Leveraging Program Analysis for Type Inference

by

Zvonimir Pavlinovic

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May 2019

Professor Thomas Wies

Acknowledgements

I would like to thank my mother, father, sister, and late grandparents for their undeniable support in pursuing my professional goals. Without their unconditional love, I would have never reached this point in my career that initially seemed as nothing more than a childish dream. This dissertation could have not been written without the guidance of my adviser, Thomas Wies. I am sincerely thankful for an unparalleled patience and countless hours spent helping me understand the phenomena of computation and the very core of being a scientist. To my friends, I thank you for being there for me despite my frequent absence and unavailability. To all the other people that helped me in any capacity during the course of my studies, I am sincerely grateful.

Abstract

Type inference is a popular feature of programming languages used to automatically guarantee the absence of certain execution errors in programs at compile time. The convenience of type inference, unfortunately, comes with a cost. Developing type inference algorithms is a challenging task that currently lacks a systematic approach. Moreover, programmers often have problems interpreting error reports produced by type inference. The overarching goal of this thesis is to provide a mathematically rigorous framework for the systematic development of sophisticated type inference algorithms that are convenient to use by the programmers. To this end, we focus on two specific problems in this thesis: (1) how to constructively design type inference algorithms that improve over the state-of-the-art and (2) how to automatically debug type errors that arise during inference. We base our approach on the observation that, similar to type inference, program analysis algorithms automatically discover various program properties that can be used to show program correctness. Type inference and program analysis techniques, although similar, have traditionally been developed independently of each other. In contrast, this thesis further explores the recent path of leveraging program analysis for type inference.

As our first contribution, we use abstract interpretation to constructively design type inference algorithms. We specifically focus on Liquid types, an advanced family of algorithms that combine classical typing disciplines and known static analyses to prove various safety properties of functional programs. By using abstract interpretation, we make the design space of Liquid type inference explicit. We also unveil the general type inference framework underlying Liquid types. By properly instantiating this general framework, one obtains novel type inference

algorithms that are sound by construction.

Our second contribution is a framework for automatically debugging type errors for languages that deploy type inference in the style of Hindley-Milner, such as OCaml and Haskell. Such languages are notorious for producing cryptic type error reports that are often not helpful in fixing the actual bug. We formulate the problem of finding the root cause of type errors as an optimization problem expressed in a formal logic. We then show how to solve this problem using automated theorem provers. We experimentally illustrate how our framework can efficiently produce type error reports that outperform the state-of-the-art solutions in identifying the true cause of type errors.

In summary, this thesis introduces a mathematical framework for the systematic design of sophisticated type inference algorithms that are sound by construction. Our results further enable automatic generation of more meaningful type error diagnostics, ultimately making type inference more usable by the programmers.

Contents

List of Figures

List of Tables

List of Appendices

Chapter 1

Introduction

Types are an integral part of modern programming languages. The role of types in a programming language is to facilitate the usage of powerful programming abstractions and, perhaps more importantly, enhance the reliability of programs. Types can be used to express information about program executions capturing the form of values being used by the program and the behavior of functions that operate over these values. For instance, types can capture program invariants stating that certain program variables always hold integer values during the execution or that functions implementing the integer division operation accept two integer arguments and produce an integer value. Programmers use types to communicate such information about their programs to the compiler that in turn uses this information to ensure the absence of certain runtime errors. Some programming languages, such as OCaml [70] and Haskell [38], automatically discover types in programs by employing type inference algorithms [74]. By using type inference, programming languages are therefore able to provide certain correctness guarantees for programs at compile time without requiring major programmer intervention.

Type inference algorithms can be seen as a form of static analysis of programs [15, 17, 16]. Both type inference and static analyses automatically deduce information about programs with the goal of showing program correctness. In type inference, programs are deemed correct if they do not exhibit a target class of execution errors, such as calling a Boolean negation operation with a string argument [74]. A deeper connection between type inference and static analyses can be observed in the way they both deal with the inherent infeasibility of automatically deciding program correctness. Static analyses approach this problem by sacrificing precision for soundness, i.e., by over-approximating how programs behave [17, 16]. That is, the behaviors of a program inferred by a static analysis always include the actual program behaviors, but they also might include same spurious behaviors. Similarly, type inference algorithms sacrifice precision by flagging some correct programs as potentially erroneous, but they ensure soundness by never classifying an erroneous program as correct. Despite many similarities [15], type inference and static analysis techniques are typically developed independently of each other. Type inference algorithms rarely reap many of the benefits provided by the advances in static analysis and program analysis in general [17, 68]. This is unfortunate, particularly since type inference algorithms and the process of their creation taken by both industry and academia have several notable shortcomings.

The presentations of many type inference algorithms found in the literature [40, 59, 78] and practical implementations [70, 38, 92] only argue the algorithm's soundness. Such works do not formally discuss the precision of the type inference being presented. The particular points where these algorithms can be made more precise, generalized, or how they can be compared against each other are not made formally explicit. Overall, the design space of existing type inference algorithms

is mathematically unclear. Developing type inference algorithms is consequently a challenging problem lacking a more systematic approach. This problem is even more severe in the context of modern programming languages that are facing demands for providing type inference with an ever increasing precision and for increasingly complex classes of errors whose absence they are supposed to guarantee. The other shortcoming associated with type inference concerns its usability. When type inference flags a given program as possibly erroneous by discovering a type error, the compiler-generated messages explaining the error are often not helpful in fixing the actual bug. The produced error explanation frequently identifies the source of the error incorrectly, thus increasing the debugging times for the programmer [54]. Even worse, cryptic type error reports can significantly hinder the process of learning languages with type inference for novice programmers [54]. The goal of this thesis is to reduce the cost of developing and using sophisticated type inference algorithms in modern programming languages by addressing the problems of (1) how to constructively design type inference algorithms that improve over the state-of-the-art and (2) how to automatically debug type errors that arise during inference. We base our approach on the close connection between static analysis and type inference described earlier.

We make two key technical contributions in this thesis. First, we use abstract interpretation [17, 16] to systematically construct Liquid type inference [92], a recently developed algorithm based on classical type inference [40, 59] and predicate abstraction [4, 27] that automatically infers rich properties of higher-order programs. Abstract interpretation is a mathematically rigorous framework for constructively designing static analyses as abstract semantics of programs, i.e., as over-approximations of the actual program behaviors. By formulating Liquid type

inference as a static analysis using abstract interpretation, we unveil the design space of such inference algorithms. Our formulation makes explicit the points where Liquid type inference loses precision and can hence be improved. Moreover, our results allow generalization of Liquid type inference and facilitate the development of powerful type inference algorithms that are sound by construction.

As a second contribution, we formally illustrate how the problem of type error debugging for languages such as OCaml and Haskell can be cast as an optimization problem that neatly separates two constituent subproblems: devising a heuristic for pinpointing the root source of a type error and searching for the root source given such a heuristic. We express the optimization problem of type error debugging using formal logic and then solve it using automated theorem provers [21, 5]. By relying on recent advances in automated theorem proving, we obtain a general, effective, and efficient framework for type error debugging.

We now discuss each of our two technical contributions in more detail.

1.1 Data Flow Refinement Type Inference

Classical type inference algorithms, such as those found in OCaml and Haskell, guarantee the absence of a certain class of runtime errors. For instance, such algorithms can capture that an array read, $\mathsf{read}\ a\ i$, will always operate on the base pointer a to some memory region that holds an array. However, to show that the array read is always within the array bounds, the types need to relate the index i at which the read occurs and the length of the array a. Automatically inferring such types is out of the scope of classical type systems. Refinement types take an exciting direction to designing static type inference algorithms that can

infer fine-grained program invariants [28, 95, 78, 92, 99, 93]. Here, the types of a classical type system are augmented with auxiliary *type refinements* that further constrain the members of the basic type.

Liquid types [78, 91, 92, 90] form an important family of refinement type systems. Here, basic types are augmented with *refinement predicates* that are drawn from some decidable logic. These predicates can express relational dependencies between inputs and outputs of functions. For example, the contract of the array read operator, read, can be expressed using the Liquid type

$$\mathsf{read} :: (a : \alpha\ \mathsf{array}) \rightarrow (i : \{\nu : \mathsf{int} \mid 0 \leq \nu < \mathsf{length}\ a\}) \rightarrow \alpha$$

This type indicates that read is a function that takes an array a over some element type α and an index i as input and returns a value of type α. The type $\{\nu : \mathsf{int} \mid 0 \leq \nu < \mathsf{length}\ a\}$ of the parameter i refines the base type int to indicate that i must be an index within the bounds of the array a. Liquid type inference uses this type to check that array accesses in a given program are within the array bounds by automatically inferring refinement types for all program expressions. For instance, consider the following illustrative OCaml program.

```
let f a i = read a (i - 1) in
let b = f [| 1; 2 |] 2 in
let c = f [| 3; 4; 5 |] 3 in
b + c
```

The above program defines a function f that accepts an integer array a, an integer i, and returns the element of a at position $i - 1$. This function is then called two times with integer arrays and integers whose value equals the length of the respective arrays. Liquid type inference first calls the OCaml type inference to

get the basic types of program expressions. For every expression, the Liquid type inference then infers a conjunction of auxiliary refinement predicates drawn from a given finite set of candidate predicates. The inferred conjunction refines the basic type of the expression and is consistent with the way the expression is used in the program. For instance, one possible inferred Liquid type for f is

$$\mathsf{f} :: (a : \mathsf{int\ array}) \to (i : \{\nu : \mathsf{int} \mid \nu = \mathsf{length}\ a \wedge 2 \leq \nu\}) \to \mathsf{int}$$

The above type correctly states that f is always called with an integer array a whose length is equal to the value of the argument i in the above program. The value of i is also correctly inferred to be greater or equal than 2. Hence, the actual access index $i - 1$ is always provably within the bounds of a for the above program. This illustrative example showcases an interesting feature of Liquid type inference. The type of f is not inferred based on the body of f in isolation, as is the case with more classical type inference [40, 59], but also on the remainder of the program: the types for parameters a and i are deduced based on how f is called in the program. The algorithm discovers that the values *flowing* to the two arguments of f are always an integer array a and an integer i such that i is in the interval $[2,\infty]$ and $i = \mathsf{length}\ a$. Intuitively, this type expresses an invariant of the program capturing how data flows to and from f.

Approach. In this thesis, we formally model the distinguishing features of Liquid type inference with the goal of generalizing it and unveiling the design space of such analyses. We shed new light on Liquid type inference in terms of abstract interpretation, a framework that enables the systematic construction of static analyses as abstractions of a concrete program semantics [16, 17]. First, we propose a new semantics of higher-order functional programs that precisely captures the pro-

gram properties abstracted by Liquid types. Our new semantics explicitly models the data flow in higher-order programs that is over-approximated by Liquid types. To the best of our knowledge, this is the first semantics where the notion of data flow is made explicit in a higher-order setting. We then systematically construct a *data flow refinement type analysis* via a sequence of Galois abstractions [16] of this concrete semantics. The resulting analysis infers valid refinement types for all subexpressions of a program and is parametric in the abstract domain used to express the type refinements. Finally, we show that Liquid type inference is a particular instance of this parametric analysis where the abstract domain of type refinements is defined by the particular choice of candidate refinement predicates made by the Liquid type inference.

Contributions. Our technical development brings several benefits. First, the points where Liquid type inference loses precision are made explicit in (1) the Galois connections used in the abstractions building up to our data flow refinement type analysis and (2) the way in which the convergence of the analysis is enforced, which we formally capture as widening operators [16, 17]. These operators are a well-known mechanism used in abstract interpretation to ensure the sound termination of static analyses. Next, we generalize from the specific choice of logical predicates as type refinements used in Liquid types by allowing for the use of arbitrary abstract domains (including e.g. polyhedra [83, 19] and automata-based domains [3, 50]). Instantiations of our parametric type analysis are sound by construction. Finally, the type inference analyses that build on our results can be more easily compared against each other and used in combination with other static analyses (e.g., via reduced products [17]).

1.2 Type Error Localization

Another important aspect of type inference besides its systematic construction is usability. For a type inference to be usable, it is important that the programmers understand the results the inference produces. This particularly applies for the cases where type inference rejects a program and reports a type error. The quality of the error report has a direct impact on the programmer's ability to fix the bug [54]. In this thesis, we focus on the problem of type error debugging for Hindley-Milner type inference [40, 59] found in languages such as OCaml and Haskell. Our focus on the Hindley-Milner algorithm is motivated by its widespread usage [70, 38] and often unsatisfactory error reports that increase debugging times for the programmers [54]. Moreover, addressing the problem of type error debugging for the Hindley-Milner type inference is an essential step in improving type error reports for the Liquid type inference, as the latter algorithm relies on the results produced by the Hindley-Milner algorithm [78].

Hindley-Milner type inference reports type errors on the fly. If the inferred type of the current program expression conflicts the inferred type of its context, the inference algorithm immediately stops and reports an error at the current program location. Although fast in practice, this approach also produces poor error diagnostics. In particular, it might be the case that the programmer made a mistake with the previous usages of the offending expression, or with some other related expressions. For example, consider the following simple OCaml program taken from the student benchmarks in [54]:

```
1 type 'a lst = Null | Cons of 'a * 'a lst
2 let x = Cons(3, Null)
```

```
3 let _ = print_string x
```

The standard OCaml compiler [70] reports a type mismatch error for expression `x` on line 3, as the code before that expression is well typed. However, perhaps the programmer defined `x` incorrectly on line 2 or misused the `print_string` function. As `x` is defined just before the error line, it seems more likely that the error is caused by a misuse of `print_string`. In fact, the student author of this code confirmed that this is the real source of the error. This simple example suggests that in order to generate useful error reports, compilers should consider several possible error causes and rank them by their relevance. Hence, there is a need for an infrastructure that can supply compilers with error sources that best match their relevance criteria. This thesis proposes a general algorithm based on constraint solving that provides this functionality.

Approach. Unlike typical type inference algorithms, we do not simply report the location of the first observed type inconsistency. Instead, we compute all minimum sets of expressions each of which, once corrected, yields a type correct program. Compilers can then use these computed sets for generating more meaningful error reports or even for providing automatic error correction. The considered optimality criterion is controlled by the compiler. For example, the compiler may only be interested in those error causes that require the fewest changes to fix the program.

The crux of our approach is to reduce type error localization to the weighted maximum satisfiability modulo theory (MaxSMT) problem. Specifically, our algorithm builds on existing work that rephrases type inference in terms of constraint solving [87, 1, 71]. Each program expression is assigned a type variable and typing information is captured in terms of constraints over those variables. If an input

program has a type error, then the corresponding set of typing constraints is unsatisfiable. We encode the compiler-specific ranking criterion by assigning weights to the generated typing constraints. A weighted MaxSMT solver then computes the satisfiable subsets of the constraints that have maximum cumulative weight. As constraints directly map to program expressions, the complements of these maximum sets represent minimum sets of program expressions that may have caused the type error.

One issue with our constraint-based approach to the type error debugging problem is that the constraints for polymorphic functions need to be freshly copied for each usage of such functions, potentially resulting in an exponential number of constraints [74]. We hence also propose an improved algorithm that deals with the inherent complexity of polymorphic typing [57]. Our new algorithm makes the optimistic assumption that the relevant type error sources only involve few polymorphic functions, even for large programs. The new algorithm abstracts polymorphic functions by principal types [40] in such a way that all potential error sources involving the definition of an abstracted function are represented by a single error source whose cost is smaller or equal to the cost of all these potential error sources. If a computed error source does not involve a usage of such an abstracted polymorphic function, the minimum error source has been found. Otherwise, we expand the instantiations of the principal type of the function involved in the discovered error source to the corresponding typing constraints and repeat the whole process of finding the minimum error source. In the worst case scenario, the algorithm terminates by solving the constraint set where all usages of polymorphic functions are expanded, ending up with the same constraint set as the naive algorithm.

Contributions. The benefits of our approach are multifold. Our approach clearly separates the problems of devising a heuristic for pinpointing the actual source of a type error, expressed as a ranking criterion, and the problem of searching for the best error source given the criterion, solved by MaxSMT solvers. The use of SMT solvers has several additional advantages. First, it allows support for a variety of type systems by instantiating the MaxSMT solver with an adequate reasoning theory. In particular, the typing constraints for the Hindley-Milner type systems [40, 59] can be encoded in the theory of inductive data types [7]. Second, the framework does not introduce a substantial implementation overhead since the SMT solver can be used as a black box. Finally, we experimentally illustrate how our type error debugging framework is able to incorporate sophisticated ranking criteria that outperform standard type inference implementations in finding the true source of type errors while being efficient even for programs having several thousands of lines of code. To the best of our knowledge, no other type error debugging technique that formally guarantees the optimality of the computed error sources has comparable performance.

1.3 Road Map

In summary, the results presented in this thesis enable a systematic development of powerful and programmer-friendly type inference algorithms. The presentation of these results in this thesis is structured as follows. We start by introducing the programming language, notation, and definitions we use throughout the thesis in Chapter 2. We present our formulation of Liquid type inference as abstract interpretation in Chapter 3. After providing an overview of the technical contri-

bution in § 3.1, we introduce our novel concrete data flow semantics of untyped higher-order languages in § 3.2 and build several subsequent abstract semantics in § 3.3, § 3.4, and § 3.5. We finally present Liquid type semantics of functional programs in § 3.6. Chapter 4 describes our solution to the problem of type error debugging aimed at type inference algorithms in the style of Hindley-Milner. We first overview our approach in § 4.1. Then, we formalize the problem of type error localization in § 4.2, present the solution in § 4.3, and discuss our implementation and evaluation in § 4.4. § 4.5 presents an improved type error localization algorithm that effectively deals with the inherent complexity of polymorphic typing. We conclude the thesis in Chapter 5.

Chapter 2

Preliminaries

We first introduce notation and definitions that we use throughout the thesis.

2.1 Notation

We often use meta-level **let** ... **in** ... and conditional **if** ... **then** ... **else** ... constructs in mathematical definitions. The context and the notation should make it clear when these constructs are used on the meta-level and as actual constructs of the programming language of investigation. We will often compress consecutive let statements **let** $x = \ldots$ **in let** $y = \ldots$ **in** ... as **let** $x = \ldots$ **;** $y = \ldots$ **in** ....

We use capital lambda notation ($\Lambda x.\ \ldots$) for defining mathematical functions. For a function $f : X \to Y$, $x \in X$, and $y \in Y$, we use $f.x : y$ and $f[x \mapsto y]$ to denote a function that maps x to y and otherwise agrees with f on every element of $X \backslash \{x\}$. For a set X we denote its powerset by $\wp(X)$. For a relation $R \subseteq X \times Y$ over sets X, Y and a natural number $n > 0$, we use $\dot{R}_n$ to refer to the pointwise lifting of R to a relation on n-tuples $X^n \times Y^n$. That is $\langle (x_1, \ldots, x_n), (y_1, \ldots, y_n) \rangle \in \dot{R}_n$ iff $\bigwedge_{1 \leq i \leq n} (x_i, y_i) \in R$. Similarly, for any nonempty set Z we denote by $\dot{R}_Z$ the

pointwise lifting of R to a relation over $(Z \to X) \times (Z \to Y)$. More precisely, if $f_1 : Z \to X$ and $f_2 : Z \to Y$, then $(f_1, f_2) \in \dot{R}_Z$ iff $\forall z \in Z.\, (f_1(z), f_2(z)) \in R$. Typically, we drop the subscripts from these lifted relations when they are clear from the context or use an alternative "dot" notation. For instance, the pointwise lifting $\dot{R}_2$ of a relation $R : X \times Y$, for some sets X and Y, is also denoted by $\ddot{R}$.

For sets X, Y, and a function $d : X \to \wp(Y)$, we use the notation $\Pi x \in X.d(x)$ to refer to the set $\{\, f : X \to Y \mid \forall x \in X.\, f(x) \in d(x) \,\}$ of all dependent functions with respect to d. Similarly, for given sets X and Y we use the notation $\Sigma x \in X.d(x)$ to refer to the set $\{\, \langle x, y \rangle : X \times Y \mid y \in d(x) \,\}$ of all dependent pairs with respect to d. We use the operators π_1 and π_2 that project to the first, respectively, second component of a pair. For set comprehensions $\{t_1 \mid t_2\}$ we assume that free variables of the term t_2 are existentially quantified unless they are free in t_1.

2.2 Language

For exposition purposes, our formal presentations are built around an idealized language: a lambda calculus with ML-style let polymorphism. The syntax of our language is defined as follows:

Expressions	$e ::= x$	variables
	$\mid c$	constants
	$\mid \mu f.\lambda x.e$	(recursive) lambda abstractions
	$\mid e_1\ e_2$	applications
	$\mid e_0\,?\,e_1 : e_2$	conditionals
	$\mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$	let bindings

The language supports constants $c \in \mathit{Cons}$ and recursive lambda abstractions. We assume the set of constants includes Booleans $b \in \mathit{Bool} \stackrel{\text{def}}{=} \{\mathit{true}, \mathit{false}\}$, which we will use to assign meaning to conditional expressions. In addition to conditional expressions and the usual constructs of the lambda calculus, our language also supports let bindings. Specifically, we use let bindings to define polymorphic functions.

Variables $x \in \mathit{Vars}$ and $f \in \mathit{Vars}$ are drawn from an infinite set that is disjoint from all other syntactic constructs. An expression e is *closed* if all using occurrences of variables within e are bound in definitions of recursive functions $\mu f.\lambda x.e$.

Let e be an expression. Each subexpression of e is uniquely annotated with a location $\ell \in \mathit{Loc}$. We use $\mathit{Loc}(e)$ to denote the set of all locations of e. A location $\ell \in \mathit{Loc}(e)$ hence uniquely identifies a subexpression of e. We denote this subexpression by $e(\ell)$. We use superscript notation to indicate these locations in (sub)expressions as in $(e_1^{\ell_1}\, e_2^{\ell_2})^{\ell}$ and $(\mu f.\lambda x.e_1^{\ell_1})^{\ell}$. Sometimes, we also use $e' \in e$ notation to denote the fact that e' is a subexpression of e. The label annotations are omitted whenever possible to avoid notational clutter. Variables are also locations $\mathit{Vars} \subseteq \mathit{Loc}$. A *program* is a closed expression where all location labels are unique.

Next, we define a mapping *Uloc* for the usage locations of a variable. Formally, *Uloc* is a partial function such that given a location ℓ of a `let` variable definition and a program p returns the set $\mathit{Uloc}_p(\ell)$ of all locations where this variable is used in p. Note that a location of a `let` variable definition is a location corresponding to the root of the defining expression. We also make use of a function for the definition location *dloc*. The function *dloc* reverses the mapping of *Uloc* for a variable usage. More precisely, $\mathit{dloc}(p, \ell)$ returns the location where the variable appearing at ℓ was defined in p. Also, for a set of locations L we define $\mathit{Vloc}(\ell)$ to

be the set of all locations in $Loc(\ell)$ that correspond to usages of `let` variables. We assume the above sets are precomputed for a given program. We do not provide detailed algorithms for computing them as they are straightforward.

2.3 Types

The types of our language are as follows:

$$\textbf{Monotypes} \qquad \tau ::= \mathsf{bool} \mid \mathsf{int} \mid \alpha \mid \tau \to \tau$$

$$\textbf{Polytypes} \qquad \sigma ::= \tau \mid \forall \alpha.\sigma$$

Monotypes τ include the *base types* bool and int, *type variables* α, which are drawn from an infinite set disjoint from the other types (as well as locations and program expressions), and *function types* $\tau \to \tau$. A monotype in which no type variables occur is called *ground*. A polytype $\forall \alpha.\sigma$ represents the intersection of all types obtained by instantiating the type variable α in σ by a ground monotype. That is, $\forall \alpha.\sigma$ binds α. We write $\forall \vec{\alpha}.\tau$ as a shorthand for $\forall \alpha_1 \ldots \forall \alpha_n.\tau$ where $\vec{\alpha} = \alpha_1, \ldots, \alpha_n$. We further denote by $fv(\sigma)$ the set of free type variables in type σ. Finally, we write $\sigma[\vec{\beta}/\vec{\alpha}]$ for capture-avoiding substitution of free occurrences of the type variables $\vec{\alpha}$ in σ by the type variables $\vec{\beta}$.

We define *typing environments* Γ as maps from variables to types. We write $\emptyset$ for the empty typing environment and extend the function fv to typing environments in the expected way.

2.4 Abstract Interpretation

A partially ordered set (poset) is a set L equipped with a relation $\sqsubseteq$ that is (1) reflexive, (2) transitive, and (3) antisymmetric. We use the notation $(L, \sqsubseteq)$ for such a poset.

Galois connections. Let $(L_1, \sqsubseteq_1)$ and $(L_2, \sqsubseteq_2)$ be two posets. We say a pair of functions $\langle \alpha, \gamma \rangle$ with $\alpha \in L_1 \to L_2$ and $\gamma \in L_2 \to L_1$ forms a *Galois connection* iff

$$\forall x \in L_1, \forall y \in L_2.\ \alpha(x) \sqsubseteq_2 y \iff x \sqsubseteq_1 \gamma(y)\ .$$

This fact is denoted by $(L_1, \sqsubseteq_1) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (L_2, \sqsubseteq_2)$. We call L_1 the *concrete* domain and L_2 the *abstract* domain of the Galois connection. Similarly, α is called *abstraction* function (or left adjoint) and γ *concretization* function (or right adjoint). Intuitively, $\alpha(x)$ is the most precise approximation of $x \in L_1$ in L_2 while $\gamma(y)$ is the least precise element of L_1 that can be approximated by $y \in L_2$.

A *complete lattice* is a tuple $\langle L, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ where $(L, \sqsubseteq)$ is a poset such that for any $X \subseteq L$, the least upper bound $\sqcup X$ (join) and greatest lower bound $\sqcap X$ (meet) with respect to $\sqsubseteq$ exist. In particular, we have $\bot = \sqcap L$ and $\top = \sqcup L$. We will often identify a complete lattice with its carrier set L. Some of our definitions for a join (meet) operator will be defined as a binary operator $\sqcup \subseteq L \times L$ ($\sqcap \subseteq L \times L$) that is lifted to arbitrary subset of L when possible.

Let $(L_1, \sqsubseteq_1, \bot_1, \top_1, \sqcup_1, \sqcap_1)$ and $(L_2, \sqsubseteq_2, \bot_2, \top_2, \sqcup_2, \sqcap_2)$ be two complete lattices. Let the pair $\langle \alpha, \gamma \rangle$, where $\alpha \in L_1 \to L_2$ and $\gamma \in L_2 \to L_1$, form a Galois connections. Each function in the pair uniquely determines the other:

$$\alpha(x) = \sqcap_2 \{ y \in L_2 \mid x \sqsubseteq_1 \gamma(y) \}$$

$$\gamma(y) = \sqcup_1\{x \in L_1 \mid \alpha(x) \sqsubseteq_2 y\}$$

Also, α is a *complete join-morphism*

$$\forall S \subseteq L_1.\ \alpha(\sqcup_1 S) = \sqcup_2\{\alpha(x) \mid x \in S\}, \alpha(\bot_1) = \bot_2$$

and γ is a *complete meet-morphism*

$$\forall S \subseteq L_2.\ \gamma(\sqcap_2 S) = \sqcap_1\{\gamma(x) \mid x \in S\}, \gamma(\top_2) = \top_1.$$

Proposition 1 ([17])**.** *The following statements are equivalent:*

1. $\langle \alpha, \gamma \rangle$ *is a Galois connection*
2. α *and* γ *are monotone,* $\alpha \circ \gamma$ *is reductive:* $\forall y \in L_2.\ \alpha(\gamma(y)) \sqsubseteq_2 y$*, and* $\gamma \circ \alpha$ *is extensive:* $\forall x \in L_1.\ x \sqsubseteq_1 \gamma(\alpha(x))$
3. γ *is a complete meet-morphism and* $\alpha = \Lambda x \in L_1.\ \bigsqcap_2\{\, y \in L_2 \mid x \sqsubseteq_1 \gamma(y)\,\}$
4. α *is a complete join-morphism and* $\gamma = \Lambda y \in L_2.\bigsqcup_1\{\, x \in L_1 \mid \alpha(x) \sqsubseteq_2 y\,\}$.

Proposition 2 ([17])**.** α *is onto iff* γ *is one-to-one iff* $\alpha \circ \gamma = \lambda y.\, y$.

Widening operators. A widening operator for a complete lattice $\langle L, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$ is a function $\triangledown : L \times L \to L$ such that: (1) $\triangledown$ is an upper bound operator, i.e., for all $x, y \in L$, $x \sqcup y \sqsubseteq x \triangledown y$, and (2) for all infinite ascending chains $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$ in L, the chain $y_0 \sqsubseteq y_1 \sqsubseteq \ldots$ eventually stabilizes, where $y_0 \stackrel{\text{def}}{=} x_0$ and $y_i \stackrel{\text{def}}{=} y_{i-1} \triangledown x_i$ for all $i > 0$.

Proposition 3 ([16])**.** *Given a complete lattice* $\langle L, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$*, let* $F : L \to L$ *be a monotone function and* $\triangledown : L \times L \to L$ *a widening operator for* L*. Let* $\bar{X}$ *be*

an (upward) iteration sequence defined over L, as follows

$$
\begin{aligned}
X^0 &\stackrel{\text{def}}{=} \bot \\
X^{i+1} &\stackrel{\text{def}}{=} \begin{cases} X^i & \text{if } F(X^i) \sqsubseteq X^i \\ X^i \triangledown F(X^i) & \text{otherwise} \end{cases}
\end{aligned}
$$

Let X_f be the least fixpoint $\mathbf{lfp}_{\bot}^{\sqsubseteq} F$ of F, which exists due to Tarski [88]. It follows then that $\bar{X}$ is stationary with the limit X such that $X_f \sqsubseteq X$.

Chapter 3

Data Flow Refinement Type Inference

We motivate our work on casting Liquid type inference as abstract interpretation by providing a more detailed introduction to the actual inference algorithm, its distinguishing features, and challenges of modeling the algorithm as an abstract semantics of programs. Proofs for all the theorems, lemmas, and corollaries stated in this chapter can be found in Appendix A, unless already presented in the chapter.

3.1 Overview

To better understand how the Liquid Type inference works, consider the following OCaml program that is slightly more involved than the example from § 1.

```
1 let dec y = y - 1 in
2 let f x g = if x >= 0 then g x else x in
3 f (f 1 dec) dec
```

The program first defines a function `dec` that decrements the input value `y` by one and then calls the higher-order function `f` twice, passing it an integer value and `dec` in each call. In turn, `f` calls the given function `g` on its parameter `x` if `x` is not negative, which is true for both calls to `f` in the program.

3.1.1 Liquid Type Inference

The Liquid type inference algorithm works as follows. First, the analysis performs a standard Hindley-Milner type inference to infer the basic shape of the refinement type for every subexpression of the program. For instance, the inferred type for the function `f` is $\mathsf{int}\to(\mathsf{int}\to\mathsf{int})\to\mathsf{int}$. For every function type $\tau_1\to\tau_2$, where τ_1 is a base type such as int, the analysis next introduces a fresh dependency variable x which stands for the function parameter, $x : \tau_1\to\tau_2$. The scope of x is the result type τ_2, i.e., refinement predicates inferred for τ_2 can express dependencies on the input τ_1 by referring to x. Further, every base type τ is replaced by a refinement type, $\{\nu : \tau \mid \phi(\nu, \vec{x})\}$, with a placeholder refinement predicate ϕ that expresses a relation between the members ν of τ and the other variables $\vec{x}$ in scope of the refinement type. For example, the augmented type for function f is

$$x : \{\nu\colon \mathsf{int} \mid \phi_1(\nu)\} \to (y : \{\nu\colon \mathsf{int} \mid \phi_2(\nu, x)\} \to \{\nu\colon \mathsf{int} \mid \phi_3(\nu, x, y)\}) \to \{\nu\colon \mathsf{int} \mid \phi_4(\nu, x)\}.$$

The algorithm then derives a system of Horn clauses modeling the subtyping constraints imposed on the refinement predicates by the program *data flow*. For example, the body of the function `f` induces the following Horn clauses over the

refinement predicates in `f`'s type:

$$x \geq 0 \wedge \phi_1(x) \Rightarrow \phi_2(x, x)$$
$$x \geq 0 \wedge \phi_1(x) \wedge \phi_2(x, x) \wedge \phi_3(\nu, x, y) \Rightarrow \phi_4(\nu, x)$$
$$x < 0 \wedge \phi_1(x) \wedge \nu = x \Rightarrow \phi_4(\nu, x)$$

The first two clauses model the data flow from the parameter `x` to the input of the function parameter `g` of `f` and, in turn, the result value of `g` to the result value of `f` in the *then* branch of the conditional. The third clause captures the flow of the parameter x to `f`'s result value in the *else* branch. Similar entailments are derived from the other subexpressions.

