theta_{\mathsf{z}}]}) \\
&\quad \wr\ \text{Refold } \mathcal{A}[\![\_]\!]\ \int \\
&= \mathcal{A}[\![\bar{\lambda}\mathsf{z}.((\bar{\lambda}\mathsf{x}.\mathsf{e}')\ \mathsf{y})]\!]_{\rho} \\
&\quad \wr\ \text{Lemma A.5}\ \int \\
&= \mathcal{A}[\![(\bar{\lambda}\mathsf{x}.\bar{\lambda}\mathsf{z}.\mathsf{e}')\ \mathsf{y}]\!]_{\rho}
\end{aligned}
$$

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

$$\mathcal{A}[\![\mathsf{e}'\ \mathsf{z}]\!]_{\rho[\mathsf{x}\mapsto\rho(\mathsf{y})]}$$

$$
\begin{aligned}
&\wr \text{ Unfold } \mathcal{A}[\![\_]\!] \int \\
&= \mathit{app}(\mathcal{A}[\![\mathrm{e}']\!]_{\rho[\mathrm{x}\mapsto\rho(\mathrm{y})]})(\rho(\mathrm{y})) \\
&\wr \text{ Induction hypothesis, monotonicity } \int \\
&\sqsubseteq \mathit{app}(\mathcal{A}[\![(\bar{\lambda}\mathrm{x}.\mathrm{e}')\ \mathrm{y}]\!]_{\rho})(\rho(\mathrm{y})) \\
&\wr \text{ Refold } \mathcal{A}[\![\_]\!] \int \\
&= \mathcal{A}[\![(\bar{\lambda}\mathrm{x}.\mathrm{e}')\ \mathrm{y}\ \mathrm{y}]\!]_{\rho} \\
&\wr \text{ Lemma A.6 } \int \\
&= \mathcal{A}[\![(\bar{\lambda}\mathrm{x}.\mathrm{e}'\ \mathrm{y})\ \mathrm{y}]\!]_{\rho} \\
&\wr \text{ Compositionality in } (\bar{\lambda}\mathrm{x}.\mathrm{e}'\ \square)\ \mathrm{y} \int \\
&= \mathcal{A}[\![(\bar{\lambda}\mathrm{x}.\mathrm{e}'\ \mathrm{x})\ \mathrm{y}]\!]_{\rho} \\
&\wr \ \mathrm{x} = \mathrm{z} \int \\
&= \mathcal{A}[\![(\bar{\lambda}\mathrm{x}.\mathrm{e}'\ \mathrm{z})\ \mathrm{y}]\!]_{\rho}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathcal{A}[\![\mathrm{e}'\ \mathrm{z}]\!]_{\rho[\mathrm{x}\mapsto\rho(\mathrm{y})]} \\
&\wr \text{ Unfold } \mathcal{A}[\![\_]\!] \int \\
&= \mathit{app}(\mathcal{A}[\![\mathrm{e}']\!]_{\rho[\mathrm{x}\mapsto\rho(\mathrm{y})]})(\rho(\mathrm{z})) \\
&\wr \text{ Induction hypothesis, monotonicity } \int \\
&\sqsubseteq \mathit{app}(\mathcal{A}[\![(\bar{\lambda}\mathrm{x}.\mathrm{e}')\ \mathrm{y}]\!]_{\rho})(\rho(\mathrm{z})) \\
&\wr \text{ Refold } \mathcal{A}[\![\_]\!] \int \\
&= \mathcal{A}[\![(\bar{\lambda}\mathrm{x}.\mathrm{e}')\ \mathrm{y}\ \mathrm{z}]\!]_{\rho} \\
&\wr \text{ Lemma A.6 } \int \\
&= \mathcal{A}[\![(\bar{\lambda}\mathrm{x}.\mathrm{e}'\ \mathrm{z})\ \mathrm{y}]\!]_{\rho}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathcal{A}[\![\textbf{let } \mathrm{z} = \mathrm{e}_1 \textbf{ in } \mathrm{e}_2]\!]_{\rho[\mathrm{x}\mapsto\rho(\mathrm{y})]} \\
&\wr \text{ Unfold } \mathcal{A}[\![\_]\!] \int \\
&= \mathcal{A}[\![\mathrm{e}_2]\!]_{\rho[\mathrm{x}\mapsto\rho(\mathrm{y}),\mathrm{z}\mapsto\mathrm{lfp}(\lambda\theta.\ \mathrm{z}\&\mathcal{A}[\![\mathrm{e}_1]\!]_{\rho[\mathrm{x}\mapsto\rho(\mathrm{y}),\mathrm{z}\mapsto\theta]})]} \\
&\wr \text{ Induction hypothesis, monotonicity } \int \\
&\sqsubseteq \mathcal{A}[\![(\bar{\lambda}\mathrm{x}.\mathrm{e}_2)\ \mathrm{y}]\!]_{\rho[\mathrm{z}\mapsto\mathrm{lfp}(\lambda\theta.\ \mathrm{z}\&\mathcal{A}[\![(\bar{\lambda}\mathrm{x}.\mathrm{e}_1)\ \mathrm{y}]\!]_{\rho[\mathrm{z}\mapsto\theta]})]} \\
&\wr \text{ Refold } \mathcal{A}[\![\_]\!] \int \\
&= \mathcal{A}[\![\textbf{let } \mathrm{z} = (\bar{\lambda}\mathrm{x}.\mathrm{e}_1)\ \mathrm{y} \textbf{ in } (\bar{\lambda}\mathrm{x}.\mathrm{e}_2)\ \mathrm{y}]\!]_{\rho}
\end{aligned}
$$

$$
\begin{aligned}
&\wr \text{ Lemma A.7 } \wr \\
&= \mathcal{A}[\![(\bar{\lambda}\mathsf{x}.\textbf{let } \mathsf{z} = \mathsf{e}_1 \textbf{ in } \mathsf{e}_2)\ \mathsf{y}]\!]_\rho
\end{aligned}
$$

□

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

Lemma A.8 (Diagonal factoring)**.** *Let ρ and ρ_Δ be two environments such that* $\forall \mathsf{x}.\ \rho(\mathsf{x}).\pi = \rho_\Delta(\mathsf{x}).\pi$.

If $\rho_\Delta.\varphi(\mathsf{x}) \sqsubseteq \rho_\Delta.\varphi(\mathsf{y})$ if and only if $\mathsf{x} = \mathsf{y}$, then every instantiation of $\mathcal{A}[\![\mathsf{e}]\!]$ factors through $\mathcal{A}[\![\mathsf{e}]\!]_{\rho_\Delta}$, that is,

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

Proof. By induction on e.

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

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

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

$$
\begin{aligned}
&\mathcal{A}[\![\bar{\lambda}\mathsf{y}.\mathsf{e}']\!]_\rho \\
&\wr \text{ Unfold } \mathcal{A}[\![\_]\!] \wr
\end{aligned}
$$

$$
\begin{aligned}
&= \mathit{fun}_{\mathsf{y}}(\lambda\theta.\ \mathcal{A}[\![\mathsf{e}']\!]_{\rho[\mathsf{y}\mapsto\theta]}) \\
&\quad \wr \text{ Property of } \mathit{fun}_{\mathsf{y}} \int \\
&= \mathit{fun}_{\mathsf{y}}(\lambda\theta.\ (\mathcal{A}[\![\mathsf{e}']\!]_{\rho[\mathsf{y}\mapsto\langle[\mathsf{y}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]})) \\
&\quad \wr \text{ Induction hypothesis } \int \\
&= \mathit{fun}_{\mathsf{y}}(\lambda\theta.\ (\mathcal{A}[\![\mathsf{e}']\!]_{\rho_\Delta[\mathsf{y}\mapsto\langle[\mathsf{y}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]})[\overline{\mathsf{x} \Longmapsto \rho(\mathsf{x}).\varphi}, \mathsf{y} \Longmapsto [\mathsf{y} \mapsto \mathsf{U}]]) \\
&\quad \wr\ \theta[\mathsf{y} \Longmapsto [\mathsf{y} \mapsto \mathsf{U}]] = \theta \int \\
&= \mathit{fun}_{\mathsf{y}}(\lambda\theta.\ (\mathcal{A}[\![\mathsf{e}']\!]_{\rho_\Delta[\mathsf{y}\mapsto\langle[\mathsf{y}\mapsto\mathsf{U}],\mathsf{Rep}\ \mathsf{U}\rangle]})[\overline{\mathsf{x} \Longmapsto \rho(\mathsf{x}).\varphi}]) \\
&\quad \wr\ \theta[\mathsf{y} \Longmapsto [\mathsf{y} \mapsto \mathsf{U}]] = \theta \int \\
&= \mathit{fun}_{\mathsf{y}}(\lambda\theta.\ (\mathcal{A}[\![\mathsf{e}']\!]_{\rho_\Delta[\mathsf{y}\mapsto\theta]})[\overline{\mathsf{x} \Longmapsto \rho(\mathsf{x}).\varphi}]) \\
&\quad \wr \text{ Property of } \mathit{fun}_{\mathsf{y}} \int \\
&= \mathit{fun}_{\mathsf{y}}(\lambda\theta.\ \mathcal{A}[\![\mathsf{e}']\!]_{\rho_\Delta[\mathsf{y}\mapsto\theta]})[\overline{\mathsf{x} \Longmapsto \rho(\mathsf{x}).\varphi}] \\
&\quad \wr \text{ Refold } \mathcal{A}[\![\_]\!] \int \\
&= (\mathcal{A}[\![\bar{\lambda}\mathsf{y}.\mathsf{e}']\!]_{\rho_\Delta})[\overline{\mathsf{x} \Longmapsto \rho(\mathsf{x}).\varphi}]
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathcal{A}[\![\textbf{let } \mathsf{y} = \mathsf{e}_1 \textbf{ in } \mathsf{e}_2]\!]_\rho \\
&\quad \wr \text{ Unfold } \mathcal{A}[\![\_]\!] \int \\
&= \mathcal{A}[\![\mathsf{e}_2]\!]_{\rho[\mathsf{y}\mapsto\mathsf{lfp}(\lambda\theta.\ \mathsf{y}\&\mathcal{A}[\![\mathsf{e}_1]\!]_{\rho[\mathsf{y}\mapsto\theta]})]} \\
&\quad \wr \text{ Induction hypothesis } \int \\
&= \mathcal{A}[\![\mathsf{e}_2]\!]_{\rho[\mathsf{y}\mapsto\mathsf{lfp}(\lambda\theta.\ \mathsf{y}\&(\mathcal{A}[\![\mathsf{e}_1]\!]_{\rho_\Delta[\mathsf{y}\mapsto\langle[\mathsf{y}\mapsto\mathsf{U}],\theta.\pi\rangle]})[\overline{\mathsf{x}\Longmapsto\rho(\mathsf{x}).\varphi},\mathsf{y}\Longmapsto\theta.\varphi])]} \\
&\quad \wr \text{ Again, backwards } \int \\
&= \mathcal{A}[\![\mathsf{e}_2]\!]_{\rho[\mathsf{y}\mapsto\mathsf{lfp}(\lambda\theta.\ \mathsf{y}\&(\mathcal{A}[\![\mathsf{e}_1]\!]_{\rho_\Delta[\mathsf{y}\mapsto\theta]})[\overline{\mathsf{x}\Longmapsto\rho(\mathsf{x}).\varphi}])]} \\
&\quad \wr \text{ Similarly for } \mathsf{e}_2\text{, push out the subst as in Lemma A.7 } \int \\
&= (\mathcal{A}[\![\mathsf{e}_2]\!]_{\rho_\Delta[\mathsf{y}\mapsto\mathsf{lfp}(\lambda\theta.\ \mathsf{y}\&\mathcal{A}[\![\mathsf{e}_1]\!]_{\rho_\Delta[\mathsf{y}\mapsto\theta]})]})[\overline{\mathsf{x} \Longmapsto \rho(\mathsf{x}).\varphi}] \\
&\quad \wr \text{ Refold } \mathcal{A}[\![\_]\!] \int \\
&= (\mathcal{A}[\![\textbf{let } \mathsf{y} = \mathsf{e}_1 \textbf{ in } \mathsf{e}_2]\!]_{\rho_\Delta})[\overline{\mathsf{x} \Longmapsto \rho(\mathsf{x}).\varphi}]
\end{aligned}
$$

□

$$\boxed{C[\![\_]\!] : \mathbb{S} \to \mathsf{AbsTy}}$$

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

Fig. A.1: Absence analysis extended to small-step configurations

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

In Figure A.1, I give the extension of $C[\![\_]\!]$ to whole machine configurations σ. Although $C[\![\_]\!]$ looks like an entirely new definition, it is actually derivative of $\mathcal{A}[\![\_]\!]$ via a context lemma à la Moran and Sands [1999, Lemma 3.2]: The environments ρ simply govern the transition from syntax to operational representation in the heap. The bindings in the heap are to be treated as mutually recursive let bindings, hence a fixpoint is needed. For safety properties such as absence, a least fixpoint is appropriate. Apply frames on the stack correspond to the application case of $\mathcal{A}[\![\_]\!]$ and invoke the summary mechanism. Update frames are ignored because my analysis is not heap-sensitive.

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

Lemma A.9 (Preservation of $C[\![\_]\!]$)**.** *If $\sigma_1 \hookrightarrow \sigma_2$, then $C[\![\sigma_1]\!] \sqsupseteq C[\![\sigma_2]\!]$.*

Proof. By cases on the transition.

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

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

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

  $$\begin{aligned}
  & C[\![\sigma_1]\!] \\
  & \quad \wr\ \text{Unfold } C[\![\sigma_1]\!]\ \wr \\
  &= \mathit{apps}_\mu(\kappa)(\mathcal{A}[\![\mathbf{let}\ \mathsf{y} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2]\!]_{\alpha(\mu)\circ\rho}) \\
  & \quad \wr\ \text{Unfold } \mathcal{A}[\![\mathbf{let}\ \mathsf{y} = \mathsf{e}_1\ \mathbf{in}\ \mathsf{e}_2]\!]\ \wr
  \end{aligned}$$

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

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

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

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

$$
\begin{aligned}
&C[\![\sigma_1]\!] \\
&\quad \wr \text{ Unfold } C[\![\sigma_1]\!] \int \\
&= \mathit{apps}_{\mu}(\mathbf{ap}(\mathsf{a}) \cdot \kappa)(\mathcal{A}[\![\bar{\lambda}\mathsf{y}.\mathsf{e}']\!]_{\alpha(\mu)\circ\rho}) \\
&\quad \wr \text{ Unfold } \mathit{apps} \int \\
&= \mathit{apps}_{\mu}(\kappa)(\mathit{app}(\mathcal{A}[\![\bar{\lambda}\mathsf{y}.\mathsf{e}']\!]_{\alpha(\mu)\circ\rho}, \alpha(\mu)(\mathsf{a}))) \\
&\quad \wr \text{ Unfold RHS of Lemma 4.3 } \int \\
&\sqsupseteq \mathit{apps}_{\mu}(\kappa)(\mathcal{A}[\![\mathsf{e}']\!]_{(\alpha(\mu)\circ\rho)[\mathsf{y}\mapsto\alpha(\mu)(\mathsf{a})]}) \\
&\quad \wr \text{ Rearrange } \int \\
&= \mathit{apps}_{\mu}(\kappa)(\mathcal{A}[\![\mathsf{e}']\!]_{(\alpha(\mu)\circ\rho[\mathsf{y}\mapsto\mathsf{a}])}) \\
&\quad \wr \text{ Refold } C[\![\sigma_2]\!] \int \\
&= C[\![\sigma_2]\!]
\end{aligned}
$$

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

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

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

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

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

  Intuitively, this is somewhat clear, because μ "evaluates to" μ' and v is the value of e', in the sense that there exists $\sigma' = (\mathsf{e}', \rho', \mu', \kappa)$ such that $\sigma' \hookrightarrow^* \sigma_1 \hookrightarrow \sigma_2$.

  Alas, who guarantees that such a σ' actually exists? We would need to rearrange the lemma for that and argue by step indexing (a.k.a. coinduction) over prefixes of *maximal traces* (to be rigorously defined later). That is, we presume that the statement

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

has been proved for all $n < k$ and proceed to prove it for $n = k$. So we presume $\sigma_0 \hookrightarrow^{k-1} \sigma_1 \hookrightarrow \sigma_2$ and $C[\![\sigma_1]\!] \sqsubseteq C[\![\sigma_0]\!]$ to arrive at a similar setup as before, only with a stronger assumption about σ_1. Specifically, due to the balanced stack discipline we know that $\sigma_0 \hookrightarrow^{k-1} \sigma_1$ factors over σ' above. We may proceed by induction over the balanced stack discipline (we will see in Section 4.4.1 that this amounts to induction over the big-step derivation) of the trace $\sigma' \hookrightarrow^* \sigma_1$ to show Equation (A.1).

This reasoning was not specific to $\mathcal{A}[\![\_]\!]$ at all. I show a more general result in Theorem 4.18 that can be reused across many more analyses.

Assuming Equation (A.1) has been proved, we proceed

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

□

Theorem 4.1 ($\mathcal{A}[\![\_]\!]$ infers absence)**.** *If* $\mathcal{A}[\![\mathsf{e}]\!]_{\rho_\mathsf{e}} = \langle \varphi, \pi \rangle$ *and* $\varphi(\mathsf{x}) = \mathsf{A}$*, then* x *is absent in* e. ↻ 76

Proof. We show the contraposition, that is, if x is used in e, then $\varphi(\mathsf{x}) = \mathsf{U}$.