The algorithm finally solves the Horn clauses using monomial predicate abstraction [52] to derive the refinement predicates ϕ_i. That is, the analysis assumes a given set of atomic predicates $Q = \{p_1(\nu, \vec{x}), \ldots, p_n(\nu, \vec{x})\}$, which are either provided by the programmer or derived from the program using heuristics, and then infers an assignment for each ϕ_i to a conjunction over Q such that all Horn clauses are valid. This can be done effectively and efficiently using the Houdini algorithm [26, 52]. For example, if we choose $Q = \{0 \leq \nu, \nu < 2, \nu = x, \nu = x - 1, \nu = y - 1, \nu = 1\}$, then the final type inferred for function `f` will be:

$$x\colon \{\nu\colon \mathsf{int} \mid 0 \leq \nu < 2\} \rightarrow (y\colon \{\nu\colon \mathsf{int} \mid 0 \leq \nu < 2 \wedge \nu = x\} \rightarrow$$
$$\{\nu\colon \mathsf{int} \mid \nu < 2 \wedge \nu = y - 1\}) \rightarrow \{\nu\colon \mathsf{int} \mid \nu < 2 \wedge \nu = x - 1\}$$

The meaning of this type is tied to the current program. For instance, the above inferred type guarantees that the program only calls `f` with an integer in the interval [0, 2) as the first argument. In other words, the inference algorithm is able to infer that only the integer values in the interval [0, 2) *flow* to `f` as inputs and

that the output values flowing out of `f` are integers smaller than 2 and are equal to the decrement of the respective inputs. Note that for a different program, the analysis would potentially infer a different refinement type for `f`. For instance, if we replaced `dec` by the identity function, the analysis would infer $\{\nu\colon \mathsf{int} \mid \nu = 1\}$ for the first input x of `f`. Further, the type inferred for the input `g` of `f` captures how functions passed as `g` to `f` are used by `f`. To see this in more detail, consider the following two types inferred for the two usages of `f` in the program:

$$x\colon \{\nu\colon \mathsf{int} \mid \nu = 1\} \rightarrow (y\colon \{\nu\colon \mathsf{int} \mid \nu = x = 1\} \rightarrow \{\nu\colon \mathsf{int} \mid \nu = x = y = 1\}) \rightarrow \{\nu\colon \mathsf{int} \mid \nu = x = 1\}$$

$$x\colon \{\nu\colon \mathsf{int} \mid \nu = 0\} \rightarrow (y\colon \{\nu\colon \mathsf{int} \mid \nu = x = 0\} \rightarrow \{\nu\colon \mathsf{int} \mid \nu = x = y = 0\}) \rightarrow \{\nu\colon \mathsf{int} \mid \nu = x = 0\}$$

Intuitively, the function `f` takes two different execution paths whose endpoints correspond to the two usages of `f`. The types inferred for these two usages capture the behavior of `f` for the inputs generated at these two call sites. The first type above, corresponding to the nested call to `f`, states that `f` is here called with integer 1 as the first input and a function accepting and producing integers as the other input. This function, once passed to `f` at this call site, is subsequently called with integer 1, also producing integer 1 as the result. Finally, the output of this invocation to `f` is also integer 1. The similar explanation can be given for the type corresponding to the outer call to `f`. Both of these two types are encompassed by the type inferred for the definition of `f`. In general, Liquid type inference infers *data flow* invariants of higher-order programs that capture not only the kind of values to which a program expression evaluates, but also how these values are used by the program from that point onward in the execution paths taken by the values.

3.1.2 Problem

Rather than indirectly describing the inference as a constraint solving problem induced by a type system with a subtyping relation, we model Liquid type inference directly as an abstraction, formalized by Galois connections, of higher-order data flow that induces the subtyping relation. One of the key technical challenges in modeling Liquid types in this way concerns the choice of the concrete semantics. While soundness of the Liquid typing rules can be proved against denotational [92] and operational semantics [78], neither is well suited for calculationally constructing Liquid type inference using abstract interpretation. Intuitively, these semantics do not explicitly capture the data flow properties of programs.

The problem with denotational semantics is that it is inherently compositional; functions are analyzed irrespective of the context in which they appear. Formally, Liquid types and their denotational interpretations do not form a Galois connection. Consider a function `let f x = x + 1` and the two types $\tau_1 = \{\nu : \mathsf{int} \mid \nu \geq 0\} \to \{\nu : \mathsf{int} \mid \nu \geq 0\}$ and $\tau_2 = \{\nu : \mathsf{int} \mid \nu < 0\} \to \{\nu : \mathsf{int} \mid \nu \leq 0\}$. The denotational semantics of `f` is a function that maps all input values that are integers to their successors and all other inputs to the error value. The type τ_1 describes all functions that map a positive integer to a positive integer and all other values to arbitrary values. The meaning of τ_2 is similar. The denotation of `f` is contained in the concretization of both τ_1 and τ_2 and hence in their intersection. However, the meet, i.e., the greatest lower bound, of τ_1 and τ_2 subject to the subtyping ordering is $\tau = \{\nu : \mathsf{int} \mid \mathit{true}\} \to \{\nu : \mathsf{int} \mid \nu = 0\}$. Clearly, the denotation of `f` is not described by τ, which means that the denotational concretization of Liquid types cannot be the right adjoint of a Galois connection [17]. One way of forming a Galois connection between types and a denotational domain is by using intersection

types [15]. However, intersection types are not supported by Liquid type inference and lead to a fundamentally different analysis [28].

Operational semantics, on the other hand, do not make explicit how a function value flows from the point of its creation to its call sites. A formal connection to Liquid type inference would then involve complex reasoning that relates steps in the program's execution trace where function values are created with subsequent steps where they are called.

3.1.3 Approach

We therefore start our technical development with a new concrete *data flow* semantics for untyped higher-order functional programs. This new semantics is inspired by minimal function graphs [46] and the exact semantics for flow analysis [41].

3.1.3.1 Data Flow Semantics

The key idea behind our data flow semantics is as follows. We first formally model unique evaluation points in a program execution. Intuitively, these points are identified by an execution environment and the unique location of the expression currently being evaluated and can be used to describe execution paths that values take in a higher-order program. As opposed to more classical semantics, such as big-step operational semantics, the meaning of programs computed by our data flow semantics is a map from execution points to *data flow* values.

Consider the illustrative program in Example 1 (left). We explain the meaning of the content to the right shortly. Each expression in the program is annotated with a unique location but we show only a portion of these locations.

Example 1. *A simple OCaml program (to the left) and its meaning (to the right)*

as given by the data flow semantics for higher-order untyped languages.

```
1  let id x = x^g in
2  let u = (id^a 1^b)^c in
3  (id^d 2^e)^f
```

$$\texttt{id} \to [a \mapsto \langle 1,1\rangle,\ d \mapsto \langle 2,2\rangle]$$
$$\texttt{u}, b, c \to 1, \quad a \to [a \mapsto \langle 1,1\rangle]$$
$$e, f \to 2, \qquad d \to [d \mapsto \langle 2,2\rangle]$$
$$g^a \to 1, \qquad g^d \to 2$$

The data flow map for our running example is shown in Example 1 to the right. Here, we identify execution points with a notation based on locations only, to avoid clutter. For instance, for the entries `b`, `c`, `u`, `e`, and `f`, the map shows the values to which the expressions annotated with those locations evaluated in the above program. As an example, the $e, f \to 2$ mapping indicates that the result of executing expressions associated with locations e and f is integer 2. The more interesting case is with execution points whose location is g. We have two such points, g^a and g^d, that store the results of evaluating the body of `id` stemming from the two invocations of `id` at call site points identified with locations a and d, respectively. In other words, we use a and d designations on g to informally describe different stack information present when the program evaluation reaches the expression with the location g two times in the above program. The valuation for points described so far are constants with the expected value, i.e., the same value one obtains using, say, a big-step operational semantics.

The more interesting cases occur at execution points that store function values. We model function values as *tables* that keep track of the call sites points at which the associated functions are called and the pair of input and output values produced. For instance, the function `id` is assigned a meaning as a table $[a \mapsto \langle 1,1\rangle,\ d \mapsto \langle 2,2\rangle]$ that stores input-output values for both call site points of `id`, a and d, where an input to `id` has been observed, i.e., where the function was

called. Interestingly, the execution points corresponding to the two usages of `id` are also tables but with only a fraction of the content computed for the definition of `id`. Intuitively, `id` takes two separate execution paths in the program, reaching locations a and d. For each execution point on these two paths, the associated tables capture how `id` has been used from that point onward in the corresponding paths. Tables stored at execution points a and d thus contain information about inputs and outputs of `id` for the call sites a and d, respectively.

As we show later, the computation in our data flow semantics boils down to propagation of information between data flow values stored at the successive execution points. On a high level, function inputs are propagated from call site points to the function definitions and the outputs are propagated back in the other direction. For instance, once it observes the input 1 generated at the call site `a`, the table assigned to the point `a` propagates this input to the table associated with the point `id`. After this input is evaluated, the produced output is stored in the table mapped to `id` and then propagated back to `a`. Note how this change of direction in which the inputs and outputs of functions are propagated intuitively corresponds to the input contravariance and output covariance of subtyping for function types.

3.1.3.2 Liquid Types as Abstract Interpretation

We continue our technical development by creating a series of semantics that abstract, in the sense of Galois connections, the concrete data flow semantics. This series of abstractions is shown in Figure 3.1. We use the concrete data flow semantics to define the properties of higher-order programs that are abstracted by the Liquid types. These properties are captured by the *collecting semantics*. We then abstract the collecting semantics by a *relational data flow semantics*. The basic

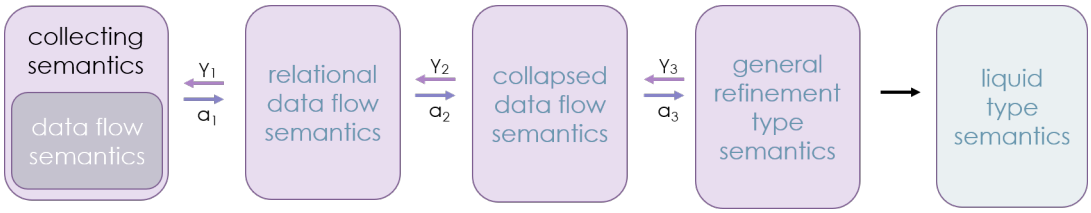

Figure 3.1: Series of abstract semantics leading up to Liquid type inference

idea behind the relational semantics is to abstract concrete data flow values by relations since Liquid types intuitively represent relations expressed in a certain decidable logic. In the next abstraction step, we introduce an abstract semantics that does not maintain the precise information on the call sites at which the inputs to functions are being generated. More precisely, the tables modeling functions in this new semantics, called the *collapsed semantics*, do not distinguish inputs by their call sites, as is the case with Liquid types and in contrast to the concrete and relational data flow semantics. The call site information in tables is collapsed and, in effect, functions are modeled essentially as single relations capturing the input-output behavior of the functions in the program. The key idea behind the *general refinement type semantics*, the next semantics in our abstraction chain, is to abstract relations with refinement types that consist of a basic type and an element of an arbitrary relational abstract domain representing type refinements. That is, the general refinement type inference is parametric in the abstract domain of type refinements. This semantics also ensures the sound termination of the type inference via widening operators [16, 17]. As we show, these operators are defined on the structure of refinement types using the available widening operators of the chosen abstract domain of type refinements. Finally, Liquid type inference is a particular instance of the general data flow refinement type inference.

Consequences. The benefits of modeling Liquid type inference as abstract interpretation are multifold. First, the design space, i.e., the precision loss points, of type inference algorithms developed using our framework are explicitly captured by Galois connections used in the abstraction chain and widening operators used to enforce the finite convergence of the analysis. Second, we obtain a generalization of the Liquid type inference in the form of general refinement type semantics. Any instantiation of this general system yields a data flow refinement type inference that is sound by construction. Lastly, type inference analyses that build on our results can be more easily compared against each other and used in combination with other static analyses (e.g., via reduced products [17]).

3.2 Data Flow Semantics

In this section, we formally introduce our data flow semantics of untyped functional programs.

Language. Our formal presentation in this chapter assumes that our language does not support let expressions. This simplification is made as we are not investigating polymorphism in this chapter. We name the resulting language λ^d. However, we will informally use let constructs in our examples to make them more readable. In general, our examples will use the constructs of more familiar programming languages, such as OCaml, that can be easily compiled to λ^d. For instance, the let constructs can be expanded into function applications of λ^d as expected [74]. The same holds for the conditional expressions.

3.2.1 Examples

Before we formally introduce the data flow semantics, we first provide more informal intuition through several additional examples.

3.2.1.1 Data Flow Computation

First, we explain how our semantics computes data flow maps. In Figure 3.2, we show the data flow map computed in each step by our data flow semantics for the program in Example 1. We only show the map entries for execution points storing tables, for simplicity. Initially, no useful information for execution points `id`, a, and d is known, denoted with $\bot$ in the map identified with (**1**). After evaluating line 1, `id`, being a function definition, is assigned the empty table $[\,]$ (**2**). This

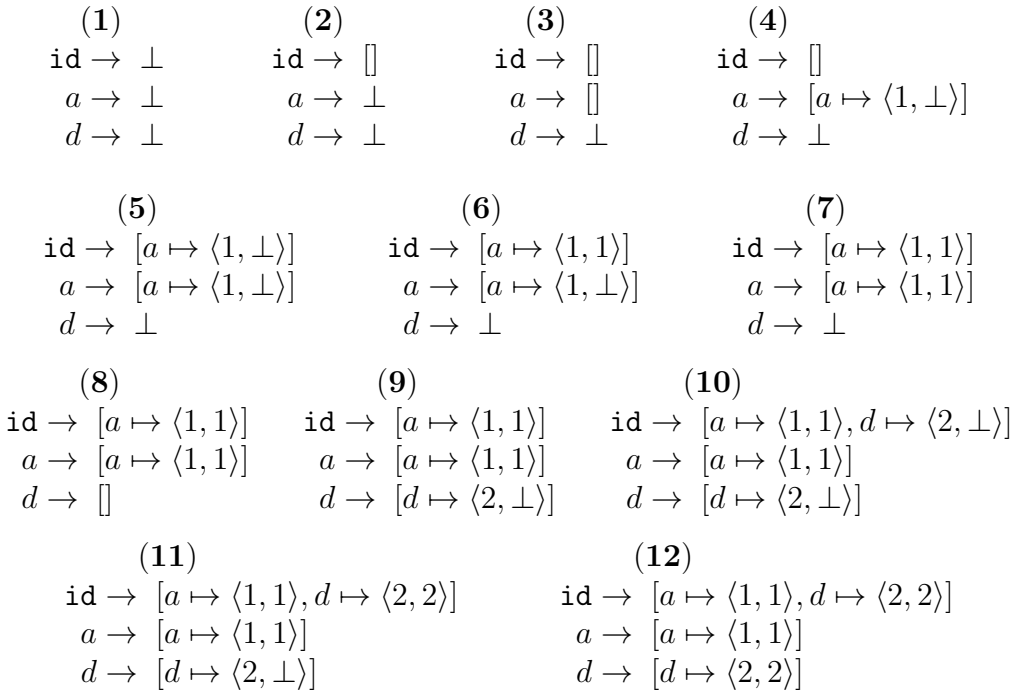

Figure 3.2: Subsequent maps computed by data flow semantics for program from Example 1. Numbers above maps are used to identify the maps.

table indicates that the meaning of `id` is a function, with currently no information on how this function is used in the program. In the next step (**3**), execution point a is reached and is also assigned [ ] based on the value of `id`. However, in the next step (**4**) our semantics observes that `id` is called with 1 at execution point a; the table associated with a hence contains this information. This information is then propagated back to the table associated with `id` in the next step (**5**). As we shall see later, this exchange of information is formally defined as propagation of data between the table associated with the point `id` and the table stored at a. Now that `id` has seen input 1, our semantics evaluates the body of `id` with 1, producing 1 as the result (**6**). In the next step (**7**), this output is propagated back in the other direction to the table stored at a. This change of direction in which inputs and outputs are propagated between tables intuitively corresponds to the contravariance of subtyping for function types [74]. After this step, the execution point d becomes reachable and is assigned the empty table [ ] (**8**). The data flow computation then continues in the similar fashion as described earlier. Note how a data flow value stored at some execution point, such as `id`, can be updated multiple times in different execution steps of our semantics.

3.2.1.2 Recursion

We now exemplify how our semantics models recursion through a program that can be found below in Example 2.

Example 2. *The OCaml program below first defines a function `f` that returns 0 when given 1 as input, and it otherwise just recurses with a decrement of the input. This function is then called two times with inputs 2 and 0, respectively. The latter call never terminates. To avoid clutter, we only annotate program expressions of*

particular interest with their locations.

```
let rec f n = if n=1 then 0 else f^a (n-1)
in (f^b 2^c) + (f^d 0^e)
```

A portion of the map computed by our data flow semantics for the above program is shown below. We again identify execution points with their locations.

$$b \rightarrow \quad [b \mapsto \langle 2, 0\rangle] \qquad\qquad d \rightarrow [d \mapsto \langle 0, \bot\rangle]$$

$$f \rightarrow \quad [b \mapsto \langle 2, 0\rangle, \;\; a^2 \mapsto \langle 1, 0\rangle,$$
$$d \mapsto \langle 0, \bot\rangle, \;\; a^0 \mapsto \langle -1, \bot\rangle, \;\; a^{-1} \mapsto \langle -2, \bot\rangle, \;\; \ldots]$$

Let us first discuss the execution point identified by b, which corresponds to the call to `f` with integer 2. The meaning assigned to b is a table indicating that the corresponding function is invoked at the call site b with integer 2, returning 0 as the output. The computation of this output clearly requires a recursive call to `f` with 1 as the input. The existence and effect of this call can be observed by the entry $a^2 \mapsto \langle 1, 0\rangle$ in the table associated with `f`. Intuitively, this entry states that `f` was called at the point identified by a, which is inside the body of `f` indicating recursion, while analyzing `f` with the previous input 2. Here, we use the integer value 2 designation on a to informally describe the stack information present when the execution reaches a for this particular case. We note that our choices of describing stack information in the examples are made for presentation purposes only; we formally explain how our data flow semantics encodes stack information in § 3.2.2.

Consider now the execution point identified by d, which corresponds to the call to `f` with integer 0. The meaning assigned to d is a table indicating that the corresponding function is invoked at the call site d with integer 0 but with no

computed output $\perp$. Clearly, there is no useful output computed since `f` does not terminate with 0 as the input. The recursive calls to `f` observed after calling `f` with 0 and their effects can be again observed in the table computed for `f`. For instance, the table entry $a^0 \mapsto \langle -1, \perp \rangle$ states that the recursive call to `f` at the call site point a has seen -1 as the input, but no output has been computed. The designation 0 on a again informally describes the stack information stating that this call to `f` is induced by the preceding call to `f` with 0 as the input. Similarly, the entry $a^{-1} \mapsto \langle -2, \perp \rangle$ indicates that `f` was recursively called at a with -2 as input and with no useful output observed. This recursive call was induced by the previous call to `f` with -1 as the input.

Observe how for every recursive call to `f` we remember the stack information describing the preceding call to `f` that led to this recursive call. That is, the table computed for `f` has entries of the form $a^i \mapsto \langle i - 1, \perp \rangle$ for every integer $i \leq 0$. Each call site point a^i explicitly stores the information about the previous call to `f`, here designated with the input integer i, that led to the current execution of the body of `f`. This way, we uniquely distinguish between different recursive calls to `f` in the program execution. We note again that the examples use a simplified notion of stack information for simplicity. As we shall see later in this section, each execution point a^i is modeled as pair of a particular execution environment and the location a. Since `f` is a recursive function, this environment will contain a certain binding for `f` itself. That binding will explicitly contain the information about the call site point, say x, of `f` that led to the current recursive execution of `f`. The call site x will in turn remember the information about the preceding call to `f` that ultimately led to x, and so on. Modeling stack information in this way allows us to uniquely identify execution points even in the presence of recursive

functions. For instance, suppose there are two calls with the same two argument values passed to a recursive function `f` in a program. These two top-level calls will induce recursive calls to `f` with the same argument values. Our semantics, however, is able to distinguish between these seemingly same recursive calls since they were induced by the two top-level calls to `f` happening at the two different call sites.

3.2.1.3 Higher-order Functions

We also illustrate how our semantics models higher-order functions of λ^d. Intuitively, a higher-order function will be modeled as a table some of whose inputs and outputs are also tables. We will work with the program shown in Example 3.

Example 3. *The OCaml program below first defines a constant function* `g1` *always returning 0, identity function* `g2`*, and a higher-order function* `h` *that accepts a function and simply calls it with integer 1. Then,* `h` *is called with* `g1` *and* `g2` *after which the respective results of these two calls are divided. This program is safe, returning 0 as the result. For simplicity, we only annotate few program expressions of interest with their locations.*

```
let h g = g 1^a in
let g1 x = 0 in
let g2 x = x in
(h^b g1^c)/(h^d g2^e)
```

The corresponding data flow map is shown below.

$$g1, c \to [a^b \mapsto \langle 1, 0\rangle] \quad g2, e \to [a^d \mapsto \langle 1, 1\rangle]$$

$$b \to [b \mapsto \langle [a^b \mapsto \langle 1, 0\rangle], 0\rangle] \quad d \to [d \mapsto \langle [a^d \mapsto \langle 1, 1\rangle], 1\rangle]$$

$$h \to [b \mapsto \langle [a^b \mapsto \langle 1, 0\rangle], 0\rangle, d \mapsto \langle [a^d \mapsto \langle 1, 1\rangle], 1\rangle]$$

Let us focus on the valuation for the function `h`. The meaning of `h` here is a table that has (non-bottom) entries for call site points identified with locations b and d, indicating that these are the two call site points from which the inputs to `h` are emitted. For the call site b, where `h` is applied to `g1`, the observed input-output pair consists of a table $T = [a^b \mapsto \langle 1, 0\rangle]$ and integer 0. Indeed, `h` produces 0 with `g1` as the input. The table T encodes the fact that `h` applies the input argument `g`, which is in this case `g1`, at the call site identified by a^b, i.e., inside the body of `h` at location a while analyzing `h` with the input seen at b. Note how we here again use the call site b to informally describe the stack information. Finally, the contents of T indicate that the input `g` of `h` was here called with integer 1 as the input, producing 0 as the output. Similar analysis applies for the second entry d of table for `h`. The explanation for other nodes in the map is similar.

We now proceed to the formal presentation of our data flow semantics for untyped higher-order programs.

3.2.2 Semantic Domains

We start our formal exposition of concrete data flow semantics by introducing semantic domains used for giving meaning to expressions of λ^d.

Nodes and environments

Every intermediate point of a program's execution is uniquely identified by an *execution node* (or simply node), $n \in \mathcal{N}$. We distinguish *expression nodes* $\mathcal{N}_e$ and *variable nodes* $\mathcal{N}_x$. An expression node $\langle E, \ell \rangle$, denoted $E \diamond \ell$, captures the execution point where a subexpression e^ℓ is evaluated in the environment E.

$$
\begin{aligned}
n \in \mathcal{N} &\stackrel{\text{def}}{=} \mathcal{N}_e \cup \mathcal{N}_x && \text{nodes} \\
\mathcal{N}_e &\stackrel{\text{def}}{=} \mathcal{E} \times \mathit{Loc} && \text{expression nodes} \\
\mathcal{N}_x &\stackrel{\text{def}}{=} \mathcal{E} \times \mathcal{N}_e \times \mathit{Vars} && \text{variable nodes} \\
E \in \mathcal{E} &\stackrel{\text{def}}{=} \mathit{Vars} \rightharpoonup_{\mathit{fin}} \mathcal{N}_x && \text{environments} \\
v \in \mathcal{V} &::= \bot \mid \omega \mid c \mid T && \text{(data flow) values} \\
T \in \mathcal{T} &\stackrel{\text{def}}{=} \mathcal{N}_e \to \mathcal{V} \times \mathcal{V} && \text{tables} \\
M \in \mathcal{M} &\stackrel{\text{def}}{=} \mathcal{N} \to \mathcal{V} && \text{execution maps}
\end{aligned}
$$

An environment E is a (finite) partial map binding variables to variable nodes. A variable node $\langle E, n_{\mathsf{cs}}, x \rangle$, denoted $E \diamond n_{\mathsf{cs}} \diamond x$, models the execution point where an argument value is bound to a formal parameter of a function. Here, E is the environment of the expression node where the function was defined. The expression node n_{cs} is referred to as the *call site* of the variable node. Intuitively, this is the node of the subexpression that emits the argument value of the function application associated with the binding. That is, if the call came from the expression $(e_1^{\ell_1}\, e_2^{\ell_2})$, then the location of n_{cs} is ℓ_2 and the environment of n_{cs} is the environment used when evaluating the function application. The call site is included in the variable node to uniquely identify the binding. This way, we implicitly model the call stack of a program in our semantics. Finally, x is the formal parameter that is

being bound to the variable node. We explain the role of expression and variable nodes in more detail later. For any node n, we denote by $env(n)$ the environment component of n and by $loc(n)$ the location component.

A pair $\langle e, E\rangle$ is called well-formed if $E(x)$ is defined for every variable x that occurs free in e.

Values and execution maps

There are four kinds of (data flow) values $v \in \mathcal{V}$. First, every constant c is also a value. The value $\bot$ stands for nontermination or unreachability of a node, and the value ω models execution errors. Functions are represented by *tables* T. A table maintains an input/output value pair for each call site. We denote by $T_\bot$ the empty table that maps every call site to the pair $\langle \bot, \bot\rangle$. We say that a table T has been called at a call site n_{cs} when the associated input value is non-bottom $n_{\mathsf{cs}} \in T \stackrel{\text{def}}{\Longleftrightarrow} \pi_1(T(n_{\mathsf{cs}})) \neq \bot$. We write $[n_{\mathsf{cs}} \mapsto \langle v_1, v_2\rangle]$ as shorthand for the table $T_\bot[n_{\mathsf{cs}} \mapsto \langle v_1, v_2\rangle]$.

The data flow semantics computes *execution maps* $M \in \mathcal{M}$, which map nodes to values. We denote by $M_\bot$ (M_ω) the execution maps that assign $\bot$ (ω) to every node. For a set of nodes N, we write $M_1 =_{\setminus N} M_2$ to express that the two maps M_1 and M_2 are pointwise equal on all nodes in $\mathcal{N} \setminus N$.

As a precursor to defining the data flow semantics, we define a *computational ordering* $\sqsubseteq$ on values. In this ordering, $\bot$ is the smallest element, ω is the largest element, and tables are ordered recursively on the pairs of values for each call site:

$$v_1 \sqsubseteq v_2 \stackrel{\text{def}}{\Longleftrightarrow} v_1 = \bot \ \vee \ v_2 = \omega \ \vee \ (v_1, v_2 \in \mathit{Cons} \ \wedge \ v_1 = v_2) \vee$$
$$(v_1, v_2 \in \mathcal{T} \wedge \forall n_{\mathsf{cs}}.\, v_1(n_{\mathsf{cs}}) \stackrel{\cdot}{\sqsubseteq} v_2(n_{\mathsf{cs}}))$$

Defining the error value as the largest element of the ordering is nonessential but simplifies the presentation. In particular, this definition ensures that the least upper bounds (lub) of arbitrary sets of values exist, which we denote by the join operator $\sqcup$.

$$\bot \sqcup v \stackrel{\text{def}}{=} v \qquad v \sqcup \bot \stackrel{\text{def}}{=} v \qquad\qquad c \sqcup c \stackrel{\text{def}}{=} c$$

$$T_1 \sqcup T_2 \stackrel{\text{def}}{=} \lambda n_{\mathsf{cs}}.\, T_1(n_{\mathsf{cs}}) \mathbin{\dot{\sqcup}} T_2(n_{\mathsf{cs}}) \qquad v_1 \sqcup v_2 \stackrel{\text{def}}{=} \omega \quad \text{(otherwise)}$$

The meet of values is defined in the expected way. These two operators together with the above ordering relation on values form a complete lattice.

Lemma 1. *Let $V \in \wp(\mathcal{V})$. Then, $\sqcup V = lub(V)$ and $\sqcap V = glb(V)$ subject to $\sqsubseteq$.*

As expected, execution maps also form a complete lattice with the ordering relation $\dot{\sqsubseteq}$, join $\dot{\sqcup}$, and meet $\dot{\sqcap}$ operators that are simply pointwise lifts of the corresponding ordering relation and operators on values.