Since x is used in e, there exists a trace

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

where $\rho_1 \triangleq \rho[\mathsf{x} \mapsto \mathsf{a}]$, $\mu_1 \triangleq \mu[\mathsf{a} \mapsto (\mathsf{x}, \rho[\mathsf{x} \mapsto \mathsf{a}], \mathsf{e}')]$. Without loss of generality, we assume the trace prefix ends at the first lookup at a, so $\mu'(\mathsf{a}) = \mu_1(\mathsf{a}) = (\mathsf{x}, \rho_1, \mathsf{e}')$. 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, $\tilde{\rho}(y).\varphi(x) = U$ implies $x = y$ for all y, because $\mu_1(a)$ is the only heap entry in which x occurs by the shadowing assumptions on syntax. By unfolding $C[\![\_]\!]$ and $\mathcal{A}[\![y]\!]$ we can see that

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

By Lemma A.9, we also have

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

And with transitivity, we get $[x \mapsto U] \sqsubseteq (C[\![(e, \rho_1, \mu_1, \kappa)]\!]).\varphi$. Since there was no other heap entry for x and a cannot occur in κ or ρ due to well-addressedness, we have $[x \mapsto U] \sqsubseteq (C[\![(e, \rho_1, \mu_1, \kappa)]\!]).\varphi$ if and only if $[x \mapsto U] \sqsubseteq (\mathcal{A}[\![e]\!]_{\tilde{\rho}}).\varphi$. With Lemma A.8, we can decompose

$$\begin{aligned}
&[x \mapsto U] \\
&\quad \wr \text{ Above result } \int \\
&\sqsubseteq (\mathcal{A}[\![e]\!]_{\tilde{\rho}}).\varphi \\
&\quad \wr\ \tilde{\rho}_\Delta(x) \triangleq \langle [x \mapsto U], \tilde{\rho}(x).\pi \rangle, \text{Lemma A.8} \int \\
&= ((\mathcal{A}[\![e]\!]_{\tilde{\rho}_\Delta})[\overline{y \Rrightarrow \tilde{\rho}(y).\varphi}]).\varphi \\
&\quad \wr\ \pi \sqsubseteq \text{Rep } U, \text{ hence } \tilde{\rho}_\Delta \sqsubseteq \tilde{\rho}_e \int \\
&\sqsubseteq ((\mathcal{A}[\![e]\!]_{\tilde{\rho}_e})[\overline{y \Rrightarrow \tilde{\rho}(y).\varphi}]).\varphi \\
&\quad \wr \text{ Definition of } _[_ \Rrightarrow _] \int \\
&= \bigsqcup \{\tilde{\rho}(y).\varphi \mid \mathcal{A}[\![e]\!]_{\tilde{\rho}_e}.\varphi(y) = U\}
\end{aligned}$$

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

A.2 Proofs for Section 4.4

↺ 96 **Theorem 4.4** (Strong Adequacy). *Let e be a closed expression, $\tau \triangleq \mathcal{S}_{\mathbf{need}}[\![e]\!]_\varepsilon(\varepsilon)$ the denotational by-need trace and init(e) ↪ … the maximal LK trace. Then*

- *τ preserves the observable termination properties of init(e) ↪ … .*

- *τ preserves the length of $init(\mathrm{e}) \hookrightarrow \ldots$.*
- *every $ev :: \mathsf{Event}$ in $\tau = \overline{\mathsf{Step}\ ev \ldots}$ refers to a transition rule in $init(\mathrm{e}) \hookrightarrow \ldots$.*

Proof. We formally define $\alpha((\sigma_i)_{i\in\overline{n}}) \triangleq \alpha_{\mathbb{S}^\infty}((\sigma_i)_{i\in\overline{n}}, \mathbf{stop})$, where $\alpha_{\mathbb{S}^\infty}$ is defined in Figure 4.13.

Then $\mathcal{S}_{\mathbf{need}}[\![e]\!]_\varepsilon(\varepsilon) = \alpha(init(\mathrm{e}) \hookrightarrow \ldots)$ follows directly from Theorem 4.17. The function $\alpha_{\mathbb{E}_\mathrm{v}}$ in Figure 4.13 formalises the intuition in which LK transitions abstract into Events. Preservation of length is proved in Lemma 4.15 and preservation of the termination observable is proved in Lemma 4.16. □

Lemma 4.14 (Characterisation of maximal traces)**.** *An LK trace $(\sigma_i)_{i\in\overline{n}}$ is maximal if and only if it is balanced, diverging or stuck.* ↻ 104

Proof.

⇒: Let $(\sigma_i)_{i\in\overline{n}}$ be maximal. Then it is interior by definition. Thus, if $n = \omega$ is infinite, then it is diverging. Otherwise, $(\sigma_i)_{i\in\overline{n}}$ is finite. If it is *not* balanced, then it is still maximal and finite, hence stuck. Otherwise, it is balanced.

⇐: Let $(\sigma_i)_{i\in\overline{n}}$ be balanced, diverging or stuck.
If $(\sigma_i)_{i\in\overline{n}}$ is balanced, then it is interior, and σ_n is a reduction state with continuation $cont(\sigma_0)$. Now suppose there would exist σ_{n+1} such that $(\sigma_i)_{i\in\overline{n+1}}$ was interior. Then the transition $\sigma_n \hookrightarrow \sigma_{n+1}$ must be one of the *reduction transition rules* Upd, App$_2$ and Case$_2$, which are the only ones in which σ_n is a reduction state (i.e. $ctrl(\sigma_n)$ is a value v). But all reduction transitions leave $cont(\sigma_0)$, which is in contradiction to interiority of $(\sigma_i)_{i\in\overline{n+1}}$. Thus, $(\sigma_i)_{i\in\overline{n}}$ is maximal.
If $(\sigma_i)_{i\in\overline{n}}$ is diverging, it is interior and infinite, hence $n = \omega$. Indeed $(\sigma_i)_{i\in\overline{\omega}}$ is maximal, because the expression $\sigma_{\omega+1}$ is undefined and hence does not exist.
If $(\sigma_i)_{i\in\overline{n}}$ is stuck, then it is maximal by definition.

□

Lemma 4.16 (Abstraction preserves termination observable)**.** *Let $(\sigma_i)_{i\in\overline{n}}$ be a maximal trace. Then $\alpha_{\mathbb{S}^\infty}((\sigma_i)_{i\in\overline{n}}, cont(\sigma_0))$ ...* ↻ 108

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

- *ends in* Ret (Stuck, _) *if and only if* $(\sigma_i)_{i \in \overline{n}}$ *is stuck.*

Proof. The second point follows by a similar inductive argument as in Lemma 4.15.

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

A.3 Proofs for Section 4.6

A.3.1 Usage Analysis Proofs

Here I give the usage analysis proofs that play a tangential role in the main body.

Abbreviation A.10 (Field access)**.** $\langle \varphi', \nu' \rangle.\varphi \triangleq \varphi'$, $\langle \varphi', \nu' \rangle.\nu = \nu'$.

↻ 141 **Lemma 4.19** (Beta-App, Semantic substitution)**.** *Let* $x ::$ Name *be fresh,* $a ::$ $\mathsf{D_U}$ *and* $f ::$ (Trace d, Domain d, HasBind d) $\Rightarrow d \to d$*. Then* $f\ a \sqsubseteq$ *apply* (*fun* $x\ f$) a *in* $\mathsf{D_U}$.

Proof. We instantiate the free theorem for f

$$\frac{R \subseteq A \times B \qquad (inst_1, inst_2) \in \mathsf{Dict}(R) \qquad (d_1, d_2) \in R}{(f_A(inst_1)(d_1), f_B(inst_2)(d_2)) \in R}$$

as follows

$$\begin{array}{c} A \triangleq B \triangleq \mathsf{D_U}, \qquad inst_1 \triangleq inst_2 \triangleq inst, \qquad d_1 \triangleq a, \qquad d_2 \triangleq pre(x) \\ R_{x,a}(d_1, d_2) \triangleq \forall g.\ d_1 = g(a) \wedge d_2 = g(pre(x)) \implies g(a) \sqsubseteq apply(fun(x, g), a) \end{array}$$

and get (translated back into Haskell)

$$\frac{(inst, inst) \in \mathsf{Dict}(R_{x,a}) \qquad (a, pre\ x) \in R_{x,a}}{(f\ a, f\ (pre\ x)) \in R_{x,a}}$$

where $pre\ x \triangleq \langle [x \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle$ defines the proxy for x, exactly as in the implementation of *fun* x, and *inst* is the canonical instance dictionary for $\mathsf{D_U}$.

We will first apply this inference rule and then show that the goal follows from $(f\ a, f\ (pre\ x)) \in R_{x,a}$.

The above inference rule can be used to show the overall goal $f\ \ a \sqsubseteq apply\ (fun\ x\ f)\ a$:

$$
\begin{array}{ll}
& (f\ a, f\ (pre\ x)) \in R_{x,a} \\
\Longleftrightarrow & \wr \text{ Definition of } R_{x,a}, apply, fun, \text{detailed reasoning below } \int \\
& f\ a \sqsubseteq \mathbf{let}\ \langle \varphi, v\rangle = f\ (pre\ x)\ \mathbf{in}\ \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi, v\rangle \\
\Longleftrightarrow & \wr \text{ refold } apply, fun \int \\
& f\ a \sqsubseteq apply\ (fun\ x\ f)\ a
\end{array}
$$

To apply the inference rule, we must prove its premises. Before we do so, it is very helpful to eliminate the quantification over arbitrary g in the relation $R_{x,a}(d_1, d_2)$. To that end, we first need to factor $fun\ x\ g = abs\ x\ (g\ (pre\ x))$, where abs is defined as follows:

$$abs\ x\ \langle \varphi, v\rangle = \langle \varphi[x \mapsto \mathsf{U}_0], \varphi\ !?\ x \fatsemi v\rangle$$

And we simplify $R_{x,a}(d_1, d_2)$, thus

$$
\begin{array}{ll}
& \forall g.\ d_1 = g\ a \wedge d_2 = g\ (pre\ x) \implies g\ a \sqsubseteq apply\ (fun\ x\ g)\ a \\
\Longleftrightarrow & \wr\ fun\ x\ g = abs\ x\ (g\ (pre\ x)) \int \\
& \forall g.\ d_1 = g\ a \wedge d_2 = g\ (pre\ x) \implies g\ a \sqsubseteq apply\ (abs\ x\ (g\ (pre\ x)))\ a \\
\Longleftrightarrow & \wr \text{ Use } d_1 = g\ a \text{ and } d_2 = g\ (pre\ x) \int \\
& \forall g.\ d_1 = g\ a \wedge d_2 = g\ (pre\ x) \implies d_1 \sqsubseteq apply\ (abs\ x\ d_2)\ a \\
\Longleftrightarrow & \wr \text{ There exists a } g \text{ satisfying } d_1 = g\ a \text{ and } d_2 = g\ (pre\ x) \int \\
& d_1 \sqsubseteq apply\ (abs\ x\ d_2)\ a \\
\Longleftrightarrow & \wr \text{ Inline } apply, abs, \text{simplify} \int \\
& d_1 \sqsubseteq \mathbf{let}\ \langle \varphi, v\rangle = d_2\ \mathbf{in}\ \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi, v\rangle
\end{array}
$$

Note that this implies $d_1.\varphi\ !?\ x = \mathsf{U}_0$, because $\varphi[x \mapsto \mathsf{U}_0]\ !?\ x = \mathsf{U}_0$ and $a.\varphi\ !?\ x = \mathsf{U}_0$ by the scoping discipline.

It turns out that $R_{x,a}$ is reflexive on all d for which $d.\varphi\ ?!\ x = \mathsf{U}_0$; indeed, then the inequality becomes an equality. (This corresponds to summarising a function that does not use its argument.) That is a fact that we need in the *stuck*, *fun*, *con* and *select* cases below, so we prove it here:

$$
\begin{array}{ll}
& \forall d.\ d \sqsubseteq \langle (d.\varphi)[x \mapsto \mathsf{U}_0] + (d.\varphi\ !?\ x) * a.\varphi, d.v\rangle \\
\Longleftrightarrow & \wr \text{ Use } (d.\varphi\ ?!\ x) = \mathsf{U}_0 \int \\
& \forall d.\ d \sqsubseteq \langle d.\varphi, d.v\rangle = d
\end{array}
$$

The last proposition is reflexivity on $\sqsubseteq$.

Now we prove the premises of the abstraction theorem:

- $(a, \mathit{pre}\ x) \in R_{x,a}$: The proposition unfolds to

$$\begin{array}{l} a \sqsubseteq \textbf{let}\ \langle \varphi, v\rangle = \mathit{pre}\ x\ \textbf{in}\ \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi, v\rangle \\ \iff \text{⟅ Unfold } \mathit{pre}\text{, simplify ⟆} \\ a \sqsubseteq \langle a.\varphi, \mathsf{Rep}\ \mathsf{U}_\omega\rangle \end{array}$$

  The latter follows from $a.v \sqsubseteq \mathsf{Rep}\ \mathsf{U}_\omega$ because $\mathsf{Rep}\ \mathsf{U}_\omega$ is the Top element.

- $(\mathit{inst}, \mathit{inst}) \in \mathsf{Dict}(R_{x,a})$: By the relational interpretation of products, we get one subgoal per instance method.

  - **Case** *step*. Goal: $\dfrac{(d_1, d_2) \in R_{x,a}}{(\mathit{step}\ \mathit{ev}\ d_1, \mathit{step}\ \mathit{ev}\ d_2) \in R_{x,a}}$.
    Assume the premise $(d_1, d_2) \in R_{x,a}$, show the goal. All cases other than $\mathit{ev} = \mathsf{Look}\ y$ are trivial, because then $\mathit{step}\ \mathit{ev}\ d = d$ and the goal follows by the premise. So let $\mathit{ev} = \mathsf{Look}\ y$. The goal is to show

$$\begin{array}{l} \mathit{step}\ (\mathsf{Look}\ y)\ d_1 \sqsubseteq \textbf{let}\ \langle \varphi, v\rangle = \mathit{step}\ (\mathsf{Look}\ y)\ d_2\ \textbf{in} \\ \qquad\qquad \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi, v\rangle \end{array}$$

    We begin by unpacking the assumption $(d_1, d_2) \in R_{x,a}$ to show it:

$$\begin{array}{l} d_1 \sqsubseteq \textbf{let}\ \langle \varphi, v\rangle = d_2\ \textbf{in}\ \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi, v\rangle \\ \impliedby \text{⟅ } \mathit{step}\ (\mathsf{Look}\ y) \text{ is monotonic ⟆} \\ \mathit{step}\ (\mathsf{Look}\ y)\ d_1 \\ \sqsubseteq \mathit{step}\ (\mathsf{Look}\ y)\ \langle (d_2.\varphi)[x \mapsto \mathsf{U}_0] + (d_2.\varphi\ !?\ x) * a.\varphi, d_2.v\rangle \\ \iff \text{⟅ Refold } \mathit{step}\ (\mathsf{Look}\ y) \text{ ⟆} \\ \mathit{step}\ (\mathsf{Look}\ y)\ d_1 \\ \sqsubseteq \langle (d_2.\varphi)[x \mapsto \mathsf{U}_0] + [y \mapsto \mathsf{U}_1] + (d_2.\varphi\ !?\ x) * a.\varphi, d_2.v\rangle \\ \iff \text{⟅ } x \neq y \text{ because } y \text{ is let-bound ⟆} \\ \mathit{step}\ (\mathsf{Look}\ y)\ d_1 \\ \sqsubseteq \textbf{let}\ \langle \varphi, v\rangle = \mathit{step}\ (\mathsf{Look}\ y)\ d_2\ \textbf{in}\ \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi, v\rangle \end{array}$$

  - **Case** *stuck*. Goal: $(\mathit{stuck}, \mathit{stuck}) \in R_{x,a}$
    Follows from reflexivity, because $\mathit{stuck} = \bot$, and $\bot.\varphi\ !?\ x = \mathsf{U}_0$.

– **Case** *fun*. Goal: $\dfrac{\forall (d_1, d_2) \in R_{x,a}.\ (f_1\ d_1, f_2\ d_2) \in R_{x,a}}{(\mathit{fun}\ y\ f_1, \mathit{fun}\ y\ f_2) \in R_{x,a}}$.
Additionally, we may assume $x \neq y$ by lexical scoping.

Now assume the premise. The goal is to show

$$\mathit{fun}\ y\ f_1 \sqsubseteq \textbf{let}\ \langle \varphi, v\rangle = \mathit{fun}\ y\ f_2\ \textbf{in}\ \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi, v\rangle$$

Recall that $\mathit{fun}\ y\ f = \mathit{abs}\ y\ (f\ (\mathit{pre}\ y))$ and that $\mathit{abs}\ y$ is monotonic.

Note that we have $(\mathit{pre}\ y, \mathit{pre}\ y) \in R_{x,a}$ because of $x \neq y$ and reflexivity. That in turn yields $(f_1\ (\mathit{pre}\ y), f_2\ (\mathit{pre}\ y)) \in R_{x,a}$ by assumption. This is useful to kick-start the following proof, showing the goal:

$$\begin{array}{ll}
& f_1\ (\mathit{pre}\ y) \\
& \quad \sqsubseteq \textbf{let}\ \langle \varphi, v\rangle = f_2\ (\mathit{pre}\ y)\ \textbf{in}\ \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi, v\rangle \\
\Longrightarrow & \quad \wr\ \text{Monotonicity of } \mathit{abs}\ y\ \int \\
& \mathit{abs}\ y\ (f_1\ (\mathit{pre}\ y)) \\
& \quad \sqsubseteq \mathit{abs}\ y\ (\textbf{let}\ \langle \varphi, v\rangle = f_2\ (\mathit{pre}\ y)\ \textbf{in}\ \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi, v\rangle) \\
\Longleftrightarrow & \quad \wr\ x \neq y \text{ and } a.\varphi\ !?\ y = \mathsf{U}_0 \text{ due to scoping, } \int \\
& \qquad \wr\ \varphi\ !?\ x \text{ unaffected by floating } \mathit{abs}\ \int \\
& \mathit{abs}\ y\ (f_1\ (\mathit{pre}\ y)) \\
& \quad \sqsubseteq \textbf{let}\ \langle \varphi, v\rangle = \mathit{abs}\ y\ (f_2\ (\mathit{pre}\ y))\ \textbf{in}\ \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi, v\rangle \\
\Longleftrightarrow & \quad \wr\ \text{Rewrite } \mathit{abs}\ y\ (f\ (\mathit{pre}\ y)) = \mathit{fun}\ y\ f\ \int \\
& \mathit{fun}\ y\ f_1 \\
& \quad \sqsubseteq \textbf{let}\ \langle \varphi, v\rangle = \mathit{fun}\ y\ f_2\ \textbf{in}\ \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi, v\rangle
\end{array}$$

– **Case** *apply*. Goal: $\dfrac{(l_1, l_2) \in R_{x,a} \qquad (r_1, r_2) \in R_{x,a}}{(\mathit{apply}\ l_1\ r_1, \mathit{apply}\ l_2\ r_2) \in R_{x,a}}$.
Assume the premises. The goal is to show

$$\mathit{apply}\ l_1\ r_1 \sqsubseteq \textbf{let}\ \langle \varphi, v\rangle = \mathit{apply}\ l_2\ r_2\ \textbf{in}\ \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi, v\rangle$$

$$\begin{array}{ll}
& \mathit{apply}\ l_1\ r_1 \\
\sqsubseteq & \quad \wr\ l_1 \sqsubseteq \mathit{apply}\ (\mathit{abs}\ x\ l_2), r_2 \sqsubseteq \mathit{apply}\ (\mathit{abs}\ x\ r_2), \text{monotonicity}\ \int \\
& \mathit{apply}\ (\textbf{let}\ \langle \varphi, v\rangle = l_2\ \textbf{in}\ \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi, v\rangle) \\
& \qquad (\textbf{let}\ \langle \varphi, v\rangle = r_2\ \textbf{in}\ \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi, v\rangle)
\end{array}$$

$$
\begin{array}{ll}
\sqsubseteq & \wr \text{ Componentwise, see below } \int \\
& \textbf{let } \langle \varphi, v \rangle = \mathit{apply}\ l_2\ r_2 \textbf{ in } \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi, v \rangle
\end{array}
$$

For the last step, we show the inequality for φ and v independently. For values, it is easy to see by calculation that the value is $v \triangleq \mathit{snd}\ (\mathit{peel}\ l_2.v)$ in both cases. The proof for the Uses component is quite algebraic; we will abbreviate $u \triangleq \mathit{fst}\ (\mathit{peel}\ l_2.v)$:

$$
\begin{array}{ll}
& (\mathit{apply}\ (\textbf{let } \langle \varphi, v \rangle = l_2 \textbf{ in } \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi, v \rangle) \\
& \qquad (\textbf{let } \langle \varphi, v \rangle = r_2 \textbf{ in } \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi, v \rangle)).\varphi \\
= & \wr \text{ Unfold } \mathit{apply} \int \\
& (l_2.\varphi)[x \mapsto \mathsf{U}_0] + (l_2.\varphi\ !?\ x) * a.\varphi \\
& + u * ((r_2.\varphi)[x \mapsto \mathsf{U}_0] + (r_2.\varphi\ !?\ x) * a.\varphi) \\
= & \wr \text{ Distribute } u * (\varphi_1 + \varphi_2) = u * \varphi_1 + u * \varphi_2 \int \\
& (l_2.\varphi)[x \mapsto \mathsf{U}_0] + (l_2.\varphi\ !?\ x) * a.\varphi \\
& + u * (r_2.\varphi)[x \mapsto \mathsf{U}_0] + u * (r_2.\varphi\ !?\ x) * a.\varphi \\
= & \wr \text{ Commute } \int \\
& (l_2.\varphi)[x \mapsto \mathsf{U}_0] + u * (r_2.\varphi)[x \mapsto \mathsf{U}_0] \\
& + (l_2.\varphi\ !?\ x) * a.\varphi + u * (r_2.\varphi\ !?\ x) * a.\varphi \\
= & \wr\ \varphi_1[x \mapsto \mathsf{U}_0] + \varphi_2[x \mapsto \mathsf{U}_0] = (\varphi_1 + \varphi_2)[x \mapsto \mathsf{U}_0], \int \\
& \wr\ u * \varphi_1 + u * \varphi_2 = u * (\varphi_1 + \varphi_2)\ \int \\
& (l_2.\varphi + u * r_2.\varphi)[x \mapsto \mathsf{U}_0] + ((l_2.\varphi + u * r_2.\varphi)\ !?\ x) * a.\varphi \\
= & \wr \text{ Refold } \mathit{apply} \int \\
& \textbf{let } \langle \varphi, \_ \rangle = \mathit{apply}\ l_2\ r_2 \textbf{ in } \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * a.\varphi
\end{array}
$$

– **Case** *con*. Goal: $\dfrac{(ds_1, ds_2) \in [R_{x,a}]}{(\mathit{con}\ k\ ds_1, \mathit{con}\ k\ ds_2) \in R_{x,a}}$.
We have shown that *apply* is compatible with $R_{x,a}$, and *foldl* is so as well by parametricity. The field denotations ds_1 and ds_2 satisfy $R_{x,a}$ by assumption; hence to show the goal it is sufficient to show that $(\langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_\omega \rangle, \langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_\omega \rangle) \in R_{x,a}$. And that follows by reflexivity since $\varepsilon\ ?!\ x = \mathsf{U}_0$.

– **Case** *select*. Goal: $\dfrac{(d_1, d_2) \in R_{x,a} \quad (fs_1, fs_2) \in \mathsf{Tag} :\rightharpoonup ([R_{x,a}] \to R_{x,a})}{(\mathit{select}\ d_1\ fs_1, \mathit{select}\ d_2\ fs_2) \in R_{x,a}}$.
Similar to the *con* case, large parts of the implementation are compatible with $R_{x,a}$ already. With $(\langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_\omega \rangle, \langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_\omega \rangle) \in R_{x,a}$ proved in the *con* case, it remains to be shown that $\mathit{lub} :: [\mathsf{D}_\mathsf{U}] \to \mathsf{D}_\mathsf{U}$

and $(\gg) :: \mathsf{D_U} \to \mathsf{D_U} \to \mathsf{D_U}$ preserve $R_{x,a}$. The proof for $(\gg)$ is very similar to but simpler than the *apply* case, where a subexpression similar to $\langle \varphi_1 + \varphi_2, b \rangle$ occurs. The proof for *lub* follows from the proof for the least upper bound operator $\sqcup$.

So let $(l_1, l_2), (r_1, r_2) \in R_{x,a}$ and show that $(l_1 \sqcup r_1, l_2 \sqcup r_2) \in R_{x,a}$. The assumptions imply that $l_1.v \sqsubseteq l_2.v$ and $r_1.v \sqsubseteq r_2.v$, so $(l_1 \sqcup r_1).v \sqsubseteq (l_2 \sqcup r_2).v$ follows by properties of least upper bound operators.

Let us now consider the Uses component. The goal is to show

$$(l_1 \sqcup r_1).\varphi \sqsubseteq (\textbf{let } \langle \varphi, v \rangle = l_2 \sqcup r_2 \textbf{ in } \langle \varphi[x \mapsto \mathsf{U_0}] + (\varphi\ !?\ x) * a.\varphi, v \rangle).\varphi$$

For the proof, we need the algebraic identity $\forall a\ b\ c\ d.\ a + c \sqcup b + d \sqsubseteq a \sqcup b + c \sqcup d$ in U. This can be proved by exhaustive enumeration of all 81 cases; the inequality is proper when $a = d = \mathsf{U_1}$ and $b = c = \mathsf{U_0}$ (or vice versa). Thus we conclude the proof:

$$\begin{aligned}
& (l_1 \sqcup r_1).\varphi = l_1.\varphi \sqcup r_1.\varphi \\
= \ & \wr \text{ By assumption, } l_1 \sqsubseteq \mathit{apply}\ (\mathit{abs}\ x\ l_2) \ \int \\
& \wr \text{ and } r_1 \sqsubseteq \mathit{apply}\ (\mathit{abs}\ x\ r_2); \text{ monotonicity } \int \\
& ((l_2.\varphi)[x \mapsto \mathsf{U_0}] + (l_2.\varphi\ !?\ x) * a.\varphi) \\
& \sqcup ((r_2.\varphi)[x \mapsto \mathsf{U_0}] + (r_2.\varphi\ !?\ x) * a.\varphi) \\
\sqsubseteq \ & \wr \text{ Follows from } \forall a\ b\ c\ d.\ a + c \sqcup b + d \sqsubseteq a \sqcup b + c \sqcup d \text{ in } \mathsf{U} \ \int \\
& ((l_2.\varphi)[x \mapsto \mathsf{U_0}] \sqcup (r_2.\varphi)[x \mapsto \mathsf{U_0}]) \\
& + ((l_2.\varphi\ !?\ x) * a.\varphi \sqcup (r_2.\varphi\ !?\ x) * a.\varphi) \\
= \ & \wr\ \varphi_1[x \mapsto \mathsf{U_0}] \sqcup \varphi_2[x \mapsto \mathsf{U_0}] = (\varphi_1 \sqcup \varphi_2)[x \mapsto \mathsf{U_0}] \ \int \\
& ((l_2 \sqcup r_2).\varphi)[x \mapsto \mathsf{U_0}] + ((l_2 \sqcup r_2).\varphi\ !?\ x) * a.\varphi \\
= \ & \wr \text{ Refold } \langle \varphi, v \rangle \ \int \\
& (\textbf{let } \langle \varphi, v \rangle = l_2 \sqcup r_2 \textbf{ in } \langle \varphi[x \mapsto \mathsf{U_0}] + (\varphi\ !?\ x) * a.\varphi, v \rangle).\varphi
\end{aligned}$$

– **Case** *bind*. Goal: $\dfrac{\forall (d_1, d_2) \in R_{x,a}.\ (f_1\ d_1, f_2\ d_2), (g_1\ d_1, g_2\ d_2) \in R_{x,a}}{(\mathit{bind}\ f_1\ g_1, \mathit{bind}\ f_2\ g_2) \in R_{x,a}}$.
By the assumptions, the definition $\mathit{bind}\ f\ g = g\ (\mathit{kleeneFix}\ f)$ preserves $R_{x,a}$ if *kleeneFix* does. Since $\mathit{kleeneFix} :: \mathsf{Lat}\ a \Rightarrow (a \to a) \to a$ is parametric, it suffices to show that the instance of Lat preserves $R_{x,a}$. We have already shown that $\sqcup$ preserves $R_{x,a}$, and we have also shown that $\mathit{stuck} = \bot$ preserves $R_{x,a}$. Hence we have shown the goal.

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

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

Fig. A.2: Syntax of by-need evaluation contexts and translation function *trans* to Mark I machine contexts

> In Section 4.5.1, I introduced a widening operator $\mathit{widen} :: \mathsf{D_U} \to \mathsf{D_U}$ to the definition of *bind*, that is, I defined $\mathit{bind}\ \mathit{rhs}\ \mathit{body} = \mathit{body}\ (\mathit{kleeneFix}\ (\mathit{widen} \circ \mathit{rhs}))$. For such an operator, I would additionally need to show that *widen* preserves $R_{x,a}$. Since the proposed cutoff operator in Section 4.5.1 only affects the $\mathsf{Value_U}$ component, the only proof obligation is to show monotonicity: $\forall\, d_1\ d_2.\ d_1.v \sqsubseteq d_2.v \implies (\mathit{widen}\ d_1).v \sqsubseteq (\mathit{widen}\ d_2).v$. This is a requirement that my widening operator must satisfy anyway.
>
> □

In the proof for Theorem 4.23 I exploit that usage analysis is somewhat invariant under wrapping of *by-need evaluation contexts*, roughly $\mathsf{U}_\omega * \mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_e} \sqsupseteq \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_\varepsilon$. To prove that, I first need to define what the by-need evaluation contexts of my 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 4.2 that uses syntactic substitution of variables instead of delayed substitution and addresses, so $\mu \in \mathsf{Var} \rightharpoonup \mathsf{Exp}$ and no closures are needed.

I follow their approach, but inline applicative contexts,[1] thus defining the by-need evaluation contexts with hole □ for our language in Figure A.2. The correspondence to Mark I machine contexts $(\Box, \mu, \kappa)$ is encoded by the translation

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

function *trans* in that same figure. It translates from mark I machine contexts $(\square, \mu, \kappa)$ to evaluation contexts E. Certainly the most interesting case is that of **upd** frames, encoding by-need memoisation. This translation function has the following property:

Lemma A.11 (Translation, without proof)**.** *$init(trans(\square, \mu, \kappa)[\mathsf{e}]) \hookrightarrow^* (\mathsf{e}, \mu, \kappa)$, and all transitions in this trace are search transitions ($\textsc{App}_1$, $\textsc{Case}_1$, $\textsc{Let}_1$, $\textsc{Look}$).*

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

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

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

Lemma 4.22 (Denotational absence)**.** *Variable x is used in e if and only if* ↻ 143 *there exists a by-need evaluation context E and expression e' such that the trace $\mathcal{S}_{\mathbf{need}}[\![E[\mathsf{Let}\ x\ e'\ e]]\!]_\varepsilon(\varepsilon)$ contains a Look x event. Otherwise, x is absent in e.*

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

$$(\textbf{let}\ \mathsf{x} = \mathsf{e}'\ \textbf{in}\ \mathsf{e}, \rho, \mu, \kappa) \hookrightarrow^* \ldots \xhookrightarrow{\textsc{Look}(\mathsf{x})} \ldots$$

We proceed as follows:

$$\begin{aligned} & (\textbf{let}\ \mathsf{x} = \mathsf{e}'\ \textbf{in}\ \mathsf{e}, \rho, \mu, \kappa) \hookrightarrow^* \ldots \xhookrightarrow{\textsc{Look}(\mathsf{x})} \ldots \\ \iff & \{\!|\ \mathsf{E} \triangleq trans(\square, \rho, \mu, \kappa)\ |\!\} \\ & init(\mathsf{E}[\textbf{let}\ \mathsf{x} = \mathsf{e}'\ \textbf{in}\ \mathsf{e}]) \hookrightarrow^* \ldots \xhookrightarrow{\textsc{Look}(\mathsf{x})} \ldots \end{aligned} \tag{A.1}$$

$$
\begin{aligned}
&\iff \text{⟅ Apply } \alpha_{\mathbb{S}^\infty} \text{ (Figure 4.13) ⟆} \\
&\quad \alpha_{\mathbb{S}^\infty}(\mathit{init}(\mathsf{E}[\textbf{let } \mathsf{x} = \mathsf{e}' \textbf{ in } \mathsf{e}]) \hookrightarrow^*, []) = \ldots\mathsf{Step}\ (\mathsf{Look}\ x)\ldots \\
&\iff \text{⟅ Theorem 4.4 ⟆} \\
&\quad \mathcal{S}_{\mathbf{need}}[\![E[\mathsf{Let}\ x\ e'\ e]]\!]_\varepsilon(\varepsilon) = \ldots\mathsf{Step}\ (\mathsf{Look}\ x)\ldots
\end{aligned}
$$

Note that the trace we start with is not necessarily a maximal trace, so step (A.1) finds a prefix that makes the trace maximal. We do so by reconstructing the syntactic *evaluation context* E with *trans* (cf. Lemma A.11) such that

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

Then the trace above is a suffix of the maximal trace that starts in state $\mathit{init}(\mathsf{E}[\textbf{let } \mathsf{x} = \mathsf{e}' \textbf{ in } \mathsf{e}])$ and it contains at least one $\textsc{Look}(\mathsf{x})$ transition. This $\textsc{Look}(\mathsf{x})$ transition cannot occur in the unshared trace prefix, because that would mean that x occurs in E, violating the distinct bound variables invariant from Section 4.1.1.

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

Lemma A.12 (Used variables are free)**.** *If x does not occur in e and in ρ (that is, $\forall y.\ (\rho\,!\,y).\varphi\ !?\ x = \mathsf{U}_0$), then $(\mathcal{S}_{\mathbf{usage}}[\![e]\!]\rho).\varphi\ !?\ x = \mathsf{U}_0$.*

Proof. By induction on e. □

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

Definition A.13 (Abstract substitution)**.** *I call $\varphi[x \Mapsto \varphi'] \triangleq \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * \varphi'$ the* abstract substitution *operation on* Uses *and overload this notation for* T_U*, so that $\langle \varphi, v\rangle[x \Mapsto \varphi'] \triangleq \langle \varphi[x \Mapsto \varphi'], v\rangle$.*

From Lemma 4.19, I can derive the following auxiliary lemma:

Lemma A.14. *If x does not occur in ρ, then*

$$\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho[x \mapsto d]} \sqsubseteq (\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho[x \mapsto \langle [x \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega\rangle]})[x \Mapsto d.\varphi].$$

Proof. Define $f\ \widehat{d} \triangleq \mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho[x \mapsto \widehat{d}]}$ and $a \triangleq d$. Note that f could be defined polymorphically as $f\ d = \mathcal{S}[\![e]\!]_{\rho[x \mapsto d]}$, for suitably polymorphic ρ. Furthermore, x could well be lambda-bound, since it does not occur in the range of ρ (and that is really what we need). Hence we may apply Lemma 4.19 to get

$$
\begin{array}{ll}
& \mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho[x \mapsto d]} \\
\sqsubseteq & \wr \text{ Lemma 4.19 } \int \\
& \mathit{apply}\ (\mathit{fun}\ x\ (\lambda\widehat{d} \to \mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho[x \mapsto \widehat{d}]}))\ d \\
= & \wr \text{ Inline } \mathit{apply}, \mathit{fun} \int \\
& \mathbf{let}\ \langle \varphi, v \rangle = \mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho[x \mapsto \langle [x \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle]}\ \mathbf{in} \\
& \langle \varphi[x \mapsto \mathsf{U}_0] + (\varphi\ !?\ x) * d.\varphi, v \rangle \\
= & \wr \text{ Refold } \_[\_ \Longmapsto \_] \int \\
& (\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho[x \mapsto \langle [x \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle]})[x \Longmapsto d].\varphi
\end{array}
$$

□

Lemma A.15 (Context closure). *Let e be an expression and E be a by-need evaluation context in which x does not occur. Then $(\mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E}).\varphi\ ?!\ x \sqsubseteq \mathsf{U}_\omega * ((\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_e}).\varphi\ !?\ x)$, where ρ_E and ρ_e are the initial environments that map free variables z to their proxy $\langle [z \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle$.*

Proof. By induction on the size of E and cases on E:

- **Case** Hole:

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

  By reflexivity.