3.2.3 Concrete Transformer

The data flow semantics of an expression e is defined as the least fixpoint of a concrete transformer, step, on execution maps. The idea is that we start with the map $M_\bot$ and then use step to consecutively update the map with new values as more and more execution nodes are reached. The signature of this transformer is as follows:

$$\mathsf{step} : \mathbb{N} \to \lambda^d \to \mathcal{E} \to \mathcal{M} \to \mathcal{M}$$

Essentially, step takes $k \in \mathbb{N}$, an expression e, and an environment E and returns a transformer $\mathsf{step}_k[\![e]\!](E) : \mathcal{M} \to \mathcal{M}$ on execution maps. The definition of the transformer is given in Fig. 3.3 using induction over both k and the structure of e. Note that in the definition we implicitly assume that $\langle e, E\rangle$ is well-formed. Let

$$\mathsf{step}_0[\![e]\!](E)(M) \stackrel{\text{def}}{=} M$$

$$\mathsf{step}_{k+1}[\![c^\ell]\!](E)(M) \stackrel{\text{def}}{=} M[E\diamond\ell \to M(E\diamond\ell) \sqcup c]$$

$$\mathsf{step}_{k+1}[\![x^\ell]\!](E)(M) \stackrel{\text{def}}{=}$$
$$\quad \textbf{let } \langle v'_x, v'\rangle = \mathsf{prop}(M(E(x)), M(E\diamond\ell))$$
$$\quad \textbf{in } M[E(x) \mapsto v'_x, E\diamond\ell \mapsto v']$$

$$\mathsf{step}_{k+1}[\![(e_1^{\ell_1} e_2^{\ell_2})^\ell]\!](E)(M) \stackrel{\text{def}}{=}$$
$$\quad \textbf{let } M_1 = \mathsf{step}_k[\![e_1]\!](E)(M);\ v_1 =_{M_1} M_1(E\diamond\ell_1) \textbf{ in}$$
$$\quad \textbf{if } v_1 \notin \mathcal{T} \textbf{ then } M_\omega \textbf{ else}$$
$$\quad\ \textbf{let } M_2 = \mathsf{step}_k[\![e_2]\!](E)(M_1);\ v_2 =_{M_2} M_2(E\diamond\ell_2) \textbf{ in}$$
$$\quad\ \textbf{let } \langle v'_1, T'\rangle = \mathsf{prop}(v_1, [E\diamond\ell_1 \mapsto \langle v_2, M_2(E\diamond\ell)\rangle])$$
$$\qquad \langle v'_2, v'\rangle = T'(E\diamond\ell_1)$$
$$\quad\ \textbf{in } M_2[E\diamond\ell_1 \mapsto v'_1, E\diamond\ell_2 \mapsto v'_2, E\diamond\ell \mapsto v']$$

$$\mathsf{step}_{k+1}[\![(\mu f.\lambda x.e_1^{\ell_1})^\ell]\!](E)(M) \stackrel{\text{def}}{=}$$
$$\quad \textbf{let } T =_M T_\perp \sqcup M(E\diamond\ell) \textbf{ in}$$
$$\quad \textbf{let } \overline{M} = \Lambda n_{\mathsf{cs}} \in T.\, \textbf{if } \pi_1(T(n_{\mathsf{cs}})) = \omega \textbf{ then } M_\omega \textbf{ else}$$
$$\quad\ \textbf{let } n_x{=}E\diamond n_{\mathsf{cs}}\diamond x;\ n_f{=}E\diamond n_{\mathsf{cs}}\diamond f;\ E_1{=}E[x{\mapsto}n_x, f{\mapsto}n_f]$$
$$\qquad \langle T'_x, T'\rangle = \mathsf{prop}([n_{\mathsf{cs}} \mapsto \langle M(n_x), M(E_1\diamond\ell_1)\rangle], T)$$
$$\qquad \langle T'', T'_f\rangle = \mathsf{prop}(T, M(n_f));\ \langle v'_x, v'_1\rangle = T'_x(n_{\mathsf{cs}})$$
$$\qquad M_1 = M[E\diamond\ell{\mapsto}T' \sqcup T'', n_f{\mapsto}T'_f, n_x{\mapsto}v'_x, E_1\diamond\ell_1{\mapsto}v'_1]$$
$$\quad\ \textbf{in } \mathsf{step}_k[\![e_1]\!](E_1)(M_1)$$
$$\quad \textbf{in } M[E\diamond\ell \mapsto T] \dot\sqcup \dot{\bigsqcup}_{n_{\mathsf{cs}}\in T} \overline{M}(n_{\mathsf{cs}})$$

$$\mathsf{step}_{k+1}[\![(e_0^{\ell_0}\ ?\ e_1^{\ell_1} : e_2^{\ell_2})^\ell]\!](E)(M) \stackrel{\text{def}}{=}$$
$$\quad \textbf{let } M_0 = \mathsf{step}_k[\![e_0]\!](E)(M);\ v_0 =_{M_0} M_0(E\diamond\ell_0) \textbf{ in}$$
$$\quad \textbf{if } v_0 \notin \mathit{Bool} \textbf{ then } M_\omega \textbf{ else}$$
$$\quad\ \textbf{let } b = \textbf{if } v_0 = \mathit{true} \textbf{ then } 1 \textbf{ else } 2 \textbf{ in}$$
$$\quad\ \textbf{let } M_b = \mathsf{step}_k[\![e_b]\!](E)(M_0);\ v_b =_{M_b} M_b(E\diamond\ell_b) \textbf{ in}$$
$$\quad\ \textbf{let } \langle v'_b, v'\rangle = \mathsf{prop}(v_b, M_b(E\diamond\ell))$$
$$\quad\ \textbf{in } M_b[E\diamond\ell \mapsto v', E\diamond\ell_b \mapsto v'_b]$$

Figure 3.3: Concrete transformer for data flow semantics

$$
\begin{array}{l}
\mathsf{prop}(T_1, T_2) \stackrel{\text{def}}{=} \\
\quad \textbf{let } T' = \Lambda n_{\mathsf{cs}}. \\
\qquad \textbf{if } n_{\mathsf{cs}} \notin T_2 \textbf{ then } \langle T_1(n_{\mathsf{cs}}), T_2(n_{\mathsf{cs}})\rangle \textbf{ else} \\
\qquad \textbf{let } \langle v_{1i}, v_{1o}\rangle = T_1(n_{\mathsf{cs}})\textbf{; } \langle v_{2i}, v_{2o}\rangle = T_2(n_{\mathsf{cs}}) \\
\qquad\quad \langle v'_{2i}, v'_{1i}\rangle = \mathsf{prop}(v_{2i}, v_{1i})\textbf{; } \langle v'_{1o}, v'_{2o}\rangle = \mathsf{prop}(v_{1o}, v_{2o}) \\
\qquad \textbf{in } (\langle v'_{1i}, v'_{1o}\rangle, \langle v'_{2i}, v'_{2o}\rangle) \\
\quad \textbf{in } \langle \Lambda n_{\mathsf{cs}}.\ \pi_1(T'(n_{\mathsf{cs}})), \Lambda n_{\mathsf{cs}}.\ \pi_2(T'(n_{\mathsf{cs}}))\rangle
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{prop}(T, \bot) \stackrel{\text{def}}{=} & \langle T, T_\bot\rangle & \mathsf{prop}(T, \omega) \stackrel{\text{def}}{=} \langle \omega, \omega\rangle \\
\mathsf{prop}(v_1, v_2) \stackrel{\text{def}}{=} & \langle v_1, v_1 \sqcup v_2\rangle & \textbf{(otherwise)}
\end{array}
$$

Figure 3.4: Value propagation in the concrete data flow semantics

us ignore the parameter k for a moment and focus on the equations that case split on the structure of e. We discuss the cases for e one at a time.

Constant $e = c^\ell$. Here, we simply set the current node to the join of the current value stored for $E{\diamond}\ell$ and the value c. The use of join is non-essential, but it simplifies our presentation; in practice, the result of the join is always c.

Variable $e = x^\ell$. This case is already more interesting. It implements the data flow propagation between the variable node $E(x)$ where x is bound and the expression node $E{\diamond}\ell$ where x is used. This is realized using the helper function prop defined in Fig. 3.4. Let $v_x = M(E(x))$ and $v = M(E{\diamond}\ell)$ be the current values stored at the two nodes in M. The function prop takes these values as input and propagates information between them, updating them to the new values v'_x and v' which are then stored back into M. The propagation works as follows. If v_x is a constant or the error value and v is still $\bot$, then we simply propagate v_x for-

ward, replacing v and leaving v_x unchanged. The interesting cases are when we propagate information between tables. The idea is that inputs in a table v flow backward to v_x whereas outputs for these inputs flow forward from v_x to v. Thus, if v_x is a table but v is still $\bot$, we initialize v to the empty table $T_\bot$ and leave v_x unchanged (because we have not yet accumulated any inputs in v). If both v_x and v are tables, we propagate inputs and outputs as described above by calling prop recursively for every call site n_{cs} that is called in v. Note how the recursive call for the propagation of the inputs $\mathsf{prop}(v_{2i}, v_{1i})$ inverts the direction of the propagation. As we shall see, this will give rise to contravariant subtyping of function types.

Function application $e = (e_1^{\ell_1}\, e_2^{\ell_2})^{\ell}$. We first evaluate e_1 to obtain the updated map M_1 and extract the new value v_1 stored at the corresponding expression node. Here, we use a special meta-level let binding $\textbf{let}\ x =_M v$. That is, $\textbf{let}\ x =_M v\ \textbf{in}\ t$ shortcircuits to M if $v = \bot$, M_ω if $v = \omega$, and otherwise yields t where x is bound to v. Hence, if the $E{\diamond}\ell_1$ node has not yet been reached ($v_1 = \bot$), we return M_1. If v_1 is not a table, then we must be attempting an unsafe call, in which case we return the error map M_ω. If v_1 is a table, we continue evaluation of e_2 obtaining the new map M_2 and value v_2 at the associated expression node $E{\diamond}\ell_2$. If v_2 is neither $\bot$ nor ω, then we need to propagate the information between this call site and v_1. To this end, we first extract the return value $v = M(E{\diamond}\ell)$ for the node of e computed thus far and create a singleton table storing the input-output $\langle v_2, v\rangle$ for this call site $E{\diamond}\ell_1$. We propagate between v_1 and this table to obtain v_1' and a table T'. Note that this propagation essentially boils to down to (1) the propagation between v_2 and the input of v_1 at $E{\diamond}\ell_1$ and (2) the propagation between the output of v_1 at $E{\diamond}\ell_1$ and the return value v. The updated table T' hence contains the

updated input and output values v_2' and v'. All of these values are stored back into M_2. Intuitively, v_1' contains the information that the corresponding function received an input coming from the call site $E \diamond \ell_1$. This information will ultimately be propagated back to the function definition that will in turn evaluate this input, as follows.

Recursive function $e = (\mu f.\lambda x.e_1^{\ell_1})^{\ell}$. We first extract the table T computed for the function thus far. Then, for every call site n_{cs} for which an input has already been back-propagated to T (i.e. $\pi_1(T(n_{\mathsf{cs}})) \neq \bot$), we do the following. First, we create a variable node n_x that will store the input value that was intuitively emitted at the call site n_{cs}. The contents of this value can potentially be updated later on during the computation if the actual input is a table that, for instance, has yet to be called. We hence define n_x using n_{cs} as a unique variable node $E \diamond n_{\mathsf{cs}} \diamond x$, thus ensuring the execution of the lambda body for other inputs does not interfere with the execution for this input. The similar reasoning underlies the creation of n_f. The unique nature of n_f allows our semantics to precisely track recursive calls to f that occurred while evaluating the input to f coming from n_{cs}. Also, note that by remembering the call site node n_{cs} in the environment nodes n_x and n_f we effectively model the program stack. That is, each call site node n_{cs} contains an environment that in turn contains environment nodes, which again keep track of the call site information, and so on.

We then propagate information between the values stored at the node for the body e_1, the nodes n_x and n_f for the bound variables, and the table T. That is, we propagate information from (1) the input of T at n_{cs} and the corresponding environment node n_x, and (2) the value assigned to the function body under the

update environment E_1 and the output of T at n_{cs}. Further, the value associated with n_f is a table that will intuitively contain information about the recursive calls to the function observed during the analysis of the body for the current input. The propagation between T and the table stored at n_f allows T to pick up such observed inputs and eventually propagate the corresponding outputs back to $M(n_f)$.

The new values obtained by propagation are stored back into the map to obtain a new map M_1 and then we evaluate the body e_1. The updated maps obtained for all call sites are then merged into a single map via a join. Note that this join does not lose precision by introducing ω. This is because the extended environment E_1 is unique for each call site, which means that the evaluation of the body e_1 for different call sites will update different expression nodes. Moreover, a concurrent update to the same variable node n in two maps $\overline{M}(n_{\mathsf{cs}})$ and $\overline{M}(n'_{\mathsf{cs}})$ can only occur if $M(n)$ stores a table and the updates are the result of back-propagation of inputs and outputs. In this case, these inputs and outputs must agree if they come from the same call site and the updated tables in $\overline{M}(n_{\mathsf{cs}})(n)$ and $\overline{M}(n'_{\mathsf{cs}})(n)$ can be safely merged via join. This reflects the deterministic nature of the concrete semantics.

Conditionals. The discussion for conditional expressions is similar to the previous expression cases.

Lemma 2. *The function* prop *is monotone and increasing.*

Lemma 3. *For every* $k \in \mathbb{N}$, $e \in \lambda^d$ *and* $E \in \mathcal{E}$ *such that* $\langle e, E\rangle$ *is well-formed, then* $\mathsf{step}_k[\![e]\!](E)$ *is monotone and increasing.*

Step-indexing. Let us now return to the discussion of the parameter k of step. Intuitively, we would like to just do away with k and define the semantics of a program e as the least fixpoint of step over the complete lattice of execution maps. However, if e does not terminate, then the tables stored in the fixpoint iterates may grow arbitrarily deep. Since tables are defined inductively, they can only have finite nesting depth, which means that the fixpoint would approximate these unbounded tables with ω. Possible solutions include to define tables coinductively or to use a different representation of tables based on Scott domains [80]. However, we here use a simpler solution inspired by step-indexing [2]. We introduce the step-index parameter k, informally called *fuel*, that bounds the number of function applications executed by $\mathsf{step}_k[\![e]\!](E)(M)$ and, in turn, the nesting depth of the tables in the resulting execution map. The fixpoint then simply computes a separate map for every *fuel* k.

We define the semantics $\mathbf{S}[\![e]\!]$ of a program e as the least fixpoint of step over the complete lattice of step-indexed execution maps $\mathbb{N} \to \mathcal{M}$, which is ordered by the pointwise lifting of the computational ordering on values:

$$\mathbf{S}[\![e]\!] \stackrel{\text{def}}{=} \mathbf{lfp}^{\ddot{\sqsubseteq}}_{\Lambda k.\, M_\perp} \Lambda \mathbf{M}.\, \Lambda k.\, \mathsf{step}_k[\![e]\!](\epsilon)(\mathbf{M}(k))$$

Here, ϵ denotes the empty environment. Lemma 3 guarantees that $\mathbf{S}[\![e]\!]$ is well-defined.

In addition to the example from the beginning of this section, we also show the final execution map for the example from § 3.1 in Appendix A.

3.2.4 Properties and Collecting Semantics

As we shall see, Liquid type systems abstract programs by *properties* $P \in \mathcal{P}$, which are step-indexed sets of execution maps: $\mathcal{P} \stackrel{\text{def}}{=} \mathbb{N} \to \wp(\mathcal{M})$. We order properties by the pointwise lifting of subset inclusion. Properties form the concrete domain of our abstract interpretation. We therefore raise the data flow semantics $\mathbf{S}$ to a *collecting semantics* $\mathbf{C} : \lambda^d \to \mathcal{P}$ that maps programs to properties:

$$\mathbf{C}[\![e]\!] \stackrel{\text{def}}{=} \Lambda k.\{\mathbf{S}[\![e]\!](k)\}$$

An example of a property is *safety*: let $\mathcal{M}_{\mathsf{safe}}$ be the set of all execution maps that map no node to the error value ω and define $P_{\mathsf{safe}} = \Lambda k.\mathcal{M}_{\mathsf{safe}}$. Then a program e is called *safe* if $\mathbf{C}[\![e]\!] \dot{\subseteq} P_{\mathsf{safe}}$.

3.3 Relational Data Flow Semantics

We now present the first abstraction of the concrete data flow semantics in the sequence of abstract semantics leading up to Liquid types. As we shall see later, a critical abstraction step performed by Liquid types is to conflate the information in tables that are propagated back from different call sites of that function. That is, a pair of input/output values that has been collected from one call site will also be considered as a possible input/output pair at other call sites to which that function flows. However, in order for Liquid type inference not to lose too much precision, output values observed for a particular call site explicitly remember the corresponding input value. We refer to this semantics as the *relational (data flow) semantics*.

3.3.1 Example

We first provide an intuition behind relational semantics through our running program from Example 1. Relational semantics abstracts the value of every node n in an execution map by a relation over the values observed at n and the values observed for the scope of n that is defined by $env(n)$.

$$\texttt{id} \to [a \to \langle\{(\nu\colon 1)\}, \{(a\colon 1, \nu\colon 1)\}\rangle, d \to \langle\{(\nu\colon 2)\}, \{(d\colon 2, \nu\colon 2)\}\rangle]$$

$$\texttt{u}, b, c \to \{(\texttt{id}\colon \mathsf{F}, \nu\colon 1)\} \qquad e, f \to \{(\texttt{id}\colon \mathsf{F}, \texttt{u}\colon 1, \nu\colon 2)\}$$

$$a \to [a \mapsto \langle\{(\texttt{id}\colon \mathsf{F}, \nu\colon 1)\}, \{(\texttt{id}\colon \mathsf{F}, a\colon 1, \nu\colon 1)\}\rangle]$$

$$d \to [d \mapsto \langle\{(\texttt{id}\colon \mathsf{F}, \texttt{u}\colon 1, \nu\colon 2)\}, \{(\texttt{id}\colon \mathsf{F}, \texttt{u}\colon 1, d\colon 2, \nu\colon 2)\}\rangle]$$

$$g^a \to \{(x^a\colon 1, \nu\colon 1)\} \qquad g^d \to \{(x^d\colon 2, \nu\colon 2)\}$$

In the above, we present the final map computed by our relational semantics for our running example. Let us explain the map by first focusing on nodes $\texttt{u}$, b, and c. The concrete semantics assigns integer 1 to these nodes, as explained in § 3.2, while relational semantics assigns the relational value $\{(\texttt{id}\colon \mathsf{F}, \nu\colon 1)\}$. This value is a relation consisting of a single tuple $(\texttt{id}\colon \mathsf{F}, \nu\colon 1)$. This tuple states that the value possibly observed for the nodes $\texttt{u}$, b, and c is integer 1, via mapping $\nu\colon 1$. Observe that, similar to Liquid types, we use the symbolic variable ν to refer to the values of the current node/expression. On the other hand, the mapping $\texttt{id}\colon \mathsf{F}$ in the tuple indicates that the node associated with $\texttt{id}$ is in the current scope and it contains some function/table value. Note how this mapping does not state what is that table exactly, yet only the fact that $\texttt{id}$ is a function. This particular design choice in modeling relations is motivated by the fact that Liquid types do not track precise dependencies on function values. The relation $\{(\texttt{id}\colon \mathsf{F}, \texttt{u}\colon 1, \nu\colon 2)\}$

computed for nodes e and f states that these nodes possibly evaluate to integer 2, where the values in the scope corresponding to `id` and `u` are a function and integer 1, respectively. Note how the relations described so far contain only a single tuple. As we show later, refinements of basic types will concretize to relations having multiple tuples of the form explained above.

Let us now turn to explaining relational tables by considering the entry for `id` node in the above map. Let us discuss the table entry $a \mapsto \langle\{(\nu : 1)\}, \{(a : 1, \nu : 1)\}\rangle$. As expected, this entry indicates that `id` is called at the node identified with a. Here, the observed input is a relational value $\{(\nu : 1)\}$ indicating that the actual input passed to `id` at b is integer 1. Note how there are no other mappings in the tuple of the input relation except $\nu : 1$. This is because the initial environment/scope, used to analyze `id`, is empty. The output relation $\{(a : 1, \nu : 1)\}$ is more interesting. This value indicates via the pair $\nu : 1$ that the actual output of `id` observed at the call site a is 1. Additionally, the output value also explicitly remembers the input value through the mapping $a : 1$ where the call site node a is used to refer to the input value. This way, relational semantics tracks dependencies between inputs and outputs, which becomes important in the abstraction steps we introduce later on.

The maps computed by the relational data flow semantics for programs of Example 3 and Example 2 are obtained similarly. For instance, here is the relational

map for our program from Example 3 exhibiting higher-order features of λ^d.

$$g1, b \to [a^b \mapsto \langle\{(\nu\colon 1)\}, \{(a^b\colon 1, \nu\colon 0)\}\rangle] \quad g2, d \to [a^d \mapsto \langle\{(\nu\colon 1)\}, \{(a^d\colon 1, \nu\colon 1)\}\rangle]$$

$$b \to [b \mapsto \langle[a^b \mapsto \langle\{(\nu\colon 1)\}, \{(a^b\colon 1, \nu\colon 0)\}\rangle], \{(b\colon \mathsf{F}, \nu\colon 0)\}\rangle]$$

$$d \to [d \mapsto \langle[a^d \mapsto \langle\{(\nu\colon 1)\}, \{(a^d\colon 1, \nu\colon 1)\}\rangle], \{(d\colon \mathsf{F}, \nu\colon 1)\}\rangle]$$

$$h \to [b \mapsto \langle[a^b \mapsto \langle\{(\nu\colon 1)\}, \{(a^b\colon 1, \nu\colon 0)\}\rangle], \{(b\colon \mathsf{F}, \nu\colon 0)\}\rangle,$$
$$d \mapsto \langle[a^d \mapsto \langle\{(\nu\colon 1)\}, \{(a^d\colon 1, \nu\colon 1)\}\rangle], \{(d\colon \mathsf{F}, \nu\colon 1)\}\rangle]$$

Note again the use call site points in tables to express input-output dependencies. To avoid clutter, we also do not show all entries for scope nodes within rows of relations. For instance, input and output relations in the relational tables of b and d should also contain entries $h\colon \mathsf{F}$, $g1\colon \mathsf{F}$, and $g2\colon \mathsf{F}$.

3.3.2 Abstract Domains

We start our formal exposition of relational semantics by introducing the abstract domains. The definitions of nodes and environments are the same as in the concrete semantics. The relational abstractions of values, constants, and tables are defined with respect to *scopes*, which are finite sets of nodes $N \subseteq \mathcal{N}$. Given a node n, we denote its scope by $N_n \stackrel{\text{def}}{=} \mathsf{rng}(env(n))$. The new domains are defined as follows:

$$\begin{array}{rcll}
r_N \in \mathcal{V}^{\mathsf{r}}_N & ::= & \bot^{\mathsf{r}} \mid \top^{\mathsf{r}} \mid R^{\mathsf{r}}_N \mid T^{\mathsf{r}}_N & \text{relational values} \\
R^{\mathsf{r}}_N \in \mathcal{R}^{\mathsf{r}}_N & \stackrel{\text{def}}{=} & \wp(\mathcal{D}^{\mathsf{r}}_N) & \text{relations} \\
D^{\mathsf{r}}_N \in \mathcal{D}^{\mathsf{r}}_N & \stackrel{\text{def}}{=} & N \cup \{\nu\} \to \mathit{Cons} \cup \{\mathsf{F}\} & \text{dependency vectors} \\
T^{\mathsf{r}}_N \in \mathcal{T}^{\mathsf{r}}_N & \stackrel{\text{def}}{=} & \Pi n_{\mathsf{cs}} \in \mathcal{N}_e.\ \mathcal{V}^{\mathsf{r}}_{N\setminus\{n_{\mathsf{cs}}\}} \times \mathcal{V}^{\mathsf{r}}_{N\cup\{n_{\mathsf{cs}}\}} & \text{relational tables} \\
M^{\mathsf{r}} \in \mathcal{M}^{\mathsf{r}} & \stackrel{\text{def}}{=} & \Pi n \in \mathcal{N}.\ \mathcal{V}^{\mathsf{r}}_{N_n} & \text{relational maps}
\end{array}$$

Relational values r_N model how concrete values, stored at some node, depend on the concrete values of nodes in the current scope N. The relational value $\bot^{\mathsf{r}}$ again models nontermination or unreachability and imposes no constraints on the values in its scope. The relational value $\top^{\mathsf{r}}$ models every possible concrete value, including ω. Concrete constant values are abstracted by relations R^{r}_N, which are sets of *dependency vectors* D^{r}_N. A dependency vector tracks the dependency between a constant value stored at the current node, represented by the special symbol ν, and the values stored at the other nodes in the current scope N. Here, we assume that ν is not contained in $\mathcal{N}$.

We only track dependencies between constant values precisely: if a node in the scope stores a table, we abstract it by the symbol F which stands for an arbitrary concrete table. We assume F to be different from all other constants. We also require that for all $D^{\mathsf{r}}_N \in r_N$, $D^{\mathsf{r}}(\nu) \in \mathit{Cons}$. We will see later that the inferred refinement relations $\phi(\nu, \vec{x})$ of Liquid types are abstractions of dependency relations. That is, every satisfying assignment of a $\phi(\nu, \vec{x})$ concretizes to a set of dependency vectors.

Relational tables T^{r} are defined analogously to concrete tables except that they now map call sites n_{cs} to pairs of relational values $\langle r_i, r_o \rangle$ rather than concrete values. The call site node is not allowed to be in the scope of the input relational r_i but is explicitly added to the scope of r_o. The idea is that r_o tracks the dependency between the output value and the input values described by r_i via n_{cs}. We denote by $T^{\mathsf{r}}_{\bot^{\mathsf{r}}}$ the empty relational table that maps every call site to the pair $\langle \bot^{\mathsf{r}}, \bot^{\mathsf{r}} \rangle$. We denote the fact that a relational table T^{r} has been invoked at a call site n_{cs} by $n_{\mathsf{cs}} \in T^{\mathsf{r}}$. Relational execution maps M^{r} assign each node n a relational value with scope N_n. We denote by $M^{\mathsf{r}}_{\bot^{\mathsf{r}}}$ ($M^{\mathsf{r}}_{\top^{\mathsf{r}}}$) the relational execution maps that assign $\bot^{\mathsf{r}}$

$(\top^{\mathsf{r}})$ to every node.

We denote by $\mathcal{V}^{\mathsf{r}}$ the union of all $\mathcal{V}^{\mathsf{r}}_N$ and use similar notation for the sets of all dependency relations and relational tables. For the members of these sets we sometimes omit the subscript indicating the scope when it is irrelevant or clear from the context. We then define a partial order $\sqsubseteq^{\mathsf{r}}$ on $\mathcal{V}^{\mathsf{r}}$, which is analogous to the partial order $\sqsubseteq$ on concrete values, except that equality on constants is replaced by subset inclusion on dependency relations:

$$r_1 \sqsubseteq^{\mathsf{r}} r_2 \iff^{\text{def}} r_1{=}\bot^{\mathsf{r}} \vee r_2{=}\top^{\mathsf{r}} \vee (r_1, r_2 \in \mathcal{R}^{\mathsf{r}}_N \wedge r_1 \subseteq r_2) \vee (r_1, r_2 \in \mathcal{T}^{\mathsf{r}}_N \wedge \forall n_{\mathsf{cs}}.\ r_1(n_{\mathsf{cs}}) \mathrel{\dot{\sqsubseteq}^{\mathsf{r}}} r_2(n_{\mathsf{cs}}))$$

We note that aside from $\bot^{\mathsf{r}}$ and $\top^{\mathsf{r}}$, only the relational values of identical scope N are comparable. This ordering induces for every N, a complete lattice $(\mathcal{V}^{\mathsf{r}}_N, \sqsubseteq^{\mathsf{r}}, \bot^{\mathsf{r}}, \top^{\mathsf{r}}, \sqcup^{\mathsf{r}}, \sqcap^{\mathsf{r}})$. The definition of join over relational values is given below.

$$\bot^{\mathsf{r}} \sqcup^{\mathsf{r}} r \stackrel{\text{def}}{=} r \qquad r \sqcup^{\mathsf{r}} \bot^{\mathsf{r}} \stackrel{\text{def}}{=} r \qquad R^{\mathsf{r}}_1 \sqcup^{\mathsf{r}} R^{\mathsf{r}}_2 \stackrel{\text{def}}{=} R^{\mathsf{r}}_1 \cup R^{\mathsf{r}}_2$$

$$T^{\mathsf{r}}_1 \sqcup^{\mathsf{r}} T^{\mathsf{r}}_2 \stackrel{\text{def}}{=} \lambda n_{\mathsf{cs}}.\, T^{\mathsf{r}}_1(n_{\mathsf{cs}}) \mathrel{\dot{\sqcup}^{\mathsf{r}}} T^{\mathsf{r}}_2(n_{\mathsf{cs}}) \qquad r_1 \sqcup^{\mathsf{r}} r_2 \stackrel{\text{def}}{=} \top^{\mathsf{r}} \quad \text{(otherwise)}$$

The meet $\sqcap^{\mathsf{r}}$ is defined similarly.

Lemma 4. *Let $V \in \wp(\mathcal{V}^{\mathsf{r}}_N)$. Then, $\sqcup^{\mathsf{r}} V = lub(V)$ and $\sqcap^{\mathsf{r}} V = glb(V)$ subject to $\sqsubseteq^{\mathsf{r}}$.*

The lattices on relational values can be lifted pointwise to a complete lattice on relational execution maps $(\mathcal{M}^{\mathsf{r}}, \mathrel{\dot{\sqsubseteq}^{\mathsf{r}}}, M^{\mathsf{r}}_{\bot^{\mathsf{r}}}, M^{\mathsf{r}}_{\top^{\mathsf{r}}}, \dot{\sqcup}^{\mathsf{r}}, \dot{\sqcap}^{\mathsf{r}})$.

Galois connections

We define the meaning of relational execution maps in terms of a Galois connection between $\mathcal{M}^{\mathsf{r}}$ and the complete lattice of sets of concrete execution maps $\wp(\mathcal{M})$, which we can then lift to properties $\mathcal{P} = \mathbb{N} \to \wp(\mathcal{M})$. To this end, we first

formalize the intuitive meaning of relational values that we have provided above. We do this using a family of Galois connections between the complete lattices $\mathcal{V}^{\mathsf{r}}_N$ and the complete lattice of sets of pairs of execution maps and values, $\wp(\mathcal{M} \times \mathcal{V})$, which is ordered by subset inclusion. These Galois connections share the same concretization function $\gamma^{\mathsf{r}} : \mathcal{V}^{\mathsf{r}} \to \wp(\mathcal{M} \times \mathcal{V})$, which is defined as:

$$\gamma^{\mathsf{r}}(\bot^{\mathsf{r}}) \stackrel{\text{def}}{=} \mathcal{M} \times \{\bot\} \qquad \gamma^{\mathsf{r}}(\top^{\mathsf{r}}) \stackrel{\text{def}}{=} \mathcal{M} \times \mathcal{V}$$

$$\gamma^{\mathsf{r}}(R^{\mathsf{r}}_N) \stackrel{\text{def}}{=} \{\langle M, c\rangle \mid D^{\mathsf{r}} \in R^{\mathsf{r}}_N \wedge D^{\mathsf{r}}(\nu) = c \wedge \forall n \in N.\, M(n) \in \gamma^{\mathsf{d}}(D^{\mathsf{r}}(n))\} \cup \gamma^{\mathsf{r}}(\bot^{\mathsf{r}})$$

$$\gamma^{\mathsf{r}}(T^{\mathsf{r}}_N) \stackrel{\text{def}}{=} \{\langle M, T\rangle \mid \forall n_{\mathsf{cs}}.\, T(n_{\mathsf{cs}}){=}\langle v_i, v_o\rangle \wedge T^{\mathsf{r}}(n_{\mathsf{cs}}){=}\langle r_i, r_o\rangle \wedge \langle M_i, v_i\rangle \in \gamma^{\mathsf{r}}(r_i) \wedge \langle M_o, v_o\rangle \in \gamma^{\mathsf{r}}(r_o) \wedge M_i =_{\setminus\{n_{\mathsf{cs}}\}} M \wedge M_o{=}M_i[n_{\mathsf{cs}} \mapsto v_i]\} \cup \gamma^{\mathsf{r}}(\bot^{\mathsf{r}})$$

Here, the function γ^{d}, which is used to give meaning to dependency relations, is defined by $\gamma^{\mathsf{d}}(c) = \{c\}$ and $\gamma^{\mathsf{d}}(\mathsf{F}) = \mathcal{T}$.

Lemma 5. *For all scopes N, γ^{r} is a complete meet-morphism between $\mathcal{V}^{\mathsf{r}}_N$ and $\wp(\mathcal{M} \times \mathcal{V})$.*

It follows that there exists a Galois connection between $\wp(\mathcal{M} \times \mathcal{V})$ and $\mathcal{V}^{\mathsf{r}}_N$ for every scope N which has γ^{r} as its right adjoint [17]. Let $\alpha^{\mathsf{r}}_N : \wp(\mathcal{M} \times \mathcal{V}) \to \mathcal{V}^{\mathsf{r}}_N$ be the corresponding left adjoint, which is uniquely determined by γ^{r} according to Proposition 1. The meaning of relational execution maps is then given in terms of γ^{r}_N by the function $\dot{\gamma}^{\mathsf{r}} : \mathcal{M}^{\mathsf{r}} \to \wp(\mathcal{M})$ as follows:

$$\dot{\gamma}^{\mathsf{r}}(M^{\mathsf{r}}) \stackrel{\text{def}}{=} \{M \mid \forall n.\, \langle M, M(n)\rangle \in \gamma^{\mathsf{r}}(M^{\mathsf{r}}(n))\}$$

From Lemma 5 it easily follows that $\dot{\gamma}^{\mathsf{r}}$ is also a complete meet-morphism. We

denote by $\dot{\alpha}^{\mathsf{r}} : \wp(\mathcal{M}) \to \mathcal{M}^{\mathsf{r}}$ the left adjoint of the induced Galois connection.

Abstract domain operations

Before we can define the abstract transformer, we need to introduce a few primitive operations for constructing and manipulating relational values and execution maps (Fig. 3.5). The operation $r_N[n_1{=}n_2]$ strengthens r_N by enforcing equality

$$
\begin{aligned}
r_N[n_1{=}n_2] &\stackrel{\text{def}}{=} \alpha^{\mathsf{r}}_N(\gamma^{\mathsf{r}}(r_N) \cap \{\langle M, v\rangle \mid M(n_1){=}M(n_2)\}) \\
r_N[\nu{=}n] &\stackrel{\text{def}}{=} \alpha^{\mathsf{r}}_N(\gamma^{\mathsf{r}}(r_N) \cap \{\langle M, v\rangle \mid v{=}M(n)\}) \\
r_N[n \leftarrow r'] &\stackrel{\text{def}}{=} \alpha^{\mathsf{r}}_N(\gamma^{\mathsf{r}}(r_N) \ \cap \{\langle M'[n \mapsto v'], v\rangle \mid \langle M', v'\rangle \in \gamma^{\mathsf{r}}(r')\}) \\
M^{\mathsf{r}}[n \leftarrow r_N] &\stackrel{\text{def}}{=} \dot{\bigsqcap}^{\mathsf{r}} \{M^{\mathsf{r}}[n' \mapsto M^{\mathsf{r}}(n')[n \leftarrow r]] \mid n' \in N\} \\
r_N[M^{\mathsf{r}}] &\stackrel{\text{def}}{=} \bigsqcap^{\mathsf{r}} \{r[n \leftarrow M^{\mathsf{r}}(n)] \mid n \in N\} \\
r_N[N'] &\stackrel{\text{def}}{=} \alpha^{\mathsf{r}}_{N'}(\gamma^{\mathsf{r}}(r_N) \ \cap \{\langle M, v\rangle \mid \forall n \in N'\backslash N.\, M(n) \notin \{\bot, \omega\}\})
\end{aligned}
$$

Figure 3.5: Operations on relational abstract domains

between n_1 and n_2 in the dependency vectors. Similarly, $r_N[\nu{=}n]$ strengthens r_N by requiring that the described value is equal to the value at node n. We use the operation $r_N[n \leftarrow r']$ to strengthen r_N with r' for n. Logically, this can be viewed as computing $r_N \wedge r'[n/\nu]$. The operation $M^{\mathsf{r}}[n \leftarrow r_N]$ lifts $r_N[n \leftarrow r']$ pointwise to relational execution maps. Note that here for every n' we implicitly rescope $M(n')$ so that its scope includes n. The operation $r_N[M^{\mathsf{r}}]$, conversely, strengthens r_N with all the relations describing the values at nodes N in M^{r}. Intuitively, this operation is used to push scope/environment assumptions to a relational value. Finally, the *rescoping* operation $r_N[N']$ changes the scope of r_N from N to N'

while strengthening it with the information that all nodes in $N' \setminus N$ are neither $\bot$ nor ω. We use the operation $\exists \hat{n}.\, r_N \stackrel{\text{def}}{=} r[N \setminus \{\hat{n}\}]$ to simply project out, i.e., remove a node $\hat{n}$ from the scope of a given relational value r_N. We (implicitly) use the rescoping operations to enable precise propagation between relational values that have incompatible scopes.