- **Case** Apply E y: Since y occurs in E, it must be different to x.

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

$$
\begin{array}{ll}
& \textbf{let } \langle \varphi, v \rangle = \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E} \textbf{ in} \\
& \textbf{case } \mathit{peel}\ v \textbf{ of } (u, v_2) \to \varphi\ !?\ x \\
= & \ \wr \text{ Refold } \langle \varphi, v \rangle.\varphi = \varphi \ \int \\
& (\mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E}).\varphi\ !?\ x \\
\sqsubseteq & \ \wr \text{ Induction hypothesis } \int \\
& \mathsf{U}_\omega * (\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_e}).\varphi\ !?\ x
\end{array}
$$

- **Case** Select E alts: Since x does not occur in alts, it is absent in alts as well by Lemma A.12. (Recall that select analyses alts with $\langle \varepsilon, \mathsf{Rep}\ \mathsf{U}_\omega \rangle$ as field proxies.)

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

- **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 $(\mathcal{S}_{\mathbf{usage}}[\![e_1]\!]_{\rho_E}).\varphi\ !?\ x = \mathsf{U}_0$ by Lemma A.12.

$$
\begin{array}{ll}
& (\mathcal{S}_{\mathbf{usage}}[\![(\mathsf{ExtendHeap}\ y\ e_1\ E)[e]]\!]_{\rho_E}).\varphi\ !?\ x \\
= & \ \wr \text{ Definition of } \_[\_] \ \int \\
& (\mathcal{S}_{\mathbf{usage}}[\![\mathsf{Let}\ y\ e_1\ E[e]]\!]_{\rho_E}).\varphi\ !?\ x \\
= & \ \wr \text{ Definition of } \mathcal{S}_{\mathbf{usage}}[\![\_]\!]_\_ \ \int \\
& \textbf{let } \mathit{rhs}\ d = \mathcal{S}_{\mathbf{usage}}[\![e_1]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Look}\ y)\ d]} \textbf{ in} \\
& (\mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Look}\ y)\ (\mathit{kleeneFix}\ \mathit{rhs})]}).\varphi\ !?\ x \\
\sqsubseteq & \ \wr \text{ Abstract substitution; Lemma A.14 } \int \\
& \textbf{let } \mathit{rhs}\ d = \mathcal{S}_{\mathbf{usage}}[\![e_1]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Look}\ y)\ d]} \textbf{ in} \\
& (\mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle]})[y \Longmapsto \\
& \quad \mathit{step}\ (\mathsf{Look}\ y)\ (\mathit{kleeneFix}\ \mathit{rhs})].\varphi\ !?\ x
\end{array}
$$

$$
\begin{aligned}
&= \text{⦇ Unfold } \_[\_ \Mapsto \_], \langle \varphi, v \rangle.\varphi = \varphi \text{ ⦈} \\
&\quad \textbf{let } \mathit{rhs}\ d = \mathcal{S}_{\mathbf{usage}}[\![e_1]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Look}\ y)\ d]} \textbf{ in} \\
&\quad \textbf{let } \langle \varphi, \_ \rangle = \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle]} \textbf{ in} \\
&\quad \textbf{let } \langle \varphi_2, \_ \rangle = \mathit{step}\ (\mathsf{Look}\ y)\ (\mathit{kleeneFix}\ \mathit{rhs}) \textbf{ in} \\
&\quad (\varphi[y \mapsto \mathsf{U}_0] + (\varphi\ !?\ y) * \varphi_2)\ !?\ x \\
&= \text{⦇ } x \text{ absent in } \varphi_2\text{, see above ⦈} \\
&\quad \textbf{let } \langle \varphi, \_ \rangle = \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle]} \textbf{ in} \\
&\quad \varphi\ !?\ x \\
&\sqsubseteq \text{⦇ Induction hypothesis ⦈} \\
&\quad \mathsf{U}_\omega * (\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_e}).\varphi\ !?\ x
\end{aligned}
$$

- **Case** $\mathsf{UpdateHeap}\ y\ E\ e_1$: Since x does not occur in e_1, and the initial environment is absent in x as well, we have $(\mathcal{S}_{\mathbf{usage}}[\![e_1]\!]_{\rho_E[y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle]}).\varphi\ !?\ x = \mathsf{U}_0$ by Lemma A.12.

$$
\begin{aligned}
&\quad (\mathcal{S}_{\mathbf{usage}}[\![(\mathsf{UpdateHeap}\ y\ E\ e_1)[e]]\!]_{\rho_E}).\varphi\ !?\ x \\
&= \text{⦇ Definition of } \_[\_] \text{ ⦈} \\
&\quad (\mathcal{S}_{\mathbf{usage}}[\![\mathsf{Let}\ y\ E[e]\ e_1]\!]_{\rho_E}).\varphi\ !?\ x \\
&= \text{⦇ Definition of } \mathcal{S}_{\mathbf{usage}}[\![\_]\!]\_ \text{ ⦈} \\
&\quad \textbf{let } \mathit{rhs}\ d = \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Look}\ y)\ d]} \textbf{ in} \\
&\quad (\mathcal{S}_{\mathbf{usage}}[\![e_1]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Look}\ y)\ (\mathit{kleeneFix}\ \mathit{rhs})]}).\varphi\ !?\ x \\
&\sqsubseteq \text{⦇ Abstract substitution; Lemma A.14 ⦈} \\
&\quad \textbf{let } \mathit{rhs}\ d = \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Look}\ y)\ d]} \textbf{ in} \\
&\quad (\mathcal{S}_{\mathbf{usage}}[\![e_1]\!]_{\rho_E[y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle]})[y \Mapsto \mathit{step} \\
&\qquad (\mathsf{Look}\ y)\ (\mathit{kleeneFix}\ \mathit{rhs})].\varphi\ !?\ x \\
&= \text{⦇ Unfold } \_[\_ \Mapsto \_], \langle \varphi, v \rangle.\varphi = \varphi \text{ ⦈} \\
&\quad \textbf{let } \mathit{rhs}\ d = \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Look}\ y)\ d]} \textbf{ in} \\
&\quad \textbf{let } \langle \varphi, \_ \rangle = \mathcal{S}_{\mathbf{usage}}[\![e_1]\!]_{\rho_E[y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle]} \textbf{ in} \\
&\quad \textbf{let } \langle \varphi_2, \_ \rangle = \mathit{step}\ (\mathsf{Look}\ y)\ (\mathit{kleeneFix}\ \mathit{rhs}) \textbf{ in} \\
&\quad (\varphi[y \mapsto \mathsf{U}_0] + (\varphi\ !?\ y) * \varphi_2)\ !?\ x \\
&= \text{⦇ } \varphi\ !?\ y \sqsubseteq \mathsf{U}_\omega, x \text{ absent in } \varphi\text{, see above ⦈} \\
&\quad \textbf{let } \mathit{rhs}\ d = \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Look}\ y)\ d]} \textbf{ in} \\
&\quad \textbf{let } \langle \varphi_2, \_ \rangle = \mathit{step}\ (\mathsf{Look}\ y)\ (\mathit{kleeneFix}\ \mathit{rhs}) \textbf{ in} \\
&\quad \mathsf{U}_\omega * \varphi_2\ !?\ x \\
&= \text{⦇ Refold } \langle \varphi, v \rangle.\varphi \text{ ⦈} \\
&\quad \textbf{let } \mathit{rhs}\ d = \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \mathit{step}\ (\mathsf{Look}\ y)\ d]} \textbf{ in}
\end{aligned}
$$

$$
\begin{aligned}
& \mathsf{U}_\omega * (\mathit{step}\ (\mathsf{Look}\ y)\ (\mathit{kleeneFix}\ \mathit{rhs})).\varphi\ !?\ x \\
= \ & \wr\ x \neq y\ \int \\
& \mathsf{U}_\omega * (\mathit{kleeneFix}\ (\lambda d \to \mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto d]})).\varphi\ !?\ x \\
= \ & \wr\ \text{Argument below}\ \int \\
& \mathsf{U}_\omega * (\mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle]}).\varphi\ !?\ x \\
\sqsubseteq \ & \wr\ \text{Induction hypothesis, } \mathsf{U}_\omega * \mathsf{U}_\omega = \mathsf{U}_\omega\ \int \\
& \mathsf{U}_\omega * (\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_e}).\varphi\ !?\ x
\end{aligned}
$$

The rationale for removing the *kleeneFix* is that under the assumption that x is absent in d (such as is the case for $d \triangleq \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle$), then it is also absent in $\mathcal{S}_{\mathbf{usage}}[\![E[e]]\!]_{\rho_E[y \mapsto d]}$ per Lemma A.12. Otherwise, we go to U_ω anyway.

UpdateHeap is why it is necessary to multiply with U_ω 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** $y = \ldots$ **in** e; for $\mathrm{e} \triangleq y$ we would infer that y is used at most once, but that is wrong in context **let** $x = \Box$ **in** $x\,x$.

□

↻ 144 **Theorem 4.23** ($\mathcal{S}_{\mathbf{usage}}[\![\_]\!]\_$ infers absence). *Let* $\rho_e \triangleq [\overline{y \mapsto \langle [y \mapsto \mathsf{U}_1], \mathsf{Rep}\ \mathsf{U}_\omega \rangle}]$ *be the initial environment with an entry for every free variable* y *of an expression* e*. If* $\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_e} = \langle \varphi, v \rangle$ *and* $\varphi\ !?\ x = \mathsf{U}_0$*, then* x *is absent in* e*.*

Proof. We show the contraposition, that is, if x is used in e, then $\varphi\ !?\ x \neq \mathsf{U}_0$.
By Lemma 4.22, there exists E, e' such that

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

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

$$
\begin{aligned}
& \mathcal{S}_{\mathbf{need}}[\![E[\mathsf{Let}\ x\ e'\ e]]\!]_\varepsilon(\varepsilon) = \ldots \mathsf{Step}\ (\mathsf{Look}\ x) \ldots && \text{(A.2)} \\
\Longrightarrow \ & \wr\ \text{Usage abstraction}\ \int \\
& (\alpha_{\mathcal{S}}(\mathcal{S}_{\mathbf{need}}[\![E[\mathsf{Let}\ x\ e'\ e]]\!]_\varepsilon)\ (\varepsilon)).\varphi \sqsupseteq [x \mapsto \mathsf{U}_1] && \text{(A.3)}
\end{aligned}
$$

$$
\begin{aligned}
&\Longrightarrow \quad \wr \text{ Corollary 4.21 } \int \\
&\quad (\mathcal{S}_{\mathbf{usage}}[\![E[\mathsf{Let}\ x\ e'\ e]]\!]_{\varepsilon}).\varphi \sqsupseteq [x \mapsto \mathsf{U}_1] \qquad \text{(A.4)} \\
&\Longrightarrow \quad \wr \text{ Lemma A.15 } \int \\
&\quad \mathsf{U}_\omega * (\mathcal{S}_{\mathbf{usage}}[\![e]\!]_{\rho_e}).\varphi = \mathsf{U}_\omega * \varphi \sqsupseteq [x \mapsto \mathsf{U}_1] \\
&\Longrightarrow \quad \wr\ \mathsf{U}_\omega * \mathsf{U}_0 = \mathsf{U}_0 \sqsubset \mathsf{U}_1 \int \\
&\quad \varphi\ !?\ x \neq \mathsf{U}_0
\end{aligned}
$$

Step (A.2) abstracts the trace by applying the usage abstraction function $\alpha_{\mathcal{S}}$. This function calls $\beta_{\mathbb{T}}$ which replaces every Step constructor with the *step* implementation of T_{U}; The Look x event on the right-hand side implies that its image under $\alpha_{\mathcal{S}}$ is at least $[x \mapsto \mathsf{U}_1]$.

Step (A.3) applies the central abstract interpretation Corollary 4.21 that is the main topic of this section, abstracting the dynamic trace property in terms of the static semantics.

Finally, step (A.4) applies Lemma A.15, which proves that absence information doesn't change when an expression is put in an arbitrary evaluation context. The final step is just algebra. □

B

Agda Code for Section 4.4.5

This Appendix contains the Agda code for Section 4.4.5. It was built and type-checked against Agda 2.6.4.

Guarded Cubical Agda Prelude

The following module is copied from the "example"[1] linked from the Agda user's guide on Guarded Cubical.[2] It can be considered part of the builtins or "runtime system" of Guarded Cubical Agda; I had no part in defining it.

Note the definition of the *later* modality ▷ in terms of a *tick abstraction*. This definition can be thought of as the Reader Tick monad, only that the monad instance is impossible to define with tick abstraction because it would lead to an unsound system. We will however use it mostly as if ▷A were just an ordinary function returning A.

The black triangle variant ▸A is the dependent variant of the tick function type ▷A; it is useful to define guarded recursive types such as EnvD later on.

```
{-# OPTIONS --guarded --cubical #-}
module Later where

open import Cubical.Core.Everything
open import Cubical.Foundations.Everything
open import Agda.Primitive.Cubical

module Prims where
  primitive
```

[1] https://github.com/agda/agda/blob/1c449e23b/test/Succeed/LaterPrims.agda
[2] https://agda.readthedocs.io/en/v2.6.4/language/guarded-cubical.html

```
    primLockUniv : Set₁

open Prims renaming (primLockUniv to LockU) public

private
  variable
    l : Level
    A B : Set l

postulate
  Tick : LockU

▷_ : ∀ {l} → Set l → Set l
▷ A = (@tick x : Tick) -> A

▶_ : ∀ {l} → ▷ Set l → Set l
▶ A = (@tick x : Tick) → A x

next : A → ▷ A
next x _ = x

_⊛_ : ▷ (A → B) → ▷ A → ▷ B
_⊛_ f x a = f a (x a)
infixr 21 _⊛_

map▷ : (f : A → B) → ▷ A → ▷ B
map▷ f x α = f (x α)

transpLater : ∀ (A : I → ▷ Set) → ▶ (A i0) → ▶ (A i1)
transpLater A u0 a = primTransp (\ i → A i a) i0 (u0 a)

postulate
  dfix : ∀ {l} {A : Set l} → (▷ A → A) → ▷ A
  pfix : ∀ {l} {A : Set l} (f : ▷ A → A) → dfix f ≡ next (f (dfix f))

fix : ∀ {l} {A : Set l} → (▷ A → A) → A
fix f = f (dfix f)
```

Partial Functions

What follows is just a simple helper module to model environments as partial functions with finite support.

```
{-# OPTIONS -cubical #-}
module PartialFunction where

open import Cubical.Relation.Nullary.Base
open import Cubical.Foundations.Prelude hiding (_[_↦_])
open import Cubical.Data.Empty.Base
open import Data.Maybe
open import Data.List
open import Data.Product
open import Function.Base

_⇀_ : ∀ {ℓ} → Set ℓ → Set ℓ → Set ℓ
A ⇀ B = A → Maybe B
infix 1 _⇀_

empty-pfun : ∀{A B : Set} → A ⇀ B
empty-pfun _ = nothing

_[_↦_] : ∀ {A B : Set} {{dec : {x y : A} → Dec (x ≡ y)}}
           → (A ⇀ B) → A → B → (A ⇀ B)
_[_↦_] {{dec}} ρ x b y with dec {x} {y}
... | yes _ = just b
... | no  _ = ρ y

_[_↦*_] : ∀ {A B : Set} {{dec : {x y : A} → Dec (x ≡ y)}}
            → (A ⇀ B) → List A → List B → (A ⇀ B)
_[_↦*_] {A} {B} {{dec}} ρ xs as = aux (Data.List.zip xs as)
  where
    aux : List (A × B) → (A ⇀ B)
    aux [] y = ρ y
    aux ((x , b) :: xs) y with dec {x} {y}
    ... | yes _ = just b
    ... | no _ = aux xs y

pmap : ∀ {A B : Set} → (A ⇀ B) → List A ⇀ List B
```

```
pmap f [] = just []
pmap {_} {B} f (a :: as) with f a
... | nothing = nothing
... | just b  = aux b (pmap f as)
  where
    aux : B → Maybe (List B) → Maybe (List B)
    aux b nothing = nothing
    aux b (just bs) = just (b :: bs)
```

Syntax

What follows is a straightforward encoding of expression syntax Exp. I use natural numbers to identify variables and constructors, because it is simpler than using strings in Agda.

```
{-# OPTIONS -cubical #-}
open import Cubical.Core.Everything hiding (_[_↦_])
open import Cubical.Foundations.Prelude hiding (_[_↦_])
open import Cubical.Data.Nat
open import Cubical.Relation.Nullary.Base
open import Data.List
open import Data.Product
open import Data.Maybe
open import Data.Bool

Var = ℕ
Con = ℕ

decEq-ℕ : (x y : ℕ) → Dec (x ≡ y)
decEq-ℕ zero zero = yes refl
decEq-ℕ zero (suc y) = no znots
decEq-ℕ (suc y) zero = no snotz
decEq-ℕ (suc x) (suc y) with decEq-ℕ x y
... | yes p = yes (cong suc p)
... | no np = no (λ p → np (injSuc p))

instance
  decEq-ℕ-imp : {x y : ℕ} → Dec (x ≡ y)
```

```
  decEq-ℕ-imp {x} {y} = decEq-ℕ x y

Alt : Set

data Exp : Set where
  ref : Var → Exp
  lam : Var → Exp → Exp
  app : Exp → Var → Exp
  let' : Var → Exp → Exp → Exp
  conapp : Con → List Var → Exp
  case' : Exp → List Alt → Exp

Alt = Con × List Var × Exp

findAlt : Con → List Alt → Maybe (List Var × Exp)
findAlt _ [] = nothing
findAlt K ((K' , vs , rhs) :: xs) with decEq-ℕ K K'
... | yes _ = just (vs , rhs)
... | no _ = findAlt K xs
```