The relational abstraction of a constant c in scope N is defined as $c^{\mathsf{r}}_N \stackrel{\text{def}}{=} \{\, D^{\mathsf{r}} \in \mathcal{D}^{\mathsf{r}}_N \mid D^{\mathsf{r}}(\nu) = c \,\}$. We define $\mathit{Bool}^{\mathsf{r}}_N \stackrel{\text{def}}{=} \mathit{true}^{\mathsf{r}}_N \sqcup^{\mathsf{r}} \mathit{false}^{\mathsf{r}}_N$.

3.3.3 Abstract Transformer

The abstract transformer, $\mathsf{step}^{\mathsf{r}}$, of the relational semantics is shown in Fig. 3.7. It closely resembles the concrete transformer, step, where relational execution maps M^{r} take the place of concrete execution maps M and relational values r take the place of concrete values v. More precisely, the actual transformer $\mathsf{Step}^{\mathsf{r}}_{k+1}[\![e]\!](E)(M^{\mathsf{r}}) \stackrel{\text{def}}{=} M^{\mathsf{r}} \mathbin{\dot{\sqcup}^{\mathsf{r}}} \mathsf{step}^{\mathsf{r}}_{k+1}[\![e]\!](E)(M^{\mathsf{r}})$ simply runs $\mathsf{step}^{\mathsf{r}}$ on a given M^{r} and joins the result with M^{r}. Defining the actual transformer this way is non-essential, but simplifies our proofs.

The case for constant values c^{ℓ} effectively replaces the join on values in the concrete transformer using a join on relational values. The constant c is abstracted by the relational value $c^{\mathsf{r}}[M^{\mathsf{r}}]$, which establishes the relation between c and the other values stored at the nodes that are bound in the environment E. In general, the transformer *pushes* environment assumptions into relational values rather than leaving the assumptions implicit. We decided to treat environment assumptions this way so we can interpret relational values independently of the environment. This idea is implemented in the rules for variables and constants, as explained above, while it is done inductively for other constructs. In the case for variables

x^ℓ, the abstract transformer replaces prop in the concrete transformer by the abstract propagation operator prop^r defined in Fig. 3.6. Note that the definition of

$$\mathsf{prop}^\mathsf{r}(T^\mathsf{r}, \bot^\mathsf{r}) \stackrel{\text{def}}{=} \langle T^\mathsf{r}, T^\mathsf{r}_{\bot^\mathsf{r}} \rangle \quad \mathsf{prop}^\mathsf{r}(T^\mathsf{r}, \top^\mathsf{r}) \stackrel{\text{def}}{=} \langle \top^\mathsf{r}, \top^\mathsf{r} \rangle$$

$$\begin{aligned}
&\mathsf{prop}^\mathsf{r}(T^\mathsf{r}_1, T^\mathsf{r}_2) \stackrel{\text{def}}{=} \\
&\quad \textbf{let } T^\mathsf{r} = \Lambda n_\mathsf{cs}. \\
&\qquad \textbf{let } \langle r_{1i}, r_{1o} \rangle = T^\mathsf{r}_1(n_\mathsf{cs}); \langle r_{2i}, r_{2o} \rangle = T^\mathsf{r}_2(n_\mathsf{cs}) \\
&\qquad\quad \langle r'_{2i}, r'_{1i} \rangle = \mathsf{prop}^\mathsf{r}(r_{2i}, r_{1i}) \\
&\qquad\quad \langle r'_{1o}, r'_{2o} \rangle = \mathsf{prop}^\mathsf{r}(r_{1o}[n_\mathsf{cs} \leftarrow r_{2i}], r_{2o}[n_\mathsf{cs} \leftarrow r_{2i}]) \\
&\qquad \textbf{in } (\langle r'_{1i}, r_{1o} \sqcup^\mathsf{r} r'_{1o} \rangle, \langle r'_{2i}, r_{2o} \sqcup^\mathsf{r} r'_{2o} \rangle) \\
&\quad \textbf{in } \langle \Lambda n_\mathsf{cs}.\, \pi_1(T^\mathsf{r}(n_\mathsf{cs})), \Lambda n_\mathsf{cs}.\, \pi_2(T^\mathsf{r}(n_\mathsf{cs})) \rangle
\end{aligned}$$

$$\mathsf{prop}^\mathsf{r}(r_1, r_2) \stackrel{\text{def}}{=} \langle r_1, r_1 \sqcup^\mathsf{r} r_2 \rangle \quad \textbf{(otherwise)}$$

Figure 3.6: Value propagation in the relational data flow semantics

prop^r assumes that its arguments range over the same scope. Therefore, whenever we use prop^r in step^r, we implicitly use the rescoping operator to make the arguments compatible. For instance, in the call to prop^r for variables x^ℓ, the scope of $M^\mathsf{r}(E(x))$ is $N_{E(x)}$ which does not include the variable node $E(x)$ itself. We therefore implicitly rescope $M^\mathsf{r}(E(x))$ to $M^\mathsf{r}(E(x))[N_{E \diamond \ell}]$ before strengthening it with the equality $\nu = E(x)$. The other cases are fairly straightforward. Observe how in the case for conditionals $(e_0^{\ell_0}\, ?\, e_1^{\ell_1} : e_2^{\ell_2})^\ell$ we analyze both branches and combine the resulting relational execution maps with a join. In each branch, the map M_0^r is strengthened with the information about the truth value stored at $E \diamond \ell_0$, reflecting the path-sensitive nature of Liquid types.

Lemma 6. *The function* prop^r *is monotone and increasing.*

Lemma 7. *For every* $k \in \mathbb{N}$, $e \in \lambda^d$ *and* $E \in \mathcal{E}$ *such that* $\langle e, E \rangle$ *is well-formed, then* $\mathsf{Step}^\mathsf{r}_k[\![e]\!](E)$ *is monotone and increasing.*

$\mathsf{step}^{\mathsf{r}}_0[\![e^\ell]\!](E)(M^{\mathsf{r}}) \stackrel{\text{def}}{=} M^{\mathsf{r}}$

$\mathsf{step}^{\mathsf{r}}_{k+1}[\![c^\ell]\!](E)(M^{\mathsf{r}}) \stackrel{\text{def}}{=} M^{\mathsf{r}}[E{\diamond}\ell \mapsto M^{\mathsf{r}}(E{\diamond}\ell) \sqcup^{\mathsf{r}} c^{\mathsf{r}}[M^{\mathsf{r}}]]$

$\mathsf{step}^{\mathsf{r}}_{k+1}[\![x^\ell]\!](E)(M^{\mathsf{r}}) \stackrel{\text{def}}{=}$
$\quad$ **let** $\langle r'_x, r'\rangle = \mathsf{prop}^{\mathsf{r}}(M^{\mathsf{r}}(E(x))[\nu{=}E(x)], M^{\mathsf{r}}(E{\diamond}\ell))$
$\quad$ **in** $M^{\mathsf{r}}[E(x) \mapsto r'_x, E{\diamond}\ell \mapsto r'[M^{\mathsf{r}}]]$

$\mathsf{step}^{\mathsf{r}}_{k+1}[\![(e^{\ell_1}_1 e^{\ell_2}_2)^\ell]\!](E)(M^{\mathsf{r}}) \stackrel{\text{def}}{=}$
$\quad$ **let** $M^{\mathsf{r}}_1 = \mathsf{Step}^{\mathsf{r}}_k[\![e_1]\!](E)(M^{\mathsf{r}})$; $r_1 =_{M^{\mathsf{r}}_1} M^{\mathsf{r}}_1(E{\diamond}\ell_1)$ **in**
$\quad$ **if** $r_1 \notin \mathcal{T}^{\mathsf{r}}$ **then** $M^{\mathsf{r}}_{\top^{\mathsf{r}}}$ **else**
$\quad\quad$ **let** $M^{\mathsf{r}}_2 = \mathsf{Step}^{\mathsf{r}}_k[\![e_2]\!](E)(M^{\mathsf{r}}_1)$; $r_2 =_{M^{\mathsf{r}}_2} M^{\mathsf{r}}_2(E{\diamond}\ell_2)$ **in**
$\quad\quad\quad \langle r'_1, T^{\mathsf{r}}_0\rangle = \mathsf{prop}^{\mathsf{r}}(r_1, [E{\diamond}\ell_1 \mapsto \langle r_2, M^{\mathsf{r}}_2(E{\diamond}\ell)\rangle])$
$\quad\quad\quad \langle r'_2, r'\rangle = T^{\mathsf{r}}_0(E{\diamond}\ell_1)$
$\quad\quad$ **in** $M^{\mathsf{r}}_2[E{\diamond}\ell_1 \mapsto r'_1, E{\diamond}\ell_2 \mapsto r'_2, E{\diamond}\ell \mapsto r']$

$\mathsf{step}^{\mathsf{r}}_{k+1}[\![(\mu f.\lambda x.e^{\ell_1}_1)^\ell]\!](E)(M^{\mathsf{r}}) \stackrel{\text{def}}{=}$
$\quad$ **let** $T^{\mathsf{r}} =_{M^{\mathsf{r}}} M^{\mathsf{r}}(E{\diamond}\ell) \sqcup^{\mathsf{r}} T^{\mathsf{r}}_\perp$ **in**
$\quad$ **let** $\overline{M^{\mathsf{r}}} = \Lambda n_{\mathsf{cs}}{\in}T^{\mathsf{r}}.$ **if** $\pi_1(T^{\mathsf{r}}(n_{\mathsf{cs}}))=\top^{\mathsf{r}}$ **then** $M^{\mathsf{r}}_{\top^{\mathsf{r}}}$ **else**
$\quad\quad$ **let** $n_x = E{\diamond}n_{\mathsf{cs}}{\diamond}x$; $n_f = E{\diamond}n_{\mathsf{cs}}{\diamond}f$; $E_1 = E[x{\mapsto}n_x, f{\mapsto}n_f]$
$\quad\quad\quad \langle T^{\mathsf{r}}_x, T^{\mathsf{r}}_1\rangle = \mathsf{prop}^{\mathsf{r}}([n_{\mathsf{cs}}{\mapsto}\langle M^{\mathsf{r}}(n_x), M^{\mathsf{r}}(E_1{\diamond}\ell_1)[n_{\mathsf{cs}}{=}n_x]\rangle], T^{\mathsf{r}})$
$\quad\quad\quad \langle T^{\mathsf{r}}_2, T^{\mathsf{r}}_f\rangle = \mathsf{prop}^{\mathsf{r}}(T^{\mathsf{r}}, M^{\mathsf{r}}(n_f))$; $\langle r'_x, r'_1\rangle = T^{\mathsf{r}}_x(n_{\mathsf{cs}})$
$\quad\quad\quad M^{\mathsf{r}}_1 = M^{\mathsf{r}}[E{\diamond}\ell \mapsto T^{\mathsf{r}}_1 \sqcup^{\mathsf{r}} T^{\mathsf{r}}_2, n_f \mapsto T^{\mathsf{r}}_f, n_x \mapsto r'_x, E_1{\diamond}\ell_1 \mapsto r'_1[n_x{=}n_{\mathsf{cs}}]]$
$\quad\quad$ **in** $\mathsf{Step}^{\mathsf{r}}_k[\![e_1]\!](E_1)(M^{\mathsf{r}}_1)$
$\quad$ **in** $M^{\mathsf{r}}[E{\diamond}\ell \mapsto T^{\mathsf{r}}] \dot{\sqcup}^{\mathsf{r}} \dot{\bigsqcup}^{\mathsf{r}}_{n_{\mathsf{cs}}\in T^{\mathsf{r}}} \overline{M^{\mathsf{r}}}(n_{\mathsf{cs}})$

$\mathsf{step}^{\mathsf{r}}_{k+1}[\![(e^{\ell_0}_0 \,?\, e^{\ell_1}_1 : e^{\ell_2}_2)^\ell]\!](E)(M^{\mathsf{r}}) \stackrel{\text{def}}{=}$
$\quad$ **let** $M^{\mathsf{r}}_0 = \mathsf{Step}^{\mathsf{r}}_k[\![e_0]\!](E)(M^{\mathsf{r}})$; $r_0 =_{M^{\mathsf{r}}_0} M^{\mathsf{r}}_0(E{\diamond}\ell_0)$ **in**
$\quad$ **if** $r_0 \not\sqsubseteq^{\mathsf{r}} \mathit{Bool}^{\mathsf{r}}$ **then** $M^{\mathsf{r}}_{\top^{\mathsf{r}}}$ **else**
$\quad\quad$ **let** $M^{\mathsf{r}}_1 = \mathsf{Step}^{\mathsf{r}}_k[\![e_1]\!](E)(M^{\mathsf{r}}_0[E{\diamond}\ell_0 \leftarrow \mathit{true}^{\mathsf{r}} \sqcap^{\mathsf{r}} r_0])$
$\quad\quad\quad M^{\mathsf{r}}_2 = \mathsf{Step}^{\mathsf{r}}_k[\![e_2]\!](E)(M^{\mathsf{r}}_0[E{\diamond}\ell_0 \leftarrow \mathit{false}^{\mathsf{r}} \sqcap^{\mathsf{r}} r_0])$
$\quad\quad\quad r_1 = M^{\mathsf{r}}_1(E{\diamond}\ell_1)$; $r_2 = M^{\mathsf{r}}_2(E{\diamond}\ell_2)$
$\quad\quad\quad \langle r'_1, r'\rangle = \mathsf{prop}^{\mathsf{r}}(r_1, M^{\mathsf{r}}_1(E{\diamond}\ell))$; $\langle r'_2, r''\rangle = \mathsf{prop}^{\mathsf{r}}(r_2, M^{\mathsf{r}}_2(E{\diamond}\ell))$
$\quad$ **in** $M^{\mathsf{r}}_0 \;\dot{\sqcup}^{\mathsf{r}}\; M^{\mathsf{r}}_1[E{\diamond}\ell_1 \mapsto r'_1, E{\diamond}\ell \mapsto r'] \,\dot{\sqcup}^{\mathsf{r}}\, M^{\mathsf{r}}_2[E{\diamond}\ell_2 \mapsto r'_2, E{\diamond}\ell \mapsto r'']$

Figure 3.7: Abstract transformer for relational data flow semantics

3.3.4 Abstract Semantics

The abstract domain of the relational abstraction is given by *relational properties* $\mathcal{P}^{\mathsf{r}} \stackrel{\text{def}}{=} \mathbb{N} \to \mathcal{M}^{\mathsf{r}}$, which are ordered by $\ddot{\sqsubseteq}^{\mathsf{r}}$, the pointwise lifting of $\dot{\sqsubseteq}^{\mathsf{r}}$ on $\mathcal{M}^{\mathsf{r}}$ to $\mathcal{P}^{\mathsf{r}}$. We likewise lift the Galois connection $\langle \dot{\alpha}^{\mathsf{r}}, \dot{\gamma}^{\mathsf{r}} \rangle$ to a Galois connection $\langle \ddot{\alpha}^{\mathsf{r}}, \ddot{\gamma}^{\mathsf{r}} \rangle$ between the concrete lattice $\mathcal{P}$ and the abstract lattice $\mathcal{P}^{\mathsf{r}}$. The relational abstract semantics $\mathbf{C}^{\mathsf{r}} : \lambda^d \to \mathcal{P}^{\mathsf{r}}$ is then defined as the least fixpoint of $\mathsf{Step}^{\mathsf{r}}$ as follows:

$$\mathbf{C}^{\mathsf{r}}[\![e]\!] \stackrel{\text{def}}{=} \mathbf{lfp}^{\ddot{\sqsubseteq}^{\mathsf{r}}}_{\Lambda k.\ M^{\mathsf{r}}_{\perp^{\mathsf{r}}}} \Lambda P^{\mathsf{r}}.\Lambda k.\ \mathsf{Step}^{\mathsf{r}}_k[\![e]\!](\epsilon)(P^{\mathsf{r}}(k))$$

Theorem 1. *The relational abstract semantics is sound, i.e., for all programs e, $\mathbf{C}[\![e]\!] \subseteq \ddot{\gamma}^{\mathsf{r}}(\mathbf{C}^{\mathsf{r}}[\![e]\!])$.*

In the proof of Theorem 1 we show that $\mathsf{prop}^{\mathsf{r}}$ and $\mathsf{step}^{\mathsf{r}}$ are sound abstractions of their concrete counterparts prop and step, from which the claim then follows easily. For the most part, these proofs are straightforward inductions because the definitions of the abstract operators are so closely aligned with the concrete ones. Most intermediate steps just follow directly from properties of the involved Galois connections. The soundness proof of $\mathsf{step}^{\mathsf{r}}$ is slightly more involved because it is not true that for all well-formed (e, E), $k \in \mathbb{N}$, $M \in \mathcal{M}$, and $M^{\mathsf{r}} \in \mathcal{M}^{\mathsf{r}}$, if $M \in \dot{\gamma}^{\mathsf{r}}(M^{\mathsf{r}})$ then $\mathsf{step}_k[\![e]\!](E)(M) \in \dot{\gamma}^{\mathsf{r}}(\mathsf{step}^{\mathsf{r}}_k[\![e]\!](E)(M^{\mathsf{r}}))$. This is because $\mathsf{step}^{\mathsf{r}}$ relies on certain invariants satisfied by the (E, M) that the concrete step maintains for the iterates of $\mathbf{S}[\![e]\!]$. For instance, to prove that the implicit uses of the rescoping operator $r[N]$ in $\mathsf{step}^{\mathsf{r}}$ are sound, we need the invariant that in the concrete, all variable nodes that are bound in E are not mapped to $\perp$ or ω by M. This can be shown with a simple induction proof. Similar invariants are needed to prove the last step in the case for recursive functions, where we need to show that the join over all call sites $\dot{\bigsqcup}_{n_{\mathsf{cs}} \in T}\ \overline{M}(n_{\mathsf{cs}})$ in the concrete transformer is abstracted by the join over all call

sites $\dot{\bigsqcup}^{\mathsf{r}}_{n_{\mathsf{cs}} \in T} \overline{M^{\mathsf{r}}}(n_{\mathsf{cs}})$ in $\mathsf{step}^{\mathsf{r}}$.

3.4 Collapsed Relational Data Flow Semantics

We now describe the key abstraction step in the construction of Liquid types, which we formalize in terms of a *collapsed (relational data flow) semantics*. This semantics abstracts the relational semantics by (1) collapsing the input/output relations for all call sites in each table into a single one, and (2) collapsing variable nodes by removing their call site components. That is, tables no longer keep track of where they are called and variable bindings no longer track the call stack.

3.4.1 Example

We first provide an intuition behind the collapsed data flow semantics through an example. For that purpose, use the program from Example 3 exhibiting higher-order features of λ^d.

The map computed by the collapsed data flow semantics is presented below. We only show map entries for a portion of reachable nodes.

$$g1, c \to \langle z, \{(\nu\colon 1), \{(z\colon 1, \nu\colon 0)\}\rangle \quad g2, e \to \langle z, \{(\nu\colon 1), \{(z\colon 1, \nu\colon 1)\}\rangle$$

$$b \to \langle g_1, \langle z, \{(\nu\colon 1)\}, \{(z\colon 1, \nu\colon 0)\}\rangle, \{(g_1\colon \mathsf{F}, \nu\colon 1), (g_1\colon \mathsf{F}, \nu\colon 0)\}\rangle$$

$$d \to \langle g_2, \langle z, \{(\nu\colon 1)\}, \{(z\colon 1, \nu\colon 1)\}\rangle, \{(g_2\colon \mathsf{F}, \nu\colon 1), (g_2\colon \mathsf{F}, \nu\colon 0)\}\rangle$$

$$h \to \langle g, \langle z, \{(\nu\colon 1)\}, \{(z\colon 1, \nu\colon 1), (z\colon 1, \nu\colon 0)\}\rangle, \{(g\colon \mathsf{F}, \nu\colon 1), (g\colon \mathsf{F}, \nu\colon 0)\}\rangle$$

As opposed to the concrete and relational semantics that keep track of execution points where the functions are being called in the program, the collapsed semantics completely disregards such information. Consider first the entry for `g1` in the map

that assigns `g1` the *collapsed* table value $\langle z, \{(\nu\colon 1), \{(z\colon 1, \nu\colon 0)\}\rangle$, stating that `g1` is a function called at some concrete call site node, symbolically represented as z, with integer 1 as the input and 0 as the output. Note that the information about the actual call site node at which `g1` is called is not maintained. Also, observe how z is used to explicitly remember the input value in the output relation. The same explanation can be given for the map entry for `g2`. The ramifications of not maintaining the precise call site information in tables are best understood by looking at the map entry for `h`. Since both `g1` and `g2` are inputs of `h` in the program and `h` cannot distinguish between these two inputs using their call site information that is absent, the analysis of the body of `h` must simultaneously consider both `g1` and `g2` as the input argument `g` of `h`. That is, the semantics infers that the input `g` of `h` is the table value $\langle z, \{(\nu\colon 1)\}, \{(z\colon 1, \nu\colon 1), (z\colon 1, \nu\colon 0)\}\rangle$ that can intuitively be seen as the join of values of `g1` and `g2`. The semantics then concludes that the output relation for `h` is the relation $\{(g\colon \mathsf{F}, \nu\colon 1), (g\colon \mathsf{F}, \nu\colon 0)\}$, containing both 0 and 1 as the possible result of calling `h`. In effect, the result for both calls to `h` at nodes b and d can be either 0 or 1. Therefore, the collapsed semantics flags the program as potentially unsafe as the value of the divisor expression can possibly be 0. Note that the concrete and relational semantics render this program safe. Overall, by not maintaining precise call site information in tables, the collapsed semantics can introduce imprecision as the input relations of functions coming from different call sites must be merged together when analyzing the function body.

We next introduce the collapsed relational data flow semantics formally. We start by explaining the abstract domains used in defining the semantics.

3.4.2 Abstract Domains

The semantic domains of the collapsed semantics are defined as follows:

$$\hat{n} \in \hat{\mathcal{N}} \stackrel{\text{def}}{=} \hat{\mathcal{E}} \times \mathit{Loc} \qquad \text{abstract nodes}$$

$$\hat{E} \in \hat{\mathcal{E}} \stackrel{\text{def}}{=} \mathit{Vars} \rightharpoonup_{\mathit{fin}} \hat{\mathcal{N}} \qquad \text{abstract environments}$$

$$u \in \mathcal{V}^{\mathsf{cr}}_{\hat{N}} ::= \bot^{\mathsf{cr}} \mid \top^{\mathsf{cr}} \mid R^{\mathsf{cr}}_{\hat{N}} \mid T^{\mathsf{cr}}_{\hat{N}} \qquad \text{collapsed values}$$

$$T^{\mathsf{cr}}_{\hat{N}} \in \mathcal{T}^{\mathsf{cr}}_{\hat{N}} \stackrel{\text{def}}{=} \Sigma z \in \mathit{DVar}.\ \mathcal{V}^{\mathsf{cr}}_{\hat{N}\setminus\{z\}} \times \mathcal{V}^{\mathsf{cr}}_{\hat{N}\cup\{z\}} \qquad \text{collapsed tables}$$

$$R^{\mathsf{cr}}_{\hat{N}} \in \mathcal{R}^{\mathsf{cr}}_{\hat{N}} \stackrel{\text{def}}{=} \wp(\mathcal{D}^{\mathsf{cr}}_{\hat{N}}) \qquad \text{collapsed relations}$$

$$D^{\mathsf{cr}}_{\hat{N}} \in \mathcal{D}^{\mathsf{cr}}_{\hat{N}} \stackrel{\text{def}}{=} (\hat{N} \cup \{\nu\}) \to (\mathit{Cons} \cup \{\mathsf{F}\}) \qquad \text{collapsed dep. vectors}$$

$$M^{\mathsf{cr}} \in \mathcal{M}^{\mathsf{cr}} \stackrel{\text{def}}{=} \Pi \hat{n} \in \hat{\mathcal{N}}.\ \mathcal{V}^{\mathsf{cr}}_{\hat{N}_{\hat{n}}} \qquad \text{collapsed maps}$$

Since we abstract from call site information in variable nodes, the new semantics no longer distinguishes between expression and variable nodes, leading to the definitions of *abstract nodes* $\hat{n} \in \hat{\mathcal{N}}$ and *abstract environments* $\hat{E} \in \hat{\mathcal{E}}$. The operations *env* and *loc* on abstract nodes are defined as for concrete nodes.

In order to be able to express input-output dependencies in tables without keeping track of call sites explicitly, the collapsed semantics introduces *dependency variables* $z \in \mathit{DVar}$, which symbolically stand for the concrete input nodes of call sites. An *abstract scope* $\hat{N}$ is then defined as a finite set of abstract nodes and dependency variables $\hat{N} \subseteq_{\mathit{fin}} (\hat{\mathcal{N}} \cup \mathit{DVar})$. Analogous to concrete scopes, we define $\hat{N}_{\hat{n}} = \mathsf{rng}(\mathit{env}(\hat{n}))$ as the abstract scope of an abstract node $\hat{n}$.

Collapsed values $u_{\hat{N}} \in \mathcal{V}^{\mathsf{cr}}_{\hat{N}}$ are implicitly indexed by an abstract scope $\hat{N}$ and closely resemble relational values. The only major difference is in the definition of tables, which now only track a single symbolic call site. A *collapsed table* $T^{\mathsf{cr}}_{N} \in \mathcal{T}^{\mathsf{cr}}_{\hat{N}}$ is a tuple $\langle z, u_1, u_2 \rangle$ where z cannot appear in the scope of the input relational value u_1 but is included in the scope of output relational value u_2. We write $\mathit{dx}(T^{\mathsf{cr}}_{N})$

to denote the dependency variable z and $io(T^{\mathsf{cr}}_{\hat{N}})$ to denote the pair $\langle u_1, u_2\rangle$. The remaining definitions are as for relational values except that abstract nodes take the place of concrete nodes.

The ordering $\sqsubseteq^{\mathsf{cr}}_{\hat{N}}$ on collapsed values $\mathcal{V}^{\mathsf{cr}}_{\hat{N}}$ resembles the ordering on relational values:

$$u_1 \sqsubseteq^{\mathsf{cr}} u_2 \overset{\text{def}}{\iff} u_1 = \bot^{\mathsf{cr}} \vee u_2 = \top^{\mathsf{cr}} \vee (u_1, u_2 \in \mathcal{R}^{\mathsf{cr}}_{\hat{N}} \wedge u_1 \subseteq u_2) \\ \vee (u_1, u_2 \in \mathcal{T}^{\mathsf{cr}}_{\hat{N}} \wedge io(u_1) \mathrel{\dot{\sqsubseteq}}^{\mathsf{cr}} io(u_2))$$

We assume that when we compare two collapsed tables, we implicitly apply α-renaming so that they range over the same dependency variable. Again, this ordering induces, for every $\hat{N}$, a complete lattice $(\mathcal{V}^{\mathsf{cr}}_{\hat{N}}, \sqsubseteq^{\mathsf{cr}}, \bot^{\mathsf{cr}}, \top^{\mathsf{cr}}, \sqcup^{\mathsf{cr}}, \sqcap^{\mathsf{cr}})$. The definition of the join $\sqcup^{\mathsf{cr}}$ operator is as follows.

$$\bot^{\mathsf{cr}} \sqcup^{\mathsf{cr}} u \overset{\text{def}}{=} u \qquad u \sqcup^{\mathsf{cr}} \bot^{\mathsf{cr}} \overset{\text{def}}{=} u \qquad R^{\mathsf{cr}}_1 \sqcup^{\mathsf{cr}} R^{\mathsf{cr}}_2 \overset{\text{def}}{=} R^{\mathsf{cr}}_1 \cup R^{\mathsf{cr}}_2$$

$$\langle z, u_1, u_2\rangle \sqcup^{\mathsf{cr}} \langle z, u_3, u_4\rangle \overset{\text{def}}{=} \langle z, u_1 \sqcup^{\mathsf{cr}} u_3, u_2 \sqcup^{\mathsf{cr}} u_4\rangle \quad u_1 \sqcup^{\mathsf{cr}} u_2 \overset{\text{def}}{=} \top^{\mathsf{cr}} \quad \text{(otherwise)}$$

The meet $\sqcap^{\mathsf{cr}}$ is defined similarly.

Lemma 8. *Let $V \in \wp(\mathcal{V}^{\mathsf{cr}}_{\hat{N}})$. Then, $\sqcup^{\mathsf{cr}} V = lub(V)$ and $\sqcap^{\mathsf{cr}} V = glb(V)$ subject to $\sqsubseteq^{\mathsf{cr}}$.*

Technically, $\sqsubseteq^{\mathsf{cr}}$ is only a pre-order due to α-renaming of tables. We implicitly identify $\mathcal{V}^{\mathsf{cr}}_{\hat{N}}$ with its quotient subject to the equivalence relation induced by $\sqsubseteq^{\mathsf{cr}}$. These lattices are lifted pointwise to a complete lattice on collapsed execution maps $(\mathcal{M}^{\mathsf{cr}}, \dot{\sqsubseteq}^{\mathsf{cr}}, M^{\mathsf{cr}}_{\bot^{\mathsf{cr}}}, M^{\mathsf{cr}}_{\top^{\mathsf{cr}}}, \dot{\sqcup}^{\mathsf{cr}}, \dot{\sqcap}^{\mathsf{cr}})$.