Denotational Interpreter

Finally, I can define the generic denotational interpreter from Figure 4.5 in Agda. I do so without defining any concrete instances; the ByName and ByNeed variants will follow in another module.

```
{-# OPTIONS -cubical -guarded #-}
module Semantics where

open import Later
open import Syntax
open import Data.Nat
open import Data.String
open import Data.List as List
open import Data.List.Membership.Propositional
open import Data.Maybe hiding (_»=_)
open import Data.Sum
open import Data.Product
```

```
open import Data.Bool hiding (T)
open import Function
open import PartialFunction
open import Cubical.Foundations.Prelude hiding (_[_↦_])
open import Cubical.Core.Everything hiding (_[_↦_])
open import Cubical.Relation.Nullary.Base
```

First I define the Event data type and the type class definitions for Trace, Domain and HasBind. Note the use of $\Sigma\ D\ p$ in fun, apply and select to characterise the subtype of denotations that will end up in the environment. Also mind the use of the later modality in step as well as bind.

```
data Event : Set where
  look : Var → Event
  update : Event
  app1 : Event
  app2 : Event
  case1 : Event
  case2 : Event
  let1 : Event

record Trace (T : Set) : Set where
  field
    step : Event → ▷ T → T
open Trace {{...}} public

record Domain (D : Set) (p : D → Set) : Set where
  field
    stuck : D
    fun : (Σ D p → D) → D
    apply : D → Σ D p → D
    con : Var → List (Σ D p) → D
    select : D → List (Var × (List (Σ D p) → D)) → D
open Domain {{...}} public

record HasBind (D : Set) : Set where
  field
    bind : ▷(▷ D → D) → (▷ D → D) → D
open HasBind {{...}} public
```

I will instantiate this predicate with the following predicate is-env, which simply expresses that any d that ends up in an environment must be of the form step (look x) $d▷$ for some x and $d▷$.

```
is-env : ∀ {D} {{trc : Trace D}} → D → Set
is-env {D} d = ∃[ x ] ∃[ d▷ ] (d ≡ step {D} (look x) d▷)
```

And finally, I can encode S⟦_⟧ in this type class algebra, pretty much as in Figure 4.5. The definition differs in three ways:

- I need to prove is-env when a let binding introduces new bindings to the environment.
- I omit tests comparing data constructor arity because that is not particularly interesting here; the mismatching cases would just return stuck.
- The definition is a bit more involved than in Haskell because of the diligent passing of Ticks. This is in order to convince Agda that $\mathcal{S}⟦\_⟧$ is productive by construction, so that no separate proof of totality is necessary.

```
𝒮⟦_⟧_ : ∀ {D} {{_ : Trace D}} {{_ : Domain D is-env}} {{_ : HasBind D}}
      → Exp → (Var ⇀ Σ D is-env) → D
𝒮⟦_⟧_ {D} e ρ = fix sem e ρ
  where
    sem : ▹(Exp → (Var ⇀ Σ D is-env) → D) → Exp → (Var ⇀ Σ D is-env) → D
    sem recurse▹ (ref x) ρ with ρ x
    ... | nothing = stuck
    ... | just (d , _) = d
    sem recurse▹ (lam x body) ρ =
      fun (λ d → step app2 (λ α → recurse▹ α body (ρ [ x ↦ d ])))
    sem recurse▹ (app e x) ρ with ρ x
    ... | nothing = stuck
    ... | just d     = step app1 (λ α → apply (recurse▹ α e ρ) d)
    sem recurse▹ (let' x e₁ e₂) ρ =
      bind  (λ α d₁ →
                recurse▹ α e₁ (ρ [ x ↦ (step (look x) d₁ , x , d₁ , refl) ]))
            (λ d₁ → step let1 (λ α →
                recurse▹ α e₂ (ρ [ x ↦ (step (look x) d₁ , x , d₁ , refl) ])))
    sem recurse▹ (conapp K xs) ρ with pmap ρ xs
    ... | nothing = stuck
    ... | just ds = con K ds
    sem recurse▹ (case' eₛ alts) ρ =
      step case1 (λ α → select (recurse▹ α eₛ ρ) (List.map alt alts))
        where
          alt : Con × List Var × Exp → Con × (List (Σ D is-env) → D)
          alt (k , xs , eᵣ) = (k , (λ ds →
            step case2 (λ α → recurse▹ α eᵣ (ρ [ xs ↦* ds ]))))
```

Concrete domain instances ByName, ByNeed

Separately from the denotational interpreter, we can prove that its instances at ByName and ByNeed are well-defined as well.

In order to do so, I first need to define the concrete type D, which needs the concrete trace type T as well as the concrete value type Value.

```
{-# OPTIONS --cubical --guarded --rewriting #-}

-- | Definitions and instances for T, Value, D, ByName, ByNeed
module Concrete where

open import Later
open import Syntax
open import Data.Nat
open import Data.String
open import Data.List as List
open import Data.List.Membership.Propositional
open import Data.Maybe hiding (_>>=_)
open import Data.Unit
open import Data.Sum
open import Data.Product
open import Data.Bool hiding (T; _∧_; _∨_)
open import Function
open import PartialFunction
open import Cubical.Foundations.Prelude hiding (_[_↦_])
open import Cubical.Foundations.Isomorphism
open import Cubical.Foundations.Transport
open import Cubical.Core.Everything hiding (_[_↦_])
open import Cubical.Relation.Nullary.Base
open import Agda.Builtin.Equality renaming (_≡_ to _≡≡_) hiding (refl)
open import Agda.Builtin.Equality.Rewrite
open import Semantics

record Monad (M : Set → Set) : Set₁ where
  field
    return : ∀ {A} → A → M A
    _>>=_ : ∀ {A} {B} → M A → (A → M B) → M B
  _>>_ : ∀ {A} {B} → M A → M B → M B
```

```
  l » r = l >>= (λ _ → r)

open Monad {{...}} public

data T (A : Set) : Set where
  ret-T : A → T A
  step-T : Event → ▹ T A → T A

Value : (Set → Set) → Set
D : (Set → Set) → Set
```

As explained in Section 4.4.5, a notable difference to the definition of Value in the main body is that I need to break the negative occurrence in fun by the use of dependent *later* ▸. This embedding is abstracted into the following type EnvD:

```
data EnvD (D : ▹ Set) : Set where
  stepLook : Var → ▸ D → EnvD D
```

Note that EnvD D is effectively the subtype of D of denotations that go into the environment ρ. One should think of stepLook x d' as a d such that d = step (look x) d'.

Actually, I would prefer to simply *express* the subtyping relationship via Σ D is-env, as in the type of fun, but the use of is-env : D T → Set requires an instance of Trace (D T) *in the type* of fun-V, leading to a circular definition of ValueF.

Defining the bijection to EnvD is easy, enough, though:

```
toSubtype : ∀ {D} {{_ : Trace D}} → EnvD (next D) → Σ D is-env
toSubtype {{_}} (stepLook x d▷) = (step (look x) d▷ , x , d▷ , refl)

fromSubtype : ∀ {D} {{_ : Trace D}} → Σ D is-env → EnvD (next D)
fromSubtype {{_}} (_ , x , d▷ , _) = stepLook x d▷
```

I can also prove that the pair indeed forms a bijection:

```
env-iso : ∀ {D} {{_ : Trace D}} → Iso (EnvD (next D)) (Σ D is-env)
env-iso = iso toSubtype fromSubtype from-to to-from
  where
    from-to : ∀ d → toSubtype (fromSubtype d) ≡ d
    from-to (d , x , d▷ , prf) i = (prf (~ i) , x , d▷ , λ i₁ → prf (i₁ ∨ (~ i)))
```

```
    to-from : ∀ d → fromSubtype (toSubtype d) ≡ d
    to-from (stepLook x d▷) = refl
```

Next up is the definition of Value, which is complicated by the fact that Agda's positivity checker has no builtin support for the later modality, so Value needs to be defined in terms of the guarded fixpoint of the signature functor ValueF defined below. I still need to turn off the positivity checker because of the recursion through τ, which however will always be instantiated with a positive functor. An alternative without this pragma would be to monomorphise ValueF for ByName and ByNeed separately.

```
{-# NO_POSITIVITY_CHECK #-}
data ValueF (τ : Set → Set) (d⁻ : ▹ Set) : Set where
  stuck-V : ValueF τ d⁻
  fun-V : (EnvD d⁻ → (D τ)) → ValueF τ d⁻
  con-V : Con → List (EnvD d⁻) → ValueF τ d⁻

Value τ = ValueF τ (dfix (τ ∘ ValueF τ))
D τ = τ (Value τ)
```

Is is easy to verify that D is the guarded fixpoint of $\tau \circ$ ValueF τ:

```
_ : ∀ {τ} → D τ ≡ fix (τ ∘ ValueF τ)
_ = refl
```

It is not completely obvious that the EnvDs that occur in a Value are still isomorphic to the subtype Σ D is-env.

```
EnvD≡is-env : ∀ τ → {{_ : Trace (D τ)}}
                  → EnvD (dfix (τ ∘ ValueF τ)) ≡ Σ (D τ) is-env
EnvD≡is-env τ = roll · subty
  where
    roll : EnvD (dfix (τ ∘ ValueF τ)) ≡ EnvD (next (D τ))
    roll i = EnvD (pfix (τ ∘ ValueF τ) i)
    subty : EnvD (next (D τ)) ≡ Σ (D τ) is-env
    subty = isoToPath (env-iso {D τ})
```

This equivalence is used to great effect in the type class instance for Domain, which otherwise is exactly as in Section 4.3.

```
return-T : ∀ {A} → A → T A
return-T = ret-T

_»=-T_ : ∀ {A} {B} → T A → (A → T B) → T B
ret-T a »=-T k = k a
step-T e τ »=-T k = step-T e (λ α → τ α »=-T k)

instance
  monad-T : Monad T
  monad-T = record { return = ret-T; _»=_ = _»=-T_ }

instance
  trace-T : ∀ {V} → Trace (T V)
  trace-T = record { step = step-T }

stuck-Value : ∀ {τ} {{_ : Monad τ}} → D τ
stuck-Value = return stuck-V

fun-Value : ∀ {τ} {{_ : Monad τ}} {{_ : Trace (D τ)}}
          → (Σ (D τ) is-env → D τ) → D τ
fun-Value {τ} f = return (fun-V (f ∘ transport (EnvD≡is-env τ)))

apply-Value : ∀ {τ} {{_ : Monad τ}} {{_ : Trace (D τ)}}
            → D τ → Σ (D τ) is-env → D τ
apply-Value {τ} dv da = dv »= aux
  where
    aux : Value τ → D τ
    aux (fun-V f) = f (transport⁻ (EnvD≡is-env τ) da)
    aux _ = stuck-Value

con-Value : ∀ {τ} {{_ : Monad τ}} {{_ : Trace (D τ)}}
          → Con → List (Σ (D τ) is-env) → D τ
con-Value {τ} K ds = return (con-V K (List.map (transport⁻ (EnvD≡is-env τ)) ds))

select-Value : ∀ {τ} {{_ : Monad τ}} {{_ : Trace (D τ)}}
             → D τ → List (Con × (List (Σ (D τ) is-env) → D τ)) → D τ
select-Value {τ} dv alts = dv »= aux alts
  where
    aux : List (Con × (List (Σ (D τ) is-env) → D τ)) → Value τ → D τ
    aux ((K' , alt) :: alts) (con-V K ds) with decEq-ℕ K K'
```

```
    ... | yes _ = alt (List.map (transport (EnvD≡is-env τ)) ds)
    ... | no _  = aux alts (con-V K ds)
    aux _ _ = stuck-Value

instance
  domain-Value : ∀ {τ} {{_ : Monad τ}} {{_ : Trace (D τ)}} → Domain (D τ) is-env
  domain-Value = record { stuck = stuck-Value;
                          fun = fun-Value; apply = apply-Value;
                          con = con-Value; select = select-Value }
```

This suffices to define the ByName interpreter. The instance of HasBind is particularly interesting, because it again employs the guarded fixpoint combinator fix:

```
record ByName (τ : Set → Set) (v : Set) : Set where
  constructor mkByName
  field get : τ v

instance
  monad-ByName : ∀ {τ} {{_ : Monad τ}} → Monad (ByName τ)
  monad-ByName =
    record { return = mkByName ∘ return;
             _»=_ = λ m k → mkByName (ByName.get m »= (ByName.get ∘ k)) }

instance
  trace-ByName : ∀ {τ} {{_ : ∀ {V} → Trace (τ V)}} {V} → Trace (ByName τ V)
  trace-ByName =
    record { step = λ e τ → mkByName (step e (λ α → ByName.get (τ α))) }

instance
  has-bind-ByName : ∀ {τ} {v} → HasBind (ByName τ v)
  has-bind-ByName {τ} =
    record { bind = λ rhs body → body (λ α → fix (λ rhs▸ → rhs α rhs▸)) }

eval-by-name : Exp → D (ByName T)
eval-by-name e = 𝒮⟦ e ⟧ empty-pfun
```

For the ByNeed instance, I need to define heaps. Heaps represent higher-order state, the total modelling of which is one of the main motivations for guarded type theory. As such, the heap is also the place where I need to break another

negative recursive occurrence through the use of the *later* modality, this time without overriding the totality checker.

Furthermore, I postulate the existence of a bump allocator nextFree as well as the well-addressedness invariant from Section 4.2, that is, any address allocated is in the domain of the heap. It would take a few tiresome and distracting invariants to turn these postulates into proofs, which is why it was not done.

```
Addr : Set
Addr = ℕ

record ByNeed (τ : Set → Set) (v : Set) : Set

Heap : ▹ Set → Set
Heap D = Addr ⇀ ▸ D
postulate nextFree : ∀ {D} → Heap D → Addr
postulate well-addressed : ∀ {D} (μ : Heap D) (a : Addr) → ∃[ d ] (μ a ≡ just d)
```

The definition of ByNeed and its type class instances are structurally the same as in the main body. However, the guarded fixpoint again requires explicit unrolling whenever the heap is accessed, for which I need to establish and transport along a few equalities. Furthermore, in step-ByNeed I need to pass around the Tick variable α.

```
ByNeedF : (Set → Set) → ▹ Set → Set → Set
ByNeedF τ d⁻ v = Heap d⁻ → τ (v × Heap d⁻)

record ByNeed τ v where
  constructor mkByNeed
  field get : ByNeedF τ (dfix (D ∘ ByNeedF τ)) v

≡-ByNeed : ∀ τ v → ByNeed τ v ≡ ByNeedF τ (dfix (D ∘ ByNeedF τ)) v
≡-ByNeed _ _ = isoToPath (iso ByNeed.get mkByNeed (λ _ → refl) (λ _ → refl))

≡-HeapD : ∀ τ → dfix (D ∘ ByNeedF τ) ≡ next (D (ByNeed τ))
≡-HeapD τ = pfix (D ∘ ByNeedF τ) · (λ i → next (D (λ v → sym (≡-ByNeed τ v) i)))

≡-▸HeapD : ∀ τ → ▸ dfix (D ∘ ByNeedF τ) ≡ ▹ D (ByNeed τ)
≡-▸HeapD τ i = ▸ ≡-HeapD τ i
```

```
≡-DByNeed :
  ∀ τ → D (ByNeed τ) ≡ ByNeedF τ (next (D (ByNeed τ))) (Value (ByNeed τ))
≡-DByNeed τ = ≡-ByNeed τ (Value (ByNeed τ))
                · (λ i → ByNeedF τ (≡-HeapD τ i) (Value (ByNeed τ)))

return-ByNeed : ∀ {τ} {{_ : Monad τ}} {v} → v → ByNeed τ v
return-ByNeed v = mkByNeed (λ μ → return (v , μ))

_»=-ByNeed_ : ∀ {τ} {{_ : Monad τ}} {a} {b}
                → ByNeed τ a → (a → ByNeed τ b) → ByNeed τ b
_»=-ByNeed_ {τ} {a} {b} m k = mkByNeed (λ μ → ByNeed.get m μ »= aux)
  where
    aux : (a × Heap (dfix (D ∘ ByNeedF τ))) → τ (b × Heap (dfix (D ∘ ByNeedF τ)))
    aux (a , μ') = ByNeed.get (k a) μ'

instance
  monad-ByNeed : ∀ {τ} {{_ : Monad τ}} → Monad (ByNeed τ)
  monad-ByNeed = record { return = return-ByNeed; _»=_ = _»=-ByNeed_ }

step-ByNeed : ∀ {τ} {v} {{_ : ∀ {V} → Trace (τ V)}}
                → Event → ▸(ByNeed τ v) → ByNeed τ v
step-ByNeed {τ} {v} e m = mkByNeed (λ μ → step e (λ α → ByNeed.get (m α) μ))

instance
  trace-ByNeed : ∀ {τ} {v} {{_ : ∀ {V} → Trace (τ V)}} → Trace (ByNeed τ v)
  trace-ByNeed = record { step = step-ByNeed }
```

The next step is to define fetch, the function that accesses the heap. Unfortunately, my definition needs to appeal to a postulate that would generally be unsafe to use. To see why this postulate is necessary and why my use of it is actually safe, consider the following definition:

```
stepLookFetch : ∀ {τ} {{_ : Monad τ}} {{_ : ∀ {V} → Trace (τ V)}}
                  → Var → Addr → D (ByNeed τ)
stepLookFetch {τ} x a = mkByNeed (λ μ →
  let d▸ = fst (well-addressed μ a) in
  step (look x) (λ α → ByNeed.get (transport (≡-▸HeapD τ) d▸ α) μ))
```

(Note that fst (well-addressed μ a) simply returns the heap entry in μ at address a, which must be present by my assumption of well-addressedness.)

This definition constructs the total Agda equivalent of the Haskell expression *step* (Look *x*) (*fetch a*), for the given variable *x* and address *a*. Ultimately, all denotations in the interpreter environment ρ will take this form under by-need evaluation. (In Definition 4.29 I define an even sharper characterisation.) In fact, *all* uses of fetch will take this form!