Galois connections

In order to provide the meaning of collapsed execution maps and values, we first need to formally relate concrete and abstract nodes, environments, and scopes. To this end, we define the *node abstraction* function $\rho : \mathcal{N} \to \hat{\mathcal{N}}$ as $\rho(n) \stackrel{\text{def}}{=} (\rho \circ env(n)) \diamond loc(n)$. In essence, ρ recursively removes call site information from concrete variable nodes. For instance, ρ maps two concrete variable nodes $E \diamond n_{\mathsf{cs}} \diamond x$ and $E \diamond n'_{\mathsf{cs}} \diamond x$ for distinct call sites n_{cs} and n'_{cs} to the same abstract node $(\rho \circ E) \diamond x$. Next, we want to define a Galois connection between relational values $\mathcal{V}^{\mathsf{r}}_N$ and collapsed values $\mathcal{V}^{\mathsf{cr}}_{\hat{N}}$ for given concrete and abstract scopes N and $\hat{N}$. More precisely, we want to abstract relations over concrete nodes in N by relations over abstract nodes in $\hat{N}$ which may contain dependency variables standing for arbitrary expression nodes at call sites. The corresponding concretization function therefore needs to be defined with respect to a specific matching $\delta : \hat{N} \to N$ of abstract to concrete nodes. These matching functions need to be consistent with node abstraction. We formalize this consistency property by defining the set of all *scope consistent matchings* $\Delta(\hat{N}, N)$ as

$$\Delta(\hat{N}, N) \stackrel{\text{def}}{=} \{\, \delta : \hat{N} \to N \mid \forall \hat{n} \in \hat{N}.(\hat{n} \in \hat{\mathcal{N}} \Rightarrow \rho(\delta(\hat{n})) = \hat{n}) \\ \wedge (\hat{n} \in DVar \Rightarrow \delta(\hat{n}) \in \mathcal{N}_e) \,\}$$

That is, we require that the matching of abstract nodes with concrete nodes is consistent with ρ. For dependency variables, we require that they are matched back to expression nodes. This second requirement formalizes the intuition that dependency variables symbolically stand for arbitrary concrete call site nodes. The meaning of collapsed values $\mathcal{V}^{\mathsf{cr}}_{\hat{N}}$ with respect to relational values $\mathcal{V}^{\mathsf{r}}_N$ is then given subject to a scope consistent matching $\delta \in \Delta(\hat{N}, N)$ by the concretization function

$\gamma^{\mathsf{cr}}_{\delta} : \mathcal{V}^{\mathsf{cr}}_{\hat{N}} \to \mathcal{V}^{\mathsf{r}}_{N}$, which is defined as follows:

$$\gamma^{\mathsf{cr}}_{\delta}(\bot^{\mathsf{cr}}) \stackrel{\text{def}}{=} \bot^{\mathsf{r}} \qquad \gamma^{\mathsf{cr}}_{\delta}(\top^{\mathsf{cr}}) \stackrel{\text{def}}{=} \top^{\mathsf{r}} \qquad \gamma^{\mathsf{cr}}_{\delta}(\langle z, u_i, u_o\rangle) \stackrel{\text{def}}{=} \Lambda n_{\mathsf{cs}}.\, \langle \gamma^{\mathsf{cr}}_{\delta}(u_i), \gamma^{\mathsf{cr}}_{\delta[z \mapsto n_{\mathsf{cs}}]}(u_o)\rangle$$

$$\gamma^{\mathsf{cr}}_{\delta}(R^{\mathsf{cr}}_{\hat{N}}) \stackrel{\text{def}}{=} \{D^{\mathsf{r}} \mid D^{\mathsf{cr}} \in R^{\mathsf{cr}} \wedge D^{\mathsf{cr}}(\nu) = D^{\mathsf{r}}(\nu) \ \wedge \forall \hat{n} \in \hat{N}.\, D^{\mathsf{r}}(\delta(\hat{n})) = D^{\mathsf{cr}}(\hat{n})\}$$

The concretization function simply renames the abstract nodes and dependency variables to concrete nodes, as given by δ. For tables, it assigns the same pair of relational values to every possible call site (modulo renaming of n_{cs} in the output relation). Note that the extended matching function $\delta[z \mapsto n_{\mathsf{cs}}]$ is again scope consistent for $\hat{N} \cup \{z\}$ and $N \cup \{n_{\mathsf{cs}}\}$.

Lemma 9. *For every $\delta \in \Delta(\hat{N}, N)$, $\gamma^{\mathsf{cr}}_{\delta}$ is a complete meet-morphism between $\mathcal{V}^{\mathsf{cr}}_{\hat{N}}$ and $\mathcal{V}^{\mathsf{r}}_{N}$.*

It follows then that there exists a unique Galois connection $\langle \alpha^{\mathsf{cr}}_{\delta}, \gamma^{\mathsf{cr}}_{\delta}\rangle$ between $\mathcal{V}^{\mathsf{r}}_{N}$ and $\mathcal{V}^{\mathsf{cr}}_{\hat{N}}$, for every $\delta \in \Delta(\hat{N}, N)$ [17]. We then lift the $\gamma^{\mathsf{cr}}_{\delta}$ to a concretization function $\dot{\gamma}^{\mathsf{cr}} : M^{\mathsf{cr}} \to M^{\mathsf{r}}$ for collapsed execution maps:

$$\dot{\gamma}^{\mathsf{cr}}(M^{\mathsf{cr}}) \stackrel{\text{def}}{=} \Lambda n. \bigsqcap\nolimits^{\mathsf{r}} \{\gamma^{\mathsf{cr}}_{\delta}(M^{\mathsf{cr}}(\rho(n))) \mid \delta \in \Delta(\hat{N}_{\rho(n)}, N_n)\}$$

That is, if $M^{\mathsf{r}} \in \dot{\gamma}^{\mathsf{cr}}(M^{\mathsf{cr}})$, then for every node n, the relational value $M^{\mathsf{r}}(n)$ must be subsumed by $\gamma^{\mathsf{cr}}_{\delta}(M^{\mathsf{cr}}(\rho(n)))$ for all possible consistent matchings δ between N_n and $\hat{N}_{\rho(n)}$. We note that for all concrete nodes n reachable in our transformer $\mathsf{step}^{\mathsf{r}}$, there is a unique $\delta \in \Delta(\hat{N}_{\rho(n)}, N_n)$. This is because $\hat{N}_{\rho(n)}$ contains no dependency variables. Moreover, the locations of the concrete variable nodes bound in reachable environments are the locations of the variables to which they are bound. Hence, they are pairwise distinct. Since ρ preserves the node location information, it is injective on N_n. Hence, δ is uniquely determined by ρ as per the definition of Δ. Since this uniqueness property does not hold in general for unreachable nodes,

the meet over all δ is needed so that $\dot{\gamma}^{\mathsf{cr}}$ is a complete meet-morphism, which follows directly from its definition and Lemma 9. As usual, the left adjoint of the induced Galois connection is denoted by $\dot{\alpha}^{\mathsf{cr}} : \mathcal{M}^{\mathsf{r}} \to \mathcal{M}^{\mathsf{cr}}$.

Abstract domain operations

In the new abstract transformer we replace the strengthening and rescoping operations on relational values such as $r_N[n_1 = n_2]$ by their most precise abstractions on collapsed values $u_{\hat{N}}$. For technical reasons in the soundness proof of the abstract transformer, these abstractions are defined with respect to a Galois connection between $\mathcal{V}^{\mathsf{cr}}_{\hat{N}}$ and the complete lattice $\Pi N.\, \Delta(\hat{N}, N) \to \mathcal{V}^{\mathsf{r}}_{N}$ where the ordering on $\mathcal{V}^{\mathsf{r}}_{N}$ and the concretizations $\gamma^{\mathsf{cr}}_{\delta}$ are simply lifted pointwise. For instance, strengthening with equality between abstract nodes, $u_{\hat{N}}[\hat{n}_1{=}\hat{n}_2]$, is defined as

$$u_{\hat{N}}[\hat{n}_1{=}\hat{n}_2] \stackrel{\text{def}}{=} \bigsqcup\nolimits^{\mathsf{cr}} \{\, \alpha^{\mathsf{cr}}_{\delta}(\gamma^{\mathsf{cr}}_{\delta}(u)[\delta(\hat{n}_1){=}\delta(\hat{n}_2)]) \mid N \subseteq \mathcal{N} \wedge \delta \in \Delta(\hat{N}, N) \,\}$$

This definition in terms of most precise abstraction should not obscure the simple nature of this operation. E.g., for collapsed dependency relations R^{cr} we simply have:

$$R^{\mathsf{cr}}_{\hat{N}}[\hat{n}_1{=}\hat{n}_2] = \{\, D^{\mathsf{cr}} {\in} R^{\mathsf{cr}}_{\hat{N}} \mid \hat{n}_1, \hat{n}_2 {\in} \hat{N} \Rightarrow D^{\mathsf{cr}}(\hat{n}_1){=}D^{\mathsf{cr}}(\hat{n}_2) \,\}$$

as one might expect. The other strengthening operations on relational values are abstracted in a similar manner.

3.4.3 Abstract Transformer

The abstract transformer for the collapsed semantics, $\mathsf{step}^{\mathsf{cr}} : \lambda^d \to \hat{\mathcal{E}} \to \mathcal{M}^{\mathsf{cr}} \to \mathcal{M}^{\mathsf{cr}}$, closely resembles the relational abstract transformer $\mathsf{step}^{\mathsf{r}}$ where the operations on relational values are simply replaced by their abstractions on collapsed

values. The only significant difference is in the case for recursive function definitions, shown in Figure 3.8. As collapsed tables do not maintain per-call-site information, the function body e_1 is now evaluated only once for the abstract node $\hat{n}_x$ that abstracts all variable nodes bound to x in the concrete. The propagation between collapsed values, shown in Figure 3.9, is similarly simplified. Note that, technically, propagation works on the representatives of equivalence classes of collapsed values modulo α-renaming of dependency variables in tables. We assume that whenever we propagate between two collapsed tables, the dependency variables in these tables are consistently renamed so that they match on both sides.

$$
\begin{array}{l}
\mathsf{step}^{\mathsf{cr}}_{k+1}[\![(\mu f.\lambda x.e_1^{\ell_1})^{\ell}]\!](\hat{E})(M^{\mathsf{cr}}) \stackrel{\text{def}}{=} \\
\quad \textbf{let } T^{\mathsf{cr}} =_{M^{\mathsf{cr}}} M^{\mathsf{cr}}(\hat{E}\diamond\ell) \sqcup^{\mathsf{cr}} T^{\mathsf{cr}}_{\bot} \textbf{ in} \\
\quad \textbf{if } \pi_1(io(T^{\mathsf{cr}})) = \bot^{\mathsf{cr}} \textbf{ then } M^{\mathsf{cr}}[\hat{E}\diamond\ell \mapsto T^{\mathsf{cr}}] \textbf{ else} \\
\quad \textbf{if } \pi_1(io(T^{\mathsf{cr}})) = \top^{\mathsf{cr}} \textbf{ then } M^{\mathsf{cr}}_{\top_{\mathsf{cr}}} \textbf{ else} \\
\quad \textbf{let } \hat{n}_x = \hat{E}\diamond x\textbf{; } \hat{n}_f = \hat{E}\diamond f\textbf{; } \hat{E}_1 = \hat{E}[x \mapsto \hat{n}_x, f \mapsto \hat{n}_f] \\
\qquad \langle T^{\mathsf{cr}}_x, T^{\mathsf{cr}}_1 \rangle = \mathsf{prop}^{\mathsf{cr}}(\langle dx(T^{\mathsf{cr}}), M^{\mathsf{r}}(n_x), M^{\mathsf{r}}(\hat{E}_1\diamond\ell_1)[dx(T^{\mathsf{cr}}){=}\hat{n}_x]\rangle, T^{\mathsf{cr}}) \\
\qquad \langle T^{\mathsf{cr}}_2, T^{\mathsf{cr}}_f \rangle = \mathsf{prop}^{\mathsf{cr}}(T^{\mathsf{cr}}_1, M^{\mathsf{cr}}(\hat{n}_f))\textbf{; } \langle u'_x, u'_1 \rangle = io(T^{\mathsf{cr}}_x) \\
\qquad M^{\mathsf{cr}}_1 = M^{\mathsf{cr}}[\hat{E}\diamond\ell \mapsto T^{\mathsf{cr}}_1 \sqcup^{\mathsf{cr}} T^{\mathsf{cr}}_2, \hat{n}_f \mapsto T^{\mathsf{cr}}_f, \hat{n}_x \mapsto u'_x, \hat{E}_1\diamond\ell_1 \mapsto u'_1[\hat{n}_x{=}dx(T^{\mathsf{cr}})]] \\
\quad \textbf{in } \mathsf{Step}^{\mathsf{cr}}_k[\![e_1]\!](\hat{E}_1)(M^{\mathsf{cr}}_1)
\end{array}
$$

Figure 3.8: Abstract transformer for collapsed semantics (recursive function definition rule)

Lemma 10. *The function* $\mathsf{prop}^{\mathsf{cr}}$ *is monotone and increasing.*

Lemma 11. *For every* $k \in \mathbb{N}$, $e \in \lambda^d$ *and* $\hat{E} \in \hat{\mathcal{E}}$ *such that* $\langle e, \hat{E} \rangle$ *is well-formed, then* $\mathsf{Step}^{\mathsf{cr}}_k[\![e]\!](\hat{E})$ *is monotone and increasing.*

In the above, the well-formedness of a pair $(e, \hat{E})$ of an expression e and abstract

$$\mathsf{prop}^{\mathsf{cr}}(T^{\mathsf{cr}}, \bot^{\mathsf{cr}}) \stackrel{\text{def}}{=} \langle T^{\mathsf{cr}}, \langle dx(T^{\mathsf{cr}}), \bot, \bot\rangle\rangle$$

$$\mathsf{prop}^{\mathsf{cr}}(T^{\mathsf{cr}}, \top^{\mathsf{cr}}) \stackrel{\text{def}}{=} \langle \top^{\mathsf{cr}}, \top^{\mathsf{cr}}\rangle$$

$$\begin{array}{l}\mathsf{prop}^{\mathsf{cr}}(T_1^{\mathsf{cr}}, T_2^{\mathsf{cr}}) \stackrel{\text{def}}{=} \\ \quad \textbf{let } \langle z, u_{1i}, u_{1o}\rangle = T_1^{\mathsf{cr}}\textbf{; } \langle z, u_{2i}, u_{2o}\rangle = T_2^{\mathsf{cr}} \\ \qquad\quad \langle u'_{2i}, u'_{1i}\rangle = \mathsf{prop}^{\mathsf{cr}}(u_{2i}, u_{1i}) \\ \qquad\quad \langle u'_{1o}, u'_{2o}\rangle = \mathsf{prop}^{\mathsf{cr}}(u_{1o}[z \leftarrow u_{2i}], u_{2o}[z \leftarrow u_{2i}]) \\ \quad \textbf{in } \langle\langle z, u'_{1i}, u_{1o} \sqcup^{\mathsf{cr}} u'_{1o}\rangle, \langle z, u'_{2i}, u_{2o} \sqcup^{\mathsf{cr}} u'_{2o}\rangle\rangle\end{array}$$

$$\mathsf{prop}^{\mathsf{cr}}(u_1, u_2) \stackrel{\text{def}}{=} \langle u_1, u_1 \sqcup^{\mathsf{cr}} u_2\rangle \qquad \textbf{(otherwise)}$$

Figure 3.9: Propagation between values in the collapsed semantics

environment $\hat{E}$ is defined in the similar way as for the concrete environments.

3.4.4 Abstract Semantics

The abstract domain of the collapsed abstract semantics is given by *collapsed properties* $\mathcal{P}^{\mathsf{cr}} \stackrel{\text{def}}{=} \mathbb{N} \to \mathcal{M}^{\mathsf{cr}}$, ordered by $\ddot{\sqsubseteq}^{\mathsf{cr}}$. The Galois connection $\langle \dot{\alpha}^{\mathsf{cr}}, \dot{\gamma}^{\mathsf{cr}}\rangle$ is lifted to the Galois connection $\langle \ddot{\alpha}^{\mathsf{cr}}, \ddot{\gamma}^{\mathsf{cr}}\rangle$ between the lattices $\mathcal{P}^{\mathsf{r}}$ and $\mathcal{P}^{\mathsf{cr}}$. The collapsed semantics $\mathbf{C}^{\mathsf{cr}}[\![\cdot]\!] : \lambda^d \to \mathcal{P}^{\mathsf{cr}}$ is then defined as

$$\mathbf{C}^{\mathsf{cr}}[\![e]\!] \stackrel{\text{def}}{=} \mathbf{lfp}^{\ddot{\sqsubseteq}^{\mathsf{cr}}}_{\Lambda k.\ M^{\mathsf{cr}}_{\bot^{\mathsf{cr}}}} \Lambda P^{\mathsf{cr}}.\Lambda k.\ \mathsf{Step}^{\mathsf{cr}}_k[\![e]\!](\epsilon)(P^{\mathsf{cr}}(k))$$

where $\mathsf{Step}^{\mathsf{cr}}_k[\![e]\!](\hat{E})(M^{\mathsf{cr}}) \stackrel{\text{def}}{=} M^{\mathsf{cr}} \dot{\sqcup}^{\mathsf{cr}} \mathsf{step}^{\mathsf{cr}}_k[\![e]\!](\hat{E})(M^{\mathsf{cr}})$.

Theorem 2. *The collapsed abstract semantics is sound, i.e., for all programs e,* $\mathbf{C}^{\mathsf{r}}[\![e]\!] \ddot{\sqsubseteq}^{\mathsf{r}} \ddot{\gamma}^{\mathsf{cr}}(\mathbf{C}^{\mathsf{cr}}[\![e]\!])$.

The proof outline is as follows. First, the collapsed domain operations are by definition sound abstractions of the corresponding relational operations. It follows that $\mathsf{prop}^{\mathsf{cr}}$ abstracts $\mathsf{prop}^{\mathsf{r}}$. These facts are then used to prove that $\mathsf{step}^{\mathsf{cr}}$ overapproximates $\mathsf{step}^{\mathsf{r}}$. The only nontrivial case is the rule for recursive function defini-

tions. The proof first shows that after evaluating the body e_1 for every call site n_{cs} in the relational semantics, the resulting map $\overline{M^{\mathsf{r}}}(n_{\mathsf{cs}}) = \mathsf{Step}^{\mathsf{r}}_{k+1}[\![e_1]\!](E_1)(M^{\mathsf{r}}_1)$ is in $\dot{\gamma}^{\mathsf{cr}}(M^{\mathsf{cr}}_2)$ where $M^{\mathsf{cr}}_2 = \mathsf{Step}^{\mathsf{cr}}_{k+1}[\![e_1]\!](\hat{E}_1)(M^{\mathsf{cr}}_1)$. It then follows that $\dot{\bigsqcup}^{\mathsf{r}}_{n_{\mathsf{cs}} \in T^{\mathsf{cr}}} \overline{M^{\mathsf{r}}}(n_{\mathsf{cs}}) \in \dot{\gamma}^{\mathsf{cr}}(M^{\mathsf{cr}}_2)$.

One important implication of node abstraction is that the collapsed semantics only reaches finitely many abstract nodes.

Lemma 12. *For all programs e and $k \in \mathbb{N}$, either $\mathbf{C}^{\mathsf{cr}}[\![e]\!](k)(\hat{n}) = M^{\mathsf{cr}}_{\top_{\mathsf{cr}}}$ or the set $\{\, \hat{n} \in \hat{\mathcal{N}} \mid \mathbf{C}^{\mathsf{cr}}[\![e]\!](k)(\hat{n}) \neq \bot^{\mathsf{cr}} \,\}$ is finite.*

In fact, it can be easily observed from the definition of the abstract transformer that, given some initial abstract environment, every subexpression will be analyzed with at most one abstract environment. This is a consequence of the fact that the transformer in the case of recursive function definitions (where variables are bound) always extends the given environment with the same unique nodes. This implies that there is a one-to-one correspondence between expression/variable nodes and expression/variable locations for the nodes that are reachable.

Example. Consider again our running program e from Example 1. The stabilized collapsed map M^{cr}_e in the fixpoint of $\mathbf{C}^{\mathsf{cr}}[\![e]\!]$ is as follows:

$$\begin{aligned}
&\mathtt{id} \to \langle z, \{(\nu\colon 1), (\nu\colon 2)\}, \{(z\colon 1, \nu\colon 1), (z\colon 2, \nu\colon 2)\}\rangle \\
&\mathtt{u}, b, c \to \{(\mathtt{id}\colon \mathsf{F}, \nu\colon 1)\} \qquad\qquad e, f \to \{(\mathtt{id}\colon \mathsf{F}, \mathtt{u}\colon 1, \nu\colon 2)\} \\
&a \to \langle z, \{(\mathtt{id}\colon \mathsf{F}, \nu\colon 1)\}, \{(\mathtt{id}\colon \mathsf{F}, z\colon 1, \nu\colon 1)\}\rangle \\
&d \to \langle z, \{(\mathtt{id}\colon \mathsf{F}, \mathtt{u}\colon 1, \nu\colon 2)\}, \{(\mathtt{id}\colon \mathsf{F}, \mathtt{u}\colon 1, z\colon 2, \nu\colon 2)\}\rangle \\
&g^b, g^e \to \{(x\colon 1, \nu\colon 1), (x\colon 2, \nu\colon 2)\}
\end{aligned}$$

Note that in this specific example there is no precision loss in terms of the input-output relations. In particular, for the nodes at locations c and f we have a

collapsed relation that contains a single vector, which means that we still have precise information about the value obtained as the result of function calls of `id`. This happens since the input values at the two calls to `id` are different and function values keep track of precise dependencies on constant inputs. The output relation of `id` at each call site is strengthened inside $\mathsf{prop}^{\mathsf{cr}}$ with the actual value of the corresponding input. For instance, at the call site identified with e, the output of `id` $\{(z:1, \nu:1), (z:2, \nu:2)\}$ is effectively strengthened with the input $\{(\mathtt{id}:\mathsf{F}, \mathtt{u}:1, z:2)\}$ emitted at e, resulting in the value $\{(\mathtt{id}:\mathsf{F}, \mathtt{u}:1, \nu:2)\}$.

3.5 Data Flow Refinement Type Semantics

Finally, we obtain a parametric generalization of Liquid types, which we refer to as the *data flow refinement type* semantics. This semantics is an abstraction of the collapsed relational semantics. The abstraction combines three ideas: (1) we abstract dependency relations over concrete constants by abstract relations drawn from some abstract domain of type refinements, (2) we fold the step-indexed properties in the collapsed semantics into single *type maps*, and (3) we introduce widening to enforce convergence of the fixpoint in a finite number of iterations. We will show that the Liquid type inference proposed in [78] is a specific instance of this parametric abstract semantics.

3.5.1 Abstract Domains

Our data flow refinement type analysis is parameterized by the choice of base types and the abstract domain representing type refinements. We assume we are given a set of base types $\mathsf{b} \in \mathsf{B}$ whose meaning is provided by the concretization function

$\gamma_B : \mathsf{B} \to \wp(Cons)$. For simplicity, base types are assumed to be disjoint and every constant has a base type $\forall c, \exists \mathsf{b}.\, c \in \gamma_B(\mathsf{b})$.

Type refinements a are drawn from a family of relational abstract lattices $\langle \mathcal{A}_{\hat{N}}, \sqsubseteq^{\mathsf{a}}, \bot^{\mathsf{a}}, \top^{\mathsf{a}}, \sqcup^{\mathsf{a}}, \sqcap^{\mathsf{a}} \rangle$ parameterized by abstract scopes $\hat{N}$. We assume that there exists a Galois connection $\langle \alpha^{\mathsf{a}}_{\hat{N}}, \gamma^{\mathsf{a}}_{\hat{N}} \rangle$ between $\mathcal{A}_{\hat{N}}$ and the complete lattice of collapsed dependency relations $\langle \mathcal{R}^{\mathsf{cr}}_{\hat{N}}, \subseteq, \emptyset, \mathcal{D}^{\mathsf{cr}}_{\hat{N}}, \cup, \cap \rangle$ for every $\hat{N}$. We further assume that the abstract domains $\mathcal{A}_{\hat{N}}$ come equipped with sound abstractions of the various strengthening operations on collapsed dependency relations. For instance, we assume that there exists an operation $a_{\hat{N}}[\hat{n}_1 = \hat{n}_2]$ such that $\alpha^{\mathsf{a}}_{\hat{N}}(\gamma^{\mathsf{a}}_{\hat{N}}(a_{\hat{N}})[\hat{n}_1 = \hat{n}_2]) \sqsubseteq^{\mathsf{a}} a_{\hat{N}}[\hat{n}_1 = \hat{n}_2]$. Finally, we assume that these abstract domains provide widening operators $\triangledown^{\mathsf{a}}_{\hat{N}} : \mathcal{A}_{\hat{N}} \times \mathcal{A}_{\hat{N}} \to \mathcal{A}_{\hat{N}}$.

The semantic domains of the data flow refinement type semantics are as follows:

$$
\begin{array}{rr}
t \in \mathcal{V}^{\mathsf{t}}_{\hat{N}} ::= \bot^{\mathsf{t}} \mid \top^{\mathsf{t}} \mid \mathcal{R}^{\mathsf{t}}_{\hat{N}} \mid T^{\mathsf{t}}_{\hat{N}} & \text{data flow refinement types} \\
T^{\mathsf{t}}_{\hat{N}} \in \mathcal{T}^{\mathsf{t}}_{\hat{N}} \stackrel{\text{def}}{=} \Sigma z \in DVar.\, \mathcal{V}^{\mathsf{t}}_{\hat{N} \setminus \{z\}} \times \mathcal{V}^{\mathsf{t}}_{\hat{N} \cup \{z\}} & \text{refinement function types} \\
R^{\mathsf{t}}_{\hat{N}} \in \mathcal{R}^{\mathsf{t}}_{\hat{N}} \stackrel{\text{def}}{=} \mathsf{B} \times \mathcal{A}_{\hat{N}} & \text{refinement base types} \\
M^{\mathsf{t}} \in \mathcal{M}^{\mathsf{t}} \stackrel{\text{def}}{=} \Pi \hat{n} \in \hat{\mathcal{N}}.\, \mathcal{V}^{\mathsf{t}}_{\hat{N}_{\hat{n}}} & \text{refinement type maps}
\end{array}
$$

The abstract values of the new semantics, which we refer to as data flow refinement types, closely resembles collapsed abstract values. The major difference is in the treatment of dependency relations. The abstraction of a dependency relation is a refinement base type $R^{\mathsf{t}}_{\hat{N}}$, which consists of a base type b and an abstract refinement relation a. We denote these pairs as $\{\nu \colon \mathsf{b} \mid a\}$ and write $\mathsf{b}(R^{\mathsf{t}}_{\hat{N}})$ for b and $a(R^{\mathsf{t}}_{\hat{N}})$ for a. Refinement function types abstract collapsed tables and consist of an input type t_i, which is bound to a dependency variable z, and an output type t_o. We use the familiar function type notation $z \colon t_i \to t_o$ for these values. As for collapsed

tables, we use the functions dx and io for extracting the dependency variable, respectively, the pair of input/output types from function types. As expected, refinement type maps associate a type of appropriate scope with each abstract node. The refinement boolean true and false values are defined as $true^{\mathsf{t}} \stackrel{\text{def}}{=} \{\nu : \mathsf{bool} \mid a_t\}$ and $false^{\mathsf{t}} \stackrel{\text{def}}{=} \{\nu\colon \mathsf{bool} \mid a_f\}$, respectively, where $Bool^{\mathsf{t}} \stackrel{\text{def}}{=} \{\nu\colon \mathsf{bool} \mid a_\top\}$.

The ordering on types is defined as:

$$t_1 \sqsubseteq^{\mathsf{t}} t_2 \iff^{\text{def}} t_1 = \bot^{\mathsf{t}} \vee t_2 = \top^{\mathsf{t}} \vee (t_1, t_2 \in \mathcal{R}^{\mathsf{t}}_{\hat{N}} \wedge \mathsf{b}(t_1) = \mathsf{b}(t_2) \wedge\ a(t_1) \sqsubseteq^{\mathsf{a}} a(t_2)) \\ \vee\ (t_1, t_2 \in \mathcal{T}^{\mathsf{t}}_{\hat{N}} \wedge io(t_1) \stackrel{\cdot}{\sqsubseteq}^{\mathsf{t}} io(t_2))$$

Again, we implicitly quotient types modulo renaming of dependency variables to obtain a partial order that induces a family of complete lattices $(\mathcal{V}^{\mathsf{t}}_{\hat{N}}, \sqsubseteq^{\mathsf{t}}, \bot^{\mathsf{t}}, \top^{\mathsf{t}}, \sqcup^{\mathsf{t}}, \sqcap^{\mathsf{t}})$. The definition of the join $\sqcup^{\mathsf{t}}$ operator is as follows.

$$\{\nu\colon \mathsf{b} \mid a_1\} \sqcup^{\mathsf{t}} \{\nu\colon \mathsf{b} \mid a_2\} \stackrel{\text{def}}{=} \{\nu\colon \mathsf{b} \mid a_1 \sqcup^{\mathsf{a}} a_2\} \qquad \bot^{\mathsf{t}} \sqcup^{\mathsf{t}} t \stackrel{\text{def}}{=} t \qquad t \sqcup^{\mathsf{t}} \bot^{\mathsf{t}} \stackrel{\text{def}}{=} t$$

$$z\colon t_1 \to t_2 \sqcup^{\mathsf{t}} z\colon t_3 \to t_4 \stackrel{\text{def}}{=} z\colon t_1 \sqcup^{\mathsf{t}} t_3 \to t_2 \sqcup^{\mathsf{t}} t_4 \qquad t_1 \sqcup^{\mathsf{t}} t_2 \stackrel{\text{def}}{=} \top^{\mathsf{t}} \quad \text{(otherwise)}$$

The meet $\sqcap^{\mathsf{t}}$ is defined similarly.

Lemma 13. *Let $V \in \wp(\mathcal{V}^{\mathsf{t}}_{\hat{N}})$. Then, $\sqcup^{\mathsf{t}} V = lub(V)$ and $\sqcap^{\mathsf{t}} V = glb(V)$ subject to $\sqsubseteq^{\mathsf{t}}$.*

These are lifted pointwise to a complete lattice of refinement type maps $(\mathcal{M}^{\mathsf{t}}, \stackrel{\cdot}{\sqsubseteq}^{\mathsf{t}}, M^{\mathsf{t}}_{\bot^{\mathsf{t}}}, M^{\mathsf{t}}_{\top^{\mathsf{t}}}, \dot{\sqcup}^{\mathsf{t}}, \dot{\sqcap}^{\mathsf{t}})$.

Galois connections

The meaning of refinement types is given by the function $\gamma^{\mathsf{t}} : \mathcal{V}^{\mathsf{t}}_{\hat{N}} \to \mathcal{V}^{\mathsf{cr}}_{\hat{N}}$ that uses the concretization functions for base types and type refinements to map a

refinement type to a collapsed value:

$$\gamma^{\mathsf{t}}(\bot^{\mathsf{t}}) \stackrel{\text{def}}{=} \bot^{\mathsf{cr}} \qquad\qquad \gamma^{\mathsf{t}}(\top^{\mathsf{t}}) \stackrel{\text{def}}{=} \top^{\mathsf{cr}}$$

$$\gamma^{\mathsf{t}}(z\colon t_i \to t_o) \stackrel{\text{def}}{=} \langle z, \gamma^{\mathsf{t}}(t_i), \gamma^{\mathsf{t}}(t_o)\rangle$$

$$\gamma^{\mathsf{t}}(\{\nu\colon \mathsf{b} \mid a\}) \stackrel{\text{def}}{=} \{D^{\mathsf{cr}} \mid D^{\mathsf{cr}} \in \gamma^{\mathsf{a}}(a) \wedge D^{\mathsf{cr}}(\nu) \in \gamma_B(\mathsf{b})\}$$

Lemma 14. *For all $\hat{N}$, $\gamma^{\mathsf{t}}_{\hat{N}}$ is a complete meet-morphism between $\mathcal{V}^{\mathsf{t}}_{\hat{N}}$ and $\mathcal{V}^{\mathsf{cr}}_{\hat{N}}$.*

We follow the usual notations for the Galois connections induced by $\gamma^{\mathsf{t}}_{\hat{N}}$, which we lift pointwise to a Galois connection $\langle \dot{\alpha}^{\mathsf{t}}, \dot{\gamma}^{\mathsf{t}}\rangle$ between type maps and collapsed maps.

3.5.2 Abstract propagation and transformer

As usual, the domain operations on refinement types can be defined as the best abstractions, using the above introduced Galois connections, of domain operations of the collapsed semantics. Here, we provide a more direct structural definition of refinement type operations by relying on the corresponding operations of the given domain $\mathcal{A}_{\hat{N}}$ of type refinements. First, $t_{\hat{N}}[z \leftarrow t] \stackrel{\text{def}}{=} t$ whenever $z \notin \hat{N}$. Otherwise, the definition goes as shown in Figure 3.10. The strengthening of a bottom and

$$\begin{array}{lll}
t[z \leftarrow \bot^{\mathsf{t}}] & \stackrel{\text{def}}{=} \bot^{\mathsf{t}} & \\
t[z \leftarrow t'] & \stackrel{\text{def}}{=} t & \text{if } t \in \{\bot^{\mathsf{t}}, \top^{\mathsf{t}}\} \text{ or } t' \in \{\top^{\mathsf{t}}\} \cup \mathcal{T}^{\mathsf{t}} \\
\{\nu\colon \mathsf{b}_1 \mid a_1\}[z \leftarrow \{\nu\colon \mathsf{b}_2 \mid a_2\}] & \stackrel{\text{def}}{=} \{\nu\colon \mathsf{b}_1 \mid a_1[z \leftarrow a_2]\} & \\
(z\colon t_1 \to t_2)[z_1 \leftarrow t] & \stackrel{\text{def}}{=} z\colon (t_1[z_1 \leftarrow t]) \to (t_2[z_1 \leftarrow \exists z.\, t]) & t \in \mathcal{R}^{\mathsf{t}}
\end{array}$$

Figure 3.10: A structural definition of strengthening operations on refinement types

top type in fact does not perform any proper strengthening. The same applies

for the cases where we strengthen with $\top^{\mathsf{t}}$ or function types. The definition of the strengthening operations for the mentioned cases is guided by simplicity and it is in line with the strengthening operations performed by Liquid type inference that does not explicitly model $\bot^{\mathsf{t}}$ and $\top^{\mathsf{t}}$ types. The strengthening of a function type is defined structurally by performing the strengthening on the corresponding input and output types. Note that the definition for strengthening an output type ensures that the dependency of the output values on the input are not changed by projecting out the dependency variable z from the scope of the type t used to perform the strengthening. The strengthening of a basic refinement type reduces to strengthening of the corresponding refinement domain elements. The structural definitions of other operations on types can be defined similarly.

Lemma 15. *Structural strengthening operations are sound.*

The propagation operation $\mathsf{prop}^{\mathsf{t}}$ on refinement types is then obtained from $\mathsf{prop}^{\mathsf{cr}}$ by replacing all operations on collapsed values with their counterparts from the refinement type semantics. We omit the definition as it is straightforward. Also, the refinement type abstracting a constant c with abstract scope $\hat{N}$ is defined as $c^{\mathsf{t}}_{\hat{N}} \stackrel{\text{def}}{=} \{\nu\colon ty(c) \mid \top^{\mathsf{a}}[\nu{=}c^{\mathsf{t}}]\}$ where $\top^{\mathsf{a}} \in \mathcal{A}_{\hat{N}}$. We define the Boolean refinement type as $Bool^{\mathsf{t}}_{\hat{N}} \triangleq false^{\mathsf{t}}_{\hat{N}} \sqcup^{\mathsf{t}} true^{\mathsf{t}}_{\hat{N}}$.

We obtain the new abstract transformer $\mathsf{Step}^{\mathsf{t}}$ from the collapsed transformer $\mathsf{step}^{\mathsf{cr}}$ in the same fashion. However, the signature of the new transformer is now $\mathsf{Step}^{\mathsf{t}} : \lambda^d \to \hat{\mathcal{E}} \to \mathcal{M}^{\mathsf{t}} \to \mathcal{M}^{\mathsf{t}}$. That is, we additionally eliminate the step-indexing parameter k. The transformer is given as $\mathsf{Step}^{\mathsf{t}}[\![e^{\ell}]\!](\hat{E})(M^{\mathsf{t}}) \stackrel{\text{def}}{=} M^{\mathsf{t}} \dot{\sqcup}^{\mathsf{t}} \mathsf{step}^{\mathsf{t}}[\![e^{\ell}]\!](\hat{E})(M^{\mathsf{t}})$ where $\mathsf{step}^{\mathsf{t}}$ is shown in Figure 3.11. As the transformer definition is parametric in the operations on the abstract domain of type refinements, different choices of refinements do not require changes in the transformer. Moreover, the analysis ab-

$$\mathsf{step}^{\mathsf{t}}[\![c^{\ell}]\!](\hat{E})(M^{\mathsf{t}}) \stackrel{\text{def}}{=} M^{\mathsf{t}}[\hat{E}\diamond\ell \mapsto M^{\mathsf{t}}(\hat{E}\diamond\ell) \sqcup^{\mathsf{t}} c^{\mathsf{t}}[M^{\mathsf{t}}]]$$

$$\mathsf{step}^{\mathsf{t}}[\![x^{\ell}]\!](\hat{E})(M^{\mathsf{t}}) \stackrel{\text{def}}{=}$$
$$\quad \textbf{let } \langle t'_x, t'\rangle = \mathsf{prop}^{\mathsf{t}}(M^{\mathsf{t}}(\hat{E}(x))[\nu{=}\hat{E}(x)], M^{\mathsf{t}}(\hat{E}\diamond\ell))$$
$$\quad \textbf{in } M^{\mathsf{t}}[\hat{E}(x) \mapsto t'_x, \hat{E}\diamond\ell \mapsto t'[M^{\mathsf{t}}]]$$

$$\mathsf{step}^{\mathsf{t}}[\![(e_1^{\ell_1}e_2^{\ell_2})^{\ell}]\!](\hat{E})(M^{\mathsf{t}}) \stackrel{\text{def}}{=}$$
$$\quad \textbf{let } M_1^{\mathsf{t}} = \mathsf{Step}^{\mathsf{t}}[\![e_1]\!](\hat{E})(M^{\mathsf{t}}); t_1 =_{M_1^{\mathsf{t}}} M_1^{\mathsf{t}}(\hat{E}\diamond\ell_1) \textbf{ in}$$
$$\quad \textbf{if } t_1 \notin \mathcal{T}^{\mathsf{t}} \textbf{ then } M^{\mathsf{t}}_{\top^{\mathsf{t}}} \textbf{ else}$$
$$\quad\ \textbf{let } M_2^{\mathsf{t}} = \mathsf{Step}^{\mathsf{t}}[\![e_2]\!](\hat{E})(M_1^{\mathsf{t}}); t_2 =_{M_2^{\mathsf{t}}} M_2^{\mathsf{t}}(\hat{E}\diamond\ell_2) \textbf{ in}$$
$$\quad\ \textbf{let } \langle t'_1, T_0^{\mathsf{t}}\rangle = \mathsf{prop}^{\mathsf{t}}(t_1, dx(t_1)\colon t_2 \to M_2^{\mathsf{t}}(\hat{E}\diamond\ell))$$
$$\qquad \langle t'_2, t'\rangle = io(T_0^{\mathsf{t}})$$
$$\quad\ \textbf{in } M_2^{\mathsf{t}}[\hat{E}\diamond\ell_1 \mapsto t'_1, \hat{E}\diamond\ell_2 \mapsto t'_2, \hat{E}\diamond\ell \mapsto t']$$

$$\mathsf{step}^{\mathsf{t}}[\![(\mu f.\lambda x.e_1^{\ell_1})^{\ell}]\!](\hat{E})(M^{\mathsf{t}}) \stackrel{\text{def}}{=}$$
$$\quad \textbf{let } T^{\mathsf{t}} =_{M^{\mathsf{t}}} M^{\mathsf{t}}(\hat{E}\diamond\ell) \sqcup^{\mathsf{t}} T_{\perp}^{\mathsf{t}} \textbf{ in}$$
$$\quad \textbf{if } \pi_1(io(T^{\mathsf{t}})) = \perp^{\mathsf{t}} \textbf{ then } M^{\mathsf{t}}[\hat{E}\diamond\ell \mapsto T^{\mathsf{t}}] \textbf{ else}$$
$$\quad \textbf{if } \pi_1(io(T^{\mathsf{t}})) = \top^{\mathsf{t}} \textbf{ then } M^{\mathsf{t}}_{\top^{\mathsf{t}}} \textbf{ else}$$
$$\quad \textbf{let } \hat{n}_x = \hat{E}\diamond x; \hat{n}_f = \hat{E}\diamond f; \hat{E}_1 = \hat{E}[x \mapsto \hat{n}_x, f \mapsto \hat{n}_f]$$
$$\quad \langle T_x^{\mathsf{t}}, T_1^{\mathsf{t}}\rangle = \mathsf{prop}^{\mathsf{t}}(dx(T^{\mathsf{t}})\colon M^{\mathsf{t}}(n_x) \to M^{\mathsf{t}}(\hat{E}_1\diamond\ell_1)[dx(T^{\mathsf{t}}){=}\hat{n}_x], T^{\mathsf{t}})$$
$$\quad \langle T_2^{\mathsf{t}}, T_f^{\mathsf{t}}\rangle = \mathsf{prop}^{\mathsf{t}}(T^{\mathsf{t}}, M^{\mathsf{t}}(\hat{n}_f)); \langle t'_x, t'_1\rangle = io(T_x^{\mathsf{t}})$$
$$\quad M_1^{\mathsf{t}} = M^{\mathsf{t}}[\hat{E}\diamond\ell \mapsto T_1^{\mathsf{t}} \sqcup^{\mathsf{t}} T_2^{\mathsf{t}}, \hat{n}_f \mapsto T_f^{\mathsf{t}}, \hat{n}_x \mapsto t'_x, \hat{E}_1\diamond\ell_1 \mapsto t'_1[\hat{n}_x{=}dx(T^{\mathsf{t}})]]$$
$$\quad \textbf{in } \mathsf{Step}^{\mathsf{t}}[\![e_1]\!](\hat{E}_1)(M_1^{\mathsf{t}})$$

$$\mathsf{step}^{\mathsf{t}}[\![(e_0^{\ell_0}\,?\,e_1^{\ell_1} : e_2^{\ell_2})^{\ell}]\!](\hat{E})(M^{\mathsf{t}}) \stackrel{\text{def}}{=}$$
$$\quad \textbf{let } M_0^{\mathsf{t}} = \mathsf{Step}^{\mathsf{t}}[\![e_0]\!](\hat{E})(M^{\mathsf{t}}); t_0 =_{M^{\mathsf{t}}} M_0^{\mathsf{t}}(\hat{E}\diamond\ell_0) \textbf{ in}$$
$$\quad \textbf{if } t_0 \not\sqsubseteq^{\mathsf{t}} \mathit{Bool}^{\mathsf{t}} \textbf{ then } M^{\mathsf{t}}_{\top^{\mathsf{t}}} \textbf{ else}$$
$$\quad \textbf{let } M_1^{\mathsf{t}} = \mathsf{Step}^{\mathsf{t}}[\![e_1]\!](\hat{E})(M_0^{\mathsf{t}}[\hat{E}\diamond\ell_0 \leftarrow \mathit{true}^{\mathsf{t}} \sqcap^{\mathsf{t}} t_0])$$
$$\qquad M_2^{\mathsf{t}} = \mathsf{Step}^{\mathsf{t}}[\![e_2]\!](\hat{E})(M_0^{\mathsf{t}}[\hat{E}\diamond\ell_0 \leftarrow \mathit{false}^{\mathsf{t}} \sqcap^{\mathsf{t}} t_0])$$
$$\qquad t_1 = M_1^{\mathsf{t}}(\hat{E}\diamond\ell_1); t_2 = M_2^{\mathsf{t}}(\hat{E}\diamond\ell_2)$$
$$\qquad \langle t'_1, t'\rangle = \mathsf{prop}^{\mathsf{t}}(t_1, M_1^{\mathsf{t}}(\hat{E}\diamond\ell)); \langle t'_2, t''\rangle = \mathsf{prop}^{\mathsf{t}}(t_2, M_2^{\mathsf{t}}(\hat{E}\diamond\ell))$$
$$\quad \textbf{in } M_0^{\mathsf{t}} \dot{\sqcup}^{\mathsf{t}} M_1^{\mathsf{t}}[\hat{E}\diamond\ell_1 \mapsto t'_1, \hat{E}\diamond\ell \mapsto t'] \dot{\sqcup}^{\mathsf{t}} M_2^{\mathsf{t}}[\hat{E}\diamond\ell_2 \mapsto t'_2, \hat{E}\diamond\ell \mapsto t'']$$

Figure 3.11: Abstract transformer for data flow refinement semantics

stracts from the specific representation of type refinements. If one uses, e.g., polyhedra for type refinements, the actual implementations can choose between constraint and generator representations of polyhedra elements [19], providing more flexibility for tuning the analysis' precision/efficiency trade-off.

Lemma 16. *The function* $\mathsf{prop}^{\mathsf{t}}$ *is increasing. Assuming that refinement domain operations are monotone,* $\mathsf{prop}^{\mathsf{t}}$ *is monotone as well.*

Lemma 17. *For every* $k \in \mathbb{N}$, $e \in \lambda^d$ *and* $\hat{E} \in \hat{\mathcal{E}}$ *such that* $\langle e, \hat{E} \rangle$ *is well-formed, then* $\mathsf{Step}^{\mathsf{t}}[\![e]\!](\hat{E})$ *is increasing. If the refinement domain operations are monotone, then* $\mathsf{Step}^{\mathsf{t}}[\![e]\!](\hat{E})$ *is also monotone.*

Widening

To ensure that the type analysis converges in finitely many steps, we need a widening operator for $\mathcal{V}^{\mathsf{t}}_{\hat{N}}$. We construct this widening operator from the widening operator $\triangledown^{\mathsf{a}}_{\hat{N}}$ on the domains of refinement relations and a *shape widening* operator. In order to define the latter, first define the *shape* of a type using the function $\mathsf{sh} : \mathcal{V}^{\mathsf{t}}_{\hat{N}} \to \mathcal{V}^{\mathsf{t}}_{\hat{N}}$

$$\mathsf{sh}(\bot^{\mathsf{t}}) \stackrel{\text{def}}{=} \bot^{\mathsf{t}} \quad \mathsf{sh}(\top^{\mathsf{t}}) \stackrel{\text{def}}{=} \top^{\mathsf{t}} \quad \mathsf{sh}(R^{\mathsf{t}}) \stackrel{\text{def}}{=} \bot^{\mathsf{t}} \qquad \mathsf{sh}(z\colon t_i \to t_o) \stackrel{\text{def}}{=} z\colon \mathsf{sh}(t_i) \to \mathsf{sh}(t_o)$$

A shape widening operator is a function $\triangledown^{\mathsf{sh}}_{\hat{N}} : \mathcal{V}^{\mathsf{t}}_{\hat{N}} \times \mathcal{V}^{\mathsf{t}}_{\hat{N}} \to \mathcal{V}^{\mathsf{t}}_{\hat{N}}$ such that (1) $\triangledown^{\mathsf{sh}}_{\hat{N}}$ is an upper bound operator and (2) for every infinite ascending chain $t_0 \sqsubseteq^{\mathsf{t}}_{\hat{N}} t_0 \sqsubseteq^{\mathsf{t}}_{\hat{N}} \ldots$, the chain $\mathsf{sh}(t'_0) \sqsubseteq^{\mathsf{t}}_{\hat{N}} \mathsf{sh}(t'_1) \sqsubseteq^{\mathsf{t}}_{\hat{N}} \ldots$ stabilizes, where $t'_0 \stackrel{\text{def}}{=} t_0$ and $t'_i \stackrel{\text{def}}{=} t'_{i-1} \triangledown^{\mathsf{sh}}_{\hat{N}} t_i$ for $i > 0$. In what follows, let $\triangledown^{\mathsf{sh}}_{\hat{N}}$ be a shape widening operator.

First, we lift $\triangledown^{\mathsf{a}}_{\hat{N}}$ to an upper bound operator $\triangledown^{\mathsf{ra}}_{\hat{N}}$ on $\mathcal{V}^{\mathsf{t}}_{\hat{N}}$:

$$
\begin{array}{ll}
t \mathbin{\triangledown^{\mathsf{ra}}_{\hat{N}}} t' \stackrel{\text{def}}{=} & \\
\quad \{\nu\colon \mathsf{b} \mid a \mathbin{\triangledown^{\mathsf{a}}_{\hat{N}}} a'\} & \textbf{if } t = \{\nu\colon \mathsf{b} \mid a\} \wedge t' = \{\nu\colon \mathsf{b} \mid a'\} \\
\quad z\colon (t_i \mathbin{\triangledown^{\mathsf{ra}}_{\hat{N}}} t'_i) \to (t_o \mathbin{\triangledown^{\mathsf{ra}}_{\hat{N} \cup \{z\}}} t'_o) & \textbf{if } t = z\colon t_i \to t_o \wedge t' = z\colon t'_i \to t'_o \\
\quad t \mathbin{\sqcup^{\mathsf{t}}_{\hat{N}}} t' & \textbf{otherwise}
\end{array}
$$

We then define the operator $\triangledown^{\mathsf{t}}_{\hat{N}} : \mathcal{V}^{\mathsf{t}}_{\hat{N}} \times \mathcal{V}^{\mathsf{t}}_{\hat{N}} \to \mathcal{V}^{\mathsf{t}}_{\hat{N}}$ as the composition of $\triangledown^{\mathsf{sh}}_{\hat{N}}$ and $\triangledown^{\mathsf{ra}}_{\hat{N}}$, that is, $t \mathbin{\triangledown^{\mathsf{t}}_{\hat{N}}} t' \stackrel{\text{def}}{=} t \mathbin{\triangledown^{\mathsf{ra}}_{\hat{N}}} (t \mathbin{\triangledown^{\mathsf{sh}}_{\hat{N}}} t')$.

Lemma 18. *$\triangledown^{\mathsf{t}}_{\hat{N}}$ is a widening operator.*

3.5.3 Abstract semantics

We identify the properties $\mathcal{P}^{\mathsf{t}}$ of the data flow refinement type semantics with type maps, $\mathcal{P}^{\mathsf{t}} \stackrel{\text{def}}{=} \mathcal{M}^{\mathsf{t}}$. Then we connect $\mathcal{P}^{\mathsf{t}}$ to collapsed properties $\mathcal{P}^{\mathsf{cr}}$ via the Galois connection induced by the concretization function $\ddot{\gamma}^{\mathsf{t}} \stackrel{\text{def}}{=} \Lambda M^{\mathsf{t}}.\Lambda k.\ \dot{\gamma}^{\mathsf{t}}(M^{\mathsf{t}})$. We lift the widening operators $\triangledown^{\mathsf{t}}_{\hat{N}}$ pointwise to an upper bound operator $\dot{\triangledown}^{\mathsf{t}}\colon \mathcal{P}^{\mathsf{t}} \times \mathcal{P}^{\mathsf{t}} \to \mathcal{P}^{\mathsf{t}}$. The data flow refinement semantics $\mathbf{C}^{\mathsf{t}}[\![\cdot]\!] : \lambda^d \to \mathcal{P}^{\mathsf{t}}$ is then defined as the least fixpoint of the widened iterates of $\mathsf{Step}^{\mathsf{t}}$:

$$\mathbf{C}^{\mathsf{t}}[\![e]\!] \stackrel{\text{def}}{=} \mathbf{lfp}^{\dot{\sqsubseteq}^{\mathsf{t}}}_{M^{\mathsf{t}}_{\perp}} \Lambda M^{\mathsf{t}}.\ (M^{\mathsf{t}} \mathbin{\dot{\triangledown}^{\mathsf{t}}} \mathsf{Step}^{\mathsf{t}}[\![e]\!](\epsilon)(M^{\mathsf{t}}))$$

Theorem 3. *The refinement type semantics is sound and terminating, i.e., for all programs e, $\mathbf{C}^{\mathsf{t}}[\![e]\!]$ converges in finitely many iterations. Moreover, $\mathbf{C}^{\mathsf{cr}}[\![e]\!] \mathrel{\ddot{\sqsubseteq}^{\mathsf{cr}}} \ddot{\gamma}^{\mathsf{t}}(\mathbf{C}^{\mathsf{t}}[\![e]\!])$.*

The soundness proof and termination follow straightforwardly from the properties of the involved Galois connections, Lemma 18, Lemma 12, and Proposition 3.

We say that a type map M^{t} is *safe* if for all abstract nodes $\hat{n}$, $M^{\mathsf{t}}(\hat{n}) \neq \top^{\mathsf{t}}$. Let further $\ddot{\gamma} \stackrel{\text{def}}{=} \ddot{\gamma}^{\mathsf{r}} \circ \ddot{\gamma}^{\mathsf{cr}} \circ \ddot{\gamma}^{\mathsf{t}}$. The next lemma states that safe type maps yield safe properties. It follows immediately from the definitions of the concretizations.

Lemma 19. *For all safe type maps* M^{t}, $\ddot{\gamma}(M^{\mathsf{t}}) \dot{\subseteq} P_{\mathsf{safe}}$.

A direct corollary of this lemma and the soundness theorems for our abstract semantics is that we can use the data flow refinement type semantics to prove the safety of programs.

Corollary 1. *For all programs* e, *if* $\mathbf{C}^{\mathsf{t}}[\![e]\!]$ *is safe then so is* e.

3.5.4 Instantiating the framework

In order to obtain an instance of our data flow refinement type analysis, one has to choose basic types, the abstract domain of type refinements with the accompanying domain operations and widening operator, and the shape widening operator.

3.6 Liquid Type Semantics

We now recast Liquid types as an instantiation of the parametric data flow refinement type inference.

Abstract domains. The set of basic types consists of types such as int, bool, etc. The abstract domain of type refinements $\mathcal{A}_{\hat{N}}$ is induced by a given set of qualifiers Q by instantiating them with the given abstract scope $\hat{N}$. Hence, the abstract domain of type refinements is $\wp(Q_{\hat{N}})$ whose elements are interpreted conjunctively and quotiented by the pre-order induced by entailment modulo some appropriate

decidable first-order theory (e.g., linear arithmetic with uninterpreted functions as in [78]). This quotient is a complete lattice.

Domain Operations. The strengthening operations on our model of Liquid types are defined structurally, as in Figure 3.10, by relying on the strengthening operations provided by the Liquid domain of type refinements. These operations correspond to the logical operations used in the original Liquid types work: a refinement domain element a can be represented as a logical predicate. Based on this representation, $a[\nu{=}\hat{n}]$ corresponds to $a \wedge \nu = \hat{n}$ and $a[\hat{n} \leftarrow a']$ is $a \wedge a'[\hat{n}/\nu]$ where the latter operation is a syntactic substitution of $\hat{n}$ for ν in $'a$. The operation $\exists z.\, t$ of projecting out a dependency variable z from a scope of a type t can be defined as removing the conjuncts of t that involve z. In practice, Liquid types avoid using the rescoping operation in the strengthening operations by renaming dependency variables, thus making sure that variables in types are always unique.

Widenings. We note that the Liquid Type inference first infer basic types and only then it infers type refinements. Our data flow refinement semantics performs the two inference phases simultaneously, thus not forcing the analysis to make an a priori decision about how the shapes of the types of recursive functions should be approximated. To obtain Liquid Types in our framework, we need a shape widening operator that mimics the approximation of recursive functions performed by the Hindley-Milner type system on which Liquid Types are based. The connection between this approximation and widening is already well understood [15]. Since the refinement domain $\mathcal{A}_{\hat{N}}$ has finite height, the type refinement widening operator $\nabla^{\mathsf{a}}_{\hat{N}}$ is simply the join operator on $\wp(Q_{\hat{N}})$.

Connection to original Liquid types. We now connect our formalization of Liquid types with the one from the original work [78]. The subtyping relation on our model of Liquid types is defined as follows

$$\{\nu\colon \mathsf{b} \mid a_1\} <:^q \{\nu\colon \mathsf{b} \mid a_2\} \overset{\text{def}}{\iff} a_1 \sqsubseteq^{\mathsf{a}} a_2$$

$$z{:}t_1{\to}t_2 <:^q z{:}t_3{\to}t_4 \overset{\text{def}}{\iff} t_3 <:^q t_1 \wedge t_2[z \leftarrow t_3] <:^q t_4[z \leftarrow t_3]$$

Our subtyping relation coincides with the original one [78, p. 6]. The main difference is the original subtyping is defined subject to typing environment assumptions that we push directly into the type refinements. It can be shown that subtyping between types implies that propagation between these types has reached a fixpoint.

Lemma 20. *If* $t_1 <:^q t_2$*, then* $\langle t_1, t_2\rangle = \mathsf{prop}^{\mathsf{t}}(t_1, t_2)$.

We use this fact to show that a valid derivation of the Liquid typing judgment relation can be obtained from the results of our analysis. To this end, we first define a typing relation $\vdash_q$ adapted from [78, p. 6]. As with Liquid types, we define the typing rules with a more standard notion of typing environments $\Gamma \in \Xi \overset{\text{def}}{=} \mathit{Vars} \rightharpoonup \mathcal{V}^{\mathsf{t}}$ that map program variables to types. In order to connect the typing rules with the results produced by our abstract interpretation, we define the rules subject to an injective mapping $\kappa : \mathit{Vars} \to \hat{\mathcal{N}}$ from variables to abstract nodes. Let Ξ^κ be the set of liquid typing environments whose domain is $\mathsf{dom}(\kappa)$. The abstract scope induced by κ is then simply $\hat{N}_\kappa \overset{\text{def}}{=} \mathsf{rng}(\kappa)$. Also, the induced abstract environment is $\kappa(\Gamma) \overset{\text{def}}{=} \Lambda x \in \mathsf{dom}(\Gamma).\kappa(x)$. Given an encoding of abstract environment nodes κ, our Liquid typing relation $\vdash_q^\kappa\colon \Sigma\Gamma \in \Xi^\kappa.\, \lambda^d \times \mathcal{V}^{\mathsf{t}}_{\hat{N}_\kappa}$ is given by the typing rules shown in Figure 3.12. We assume we are given an existing base typing function $\mathit{ty} : \mathit{Cons} \to \mathsf{B}$ (e.g., Hindley-Milner) for constants and an existing strengthening operation $a[\nu = c]$ on type refinements that sets ν to c in the concrete (implemented

$$\frac{x \in \mathsf{dom}(\Gamma) \quad \Gamma(x) <:^q t \quad t = \Gamma(x)[\nu{=}\kappa(x)][\Gamma]^\kappa}{\Gamma \vdash^\kappa_q x : t} \qquad \frac{}{\Gamma \vdash^\kappa_q c : c^{\mathsf{t}}[\Gamma]^\kappa}$$

$$\frac{\Gamma \vdash^\kappa_q e_1 : t_1 \quad \Gamma \vdash^\kappa_q e_2 : t_2 \quad t_1 <:^q dx(t_1) \colon t_2 \to t}{\Gamma \vdash^\kappa_q e_1 e_2 : t}$$

$$\frac{\begin{array}{cc} t <:^q t_f & dx(t) \colon t_x \to t_1 <:^q t \\ \kappa_1 = \kappa.x \colon \kappa(\Gamma) \diamond x.f \colon \kappa(\Gamma) \diamond f & \Gamma.x : t_x.f : t_f \vdash^{\kappa_1}_q e : t_1 \end{array}}{\Gamma \vdash^\kappa_q \mu f \lambda x.\, e : t}$$

$$\frac{\begin{array}{cc} \Gamma \vdash^\kappa_q x : t_0 \quad t_0 <:^q \mathit{Bool}^{\mathsf{t}} & t_1 <:^q t \quad t_2 <:^q t \\ \Gamma_1 = \Lambda x \in \mathsf{dom}(\Gamma).\, \Gamma(x)[\kappa(x) \leftarrow t_0 \sqcap^{\mathsf{t}} \mathit{true}^{\mathsf{t}}] & \Gamma_1 \vdash^\kappa_q e_1 : t_1 \\ \Gamma_2 = \Lambda x \in \mathsf{dom}(\Gamma).\, \Gamma(x)[\kappa(x) \leftarrow t_0 \sqcap^{\mathsf{t}} \mathit{false}^{\mathsf{t}}] & \Gamma_2 \vdash^\kappa_q e_2 : t_2 \end{array}}{\Gamma \vdash^\kappa_q (x \,?\, e_1 : e_2) : t}$$

Figure 3.12: Adapted Liquid typing rules

as adding a logical conjunct $\nu = c$ to a). We use these operations and κ to properly strengthen constants with the environment assumptions. More precisely, $c^{\mathsf{t}}[\Gamma]^\kappa$ is a shorthand for $c^{\mathsf{t}}[M^{\mathsf{t}}]$ where $M^{\mathsf{t}} = \Lambda \hat{n}.\, \hat{n} \in \mathsf{rng}(\kappa) \,?\, \Gamma(\kappa^{-1}(\hat{n})) : \bot^{\mathsf{t}}$.

The first obvious difference between our typing rules and the original Liquid typing rules is that we need to maintain the mapping κ between program variables and environment nodes. Again, we use this mapping to easier connect the typing rules to our semantics and properly push environment assumptions to types that happens on the leaves of typing derivations. Second, the rules in the original paper are nondeterministic: the subtyping rule can be applied at any point in the typing derivation. Our rules, on the other hand, are deterministic as we directly push subtyping constraints at any point where there is a potential flow of data. Lastly, we assume for simplicity that conditionals are A-normalized, as is the case with the original Liquid type inference algorithm implementation [78, sec. 4.4].

It can then be shown that the Liquid typing rules specify a fixpoint of our

Liquid type semantics. Let *valid* be typing environments that do not contain $\bot^{\mathsf{t}}$ and $\top^{\mathsf{t}}$ types. We can now formalize the result that a solution computed by type inference based on the Liquid typing rules can be obtained using our abstract interpretation.

Theorem 4. *Let e^{ℓ} be an expression, κ an injective encoding of abstract variable nodes, $\Gamma \in \Xi^{\kappa}$ a valid typing environment, and $t \in \mathcal{V}^{\mathsf{t}}_{\hat{N}_{\kappa}}$. If $\Gamma \vdash^{\kappa}_{q} e^{\ell} : t$, then there exists a safe type map M^{t} such that $\mathsf{Step}^{\mathsf{t}}[\![e^{\ell}]\!](\kappa(\Gamma)\diamond\ell)(M^{\mathsf{t}}) = M^{\mathsf{t}}$ and $M^{\mathsf{t}}(\kappa(\Gamma)\diamond\ell) = t$.*

The proof goes by showing that one can construct a safe type map from the typing derivation corresponding to $\Gamma \vdash^{\kappa}_{q} e^{\ell} : t$. This map is then shown to be a fixpoint of the abstract transformer.

Design space. We note that our Liquid type inference is sound by construction and the points where the analysis loses precision are made explicit in our choice of widening operators and Galois connections used in the relational, collapsed, and refinement type semantics. This is in contrast to the original Liquid types work where the soundness of the inference is argued as follows. Given an expression e, a Liquid type environment Γ, and a qualifier set Q, it is first shown that for a Liquid type τ inferred by the algorithm $\mathsf{Solve}(\Gamma, e, Q)$, the original Liquid typing relation holds $\Gamma \vdash_{Q} e : \tau$ [78, Theorem 2]. Then, this relation is shown to be conservative subject to an "exact" typing relation $\Gamma \vdash e : \tau$ [78, Theorem 1], which is in turn proven to be sound subject to a standard operational semantics elsewhere [51]. This indirect soundness argument obscures where the analysis loses precision. In particular, it does not make formally explicit how $\vdash_{Q}$ abstracts $\vdash$ and how $\vdash$ abstracts the concrete semantics.

3.7 Related Work

We now discuss work related to refinement type inference.

Refinement type systems. One of the earliest works on refinement types goes back to [28]. The authors present a type system for ML where a programmer-annotated definition of an algebraic datatype induces a finite set of its subtypes, called refinements. These refinements constitute a finite-height (non-relational) abstract domain of type refinements. The presented type system uses intersection and union types in order to perform modular refinement type inference. Xi and Pfenning [96] introduce a modular and semi-automatic dependent type analysis for ML where refinement types (called index types) are defined over a parametric constraint domain. Techniques used in the original Liquid types [78] work were extended to support Haskell [92], abstractions over type refinements [91, 90], and inference of heap invariants for C programs [79]. Our results can serve as the basis for reformulating these works in terms of abstract interpretation. We also note that the work in [92] provides a denotational semantic model of Liquid types. However, this model is used for proving the soundness of Liquid typing rules and the inference algorithm. This is a different task than systematically and constructively designing Liquid type inference as an abstract semantics of programs, for which denotational semantics is not a good fit as it is inherently modular and it does not make explicit how data flows in programs. Zhu and Jagannathan [99] present an algorithm that infers data flow invariants of functional programs using a refinement type inference algorithm that solves subtyping constraints per function body and then lazily propagates the information from call sites to corresponding functions, akin to our data flow refinement type system. Moreover, their system

uses intersection types labeled by call site information to infer the type of a function based on different call sites. This suggests that their system can be formalized using our framework where the collapsed semantics is refined by a partition of call sites. Their inference algorithm, however, is not guaranteed to terminate. Viewed using our framework, this is a consequence of the fact that the abstract domain of type refinements is of infinite height and the fixpoint computation is not using a widening operator. Knowles and Flanagan [51] introduce a general refinement type system that infers data flow invariants for functional programs where refinements are conjunctions and disjunctions of unrestricted Boolean expressions drawn from the language of programs. Similar to Liquid types, their system first computes the basic type structure of the program and then sets up constraints encoding the program data flow. When viewed through the lens of our work, their inference algorithm works over an infinite-height abstract domain and uses certain widening operator to ensure finite convergence. When analyzing recursive functions, the resulting function type is itself defined as a least fixpoint, since the fixpoint operator is anyhow available in the language. The inference algorithm proposed in [43] can be seen as an implementation of our data flow refinement type system where (collapsed) data flow equations of a higher-order program are transformed into a first-order program that is then analyzed using various abstract domains. The results are then pushed back to the original higher-order program. Refinement type techniques have also been used in the context of dynamic languages [93, 48, 53]. The work in [53] uses Galois connections to formalize the intuition that gradual refinement types abstract static refinement types. However, the resulting gradual type system does not take full advantage of the framework of abstract interpretation, i.e., the type system is not calculationally constructed as an abstraction

of a concrete program semantics. We note that none of the above works formally describe their refinement type systems as abstract interpretations in this original sense.