Unfortunately, it is hard to decompose stepLookFetch into separate function calls to step and fetch : Addr → ▷(D (ByNeed T)), because the latter will then need to bind the tick variable α (part of ▷) before the heap μ (part of D (ByNeed T)). This is in contrast to the order of binders in stepLookFetch, which may bind μ before α, because look steps leave the heap unchanged. (See step-ByNeed above for confirmation, which is inlined into stepLookFetch).

The flipped argument order is problematic for my definition of fetch, because ticked type theory conservatively assumes that μ might depend on α — when in reality it does not in stepLookFetch! The result is that the subexpression ByNeed.get ($d \triangleright\ \alpha$) μ would not be well-typed under the flipped order, because

- $d\triangleright$ comes from μ, and
- μ might already depend on α, because
- μ was introduced after α, and hence
- $d\triangleright$ may not be applied to α again in ticked type theory.

I currently know of no way to encode this knowledge without a postulate of the following form:

```
postulate
  flip-tick      : ∀ {A B : Set} → (A → ▷ B) → ▷ (A → B)
  flip-tick-beta : ∀ {A B : Set} {f : A → ▷ B} {μ : A} {@tick α : Tick}
                   → flip-tick f α μ ≡≡ f μ α
{-# REWRITE flip-tick-beta #-}
```

It is most helpful to look at the postulated "implementation rule" flip-tick-beta to see when use of flip-tick is safe: Given some f and a heap μ that *does not depend* on some tick variable α, call f with μ first instead of α. So flip-tick literally flips around the arguments it receives before calling f, and unless μ does not depend on α, the application of flip-tick is stuck because the rule does not apply.

I use flip-tick in the implementation of fetch exactly to flip back the binding order to what it will be in the use site stepLookFetch:

```
fetch : ∀ {τ} {{_ : Monad τ}} → Addr → ▹(D (ByNeed τ))
fetch {τ} a = map▹ mkByNeed (flip-tick (λ μ →
  let d▹ = fst (well-addressed μ a) in
  (λ α → ByNeed.get (transport (≡-▸HeapD τ) d▹ α) μ)))
```

Agda is able to calculate that this definition of fetch is equivalent to the one inlined into stepLookFetch:

```
postulate-ok : ∀ {τ x a} {{_ : Monad τ}} {{_ : ∀ {V} → Trace (τ V)}}
                 → step (look x) (fetch {τ} a) ≡ stepLookFetch x a
postulate-ok = refl
```

(Note that this proof automatically applies flip-tick-beta by the REWRITE pragma above.)

This should be sufficient justification for my use of flip-tick. The definition of memo is a bit more involved but does not need any postulates at all:

```
memo : ∀ {τ} {{_ : Monad τ}} {{_ : ∀ {V} → Trace (τ V)}}
       → Addr → ▹(D (ByNeed τ)) → ▹(D (ByNeed τ))
memo {τ} a d▹ = fix memo' d▹
  where
    memo' : ▹(▹(D (ByNeed τ)) → ▹(D (ByNeed τ)))
            → ▹(D (ByNeed τ)) → ▹(D (ByNeed τ))
    memo' rec▹ d▹ α₁ = do
      v ← d▹ α₁
      step update (λ _α₂ → mkByNeed (λ μ →
        return (v , μ [ a ↦ transport⁻ (≡-▸HeapD τ)
                                       (rec▹ α₁ (λ _ → return v)) ])))
```

Building on fetch and memo, I define the HasBind instance as follows

```
bind-ByNeed : ∀ {τ} {{_ : Monad τ}} {{_ : ∀ {V} → Trace (τ V)}}
              → ▹ (▹(D (ByNeed τ)) → D (ByNeed τ))
              →   (▹(D (ByNeed τ)) → D (ByNeed τ))
              →   D (ByNeed τ)
bind-ByNeed {τ} rhs body = do
  a ← mkByNeed (λ μ → return (nextFree μ , μ))
  mkByNeed (λ μ →
    return (tt , μ [ a ↦ transport⁻ (≡-▸HeapD τ)
```

```
                                        (memo a (λ α → rhs α (fetch a))) ]))
  step let1 (λ _α → body (fetch a))

instance
  has-bind-ByNeed : ∀ {τ} {{_ : Monad τ}} {{_ : ∀ {V} → Trace (τ V)}}
                    → HasBind (D (ByNeed τ))
  has-bind-ByNeed = record { bind = bind-ByNeed }

eval-by-need : Exp → T (Value (ByNeed T) × Heap (next (D (ByNeed T))))
eval-by-need e = transport (≡-DByNeed T) (𝒮⟦ e ⟧ empty-pfun) empty-pfun
```

This completes the definition of eval-by-need which is thus proven total.

C

Denotational Interpreter for GHC Core in Section 4.5.5

The following code lists the main module of my fork of GHC[1], relative to a development version of GHC 9.10[2]. It implements the denotational interpreter for GHC Core, and GHC's Demand Analysis can be made an instance of this interpeter.

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
{-# LANGUAGE QuantifiedConstraints #-}
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE LambdaCase #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE UndecidableInstances #-}
module GHC.Core.Semantics where

import GHC.Prelude

import GHC.Core
import GHC.Core.Coercion
import GHC.Core.DataCon

import qualified GHC.Data.Word64Map as WM

import GHC.Types.Literal
import GHC.Types.Id
import GHC.Types.Name
import GHC.Types.Var.Env
import GHC.Types.Unique.Set
```

[1] https://gitlab.haskell.org/ghc/ghc/-/tree/33f4bdd
[2] https://gitlab.haskell.org/ghc/ghc/-/tree/1350345

```
import GHC.Utils.Misc
import GHC.Utils.Outputable
import Control.Monad
import Control.Monad.Trans.State
import Data.Word
import GHC.Core.Utils hiding (findAlt)
import GHC.Core.Type
import GHC.Builtin.PrimOps
import GHC.Builtin.Types
import GHC.Types.Var
import GHC.Core.TyCo.Rep
import GHC.Core.FVs
import GHC.Core.Class
import GHC.Types.Id.Info
import GHC.Types.Unique
import GHC.Builtin.Names

data Event
  = Look Id
  | LookArg CoreExpr
  | Update
  | App1
  | App2
  | Case1
  | Case2
  | Let1

class Trace d where
  step :: Event → d → d

type DAlt d = (AltCon, [Id], d → [d] → d)

class Domain d where
  stuck :: d
  erased :: d
  lit :: Literal → d
  global :: Id → d
  classOp :: Id → Class → d
  primOp :: Id → PrimOp → d
  fun :: Var → (d → d) → d
```

```
  con :: DataCon → [d] → d
  apply :: d → (Bool, d) → d
  select :: d → CoreExpr → Id → [DAlt d] → d
  keepAlive :: [d] → d → d

data BindHint
  = BindArg Id
  | BindLet CoreBind

class HasBind d where
  bind :: BindHint → ([d] → d] → ([d] → d) → d

keepAliveVars :: Domain d ⇒ [Id] → IdEnv d → d → d
keepAliveVars xs ρ
  | Just ds ← traverse (lookupVarEnv ρ) xs = keepAlive ds
  | otherwise = const stuck

keepAliveCo :: Domain d ⇒ Coercion → IdEnv d → d → d
keepAliveCo co = keepAliveVars (nonDetEltsUniqSet $ coVarsOfCo co)

keepAliveUnfRules :: Domain d ⇒ Id → IdEnv d → d → d
keepAliveUnfRules x =
  keepAliveVars (nonDetEltsUniqSet $ bndrRuleAndUnfoldingIds x)

feignBndr :: Name → PiTyBinder → Var
feignBndr n (Anon (Scaled mult ty) _) = mkLocalIdOrCoVar n mult ty
feignBndr n (Named (Bndr tcv _))      = tcv `setVarName` n

feignId :: Name → Type → Id
feignId n ty = mkLocalIdOrCoVar n ManyTy ty

mkPap :: (Trace d, Domain d) ⇒ [PiTyBinder] → ([d] → d) → d
mkPap arg_bndrs app_head =
  go [] (zipWith feignBndr localNames arg_bndrs)
  where
    go ds [] = app_head (reverse ds)
    go ds (x : xs) = fun x (λd → step App2 $ go (d : ds) xs)

x1, x2 :: Name
localNames :: [Name]
localNames@(x1 : x2: _) =
  [mkSystemName (mkUniqueInt 'I' i) (mkVarOcc "local") | i ← [0..]]

anfise
  ::    (Trace d, Domain d, HasBind d)
```

```
    ⇒ [CoreExpr] → IdEnv d → ([d] → d) → d
anfise es ρ k = go (zip localNames es) []
  where
    go [] ds = k (reverse ds)
    go ((x, e) : es) ds = anf_one x e ρ $ λd → go es (d : ds)
    anf_one _ (Lit l) _ k = k (lit l)
    anf_one _ (Var x) ρ k = evalVar x ρ k
    anf_one _ (Coercion co) ρ k = keepAliveCo co ρ (k erased)
    anf_one _ (Type _ty) _ k = k erased
    anf_one x (Tick _t e) ρ k = anf_one x e ρ k
    anf_one x (Cast e co) ρ k = keepAliveCo co ρ (anf_one x e ρ k)
    anf_one x e ρ k =
      bind (BindArg (feignId x e_ty)) [\_ → 𝒮⟦e⟧ρ]
        (λds → let d = step (LookArg e) (only ds) in
               if isUnliftedType e_ty && not (exprOkForSpeculation e)
                 then seq_ (d, e, e_ty) (k d)
                 else k d)
      where
        e_ty = exprType e
        seq_ :: Domain d ⇒ (d, CoreExpr, Type) → d → d
        seq_ (a, e, ty) b = select a e wildCardId [(DEFAULT, [], \_a _ds → b)]
          where
            wildCardId :: Id
            wildCardId =
              feignBndr wildCardName (Anon (Scaled ManyTy ty) FTF_T_T)
evalConApp :: (Trace d, Domain d, HasBind d) ⇒ DataCon → [d] → d
evalConApp dc args = case compareLength args rep_ty_bndrs of
  EQ  → con dc args
  GT  → stuck
  LT  → mkPap rest_bndrs $ λetas → con dc (args ++ etas)
  where
    rep_ty_bndrs = fst $ splitPiTys (dataConRepType dc)
    rest_bndrs = dropList args rep_ty_bndrs
evalVar :: (Trace d, Domain d, HasBind d) ⇒ Var → IdEnv d → (d → d) → d
evalVar x ρ k = case idDetails x of
  _ | isTyVar x          → k erased
  DataConWorkId dc → k (evalConApp dc [])
```

```
    DataConWrapId _  → k (S⟦unfoldingTemplate (idUnfolding x)⟧_emptyVarEnv)
    PrimOpId op _    → k (primOp x op)
    ClassOpId cls _  → k (classOp x cls)
    _ | isGlobalId x → k (global x)
    _                → maybe stuck k (lookupVarEnv ρ x)
S⟦_⟧_ :: (Trace d, Domain d, HasBind d) ⇒ CoreExpr → IdEnv d → d
S⟦Coercion co⟧ρ = keepAliveCo co ρ erased
S⟦Type _ty⟧_ = erased
S⟦Lit l⟧_ = lit l
S⟦Tick _t e⟧ρ = S⟦e⟧ρ
S⟦Cast e co⟧ρ = keepAliveCo co ρ (S⟦e⟧ρ)
S⟦Var x⟧ρ = evalVar x ρ id
S⟦Lam x e⟧ρ =
  fun x (λd → step App2 (S⟦e⟧_extendVarEnv ρ x d))
S⟦e⟧_@App { } ρ
  | Var v ← f, Just dc ← isDataConWorkId_maybe v
  = anfise as ρ (evalConApp dc)
  | otherwise
  = anfise (f : as) ρ $ λ(df : das) →
      go df (zipWith (λd a → (d, isTypeArg a)) das as)
  where
    (f, as) = collectArgs e
    go df [ ] = df
    go df ((d, is_ty) : ds) = go (step App1 $ apply df (is_ty, d)) ds
S⟦Let b@(NonRec x rhs) body⟧ρ =
  bind (BindLet b)
       [\_ → keepAliveUnfRules x ρ $
             S⟦rhs⟧ρ]
       (λds → step Let1 $
              S⟦body⟧_extendVarEnv ρ x (step (Look x) (only ds)))
S⟦Let b@(Rec binds) body⟧ρ =
  bind (BindLet b)
       [λds → keepAliveUnfRules x (new_ρ ds) $
              S⟦rhs⟧_new_ρ ds | (x, rhs) ← binds]
       (λds → step Let1 (S⟦body⟧_new_ρ ds))
  where
    xs = map fst binds
```

```
    new_ρ ds = extendVarEnvList ρ $
      zipWith (λx d → (x, step (Look x) d)) xs ds
𝒮⟦Case e b _ty alts⟧_ρ = step Case₁ $
  select      (𝒮⟦e⟧_ρ) e b
              [(con, xs, cont xs rhs) | Alt con xs rhs ← alts]
  where
    cont xs rhs scrut ds = step Case₂ $ 𝒮⟦rhs⟧_ $ extendVarEnvList ρ $
      zipEqual "eval Case{}" (b : xs) (scrut : ds)

data T v = Step Event (T v) | Ret v
  deriving Functor
instance Applicative T where
  pure = Ret
  (<*>) = ap
instance Monad T where
  Ret a >>= f = f a
  Step ev t >>= f = Step ev (t >>= f)
instance Trace (T v) where
  step = Step

type D τ = τ (Value τ)
data Value τ
  = Stuck
  | Erased
  | Litt Literal
  | Fun (D τ → D τ)
  | Con DataCon [D τ]

instance (Trace (D τ), Monad τ) ⇒ Domain (D τ) where
  stuck = return Stuck
  lit l = return (Litt l)
  fun _x f = return (Fun f)
  con k ds = return (Con k ds)
  apply d (_b, a) = d >>= λcase Fun f → f a; _ → stuck
  select d _f _b fs = d >>= λv → case v of
    Stuck                                          → stuck
    Con k ds | Just (_,_,f) ← findAlt (DataAlt k) fs → f (return v) ds
    Litt l   | Just (_,_,f) ← findAlt (LitAlt l)  fs → f (return v) []
    _        | Just (_,_,f) ← findAlt DEFAULT fs    → f (return v) []
    _                                              → stuck
```

```
  global _          = stuck
  classOp _x _cls = stuck
  primOp _x op = case op of
    IntAddOp → intop (+)
    IntMulOp → intop (*)
    IntRemOp → intop rem
    _        → stuck
    where
      intop op = binop int_ty int_ty $ λv1 v2 → case (v1, v2) of
        (Litt (LitNumber LitNumInt i1), Litt (LitNumber LitNumInt i2))
          → Litt (LitNumber LitNumInt (i1 `op` i2))
        _ → Stuck
      binop ty1 ty2 f = mkPap [ty1, ty2] $ λ[d1, d2] → f <$> d1 <*> d2
      int_ty = Anon (Scaled ManyTy intTy) FTF_T_T
  erased = return Erased
  keepAlive _ d = d

findAlt :: AltCon → [DAlt d] → Maybe (DAlt d)
findAlt con alts
  = case alts of
      (deflt@(DEFAULT, _, _) : alts) → go alts (Just deflt)
      _ → go alts Nothing
  where
    go [] deflt = deflt
    go (alt@(con1, _, _) : alts) deflt
      = case con `cmpAltCon` con1 of
          LT → deflt
          EQ → Just alt
          GT → go alts deflt

type Addr = Word64
type Heap τ = WM.Word64Map (D τ)
newtype ByNeed τ v = ByNeed {runByNeed :: StateT (Heap (ByNeed τ)) τ v}
  deriving (Functor, Applicative, Monad)

instance (∀v. Trace (τ v)) ⇒ Trace (ByNeed τ v) where
  step ev (ByNeed (StateT m)) = ByNeed $ StateT $ step ev ∘ m

fetch :: Monad τ ⇒ Addr → D (ByNeed τ)
fetch a = ByNeed get ≫= λμ → μWM.! a
```

```
memo :: ∀τ. (Monad τ, ∀ν. Trace (τ ν))
     ⇒ Addr → D (ByNeed τ) → D (ByNeed τ)
memo a d = d >>= ByNeed ∘ StateT ∘ upd
  where upd Stuck μ = return (Stuck :: Value (ByNeed τ), μ)
        upd v   μ = step Update (return (v, WM.insert a (memo a (return v)) μ))

freeList :: Heap τ → [Addr]
freeList μ = [a..]
  where a = case WM.lookupMax μ of Just (a, _) → a + 1; _ → 0

instance (Monad τ, ∀ν. Trace (τ ν)) ⇒ HasBind (D (ByNeed τ)) where
  bind _hint rhss body = do
    as ← take (length rhss) ∘ freeList <$> ByNeed get
    let ds = map fetch as
    ByNeed $ modify (λμ → foldr (λ(a, rhs) →
      WM.insert a (memo a (rhs ds))) μ (zip as rhss))
    body ds

evalByNeed :: CoreExpr → T (Value (ByNeed T), Heap (ByNeed T))
evalByNeed e = runStateT (runByNeed (𝒮⟦e⟧_emptyVarEnv)) WM.empty
```

D

Extracted Haskell code for Chapter 4

Here I list the complete code for Chapter 4.

The following module defines the expression data type and its helpers:

```
module Exp where

import qualified Data.Map as Map

type Name = String  -- [a-z][a-zA-Z0-9]+
data Tag
    = FF | TT | None | Some | Pair | S | Z
  deriving (Show, Read, Eq, Ord, Enum, Bounded)

conArity :: Tag → Int
conArity Pair = 2
conArity Some = 1
conArity S = 1
conArity _ = 0

data Exp
    = Var Name
    | App Exp Name
    | Lam Name Exp
    | Let Name Exp Exp
    | ConApp Tag [Name]
    | Case Exp Alts
type Alts = Map.Map Tag ([Name], Exp)

type Label = String
label :: Exp → Label
label e = case e of
```

```
  Lam x _ → "\\lambda " ++ x ++ ". "
  ConApp k xs → show k ++ "(" ++
    showSep (showString ",") (map showString xs) [] ++ ")"
  _           → undefined
showSep :: ShowS → [ShowS] → ShowS
showSep _ [] = id
showSep _ [s] = s
showSep sep (s : ss) = s ∘ sep ∘ showString "  " ∘ showSep sep ss
```

The full definitions for Section 4.3 follow:

```
{# LANGUAGE DerivingVia #}
{# LANGUAGE PartialTypeSignatures #}
{# LANGUAGE QuantifiedConstraints #}
{# LANGUAGE UndecidableInstances #}

module Interpreter where

import Data.Map (Map)
import qualified Data.Map as Map
import Data.Set (Set)
import qualified Data.Set as Set
import Data.List (foldl')
import Text.Show (showListWith)
import Data.Functor.Identity
import Control.Applicative
import Control.Monad
import Control.Monad.Fix
import Control.Monad.Trans.State
import Exp

-- Finite maps
type (:⇀) = Map
ε          :: Ord k ⇒ k :⇀ v
_[_ ↦ _]   :: Ord k ⇒ (k :⇀ v) → k → v → (k :⇀ v)
_[_ ↦ _]   :: Ord k ⇒ (k :⇀ v) → [k] → [v] → (k :⇀ v)
(!)        :: Ord k ⇒ (k :⇀ v) → k → v
dom        :: Ord k ⇒ (k :⇀ v) → Set k
(∈)        :: Ord k ⇒ k → Set k → Bool
(◁)        :: (b → c) → (a :⇀ b) → (a :⇀ c)
assocs     :: (k :⇀ v) → [(k, v)]
```

```
ε = Map.empty
ρ[x ↦ d] = Map.insert x d ρ
ρ[xs ↦ ds] = foldl' (uncurry ∘ _[_ ↦ _]) ρ (zip xs ds)
[x ↦ d] = Map.singleton x d
(◁) = Map.map
infixr 9 ◁
(!) = (Map.!)
dom = Map.keysSet
(∈) = Set.member
assocs = Map.assocs
```

```
-- Semantic domain: values and traces
type D τ = τ (Value τ);
data T v = Step Event (T v) | Ret v
```

```
takeT :: Int → T a → T (Maybe a)
takeT 0 _ = return Nothing
takeT _ (Ret a) = return (Just a)
takeT n (Step e t) = Step e (takeT (n − 1) t)
```

```
data Value τ = Stuck | Fun (D τ → D τ) | Con Tag [D τ]
data Event   = Look Name | Upd | App₁ | App₂
             | Let₀ | Let₁ | Case₁ | Case₂
```

```
instance Functor T where
  fmap f (Ret a) = Ret (f a)
  fmap f (Step e t) = Step e (fmap f t)
instance Applicative T where
  pure = Ret
  (<*>) = ap
instance Monad T where
  Ret v >>= k = k v
  Step e τ >>= k = Step e (τ >>= k)
```

```
-- Type class algebra
class Trace d where
  step :: Event → d → d
```

```
class Domain d where
  stuck :: d
  fun :: Name → Label → (d → d) → d
  apply :: d → d → d
```

```
  con :: Label → Tag → [d] → d
  select :: d → (Tag :⇀ ([d] → d)) → d
class HasBind d where
  bind :: Name → (d → d) → (d → d) → d
instance Trace (T v) where
  step = Step
instance Monad τ ⇒ Domain (D τ) where
  stuck = return Stuck
  fun _ _ f = return (Fun f)
  apply  d a = d >>= λv → case v of
    Fun f → f a
    _     → stuck
  con _ k ds = return (Con k ds)
  select dv alts = dv >>= λv → case v of
    Con k ds | k ∈ dom alts → (alts ! k) ds
    _                       → stuck
-- Generic denotational interpreter S
𝒮⟦_⟧_ :: (Trace d, Domain d, HasBind d) ⇒ Exp → (Name :⇀ d) → d
𝒮⟦e⟧ρ = case e of
  Var x | x ∈ dom ρ → ρ ! x
        | otherwise → stuck
  Lam x body → fun x (label e) $ λd → step App2 (𝒮⟦body⟧(ρ[x↦d]))
  App e x | x ∈ dom ρ → step App1 $ apply (𝒮⟦e⟧ρ) (ρ ! x)
          | otherwise → stuck
  Let x e1 e2 → bind x (λd1 → 𝒮⟦e1⟧ρ[x↦step (Look x) d1])
                       (λd1 → step Let1 (𝒮⟦e2⟧ρ[x↦step (Look x) d1]))
  ConApp k xs | all (∈ dom ρ) xs, length xs == conArity k
                → con (label e) k (map (ρ !) xs)
              | otherwise
                → stuck
  Case e alts → step Case1 $ select (𝒮⟦e⟧ρ) (cont ◁ alts)
    where
      cont (xs, er) ds | length xs == length ds = step Case2 (𝒮⟦er⟧ρ[xs↦ds])
                       | otherwise              = stuck
-- By-name semantics
𝒮name⟦_⟧_ :: Exp → (Name :⇀ Dna) → Dna
```

```
𝒮name⟦e⟧ρ = 𝒮⟦e⟧ρ :: Dna
newtype ByName τ v = ByName { unByName :: (τ v) }
  deriving newtype (Functor, Applicative, Monad)
type Dna = D (ByName T)
instance Trace (τ v) ⇒ Trace (ByName τ v) where
  step e = ByName ∘ step e ∘ unByName
instance HasBind (D (ByName τ)) where
  bind _ rhs body = body (fix rhs)
takeName :: Int → ByName T a → T (Maybe a)
takeName n (ByName τ) = takeT n τ
 -- Heaps
type Addr    = Int
type Heap τ = Addr :⇀ D τ
nextFree :: Heap τ → Addr
nextFree h = case Map.lookupMax h of
  Nothing   → 0
  Just (k, _) → k + 1
 -- By-need semantics
𝒮need⟦_⟧_(_) :: Exp → (Name :⇀ Dne) → Heapne → T (Valuene, Heapne)
𝒮need⟦e⟧ρ(μ) = unByNeed (𝒮⟦e⟧ρ :: Dne) μ
newtype ByNeed τ v
   = ByNeed { unByNeed :: Heap (ByNeed τ) → τ (v, Heap (ByNeed τ)) }
type Dne        = D (ByNeed T)
type Valuene    = Value (ByNeed T)
type Heapne     = Heap (ByNeed T)
get       :: Monad τ ⇒ ByNeed τ (Heap (ByNeed τ))
get       = ByNeed (λμ → return (μ, μ))
put       :: Monad τ ⇒ Heap (ByNeed τ) → ByNeed τ ()
put μ     = ByNeed (λ_ → return ((), μ))
instance (∀v. Trace (τ v)) ⇒ Trace (ByNeed τ v) where
  step e m = ByNeed (step e ∘ unByNeed m)
fetch :: Monad τ ⇒ Addr → D (ByNeed τ)
fetch a = get ≫= λμ → μ ! a
memo :: ∀τ. (Monad τ, ∀v. Trace (τ v)) ⇒ Addr → D (ByNeed τ) → D (ByNeed τ)
```

```
memo a d = d >>= λv → ByNeed (upd v)
  where upd Stuck μ = return (Stuck :: Value (ByNeed τ), μ)
        upd v     μ = step Upd (return (v, μ[a ↦ memo a (return v)]))
instance (Monad τ, ∀v. Trace (τ v)) ⇒ HasBind (D (ByNeed τ)) where
  bind η rhs body = do μ ← get
                       let a = nextFree μ
                       put μ[a ↦ memo a (rhs (fetch a))]
                       body (fetch a)
deriving via StateT (Heap (ByNeed τ)) τ instance Functor τ ⇒
  Functor (ByNeed τ)
deriving via StateT (Heap (ByNeed τ)) τ instance Monad τ  ⇒
  Applicative (ByNeed τ)
deriving via StateT (Heap (ByNeed τ)) τ instance Monad τ  ⇒
  Monad (ByNeed τ)
-- Partial by-value interpreter
S_value⟦_⟧_ :: Exp → (Name :⇀ D (ByValue T)) → D (ByValue T)
S_value⟦e⟧ρ = S⟦e⟧ρ :: D (ByValue T)
newtype ByValue τ v = ByValue { unByValue :: τ v }
instance Trace (τ v) ⇒ Trace (ByValue τ v) where
  step e (ByValue τ) = ByValue (step e τ)
class Extract τ where getValue :: τ v → v
instance Extract T where
  getValue (Ret v)    = v
  getValue (Step _ τ) = getValue τ
instance (Trace (D (ByValue τ)), Monad τ, Extract τ)
         ⇒ HasBind (D (ByValue τ)) where
  bind η rhs body = step Let0 $ do
    let  d = rhs (return v)            :: D (ByValue τ)
         v = getValue (unByValue d) :: Value (ByValue τ)
    v1 ← d
    body (return v1)
deriving instance Functor τ ⇒ Functor (ByValue τ)
deriving instance Applicative τ ⇒ Applicative (ByValue τ)
deriving instance Monad τ  ⇒ Monad (ByValue τ)
-- By-value semantics with lazy initialisation
S_vinit⟦_⟧_(_) :: Exp → (Name :⇀ D (ByVInit T)) → Heap _ → T (Value _, Heap _)
```

```
𝒮vinit⟦e⟧ρ(μ) = unByVInit (𝒮⟦e⟧ρ :: D (ByVInit T)) μ
newtype ByVInit τ v
  = ByVInit {unByVInit :: Heap (ByVInit τ) → τ (v, Heap (ByVInit τ))}
instance (Monad τ, ∀v. Trace (τ v)) ⇒ HasBind (D (ByVInit τ)) where
  bind η rhs body = do μ ← get
                       let a = nextFree μ
                       put μ[a ↦ stuck]
                       step Let₀ (memo a (rhs (fetch a))) >=> body ∘ return
deriving via StateT (Heap (ByVInit τ)) τ instance Functor τ ⇒
  Functor (ByVInit τ)
deriving via StateT (Heap (ByVInit τ)) τ instance Monad τ ⇒
  Applicative (ByVInit τ)
deriving via StateT (Heap (ByVInit τ)) τ instance Monad τ ⇒
  Monad (ByVInit τ)
get :: Monad τ ⇒ ByVInit τ (Heap (ByVInit τ))
get = ByVInit (λμ → return (μ, μ))
put :: Monad τ ⇒ Heap (ByVInit τ) → ByVInit τ ()
put μ = ByVInit (λ_ → return ((), μ))
instance (∀v. Trace (τ v)) ⇒ Trace (ByVInit τ v) where
  step e m = ByVInit (step e ∘ unByVInit m)
fetch :: Monad τ ⇒ Addr → D (ByVInit τ)
fetch a = get >=> λμ → μ ! a
memo :: ∀τ. (Monad τ, ∀v. Trace (τ v)) ⇒ Addr → D (ByVInit τ) → D (ByVInit τ)
memo a d = d >=> λv → ByVInit (upd v)
  where upd Stuck μ = return (Stuck :: Value (ByVInit τ), μ)
        upd v     μ = return (v, μ[a ↦ memo a (return v)])
-- Partial clairvoyant interpreter
𝒮clair⟦_⟧_ :: Exp → (Name :⇀ D (Clairvoyant T)) → T (Value (Clairvoyant T))
𝒮clair⟦e⟧ρ = runClair $ 𝒮⟦e⟧ρ
data Fork f a = Empty | Single ! a | Fork (f a) (f a)
  deriving Functor
newtype ParT τ a = ParT {unParT :: τ (Fork (ParT τ) a)}
  deriving Functor
instance Monad τ ⇒ Applicative (ParT τ) where
  pure a = ParT (pure (Single a))
```

```
  (<*>) = ap
instance Monad τ ⇒ Monad (ParT τ) where
  ParT mas >>= k = ParT $ mas >>= λx → case x of
    Empty → pure Empty
    Single a → unParT (k a)
    Fork l r → pure (Fork (l >>= k) (r >>= k))
instance Monad τ ⇒ Alternative (ParT τ) where
  empty = ParT (pure Empty)
  l <|> r = ParT (pure (Fork l r))

newtype Clairvoyant τ a = Clairvoyant {unClair :: ParT τ a}
  deriving newtype (Functor, Applicative, Monad)

instance (∀v. Trace (τ v)) ⇒ Trace (Clairvoyant τ v) where
  step e (Clairvoyant (ParT mforks)) = Clairvoyant $ ParT $ step e mforks

leftT :: Monad τ ⇒ ParT τ a → ParT τ a
leftT (ParT τ) = ParT $ τ >>= λx → case x of
  Fork l _ → unParT l
  _ → undefined

rightT :: Monad τ ⇒ ParT τ a → ParT τ a
rightT (ParT τ) = ParT $ τ >>= λx → case x of
  Fork _ r → unParT r
  _ → undefined

parFix :: (Extract τ, Monad τ) ⇒ (Fork (ParT τ) a → ParT τ a) → ParT τ a
parFix f = ParT $ fix (unParT ∘ f ∘ getValue) >>= λx → case x of
    Empty → pure Empty
    Single a → pure (Single a)
    Fork _ _ → pure (Fork (parFix (leftT ∘ f)) (parFix (rightT ∘ f)))

instance (Extract τ, Monad τ, ∀v. Trace (τ v)) ⇒ HasBind (D (Clairvoyant τ)) where
  bind _ rhs body = Clairvoyant (skip <|> let') >>= body
    where
      skip = return (Clairvoyant empty)
      let' = fmap return $ unClair $ step Let0 $ Clairvoyant $ parFix $
        unClair ∘ rhs ∘ Clairvoyant ∘ ParT ∘ return

headParT :: (Monad τ, Extract τ) ⇒ ParT τ v → τ (Maybe v)
headParT τ = go τ
  where
    go :: (Monad τ, Extract τ) ⇒ ParT τ v → τ (Maybe v)
```

```
    go (ParT τ) = τ ≫= λx → case x of
      Empty    → pure Nothing
      Single a → pure (Just a)
      Fork l r → case getValue (go l) of
        Nothing → go r
        Just _  → go l
  runClair :: (Monad τ, Extract τ) ⇒ D (Clairvoyant τ) → τ (Value (Clairvoyant τ))
  runClair (Clairvoyant m) = headParT m ≫= λx → case x of
    Nothing → error "Expected at least one Clairvoyant trace"
    Just t  → pure t
```

A bit of order theory:

```
module Order where

import Data.Map (Map)
import qualified Data.Map as Map
import Data.Set (Set)
import qualified Data.Set as Set

class Eq a ⇒ Lat a where
  ⊥ :: a
  (⊔) :: a → a → a;

(⊑) :: Lat a ⇒ a → a → Bool
x ⊑ y = x ⊔ y == y

lub :: (Foldable f, Lat a) ⇒ f a → a
lub = foldr (⊔) ⊥

instance (Ord k, Lat v) ⇒ Lat (Map k v) where
  ⊥ = Map.empty
  (⊔) = Map.unionWith (⊔)

instance (Ord a, Lat a) ⇒ Lat (Set a) where
  ⊥ = Set.empty
  (⊔) = Set.union

instance (Lat a, Lat b) ⇒ Lat (a, b) where
  ⊥ = (⊥, ⊥)
  (a1, b1) ⊔ (a2, b2) = (a1 ⊔ a2, b1 ⊔ b2)

instance Lat a ⇒ Lat [a] where
  ⊥ = []
```

```
    [] ⊔ ys = ys
    xs ⊔ [] = xs
    (x : xs) ⊔ (y : ys) = x ⊔ y : xs ⊔ ys
kleeneFixAbove :: Lat a ⇒ a → (a → a) → a
kleeneFixAbove a f = stationary $ iterate f a
  where stationary (a : b : r) = if b ⊑ a then b else stationary (b : r)
kleeneFix :: Lat a ⇒ (a → a) → a
kleeneFix = kleeneFixAbove ⊥
kleeneFixAboveM :: (Monad m, Lat a) ⇒ a → (a → m a) → m a
kleeneFixAboveM a f = f a ≫= λb → if b ⊑ a then return b else kleeneFixAboveM b f
```

And finally the definitions for Section 4.5:

```
{# LANGUAGE TypeFamilies #}
{# LANGUAGE DerivingVia #}
module StaticAnalysis where
import Prelude hiding ((+), (*))
import qualified Data.Map as Map
import Data.Set (Set)
import qualified Data.Set as Set
import Data.Functor.Identity
import Control.Monad
import Control.Monad.ST
import Control.Monad.Trans.Reader
import Control.Monad.Trans.State
import Data.STRef
import Data.Foldable
import Data.Coerce
import qualified Data.List as List
import Exp
import Order
import Interpreter
  -- Usage cardinality U
data U = U0 | U1 | Uω
type Uses = Name :⇀ U
class UVec a where
  (+) :: a → a → a
```

```
  (*) :: U → a → a
infixl 6 +
infixl 7 *
instance UVec U where
  U1 + U1 = Uω
  u1 + u2 = u1 ⊔ u2
  U0 * _ = U0
  _ * U0 = U0
  U1 * u = u
  Uω * _ = Uω
deriving instance Eq U
instance Lat U where
  ⊥ = U0
  U0  ⊔ u   = u
  u   ⊔ U0  = u
  U1  ⊔ U1  = U1
  _   ⊔ _   = Uω
instance UVec Uses where
  (+) = Map.unionWith (+)
  u * m = Map.map (u*) m
-- Usage trace UT and usage domain UD
data TU v = ⟨Uses, v⟩
instance Trace (TU v) where
  step (Look x) ⟨φ, v⟩ = ⟨[x ↦ U1] + φ, v⟩
  step _        τ      = τ
deriving instance Functor TU
instance Applicative TU where
  pure a = ⟨ε, a⟩
  (<*>) = ap
instance Monad TU where
  return a = ⟨ε, a⟩
  ⟨φ1, a⟩ >>= k = let ⟨φ2, b⟩ = k a in ⟨φ1 + φ2, b⟩
deriving instance Eq a ⇒ Eq (TU a)
instance Extract TU where getValue ⟨_, v⟩ = v
data ValueU = U :: ValueU | Rep U
type DU = TU ValueU
```