Semantics of higher-order programs. Work in [41] and similarly [76] use flow graphs to assign concrete meaning to higher-order programs. Nodes in the graph correspond to nodes of our data flow semantics. Their semantics models functions as expression nodes storing the location of the function definition expression. Argument values then flow directly from call sites to the function. Hence, this semantics has to make non-local changes when analyzing function applications. In contrast, our data flow semantics treats functions as tables and propagates values backwards in lockstep with the program structure, which is more suitable for building type systems. The minimal function graph semantics [46, 45] models functions as tables whose domains only contain inputs observed during program execution, similar to our data flow semantics. However, minimal function graphs do not explicitly model flow of information in a program, which is not well suited for designing data flow refinement type systems. More precisely, similar to minimal function graphs, our new semantics computes for every function, the pairs of input and output values for the calls to that function that are actually observed during program execution. While minimal function graph semantics accumulates this information in a separate dedicated global structure, our semantics stores information on function calls more locally. That is, at each point in the flow graph where a function value is observed, we record those calls to the function that happen from that point onward as the function value continues to flow through the program. The semantics makes explicit how input values propagate backwards from the call sites of a function

to the function definition and, conversely, how output values propagate from the definition back to the call sites. Our semantics precisely captures the properties abstracted by Liquid types and is defined structurally on the program syntax, allowing for an easier formal connection to the typing rules underlying Liquid type inference.

There is a large body of work on control and data flow analysis on higher-order programs and the concrete semantics they overapproximate [58, 69, 65, 44, 18]. However, these works either do not make program data flow explicit or they treat program functions as graph nodes, continuations, or closures, all of which are not well-suited for developing data flow refinement type systems as an abstract semantics since they do not make explicit how data flows in functional programs, i.e., how inputs flow from function call sites to the function definition and how the outputs flow back in the other direction. To the best of our knowledge, the data flow semantics introduced in this work is the first concrete semantics where the notion of data flow is made explicit and the concrete transformer only makes local changes to the global execution map, enabling abstractions that naturally lead to structural typing rules.

Types as abstract interpretations. The formal connection between types and abstract interpretation is studied by Cousot [15]. His paper shows how to construct standard polymorphic type systems via a sequence of abstractions starting from a concrete denotational call-by-value semantics. Monsuez [62] similarly shows how to use abstract interpretation to design polymorphic type systems for call-by-name semantics and model advanced type systems such as System F [60, 61, 63, 64]. Techniques presented in [32, 31] use the abstract interpretation view of types

to design new type inference algorithms for ML-like languages by incorporating widening operators that infer more precise types for recursive functions. Unlike these works that focus on modular type inference, this thesis is instead focused on refinement type systems that perform a whole program analysis, an example of which are Liquid types. The work in [42] uses abstract interpretation techniques to develop a type system for JavaScript that is able to guarantee absence of common errors such as invoking a non-function value or accessing an undefined property. Further, the work in [29] uses Galois connections to relate gradual types with abstract static types to build a gradual type system from a static one. Harper [37] introduces a framework for constructing dependent type systems from operational semantics based on the PER model of types. Although this work does not build on abstract interpretation, it relies on the idea that types overapproximate program behaviors and that they can be derived using a suitable notion of abstraction.

Chapter 4

Type Error Localization

The contents of this chapter are previously published in a slightly modified form in [72, 73]. Proofs for all the theorems, lemmas, and corollaries stated in this chapter can be found in Appendix B, unless already presented in the chapter.

4.1 Overview

In this section, we provide an overview of our approach to the problem of type error localization introduced in § 1 and explain it through several illustrative examples.

The high-level execution flow of our constraint-based type error localization is shown in Figure 4.1. The framework can be viewed as a compiler plug-in. When type inference takes place, our algorithm starts by generating typing assertions for the given input program. The constraint generation incorporates a compiler-specific criterion for ranking type error sources by appropriately assigning weights to the assertions. After the constraint generation finishes, the produced annotated assertions, constituting a typing constraint, are passed to a weighted MaxSMT solver. If the input program has a type error, the generated constraint is unsatis-

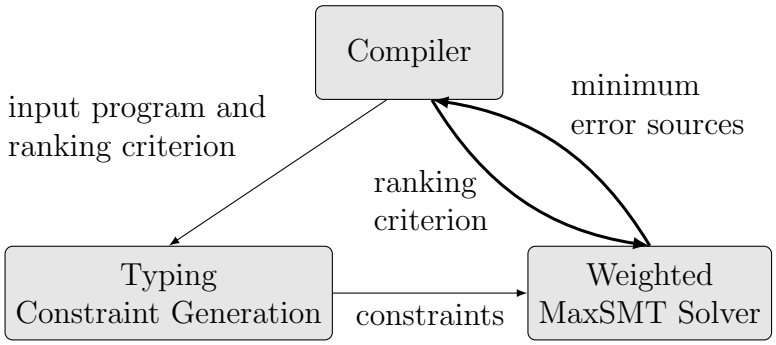

Figure 4.1: High-level overview of constraint-based type error localization. Thick arrows represent a looping interaction between a compiler and the SMT solver.

fiable. In this case, the MaxSMT solver finds all error sources that are minimum subject to the specified ranking criterion. The compiler can then iteratively interact with the solver to further rank and filter the error sources to generate an appropriate error message. This way, our framework can support interaction with the programmer. In particular, the compiler can take feedback from the programmer and pass it to the solver to guide the search for the error source the programmer needs. In the following, we describe the actual algorithm in more detail and highlight the main features of our approach.

4.1.1 Minimum Error Sources

First, let us make the notion of minimum error source more precise. An error source is a set of program locations that need to be fixed so that the erroneous program becomes well typed. Usually, not all error sources are equally likely to cause the error, and the compiler might prefer some error sources over others. In our framework, the compiler provides a criterion for ranking error sources by assigning weights to program locations. Our algorithm then finds the error sources with minimum cumulative weight. As an example, consider the following OCaml

program:

```
1 let f x = print_int x in
2 let g x = x + 1 in
3 let x = "hi" in
4   f x;
5   g x
```

Note that the program cannot be typed since the functions `f` and `g` expect an argument of type `int`, but both are applied to `x` whose type is `string`. The program has several possible error sources. One error source is the declaration of `x`. Namely, changing the declaration of `x` to, say, an integer constant would make the program well typed. Another error source is given by the two function applications of `f` and `g`. For example, replacing both `f` and `g` by `print_string` would yield another well-typed program.

Now consider a ranking criterion that assigns each program location equal weight 1. Then the cumulative weight of the first error source is 1, while the weight of the second error source is 2. Our algorithm would therefore report the first error source and discard the second one. This simple ranking criterion thus minimizes the number of program locations that need to be modified so that the program becomes well typed. Later in this chapter, we discuss more elaborate ranking criteria that are useful for type error localization.

4.1.2 Reduction to MaxSMT

Next, we show how the reduction to weighted MaxSMT works through another example. Consider the following OCaml program:

```
let x = "hi" in not x
```

Clearly, the program is not well typed as the operation `not` on Booleans is applied to a variable `x` of type `string`.

Our constraint generation procedure takes the program and generates a set of typing assertions using the OCaml type inference rules. This set of assertions constitutes a typing constraint. A formal description of the constraint generation can be found in § 4.3.2. For our example program, the constraint generation produces the following set of assertions:

$$\alpha_{not} = \mathsf{fun}(\mathsf{bool}, \mathsf{bool}) \qquad \text{[Def. of } \texttt{not}\text{]} \qquad (4.1)$$

$$\alpha_{app} = \mathsf{fun}(\alpha_i, \alpha_o) \qquad \texttt{not x} \qquad (4.2)$$

$$\alpha_{app} = \alpha_{not} \qquad \texttt{not} \qquad (4.3)$$

$$\alpha_i = \alpha_x \qquad \texttt{x} \qquad (4.4)$$

$$\alpha_x = \mathsf{string} \qquad \texttt{x = "hi"} \qquad (4.5)$$

The constraint encoding preserves the mapping between assertions and associated program locations. That is, each assertion comes from a particular program expression shown to the right of the assertion. The assertion (4.1) is generated from the definition of the function `not` in OCaml's standard library [55]. It specifies the type α_{not} of `not` as a function type from bool to bool. The assertions (4.2) to (4.4) specify the types of all subexpressions of the expression `not x` in the program. Finally, the assertion (4.5) specifies the type of `x`, according to its declaration.

The generated typing constraint is interpreted in the theory of inductive data types, which is supported by many SMT solvers [5, 21, 23]. In this theory, the terms α_{app}, α_i, α_o, α_{not}, and α_x stand for type variables, while the terms bool

and string are type constants. The function fun is a type constructor that maps an argument type α_i and a result type α_o to a function type from α_i to α_o. The theory interprets fun as an injective constructor. Hence, the assertions (4.1) to (4.3) together imply $\alpha_i = \mathsf{bool}$. Type constants such as bool and string are interpreted as pairwise distinct values. The assertions (4.4) and (4.5) imply $\alpha_i = \mathsf{string}$, so we conclude $\mathsf{bool} = \mathsf{string}$ together with the assertions (4.1) to (4.3) – a contradiction. Consequently, the generated typing constraint is unsatisfiable, confirming there is a type error.

As in the previous example, we first assume a ranking criterion that assigns each program location equal weight 1. The problem of finding the minimum error sources in the program is then encoded into a weighted MaxSMT problem by assigning weight 1 to each assertion in the generated constraint. The weighted MaxSMT solver computes all subsets of assertions in the typing constraint that are satisfiable and have maximum cumulative weight. Thus, the complements of those sets encode the minimum error sources.

For the running example, there are altogether five complement sets of maximum satisfiable subsets, each consisting of a single assertion (4.1)-(4.5). Removing any single assertion leaves the remaining assertions satisfiable. In other words, by fixing the program expression associated with the removed assertion, the input program becomes well typed. For example, consider the assertion (4.3). Removal of that assertion can be interpreted as using a different function instead of `not` in the application `not x`. Removal of the assertion (4.1) can be seen as changing the internals of `not` so that it applies to string values. Hence, the meaning associated with removing a certain assertion can be utilized for suggesting possible error fixes.

4.1.3 Ranking Criteria

Not all error sources are equally useful for fixing the type error. In the previous example, it is unlikely that the true error source is located in the implementation of the function `not`, since it is part of the standard library. Compilers might want to eliminate such error sources from consideration. Similarly, the error source corresponding to the removal of assertion (4.2) implies that the programmer should use some other expression than `not x` in the program. This error source seems less specific than the remaining error sources, so compilers might want to rank it as less likely. In general, compilers might want to rank error sources by some definition of usefulness. In our framework, such rankings are encoded by assigning appropriate weights to program locations, and hence assertions in the generated typing constraint.

Hard Assertions. One way of incorporating ranking criteria is to specify that certain assertions must hold, no matter what. Such assertions are commonly referred to as *hard assertions*. Assertions that the MaxSMT solver is allowed to remove from the assertion set are called *soft*. The compiler may, for instance, annotate all assertions as hard that come from the definitions of functions provided in the standard library. This would encode that all type error sources must be located in user-defined functions. If we apply this criterion to our previous example, we annotate the assertion (4.1) as hard. This eliminates the error source implying that the implementation of `not` must be modified. Other assertions that might be considered hard are assertions that come from user-provided type annotations in the program.

Weighted Clauses. Our approach supports more sophisticated ranking criteria than distinguishing only between hard and soft assertions. Such criteria are encoded by assigning weights to assertions. Going back to our running example, compilers might favor error sources consisting of smaller program expressions. For instance, each assertion can be assigned a weight equal to the size of the corresponding expression in the abstract syntax tree. This way, `not x` in our previous example is not reported, as desired.

To demonstrate the power of weighted assertions, we consider another example from the student benchmarks in [54], which served as the motivating example for the type error localization technique presented in [97]:

```
1  let f(lst:move list):
2      (float*float) list = ...
3    let rec loop lst x y dir acc =
4      if lst = [] then
5        acc
6      else
7        print_string "foo"
8    in
9    List.rev
10     (loop lst 0.0 0.0 0.0 [(0.0,0.0)])
```

The standard OCaml compiler reports a type error in the shaded expression on line 10. However, the actual error cause lies in the misuse of the `print_string` function on line 7, as reported by the authors of [97]. The technique of Zhang and Myers [97] correctly reports the expression on line 7 as the most likely error cause. With the ranking criteria that we introduced so far, our algorithm generates

two error sources that both have minimum weight. These error sources can be interpreted as follows:

1. Replace the function `print_string` on line 7.

2. Replace the function `loop` on line 10.

However, the second error source is not likely the actual cause of the error: using some other function than `loop` on line 10 would mean that the `loop` function is not used at all in the program. In fact, the OCaml compiler would generate a warning for the fixed program.

The compiler can thus analyze the produced error sources and find those sources that correspond to the removal of entire functions from the program. Assertions associated with such error sources can then be set as hard. In the running example, the solver will then produce just a single error source, indicating the actual error cause.

This additional ranking criterion can also be encoded directly in the typing constraint without a second round of interaction with the solver. Suppose the program contains a user-defined function `f` whose type is described by a type variable α_f. Suppose further that `f` is used n times in the program and for i, $1 \leq i \leq n$, let the type variable α_i indicate the type imposed on `f` by the context of the i-th usage of `f`. Then we can add the following additional hard assertion to the typing constraint: $\alpha_1 = \alpha_f \vee \ldots \vee \alpha_n = \alpha_f$. This assertion expresses that `f` needs to be used at least once, effectively encoding the additional criterion.

4.2 Problem

We now formally define the problem of computing minimum error sources for a given ranking criterion.

Language and Semantics. For the purposes of type error localization, we assume in this chapter that constants of our language besides Booleans also include integers $n \in \mathit{Int}$ and a constant hole $\bot$. Please note that the constant $\bot$ here plays a different role than the same symbol in Chapter 3. We explain the role of holes in more detail below. We are also only interested in non-recursive lambda abstractions $\lambda x.e$. This simplification is introduced for presentation purposes without loss of generality. We name the resulting language $\lambda^{\bot}$. We also assume standard operational semantics for $\lambda^{\bot}$.

As for other Hindley-Milner type systems, type inference is decidable for $\lambda^{\bot}$. Expressions therefore do not require explicit type annotations. Typing judgments take the form $\Gamma \vdash e : \tau$. We say that a program $p \in \lambda^{\bot}$ is *well typed* iff there exists a type σ such that $\emptyset \vdash p : \sigma$.

The actual typing rules are shown in Figure 4.2. The rules are standard, with the exception of the rule [Hole], which we describe in more detail. The value $\bot$ has the polytype $\forall\alpha.\alpha$. Therefore, the typing rule [Hole] assigns a fresh unconstrained type variable to each usage of $\bot$. Intuitively, $\bot$ is a placeholder for another program expression. It can be safely used in any context without causing a type error. Changes to hole expressions in an ill-typed program cannot make the program well typed[1].

[1] A similar concept was previously used in [54] where a hole in an OCaml program is represented by an expression that raises an exception.

$$\frac{x : \forall\vec{\alpha}.\tau \in \Gamma \quad \vec{\beta}\ \text{new}}{\Gamma \vdash x : \tau[\vec{\beta}/\vec{\alpha}]}\ [\textsc{Var}] \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ [\textsc{App}]$$

$$\frac{b \in \mathit{Bool}}{\Gamma \vdash b : \mathsf{bool}}\ [\textsc{Bool}] \qquad \frac{\Gamma.x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ [\textsc{Abs}]$$

$$\frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1\,?\,e_2 : e_3 : \tau}\ [\textsc{Cond}] \qquad \frac{n \in \mathit{Int}}{\Gamma \vdash n : \mathsf{int}}\ [\textsc{Int}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma.x : \forall\vec{\alpha}.\tau_1 \vdash e_2 : \tau_2 \quad \vec{\alpha} = \mathit{fv}(\tau_1) \setminus \mathit{fv}(\Gamma)}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : \tau_2}\ [\textsc{Let}]$$

$$\frac{\alpha\ \text{new}}{\Gamma \vdash \bot : \alpha}\ [\textsc{Hole}]$$

Figure 4.2: Typing rules for $\lambda^{\bot}$

Minimum Error Sources. Without loss of generality, we assume that locations of expressions are implemented as sequences of natural numbers. That is, we call a path $\ell \in \mathbb{N}^*$ in the abstract syntax tree representation of e a *location* of e. Now, let *mask* be the function that maps an expression e and a location $\ell \in \mathit{Loc}(e)$ to the expression obtained from e by replacing $e(\ell)$ with $\bot$. We extend *mask* to sets of locations in the expected way.

Definition 1 (Error source)**.** *Let p be a program. A set of locations $L \subseteq \mathit{Loc}(p)$ is an* error source *of p if*

1. *$\mathit{mask}(p, L)$ is well typed*
2. *for all strict subsets L' of L, $\mathit{mask}(p, L')$ is not well typed.*

A ranking criterion allows the compiler to favor error sources of particular interest by assigning appropriate weights to locations. Formally, a *ranking criterion*

is a function R that maps a program p to a partial function $R(p) : Loc(p) \rightharpoonup \mathbb{N}_+$. The locations in $Loc(p) \backslash \mathsf{dom}(R(p))$ are considered *hard* locations, i.e., R disregards these locations as causes of type errors. We extend $R(p)$ to a set of locations $L \subseteq Loc(p)$ by defining

$$R(p)(L) = \sum_{\ell \in \mathsf{dom}(R(p)) \cap L} R(p)(\ell) \ .$$

Minimum error sources are error sources that minimize the given ranking criterion.

Definition 2 (Minimum error source)**.** *Let R be a ranking criterion and p a program. An error source $L \subseteq Loc(p)$ is called* minimum error source *of p subject to R if for all other error sources L' of p, $R(p)(L) \leq R(p)(L')$.*

We are interested in the problem of finding a minimum error source for a given program p subject to a given ranking criterion R, respectively, finding all such minimum error sources.

4.3 Algorithm

In this section, we present our algorithm for computing the minimum error sources. More precisely, we show how the problem of finding minimum type error sources can be reduced to the weighted MaxSMT problem that can in turn be solved off-the-shelf using automated theorem provers. We first formally define the weighted MaxSMT problem and then explain the actual reduction.

Weighted MaxSMT. The MaxSAT problem takes as input a finite set of propositional clauses $\mathcal{C}$ and finds an assignment that maximizes the number of clauses K that are simultaneously satisfied [56]. MaxSAT can alternatively be viewed as

finding the largest subset $\mathcal{C}'$ of clauses $\mathcal{C}$ such that $\mathcal{C}'$ is satisfiable and $\mathcal{C}'$ is a maximum satisfiable subset, $|\mathcal{C}'| = K$. Partial MaxSAT partitions $\mathcal{C}$ into *hard* and *soft* clauses. The hard clauses $\mathcal{C}_H$ are assumed to hold and the goal is to find a maximizing subset $\mathcal{C}'$ of the soft clauses such that $\mathcal{C}' \cup \mathcal{C}_H$ is satisfiable. Weighted Partial MaxSAT, for simplicity referred to as only weighted MaxSAT, adds an integer weight $w_i = w(C_i)$ to each soft clause C_i and asks for a satisfiable subset $\mathcal{C}'$ that maximizes the weighted score:

$$\sum_{C_i \in \mathcal{C}'} w_i \text{ subject to } \mathcal{C}_H \cup \mathcal{C}' \text{ is satisfiable.} \tag{4.6}$$

The MaxSMT problem generalizes MaxSAT from working over propositional clauses to a set of assertion formulas $\mathcal{A}$ where each assertion belongs to a fixed first-order theory $\mathcal{T}$. Most concepts directly generalize from MaxSAT to MaxSMT: satisfiability is replaced by Satisfiability Modulo Theories [6], partial MaxSMT has hard and soft assertions ($\mathcal{A}_H$ and $\mathcal{A}_S$), weighted partial MaxSMT assigns an integer weight to each soft assertion. We represent weighted MaxSMT instances as tuples $(w, \mathcal{A}_H, \mathcal{A}_S)$ where w is the weight function assigning the weights to the soft assertions.

Theory of Inductive Data Types. Our reduction to the weighted MaxSMT problem generates a typing constraint from the given input program. This constraint is satisfiable iff the input program is well typed. The constraint is then passed to the MaxSMT solver, which computes the minimum error sources. The generated typing constraint hence needs to be expressed in terms of assertions that are interpreted in an appropriate first order theory. We use the theory of inductive data types [7] for this purpose. In this theory, we can define our own inductive

data types and then state equality constraints over the terms of that data type. For our encoding, we define an inductive data type Types that represents the set of all ground monotypes of $\lambda^{\perp}$:

$$t \in \mathsf{Types} ::= \mathsf{int} \mid \mathsf{bool} \mid \mathsf{fun}(t, t)$$

Here, the term constructor fun is used to encode the ground function types.

The terms in Types are interpreted in the models of the theory of inductive data types. A model of this theory is a first-order structure that interprets the type constructors such that the following axioms are satisfied:

$$\mathsf{int} \neq \mathsf{bool}$$

$$\forall \alpha, \beta \in \mathsf{Types}.\ \mathsf{fun}(\alpha, \beta) \neq \mathsf{int} \wedge \mathsf{fun}(\alpha, \beta) \neq \mathsf{bool}$$

$$\forall \alpha, \beta, \gamma, \delta \in \mathsf{Types}.\ \mathsf{fun}(\alpha, \beta) = \mathsf{fun}(\gamma, \delta) \Rightarrow \alpha = \gamma \wedge \beta = \delta$$

That is, the term constructor fun must be interpreted by an injective function, and the interpretation of the terms int and bool is distinct from the interpretation of all other terms. Hence, the axioms exactly encode the equality relation on ground monotypes of $\lambda^{\perp}$. For the type systems of actual languages such as OCaml, we extend Types with additional type constructors, e.g., to encode user-defined algebraic data types in OCaml programs.

Overview of the Reduction. Next, we describe the actual reduction to weighted MaxSMT. In the following, let p be the program under consideration and let R be the ranking criterion of interest.

At the heart of our reduction is a constraint generation procedure that traverses p and generates a set of assertions in our theory of Types. These assertions encode

the typing rules shown in Figure 4.2. The constraint generation procedure can produce multiple assertions associated with a single location ℓ of p. Suppose R considers a location ℓ to be soft. That is, while searching for minimum error sources the solver must consider two possibilities. If ℓ is in the minimum error source, then none of the assertions generated from the expression $p(\ell)$ need to be satisfied. If on the other hand ℓ is not in the minimum error source, then the type of the expression $p(\ell)$ must be consistent with its context. That is, all the assertions directly associated with ℓ must be satisfied simultaneously. However, some of the assertions generated from subexpressions of $p(\ell)$ may be dropped because the corresponding locations may still be in the minimum error source. Thus, to properly encode the problem of finding minimum error sources, we need to link the assertions of location ℓ together so that the solver considers them as a single logical unit. Also, we need to capture the relationship between the locations according to the structure of p. For this purpose, we associate a propositional variable T_ℓ with every location ℓ. By setting T_ℓ to true, the solver decides that T_ℓ is not in the minimum error source. That is, each assertion associated with a location ℓ_o takes the following form:

$$T_{\ell_n} \Rightarrow \cdots \Rightarrow T_{\ell_1} \Rightarrow T_{\ell_0} \Rightarrow t_1 = t_2 \tag{4.7}$$

Here, $\ell_1, \ldots, \ell_n$ are the locations that are reached along the path to ℓ_0 in p (i.e., all proper prefixes of ℓ_0). The terms t_1 and t_2 are terms in our theory of Types. These terms may include logical variables that need to be unified by the MaxSMT solver. Note that the assertion (4.7) ensures that if any of the T_{ℓ_i} is set to false (i.e., ℓ_i is in the minimum error source), then all the assertions that depend on ℓ_0 are immediately satisfied. In order to simplify the presentation in § 4.1, we have

omitted the additional propositional variables in our discussion of the examples.

The assertions that encode the typing rules are considered hard. To reduce the problem of finding minimum error sources to weighted MaxSMT, we add to these hard assertions an additional set of clauses, each of which consists of one of the propositional variables T_ℓ. For each ℓ, if $R(p)(\ell)$ is defined, the clause T_ℓ is soft with associated weight $R(p)(\ell)$. Otherwise, T_ℓ is hard. Each solution to this weighted MaxSMT instance then yields a minimum error source by taking all locations ℓ whose clause T_ℓ is not in the solution set.

We now explain these steps in more detail, starting with the generation of the assertions that encode the typing rules.

4.3.1 Assertion Generation

We build on existing work for constrained-based type checking [87, 1, 71] and adapt it for our purposes.

We formalize the assertion generation in terms of a constraint typing relation $\mathcal{A}, \Gamma \vdash p : \alpha$. Here, $\mathcal{A}$ is a set of typing assertions of the form (4.7). The rules that define the constraint typing relation are given in Figure 4.3. They are defined recursively on the structure of expressions. To relate the generated typing assertions to p's locations, we assume that every expression is annotated with its location ℓ in p. Note that for a set of assertions $\mathcal{A}$, we write $T_\ell \Rightarrow \mathcal{A}$ to mean $\{\, T_\ell \Rightarrow A \mid A \in \mathcal{A} \,\}$.

We focus on the rules [A-Let] and [A-Var] as they are the most complex ones. The [A-Let] rule handles let bindings $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ by first computing a set of typing assertions $\mathcal{A}_1$ for e_1. The assertions in $\mathcal{A}_1$ encode all typing assertions imposed by e_1 under the current environment Γ. In particular, they capture the

$$\frac{\mathcal{A}, \Gamma.x : \alpha \vdash e : \beta \quad \gamma \text{ new}}{T_\ell \Rightarrow (\{\gamma = \mathsf{fun}(\alpha, \beta)\} \cup \mathcal{A}), \Gamma \vdash (\lambda x.e)^\ell : \gamma} \text{ [A-Abs]}$$

$$\frac{\mathcal{A}_1, \Gamma \vdash e_1 : \alpha \quad \mathcal{A}_2, \Gamma \vdash e_2 : \beta \quad \gamma \text{ new}}{T_\ell \Rightarrow (\{\alpha = \mathsf{fun}(\beta, \gamma)\} \cup \mathcal{A}_1 \cup \mathcal{A}_2), \Gamma \vdash (e_1\ e_2)^\ell : \gamma} \text{ [A-App]}$$

$$\frac{\begin{array}{c}\mathcal{A}_1, \Gamma \vdash e_1 : \alpha \quad \mathcal{A}_2, \Gamma \vdash e_2 : \beta \quad \mathcal{A}_3, \Gamma \vdash e_3 : \gamma \quad \delta \text{ new} \\ \mathcal{A} = \{(T_{\ell_1} \Rightarrow \alpha = \mathsf{bool}), (T_{\ell_2} \Rightarrow \beta = \delta), (T_{\ell_3} \Rightarrow \gamma = \delta)\} \cup \mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3\end{array}}{T_\ell \Rightarrow \mathcal{A}, \Gamma \vdash (e_1^{\ell_1}\ ?\ e_2^{\ell_2} : e_3^{\ell_3})^\ell : \delta} \text{ [A-Cond]}$$

$$\frac{\alpha \text{ new}}{\emptyset, \Gamma \vdash \bot : \alpha} \text{ [A-Hole]} \qquad \frac{x : \forall \vec{\alpha}.(\mathcal{A} \Rightarrow \alpha) \in \Gamma \quad \vec{\beta}, \gamma \text{ new}}{T_\ell \Rightarrow (\{\gamma = \alpha[\vec{\beta}/\vec{\alpha}]\} \cup \mathcal{A}[\vec{\beta}/\vec{\alpha}]), \Gamma \vdash x^\ell : \gamma} \text{ [A-Var]}$$

$$\frac{b \in \mathit{Bool} \quad \alpha \text{ new}}{\{T_\ell \Rightarrow \alpha = \mathsf{bool}\}, \Gamma \vdash b^\ell : \alpha} \text{ [A-Bool]} \qquad \frac{n \in \mathit{Int} \quad \alpha \text{ new}}{\{T_\ell \Rightarrow \alpha = \mathsf{int}\}, \Gamma \vdash n^\ell : \alpha} \text{ [A-Int]}$$

$$\frac{\mathcal{A}_1, \Gamma \vdash e_1 : \alpha_1 \quad \mathcal{A}_2, \Gamma.x : \forall \vec{\alpha}.(\mathcal{A}_1 \Rightarrow \alpha_1) \vdash e_2 : \alpha_2 \quad \vec{\alpha} = \mathit{fv}(\alpha) \setminus \mathit{fv}(\Gamma) \quad \vec{\beta}, \gamma \text{ new}}{T_\ell \Rightarrow (\{\gamma = \alpha_2\} \cup \mathcal{A}_1[\vec{\beta}/\vec{\alpha}] \cup \mathcal{A}_2), \Gamma \vdash (\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2)^\ell : \gamma} \text{ [A-Let]}$$

Figure 4.3: Rules defining the constraint typing relation for $\lambda^\perp$

assertions on the type of e_1 itself, which is described by a fresh type variable α_1. To compute the typing assertions $\mathcal{A}_2$ for e_2, the variable x is added to the typing environment Γ. The type of x is described by a typing schema of the form:

$$\forall \vec{\alpha}.(\mathcal{A}_1 \Rightarrow \alpha_1)$$

The typing schema is needed to properly handle polymorphism. It remembers the set of assertions $\mathcal{A}_1$ along with the type variable α_1 that represents the type of e_1 and x. Note that the schema quantifies over all type variables $\vec{\alpha}$ that have been freshly introduced when analyzing e_1 (including α_1). Whenever x is used inside the body e_2 of the let binding, the [A-Var] rule instantiates $\mathcal{A}_1$ by replacing the

type variables $\vec{\alpha}$ with fresh type variables $\vec{\beta}$. The instantiated copy is then added to the other generated assertions. The fresh instances of $\mathcal{A}_1$ ensure that each usage of x in e_2 is consistent with the typing assertions imposed by e_1.

The following lemma states the correctness of the constraint typing relation.

Lemma 21. *Let p be a program, $L \subseteq Loc(p)$, $\mathcal{A}$ a set of typing assertions, and α a type variable such that $\mathcal{A}, \emptyset \vdash p : \alpha$. Then $mask(L, p)$ is well typed iff $\mathcal{A} \cup \{ T_\ell \mid \ell \notin L \}$ is satisfiable in the theory of* Types.

The proof of Lemma 21 closely follows that of [87, Lemma 2], modulo reasoning about the auxiliary propositional variables.

4.3.2 Reduction to Weighted Partial MaxSMT

To compute minimum error sources, we generate a weighted MaxSMT instance $I(p, R) = (w, \mathcal{A}_H, \mathcal{A}_S)$ as follows. Let $\mathcal{A}$ be a set of assertions such that $\mathcal{A}, \emptyset \vdash p : \alpha$ for some type variable α. Then define:

$$\mathcal{A}_H = \mathcal{A} \cup \{ T_\ell \mid \ell \notin \mathsf{dom}(R(p)) \}$$

$$\mathcal{A}_S = \{ T_\ell \mid \ell \in \mathsf{dom}(R(p)) \}$$

$$w(T_\ell) = R(p)(\ell), \text{ for all } T_\ell \in \mathcal{A}_S$$

Let $\mathcal{S}$ be a solution of $I(p, R)$. Then define $E(\mathcal{S}) = \{ \ell \in Loc(p) \mid T_\ell \notin \mathcal{S} \}$. The following theorem states the correctness of our reduction.