```
instance Eq Value_U where
  Rep u1 == Rep u2 = u1 == u2
  v1     == v2     = peel v1 == peel v2
instance Lat Value_U where
  ⊥ = (Rep U0)
  Rep u1 ⊔ Rep u2 = Rep (u1 ⊔ u2)
  Rep u1 ⊔ v      = u1 ⨾ Rep u1 ⊔ v
  v ⊔ Rep u2      = v ⊔ u2 ⨾ Rep u2
  u1 ⨾ v1 ⊔ u2 ⨾ v2 = u1 ⊔ u2 ⨾ v1 ⊔ v2
instance Lat D_U where
  ⊥ = ⟨⊥, ⊥⟩
  ⟨φ1, v1⟩ ⊔ ⟨φ2, v2⟩ = ⟨φ1 ⊔ φ2, v1 ⊔ v2⟩
  -- Usage analysis
S_usage⟦_⟧_ :: Exp → (Name :⇀ D_U) → D_U
S_usage⟦e⟧ρ = S⟦e⟧ρ
instance Domain D_U where
  stuck          = ⊥
  fun x _ f      = case f ⟨[x ↦ U1], Rep Uω⟩ of
    ⟨φ, v⟩ → ⟨φ[x ↦ U0], φ !? x ⨾ v⟩
  apply ⟨φ1, v1⟩ ⟨φ2, _⟩ = case peel v1 of
    (u, v2) → ⟨φ1 + u * φ2, v2⟩
  con _ _ ds = foldl apply ⟨ε, Rep Uω⟩ ds
  select d fs = d ≫ lub [ f (replicate (conArity k) ⟨ε, Rep Uω⟩)
                        | (k, f) ← assocs fs]
peel :: Value_U → (U, Value_U)
peel (Rep u)   = (u, Rep u)
peel (u ⨾ v)   = (u, v)
(!?) :: Uses → Name → U
φ !? x | x ∈ dom φ = φ ! x
       | otherwise = U0
instance HasBind D_U where
  bind _ rhs body = body (kleeneFix rhs)
  -- Types
data TyCon = BoolTyCon | NatTyCon | OptionTyCon | PairTyCon
  deriving (Eq, Enum, Bounded)
data Type = Type :→: Type | TyConApp TyCon [Type] | TyVar Name | Wrong
```

```
  deriving (Eq)
data PolyType = PT [Name] Type

freeVars :: Type → Set Name
freeVars (TyVar x) = Set.singleton x
freeVars (a :→: r) = freeVars a `Set.union` freeVars r
freeVars (TyConApp _ as) = Set.unions (map freeVars as)
freeVars Wrong = Set.empty

splitFunTys :: Type → ([Type], Type)
splitFunTys ty = go [] ty
  where
    go as (a :→: r) = go (a : as) r
    go as ty = (reverse as, ty)

conTy :: Tag → PolyType
conTy TT = PT [] (TyConApp BoolTyCon [])
conTy FF = PT [] (TyConApp BoolTyCon [])
conTy Z = PT [] (TyConApp NatTyCon [])
conTy S = PT [] (TyConApp NatTyCon [] :→: TyConApp NatTyCon [])
conTy None = PT ["a_none"]
  (TyConApp OptionTyCon [TyVar "a_none"])
conTy Some = PT ["a_some"]
  (TyVar "a_some" :→: TyConApp OptionTyCon [TyVar "a_some"])
conTy Pair = PT ["a_pair", "b_pair"]
  (TyVar "a_pair" :→: TyVar "b_pair" :→:
    TyConApp PairTyCon [TyVar "a_pair", TyVar "b_pair"])

tyConTags :: TyCon → [Tag]
tyConTags tc =
  [k | k ← [minBound .. maxBound]
     , let PT _ ty = conTy k
     , TyConApp tc' _ ← [snd (splitFunTys ty)]
     , tc == tc']

-- Domain of Algorithm J
type Subst = Name :⇀ Type
type Constraint = (Type, Type)

newtype J a = J {unJ :: StateT (Set Name, Subst) Maybe a}

deriving instance Functor J
instance Applicative J where
```

```
  pure = J ∘ pure
  (<*>) = ap
instance Monad J where
  J m >>= k = J (m >>= unJ ∘ k)

runJ :: J PolyType → PolyType
runJ (J m) = case evalStateT m (Set.empty, ε) of
  Just ty   → ty
  Nothing → PT [ ] Wrong

applySubst :: Subst → Type → Type
applySubst subst ty@(TyVar y)
  | Just ty ← Map.lookup y subst = ty
  | otherwise = ty
applySubst subst (a :→: r) =
  applySubst subst a :→: applySubst subst r
applySubst subst (TyConApp k tys) =
  TyConApp k (map (applySubst subst) tys)
applySubst _ ty = ty

addCt :: Constraint → Subst → Maybe Subst
addCt (l, r) subst = case (applySubst subst l, applySubst subst r) of
  (l, r) | l == r → Just subst
  (TyVar x, ty)
    | not (occurs x ty)
    → Just (Map.insert x ty subst)
  (_, TyVar _) → addCt (r, l) subst
  (a1 :→: r1, a2 :→: r2) → addCt (a1, a2) subst >>= addCt (r1, r2)
  (Wrong, Wrong) → Just subst
  (TyConApp k1 tys1, TyConApp k2 tys2) | k1 == k2 →
    foldrM addCt subst (zip tys1 tys2)
  _ → Nothing
  where
    occurs x ty = applySubst [x ↦ ty] ty /= ty

unify :: Constraint → J ()
unify ct = J $ StateT $ λ(names, subst) → case addCt ct subst of
  Just subst' → Just ((), (names, subst'))
  Nothing   → Nothing

freshTyVar        :: J Type
```

```
freshTyVar = J $ state $ λ(ns, subst) →
  let n = "\\alpha_{" ++ show (Set.size ns) ++ "}"
  in (TyVar n, (Set.insert n ns, subst))

freshenVars :: [Name] → J Subst
freshenVars alphas = foldM one ε alphas
  where
    one subst alpha = do
      beta ← freshTyVar
      pure subst[alpha ↦ beta]

instantiatePolyTy :: PolyType → J Type
instantiatePolyTy (PT alphas ty) = do
  subst ← freshenVars alphas
  return (applySubst subst ty)

instantiateCon :: Tag → J Type
instantiateCon k = instantiatePolyTy (conTy k)

generaliseTy :: J Type → J PolyType
generaliseTy (J m) = J $ do
  (outer_names, _) ← get
  ty ← m
  (_names', subst) ← get
  let ty' = applySubst subst ty
  let one n = freeVars $ applySubst subst (TyVar n)
  let fvΓ = Set.unions (Set.map one outer_names)
  let generics = freeVars ty' `Set.difference` fvΓ
  return (PT (Set.toList generics) ty')

closedType :: J Type → PolyType
closedType d = runJ (generaliseTy d)

  -- Type analysis
S_type⟦_⟧ :: Exp → PolyType
S_type⟦e⟧ = closedType (S⟦e⟧ε) :: PolyType

instance Trace (J v) where step _ = id
instance Domain (J Type) where
  stuck = return Wrong
  fun _ _ f = do
    θα ← freshTyVar
    θ  ← f (return θα)
```

```
      return (θα :⟶: θ)
    con _ k ds = do
      con_app_ty ← instantiateCon k
      arg_tys ← sequence ds
      res_ty ← freshTyVar
      unify (con_app_ty, foldr (:⟶:) res_ty arg_tys)
      return res_ty
    apply v a = do
      θ1    ← v
      θ2    ← a
      θα    ← freshTyVar
      unify (θ1, θ2 :⟶: θα)
      return θα
    select dv fs =    case Map.assocs fs of
      []              → stuck
      fs@((k, _): _) → do
        con_ty ← dv
        res_ty ← snd ∘ splitFunTys <$> instantiateCon k
        let TyConApp tc tc_args = res_ty
        unify (con_ty, res_ty)
        ks_tys ← enumerateCons tc tc_args
        tys ← forM ks_tys $ λ(k, tys) →
          case List.find (λ(k', _) → k' == k) fs of
            Just (_, f) → f tys
            _           → stuck
        case tys of
          []     → stuck
          ty : tys' → mapM (λty' → unify (ty, ty')) tys' ≫ return ty
enumerateCons :: TyCon → [Type] → J [(Tag, [J Type])]
enumerateCons tc tc_arg_tys = forM (tyConTags tc) $ λk → do
  ty ← instantiateCon k
  let (field_tys, res_ty) = splitFunTys ty
  unify (TyConApp tc tc_arg_tys, res_ty)
  return (k, map pure field_tys)
instance HasBind (J Type) where
  bind _ rhs body = do
    σ ← generaliseTy (uniFix rhs)
```

```
    body (instantiatePolyTy σ)
uniFix :: (J Type → J Type) → J Type
uniFix rhs = do
  θα    ← freshTyVar
  θ     ← rhs (return θα)
  unify (θα, θ)
  return θα
 -- Domain of control-flow analysis CD
newtype Labels = Lbls (Set Label) deriving (Eq, Ord)
instance Lat Labels where
  ⊥ = Lbls Set.empty
  Lbls l ⊔ Lbls r = Lbls (Set.union l r)
 -- If I were serious, I should have used the flat lattice over 'Tag'.
instance Lat Tag where
  ⊥ = error "no bottom Tag"
  k1 ⊔ k2 = if k1 /= k2 then error "k1 /= k2" else k1
type ConCache = (Tag, [Labels])
data FunCache = FC (Maybe (Labels, Labels)) (DC → DC)
data Cache = Cache {cCons :: Label :⇀ ConCache, cFuns :: Label :⇀ FunCache}
type DC = State Cache Labels
runCFA :: DC → Labels
runCFA m = evalState m (Cache ⊥ ⊥)
overCons :: ((Label :⇀ ConCache) → (Label :⇀ ConCache)) → Cache → Cache
overCons f (Cache cons funs) = Cache (f cons) funs
overFuns :: ((Label :⇀ FunCache) → (Label :⇀ FunCache)) → Cache → Cache
overFuns f (Cache cons funs) = Cache cons (f funs)
updConCache :: Label → Tag → [Labels] → State Cache ()
updConCache ℓ k vs = modify $ overCons $ λcons →
  Map.singleton ℓ (k, vs) ⊔ cons
updFunCache :: Label → (DC → DC) → State Cache ()
updFunCache ℓ f = modify $ overFuns $ λfuns →
  Map.singleton ℓ (FC Nothing f) ⊔ funs
cachedCall :: Labels → Labels → DC
cachedCall (Lbls ℓ̄) v = fmap lub $ forM (Set.toList ℓ̄) $ λℓ → do
  FC cache f ← gets (Map.findWithDefault ⊥ ℓ ∘ cFuns)
```

```
    let call in_ out = do
          let in_' = in_ ⊔ v
          modify $ overFuns (Map.insert ℓ (FC (Just (in_', out)) f))
          out' ← f (return in_')
          modify $ overFuns (Map.insert ℓ (FC (Just (in_', out')) f))
          return out'
    case cache of
      Just (in_, out)
        | v ⊑ in_    → return out
        | otherwise → call in_ out
      Nothing        → call ⊥ ⊥
cachedCons :: Labels → State Cache (Tag :⇀ [Labels])
cachedCons (Lbls ℓ̄) = do
  cons ← cCons <$> get
  return $ Map.fromListWith (⊔)
    [cons ! ℓ | ℓ ← Set.toList ℓ̄, ℓ ∈ dom cons]
instance Eq FunCache where
  FC cache1 _ == FC cache2 _ = cache1 == cache2
instance Lat FunCache where
  ⊥ = FC Nothing (const (return ⊥))
  FC cache1 f1 ⊔ FC cache2 f2 = FC cache' f'
    where
      f' d = do
        v ← d
        lv ← f1 (return v)
        rv ← f2 (return v)
        return (lv ⊔ rv)
      cache' = case (cache1, cache2) of
        (Nothing, Nothing)              → Nothing
        (Just c1, Nothing)              → Just c1
        (Nothing, Just c2)              → Just c2
        (Just (in_1, out1), Just (in_2, out2))
          | in_1 ⊑ in_2, out1 ⊑ out2    → Just (in_2, out2)
          | in_2 ⊑ in_1, out2 ⊑ out1    → Just (in_1, out1)
          | otherwise                   → error "uh oh"
instance Eq Cache where
  c1 == c2 = cFuns c1 == cFuns c2 && cCons c1 == cCons c2
```

```
instance Lat Cache where
  ⊥ = Cache Map.empty Map.empty
  c1 ⊔ c2 = Cache (f cCons) (f cFuns)
    where
      f :: Lat fld ⇒ (Cache → fld) → fld
      f fld = fld c1 ⊔ fld c2

 -- Control-flow analysis
𝒮cfa⟦_⟧ :: Exp → Labels
𝒮cfa⟦e⟧ = runCFA (𝒮⟦e⟧ε)
instance HasBind DC where
  bind _ rhs body = go ⊥ ≫= body ∘ return
    where
      go :: Labels → DC
      go v = do
        cache ← get
        v′ ← rhs (return v)
        cache′ ← get
        if v′ ⊑ v && cache′ ⊑ cache
          then return v′
          else go v′
instance Trace DC where step _ = id
instance Domain DC where
  stuck = return ⊥
  fun _ ℓ f = do
    updFunCache ℓ f
    return (Lbls (Set.singleton ℓ))
  apply dv da = do
    v ← dv
    a ← da
    cachedCall v a
  con ℓ k ds = do
    lbls ← sequence ds
    updConCache ℓ k lbls
    return (Lbls (Set.singleton ℓ))
  select dv fs = do
    v ← dv
    tag2flds ← cachedCons v
```

```
      lub <$> sequence [ f (map return (tag2flds ! k))
                       | (k, f) ← Map.assocs fs, k ∈ dom tag2flds]
  -- Stateful analysis, Static domain
class Domain d ⇒ StaticDomain d where
  type Ann d :: ∗
  extractAnn :: Name → d → (d, Ann d)
  funS       :: Monad m ⇒ Name → Label → (m d → m d) → m d
  selectS    :: Monad m ⇒ m d → (Tag :⇀ ([m d] → m d)) → m d
  bindS      :: Monad m ⇒ Name → d → (d → m d) → (d → m d) → m d
fun′ :: StaticDomain d ⇒ Name → Label → (d → d) → d
fun′ x lbl f = runIdentity (funS x lbl (coerce f))
select′ :: StaticDomain d ⇒ d → (Tag :⇀ ([d] → d)) → d
select′ d fs = runIdentity (selectS (Identity d) (coerce fs))
bind′ :: (Lat d, StaticDomain d) ⇒ Name → (d → d) → (d → d) → d
bind′ x rhs body = runIdentity (bindS x ⊥ (coerce rhs) (coerce body))
data Refs s d = Refs (STRef s (Name :⇀ d)) (STRef s (Name :⇀ Ann d))
newtype AnnT s d a = AnnT (Refs s d → ST s a)
type AnnD s d = AnnT s d d
deriving via ReaderT (Refs s d) (ST s) instance Functor (AnnT s d)
deriving via ReaderT (Refs s d) (ST s) instance Applicative (AnnT s d)
deriving via ReaderT (Refs s d) (ST s) instance Monad (AnnT s d)
instance Trace d ⇒ Trace (AnnD s d) where
  step ev (AnnT f) = AnnT (λrefs → step ev <$> f refs)
instance StaticDomain d ⇒ Domain (AnnD s d) where
  stuck = return stuck
  fun x l f = funS x l f
  con l k ds = con l k <$> sequence ds
  apply f d = apply <$> f <*> d
  select md mfs = selectS md mfs
instance (Lat d, StaticDomain d) ⇒ HasBind (AnnD s d) where
  bind x rhs body = do
    init ← readCache x
    let rhs′ d1 = do d2 ← rhs (return d1); writeCache x d2; return d2
    annotate x (bindS x init rhs′ (body ∘ return))
readCache :: Lat d ⇒ Name → AnnD s d
readCache n = AnnT $ λ(Refs cache _) → do
```

```
    c ← readSTRef cache
    return (Map.findWithDefault ⊥ n c)
writeCache :: Name → d → AnnT s d ()
writeCache n d = AnnT $ λ(Refs cache _) →
    modifySTRef' cache $ λc → c[n ↦ d]
annotate :: StaticDomain d ⇒ Name → AnnD s d → AnnD s d
annotate x ad = do
    d ← ad
    let (d', ann) = extractAnn x d
    AnnT $ λ(Refs _ anns) → modifySTRef' anns $ λa → a[x ↦ ann]
    return d'
runAnn       :: (∀s. AnnD s d) → (d, Name :⇀ Ann d)
runAnn m = runST $ do
    r@(Refs _ anns) ← Refs <$> newSTRef ε <*> newSTRef ε
    d ← case m of AnnT f → f r
    anns ← readSTRef anns
    return (d, anns)
  -- Stateful usage analysis
S_usage⇝ ⟦_⟧_ :: Exp → (Name :⇀ D_U) → (D_U, Name :⇀ U)
S_usage⇝ ⟦e⟧ρ = runAnn (S⟦e⟧_return◁ρ)
instance StaticDomain D_U where
    type Ann D_U = U
    extractAnn x ⟨φ, v⟩ = (⟨Map.delete x φ, v⟩, φ !? x)
    funS x η f = do
        ⟨φ, v⟩ ← f (return ⟨[x ↦ U_1], Rep U_ω⟩)
        return ⟨φ[x ↦ U_0], φ !? x ⨾ v⟩
    selectS md mfs = do
        d ← md
        alts ← sequence [ f (replicate (conArity k) (return ⟨ε, Rep U_ω⟩))
                        | (k, f) ← Map.assocs mfs]
        return (d ≫ lub alts)
    bindS _ init rhs body = kleeneFixAboveM init rhs ≫= body
```

Index of Definitions