Theorem 5. *Let p be a program and R a ranking criterion. Then $L \subseteq Loc(p)$ is a minimum error source of p subject to R iff there exists a solution $\mathcal{S}$ of $I(p, R)$ such that $L = E(\mathcal{S})$.*

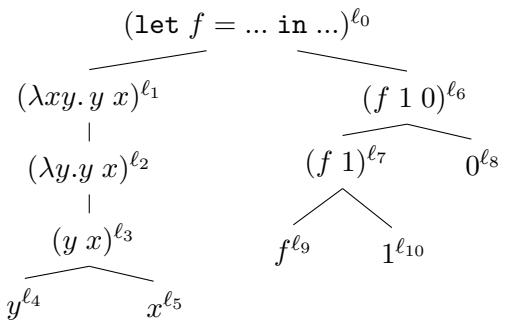

Figure 4.4: Labeled abstract syntax tree for the program p

We conclude the presentation of our algorithm with an example that demonstrates how our approach deals with polymorphic functions.

Example 4. *Let p be the following ill-typed program:*

$$\texttt{let}\ f = \lambda x\, y.\ y\ x\ \texttt{in}\ f\ 1\ 0$$

The most general type that can be inferred for f from its defining expression is

$$f : \forall \alpha\, \beta.\, \alpha \to (\alpha \to \beta) \to \beta\ \ .$$

However, the type of 0 *is* int *and not a function type. Hence, applying* 0 *as second argument to f in the body of the let violates the typing rules.*

Figure 4.4 shows the abstract syntax tree of the program p with each node labeled by its location. Applying the constraint generation rules from Figure 4.3 then yields the following set of assertions $\mathcal{A}$:

$$
\begin{aligned}
T_{\ell_0} \Rightarrow \{&\alpha_0 = \alpha_6,\ C_f(\alpha_1'', \alpha_2'', \alpha_3'', \alpha_4'', \alpha_5'', \beta_1'', \beta_2''),\\
&T_{\ell_6} \Rightarrow \{\alpha_7 = \mathsf{fun}(\alpha_8, \alpha_6),\\
&\quad T_{\ell_7} \Rightarrow \{\alpha_9 = \mathsf{fun}(\alpha_{10}, \alpha_7),\\
&\quad\quad T_{\ell_8} \Rightarrow \alpha_8 = \mathsf{int},\\
&\quad\quad T_{\ell_9} \Rightarrow \{\alpha_9 = \alpha_1'',\\
&\quad\quad\quad C_f(\alpha_1', \alpha_2', \alpha_3', \alpha_4', \alpha_5', \beta_1', \beta_2')\}\\
&\quad\quad \},\\
&\quad T_{\ell_{10}} \Rightarrow \alpha_{10} = \mathsf{int}\}\}
\end{aligned}
$$

where

$$
\begin{aligned}
&C_f(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \beta_1, \beta_2) \equiv\\
&\quad T_{\ell_1} \Rightarrow \{\ \alpha_1 = \mathsf{fun}(\beta_1, \alpha_2),\\
&\quad\quad T_{\ell_2} \Rightarrow \{\ \alpha_2 = \mathsf{fun}(\beta_2, \alpha_3),\\
&\quad\quad\quad T_{\ell_3} \Rightarrow \{\ \alpha_4 = \mathsf{fun}(\alpha_5, \alpha_3),\\
&\quad\quad\quad\quad T_{\ell_4} \Rightarrow \alpha_4 = \beta_2,\\
&\quad\quad\quad\quad T_{\ell_5} \Rightarrow \alpha_5 = \beta_1\}\}\}
\end{aligned}
$$

Note that we introduced a predicate C_f to serve as a shorthand for the instantiated assertions that are associated with the bound variable f.

The assertions in $\mathcal{A}$ are satisfiable in Types if one of the following location variables is assigned false: T_{ℓ_4}, T_{ℓ_8}, T_{ℓ_9}, or any other T_{ℓ_i} where ℓ_i is on the path to one of the locations ℓ_4, ℓ_8, or ℓ_9.

Now consider the ranking criterion R that defines ℓ_9 hard and all other locations ℓ_i soft with rank $R(p)(\ell_i) = n_i$, where n_i is the number of nodes in the subtree of the AST whose root ℓ_i identifies. Solving the weighted MaxSMT instance $I(p, R)$ yields two minimum error sources, each of which contains one of the locations ℓ_4 and ℓ_8.

4.4 Implementation and Evaluation

We have implemented our weighted MaxSMT based algorithm for computing minimum type error sources for the Caml subset of the OCaml language and evaluated it on the OCaml benchmark suite from [54]. The implementation of our algorithm was able to find a minimum error source for 98% of the benchmarks in a reasonable time using a typical ranking criterion.

4.4.1 Implementation

Our implementation bundles together the EasyOCaml [24] tool and the MaxSMT solver νZ [11, 10]. The νZ solver is available as a branch of the SMT solver Z3 [21]. We use EasyOCaml for generating typing constraints for OCaml programs. Once we convert the constraints to the weighted MaxSMT instances, we use Z3's weighted MaxRes [66] algorithm to compute a minimum error source.

Assertion Generation. EasyOCaml implements the type error localization approach described in [33] for a subset of the OCaml language. Similar to our approach, EasyOCaml uses constraint solving for error localization. We tapped into the corresponding code of EasyOCaml and modified it to produce the weighted MaxSMT instances from the input program. The implementation begins by running EasyOCaml on the input program. EasyOCaml produces typing assertions, which we annotate with the locations of the corresponding program expressions in the input source code. These assertions only encode the typing relation. In addition, we output the structure of the input program in terms of its abstract syntax tree edges where the nodes are the locations associated with the typing assertions. This information is sufficient to generate the final typing constraint as described

in § 4.3.2. In our evaluation, we used a fixed ranking criterion that is defined as follows: (1) all assertions that come from expressions in external libraries are set as hard; (2) all assertions that come from user-provided type annotations are set as hard; and (3) all remaining assertions have a weight equal to the size of the corresponding expression in the abstract syntax tree. We encode the generated assertions in an extension of the `SMT-LIB` 2 language [8] to handle the theory of inductive data types.

Solving the Weighted MaxSMT Instances. We compute the weighted partial MaxSMT solution for the encoded typing constraints by using Z3's weighted partial MaxSMT facilities. In particular, we configure the solver to use the MaxRes [66] algorithm for solving the weighed partial MaxSMT problem.

4.4.2 Evaluation

We now discuss in more details the effectiveness of minimum error sources as well as the efficiency of computing them using our implementation.

Quality of Computed Minimum Error Sources. In our first experiment, we assessed how good the produced minimum error sources are at pinpointing the true error source. To this end, we chose 20 programs from the benchmark suite at random, identified the true error source in each program, and compared it against the first minimum error source that was computed by our implementation. As mentioned earlier, we used the ranking criterion that favors error sources of smaller code size and that excludes possible errors sources coming from external functions and type annotations. To identify the true error source we used additional infor-

mation that was provided by the benchmark suite. Namely, the benchmark suite does not consist of individual programs. Instead, it consists of sequences of modified versions of programs. Each such program sequence was recorded in a single session of interactions between a student and the OCaml compiler. By comparing subsequent program versions within one sequence, we can identify the changes that the student made to fix the reported type errors and thereby infer the true error sources.

We classified the result of the comparison as either "hit", "close", or "miss". Here, "hit" means that the computed minimum error source exactly matches the true error source, and "miss" means that there is no relationship between the two. A "close" result means that the locations in the computed minimum error source were close enough to the true error locations so that a programmer could easily understand and fix the type error. For example, if the minimum error source reported the function in a function application instead of the argument of the application (or vice versa), then we considered the result to be close to the true error source.

We then repeated the same experiment but this time using OCaml's type checker instead of our implementation. For each program, we recorded the result of the two experiments. Figure 4.5 shows the number of obtained result pairs, aggregated over the 20 benchmark programs. As can be seen, on the randomly chosen programs, our approach identifies the true error source more often than the OCaml type checker, even though we were using a rather simplistic ranking criterion. Specifically, our approach missed the true error source in only three programs, whereas OCaml did so in six programs. Despite the subjective nature of true error sources, we believe that this experiment shows the potential of our

AST size criterion	OCaml	# of outcomes
hit	miss	3
hit	close	2
hit	hit	4
close	miss	1
close	close	6
close	hit	1
miss	miss	2
miss	close	1
miss	hit	0

Figure 4.5: Quality of AST ranking criterion compared to OCaml's type checker in pinpointing the true error source in 20 benchmark programs

approach to providing helpful error reports to the user.

Computing minimum error sources. The last experiment we performed had the goal of measuring the time spent computing the minimum error sources. We broke the student benchmark suite consisting of 356 programs into 8 groups according to their size in the number of lines of code. The first group includes programs consisting of 0 and 50 lines of code, the second group includes programs of size 50 to 100, and so on as shown in the first column of Figure 4.6. The numbers in parenthesis indicate the number of programs in each group. The figure shows statistics about the execution time for computing a minimum error source. We show the minimum, average, and maximum execution times for each program family. As it can be seen, the execution times are reasonably small. However, for some programs the framework spends around a dozen seconds computing the solution.

The slowdown in our implementation mostly occurs in the cases where a huge number of constraints has been generated. This is confirmed by statistics shown in Figure 4.7. As it can be seen, even for small to moderate programs the number of

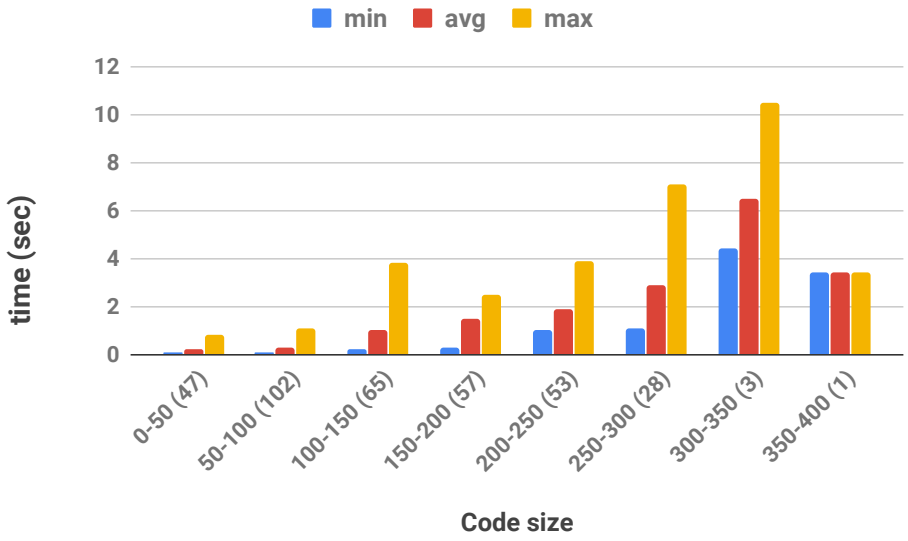

Figure 4.6: Maximum, average, and minimum execution times for computing a minimum error source

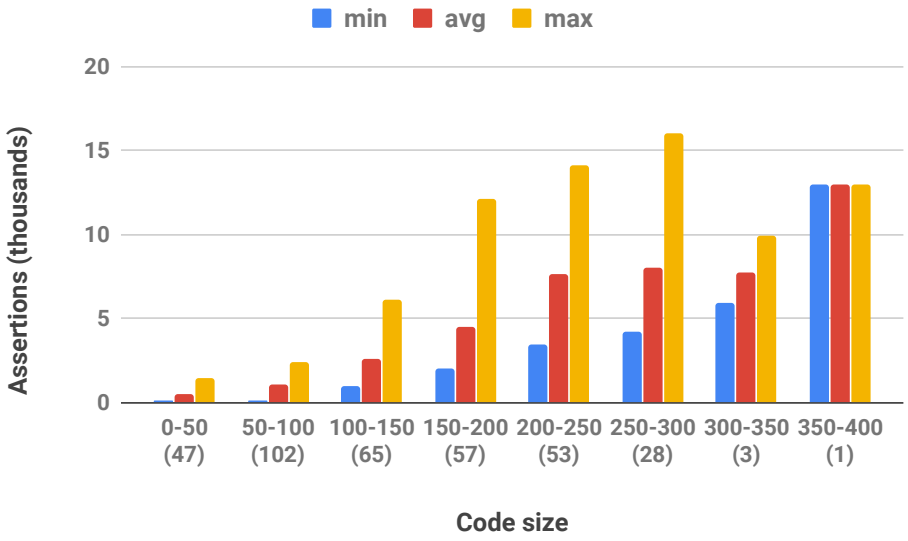

Figure 4.7: Maximum, average, and minimum number of generated assertions (in thousands) for computing a minimum error source

generated assertions can reach more than ten thousand. In general, the number of assertions does not grow linearly with the size of the program. In fact, our tool can

generate an exponential number of assertions for certain programs. As we discuss next, this phenomena is tied to an inherent property of polymorphic typing.

More experiments and accompanying statistics can be found in the original paper [72].

4.5 Taming The Exponential Explosion

As observed in the previous section, the number of typing constraints can grow exponentially in the size of the program. This is because the constraints associated with polymorphic functions are duplicated each time these functions are used, as illustrated in the discussion of Example 4.

Exponential Explosion Problem. Let us again briefly provide the intuition behind copying constraints for polymorphic functions. Consider the typing constraints for the OCaml function `let f (a, b, c) = (a, c)` that simply takes a first and last element of a triple and returns them as a pair.

$$\alpha_f = \mathsf{fun}(\alpha_i, \alpha_o)$$

$$\alpha_i = \mathsf{triple}(\alpha_a, \alpha_b, \alpha_c)$$

$$\alpha_o = \mathsf{pair}(\alpha_a, \alpha_c)$$

The above constraints state that the type of `f`, represented by the type variable α_f, is a function type that accepts some triple and returns a pair whose constituent types are equal to the type of the first and last component of that triple. When a polymorphic function, such as `f`, is called in the program, the associated set of typing constraints needs to be *instantiated* and the new copy has to be added to the whole set of typing constraints. Instantiation of typing constraints involves

copying the constraints and replacing free type variables in the copy with fresh type variables. If the constraints of polymorphic function were not freshly instantiated for each usage of the function, the same type variable would be constrained by the context of each usage, potentially resulting in a spurious type error. Introducing a fresh copy of typing constraints for functions is a standard approach for dealing with polymorphism in purely constraint based type inference and checking [74].

The exponential explosion in the constraint size does not seem to be avoidable. The type inference problem for polymorphic type systems is known to be EXPTIME-complete [57, 49]. However, in practice, compilers successfully avoid the explosion by computing the principal type [20] of each polymorphic function, such as the one shown in Example 4, and then instantiating a fresh copy of this type for each usage. The resulting constraints are much smaller in practice. Since these constraints are equisatisfiable with the original constraints, the resulting algorithm is a decision procedure for the type checking problem [20]. Unfortunately, this technique cannot be applied immediately to the optimization problem of type error localization. If the minimum cost error source is located inside of a polymorphic function, then abstracting the constraints of that function by its principle type will hide this error source. Thus, this approach can yield incorrect results.

Iterative Expansion. In this section, we propose an improved reduction to the MaxSMT problem that abstracts polymorphic functions by principal types and still guarantees the optimum solution. Our reduction is based on the optimistic assumption that type error sources only involve few polymorphic functions, even for large programs. Our new reduction to MaxSMT relies on principal type abstraction that is done in such a way that all potential error sources involving the definition

of an abstracted function are represented by a single error source whose cost is smaller or equal to the cost of all these potential error sources. The algorithm then iteratively computes minimum error sources for abstracted constraints. If an error source involves a usage of a polymorphic function, the corresponding instantiations of the principal type of that function are expanded to the actual typing constraints. Usages of polymorphic functions that are exposed by the new constraints are expanded if they are relevant for the minimum error source in the next iteration. The algorithm eventually terminates when the computed minimum error source no longer involves any usages of abstracted polymorphic functions. Such error sources are guaranteed to have minimum cost for the fully expanded constraints, even if the final constraint is not yet fully expanded.

Our core technical contribution is a refinement of the typing relation introduced in § 4.3.1. The novelty of this new typing relation is the ability to specify a set of variable usage locations whose typing constraints are abstracted as the principal type of the variable. We then describe an algorithm that iteratively uses this typing relation to find a minimum error source while expanding only those principal type usages that are relevant for the minimum source.

4.5.1 Assertion Generation

For the rest of this section, we assume a fixed program $p \in \lambda^{\perp}$. We first formally introduce the notion of principal types.

Principal Types. Standard type inference implementations handle expressions of the form `let` $x = e_1$ `in` e_2 by computing the *principal* type of e_1, binding x to the principal type σ_p in the environment $\Gamma.x : \sigma_p$, and proceeding to perform type

$$\frac{\Pi.x : \alpha, \Gamma.x : \alpha \vdash_{\mathfrak{L}} e : \beta \mid \mathcal{A} \qquad \gamma \text{ new}}{\Pi, \Gamma \vdash_{\mathfrak{L}} (\lambda x.e)^{\ell} : \gamma \mid \{T_\ell \Rightarrow (\{\gamma = \mathsf{fun}(\alpha, \beta)\} \cup \mathcal{A})\}} \text{ [A-Abs]}$$

$$\frac{\Pi, \Gamma \vdash_{\mathfrak{L}} e_1 : \alpha \mid \mathcal{A}_1 \qquad \Pi, \Gamma \vdash_{\mathfrak{L}} e_2 : \beta \mid \mathcal{A}_2 \qquad \gamma \text{ new}}{\Pi, \Gamma \vdash_{\mathfrak{L}} (e_1\ e_2)^{\ell} : \gamma \mid \{T_\ell \Rightarrow (\{\alpha = \mathsf{fun}(\beta, \gamma)\} \cup \mathcal{A}_1 \cup \mathcal{A}_2)\}} \text{ [A-App]}$$

$$\frac{\begin{array}{l}\Pi, \Gamma \vdash_{\mathfrak{L}} e_1 : \alpha_1 \mid \mathcal{A}_1 \qquad \Pi, \Gamma \vdash_{\mathfrak{L}} e_2 : \alpha_2 \mid \mathcal{A}_2 \qquad \Pi, \Gamma \vdash_{\mathfrak{L}} e_3 : \alpha_3 \mid \mathcal{A}_3 \qquad \gamma \text{ new} \\ \mathcal{A}_4 = \{(T_{\ell_1} \Rightarrow \alpha_1 = \mathsf{bool}), (T_{\ell_2} \Rightarrow \alpha_2 = \gamma), (T_{\ell_3} \Rightarrow \alpha_3 = \gamma)\}\end{array}}{\Pi, \Gamma \vdash_{\mathfrak{L}} (e_1^{\ell_1}\ ?\ e_2^{\ell_2} : e_3^{\ell_3})^{\ell} : \gamma \mid \{T_\ell \Rightarrow (\mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3 \cup \mathcal{A}_4)\}} \text{ [A-Cond]}$$

$$\frac{\alpha \text{ new}}{\Pi, \Gamma \vdash_{\mathfrak{L}} \bot : \alpha \mid \emptyset} \text{ [A-Hole]}$$

$$\frac{b \in \mathit{Bool} \qquad \alpha \text{ new}}{\Pi, \Gamma \vdash_{\mathfrak{L}} b^{\ell} : \alpha \mid \{T_\ell \Rightarrow \alpha = \mathsf{bool}\}} \text{ [A-Bool]}$$

$$\frac{n \in \mathit{Int} \qquad \alpha \text{ new}}{\Pi, \Gamma \vdash_{\mathfrak{L}} n^{\ell} : \alpha \mid \{T_\ell \Rightarrow \alpha = \mathsf{int}\}} \text{ [A-Int]}$$

$$\frac{\ell \in \mathfrak{L} \qquad x : \forall \vec{\alpha}.(\mathcal{A} \Rrightarrow \alpha) \in \Gamma \qquad \vec{\beta}, \gamma \text{ new}}{\Pi, \Gamma \vdash_{\mathfrak{L}} x^{\ell} : \gamma \mid \{T_\ell \Rightarrow (\{\gamma = \alpha[\vec{\beta}/\vec{\alpha}]\} \cup \mathcal{A}[\vec{\beta}/\vec{\alpha}])\}} \text{ [A-Var-Exp]}$$

$$\frac{\ell \notin \mathfrak{L} \qquad x : \forall \vec{\alpha}.(\mathcal{A} \Rrightarrow \alpha) \in \Pi \qquad \vec{\beta}, \gamma \text{ new}}{\Pi, \Gamma \vdash_{\mathfrak{L}} x^{\ell} : \gamma \mid \{T_\ell \Rightarrow (\{\gamma = \alpha[\vec{\beta}/\vec{\alpha}]\} \cup \mathcal{A}[\vec{\beta}/\vec{\alpha}])\}} \text{ [A-Var-Prin]}$$

$$\frac{\begin{array}{l}\ell_1 \in \mathfrak{L} \\ \Pi, \Gamma \vdash_{\mathfrak{L}} e_1 : \alpha_1 \mid \mathcal{A}_1 \qquad \vec{\alpha} = \mathit{fv}(\mathcal{A}_1) \setminus \mathit{fv}(\Gamma) \qquad \tau_{exp} = \forall \vec{\alpha}.(\mathcal{A}_1 \Rrightarrow \alpha_1) \\ \Pi, \Gamma.x : \tau_{exp} \vdash_{\mathfrak{L}} e_2 : \alpha_2 \mid \mathcal{A}_2 \qquad \vec{\beta}, \gamma \text{ new}\end{array}}{\Pi, \Gamma \vdash_{\mathfrak{L}} (\mathtt{let}\ x = e_1^{\ell_1}\ \mathtt{in}\ e_2)^{\ell} : \gamma \mid \{T_\ell \Rightarrow (\{\gamma = \alpha_2\} \cup \mathcal{A}_1[\vec{\beta}/\vec{\alpha}] \cup \mathcal{A}_2)\}} \text{ [A-Let-Exp]}$$

$$\frac{\begin{array}{l}\ell_1 \notin \mathfrak{L} \\ \rho(\Pi, e_1) = \forall \vec{\delta}.\tau_p \qquad \alpha \text{ new} \qquad \tau_{prin} = \forall \alpha, \vec{\delta}.(\{P_{\ell_1} \Rightarrow \alpha = \tau_p\} \Rrightarrow \alpha) \\ \Pi, \Gamma \vdash_{\mathfrak{L}} e_1 : \alpha_1 \mid \mathcal{A}_1 \qquad \vec{\alpha} = \mathit{fv}(\mathcal{A}_1) \setminus \mathit{fv}(\Gamma) \qquad \tau_{exp} = \forall \vec{\alpha}.(\mathcal{A}_1 \Rrightarrow \alpha_1) \\ \Pi.x : \tau_{prin}, \Gamma.x : \tau_{exp} \vdash_{\mathfrak{L}} e_2 : \alpha_2 \mid \mathcal{A}_2 \qquad \vec{\beta}, \gamma \text{ new}\end{array}}{\Pi, \Gamma \vdash_{\mathfrak{L}} (\mathtt{let}\ x = e_1^{\ell_1}\ \mathtt{in}\ e_2)^{\ell} : \gamma \mid \{T_\ell \Rightarrow (\{\gamma = \alpha_2\} \cup \mathcal{A}_1[\vec{\beta}/\vec{\alpha}] \cup \mathcal{A}_2)\}} \text{ [A-Let-Prin]}$$

Figure 4.8: Rules defining the refined constraint typing relation for $\lambda^{\perp}$

inference on e_1 [20]. Given an environment Γ, the type σ_p is the principal type for e if $\Gamma \vdash e : \sigma_p$ and for any other σ such that $\Gamma \vdash e : \sigma$ then σ is a generic instance of σ_p. Note that a principal type is unique, subject to e and Γ, up to the renaming of bound type variables in σ_p. We define ρ to be a partial function accepting an expression e and a typing environment Γ where $\rho(\Gamma, e)$ returns a principal type of e subject to Γ. If e is not typeable in Γ, then $(\Gamma, e) \notin \mathsf{dom}(\rho)$. We assume that the principal type function ρ is precomputed for p. We do not provide a detailed algorithms for computing ρ since it is well-known from the literature. For instance, the ρ function can be implemented using the classical W algorithm [20].

The main idea behind our improved algorithm is to iteratively discover which principal type usages must be expanded to compute a minimum error source. The technical core of the algorithm is a new typing relation that produces typing constraints subject to a set of locations where principal type usages must be expanded.

As before, we use $\mathcal{A}$ to denote a set of logical assertions in the signature of Types that represent typing constraints. Henceforth, when we refer to types we mean terms over Types. *Expanded locations* are a set of locations $\mathfrak{L}$ such that $\mathfrak{L} \subseteq Loc(p)$. Intuitively, this is a set of locations corresponding to usages of `let` variables x where the typing of x in the current iteration of the algorithm is expanded into the corresponding typing constraints. Those locations of usages of x that are not expanded will treat x using its principal type. We also introduce a set of locations whose usages must be expanded $\mathfrak{L}_0$. We will always assume $\mathfrak{L}_0 \subseteq \mathfrak{L}$. Formally, $\mathfrak{L}_0$ is the set of all program locations in p except the locations of well-typed `let` variables and their usages. This definition enforces that usages of variables that have no principal type are always expanded. In summary, $\mathfrak{L}_0 \subseteq \mathfrak{L} \subseteq Loc(p)$.

We define a typing relation $\vdash_{\mathfrak{L}}$ over $(\Pi, \Gamma, e, \alpha, \mathcal{A})$ which is parameterized by

$\mathfrak{L}$. The relation is given by judgments of the form:

$$\Pi, \Gamma \vdash_{\mathfrak{L}} e : \alpha \mid \mathcal{A}.$$

Intuitively, the relation holds iff expression e in p has type α under typing environment Γ if we solve the constraints $\mathcal{A}$ for α. (We make this statement formally precise later.) The relation depends on $\mathfrak{L}$, which controls whether a usage of a `let` variable is typed by the principal type of the `let` definition or the expanded typing constraints of that definition.

For technical reasons, the principal types are computed in tandem with the expanded typing constraints. This is because both the expanded constraints and the principal types may refer to type variables that are bound in the environment, and we have to ensure that both agree on these variables. We therefore keep track of two separate typing environments:

- the environment Π binds `let` variables to the principal types of their defining expressions if the principal type exists with respect to Π, and
- the typing environment Γ binds `let` variables to their expanded typing constraints (modulo $\mathfrak{L}$).

The typing relation ensures that the two environments are kept synchronized. To properly handle polymorphism, the bindings in Γ are represented by typing schemas:

$$x : \forall \vec{\alpha}.(\mathcal{A} \Rrightarrow \alpha)$$

The schema states that x has type α if we solve the typing constraints $\mathcal{A}$ for the variables $\vec{\alpha}$. To simplify the presentation, we also represent bindings in Π as type schemas. Note that we can represent an arbitrary type t by the schema

$\forall\alpha.(\{\alpha = t\} \Rrightarrow \alpha)$ where $\alpha \notin fv(t)$. The symbol $\Rrightarrow$ is used here to suggest, but keep syntactically separate, the notion of logical implication $\Rightarrow$ that is implicitly present in the schema.

The typing relation $\Pi, \Gamma \vdash_{\mathfrak{L}} e : \alpha \mid \mathcal{A}$ is defined in Figure 4.8. It can be seen as a constraint generation procedure that goes over an expression e at location ℓ and generates a set of typing constraints $\mathcal{A}$. Again, we associate with each location ℓ a propositional variable T_ℓ for the same purpose of identifying minimum error sources as defined in § 4.2.

The rules A-Let-Prin and A-Let-Exp govern the computation and binding of typing constraints and principal types for `let` definitions $(\texttt{let}\ x = e_1^{\ell_1}\ \texttt{in}\ e_2)^{\ell}$. If e_1 has no principal type under the current environment Π, then $\ell_1 \in \mathfrak{L}$ by the assumption that $\mathfrak{L}_0 \subseteq \mathfrak{L}$. Thus, when rule A-Let-Prin applies, $\rho(\Pi, e_1)$ is defined. The rule then binds x in Π to the principal type and binds x in Γ to the expanded typing constraints obtained from e_1.

The [A-Let-Prin] rule binds x in both Π and Γ as it is possible that in the current iteration some usages of x need to be typed with principal types and some with expanded constraints. For instance, our algorithm can expand usages of a function, say f, in the first iteration, and then expand all usages of, say g, in the next iteration. If g's defining expression in turn contains calls to f, those calls will be typed with principal types. This is done because there may exist a minimum error source that does not require that the calls to f in g are expanded.

After extending the typing environments, the rule recurses to compute the typing constraints for the body e_2 with the extended environments. Note that the rule introduces an auxiliary propositional variable P_{ℓ_1} that guards all the typing constraints of the principal type before x is bound in Π. This step is crucial for the

correctness of the algorithm. We refer to the variables as *principal type correctness variables*. That is, if P_{ℓ_1} is `true` then this means that the definition of the variable bound at ℓ_1 is not involved in the minimum error source and the principal type safely abstracts the associated unexpanded typing constraints.

The rule A-Let-Exp applies whenever $\ell_1 \in \mathfrak{L}$. The rule is almost identical to the A-Let-Prin rule, except that it does not bind x in Π to τ_{prin} (the principal type). This will have the effect that for all usages of x in e_2, the typing constraints for e_1 to which x is bound in Γ will always be instantiated. By the way the algorithm extends the set $\mathfrak{L}$, $\ell_1 \in \mathfrak{L}$ implies that $\ell_1 \in \mathfrak{L}_0$, i.e., the defining expression of x is ill-typed and does not have a principal type.

The A-Var-Prin rule instantiates the typing constraints of the principal type of a `let` variable x if x is bound in Π and the location of x is not marked to be expanded. Instantiation is done by substituting the type variables $\vec{\alpha}$ that are bound in the schema of the principle type with fresh type variables $\vec{\beta}$. The A-Var-Exp rule is again similar, except that it handles all usages of `let` variables that are marked for expansion, as well as all usages of variables that are bound in lambda abstractions.

The remaining rules are relatively straightforward. The rule A-Abs is noteworthy as it simultaneously binds the abstracted variable x to the same type variable α in both typing environments. This ensures that the two environments consistently refer to the same bound type variables when they are used in the subsequent constraint generation and principal type computation within e.

4.5.2 Reduction to Weighted Partial MaxSMT

The reduction to MaxSMT follows the reduction introduced in § 4.3.2. Additionally, the new reduction also must take care of assertions generated for usages of principal types.

Given a cost function R for program p and a set of locations $\mathfrak{L}$ where $\mathfrak{L}_0 \subseteq \mathfrak{L}$, we generate a WPMaxSMT instance $I(p, R, \mathfrak{L}) = (\mathcal{A}_H, \mathcal{A}_S, w)$ as follows. Let $\mathcal{A}_{p,\mathfrak{L}}$ be a set of constraints such that $\emptyset, \emptyset \vdash_{\mathfrak{L}} p : \alpha \mid \mathcal{A}_{p,\mathfrak{L}}$ for some type variable α. Then define

$$\begin{aligned} \mathcal{A}_H &= \mathcal{A}_{p,\mathfrak{L}} \cup \{\, T_\ell \mid \ell \notin \mathsf{dom}(R(p)) \,\} \cup \mathit{PDefs}(p) \\ \mathcal{A}_S &= \{\, T_\ell \mid \ell \in \mathsf{dom}(R(p)) \,\} \\ w(T_\ell) &= R(p)(\ell), \text{ for all } T_\ell \in \mathcal{A}_S \end{aligned}$$

The set of assertions $\mathit{PDefs}(p)$ contains the definitions for the principal type correctness variables P_ℓ. For a `let` variable x that is defined at some location ℓ, the variable P_ℓ is defined to be `true` iff

- each location variable $T_{\ell'}$ for a location ℓ' in the defining expression of x is `true`, and
- each principal type correctness variable $P_{\ell'}$ for a `let` variable that is defined at ℓ' and used in the defining expression of x is `true`.

Formally, $\mathit{PDefs}(p)$ defines the set of formulas

$$\begin{aligned} \mathit{PDefs}(p) &= \{\, \mathit{PDef}_\ell \mid \ell \in \mathsf{dom}(\mathit{Uloc}_p) \,\} \\ \mathit{PDef}_\ell &= \left(P_\ell \Leftrightarrow \bigwedge_{\ell' \in \mathit{Loc}(\ell)} T_{\ell'} \quad \wedge \bigwedge_{\ell' \in \mathit{Vloc}(\ell)} P_{\mathit{dloc}(\ell')} \right) \end{aligned}$$

Setting the P_ℓ to `false` thus captures all possible error sources that involve some of the locations in the defining expression of x, respectively, the defining expressions of other variables that x depends on. Recall that the propositional variable P_ℓ is used to guard all the instances of the principal types of x in $\mathcal{A}_{p,\mathfrak{L}}$. Thus, setting P_ℓ to `false` will make all usage locations of x well-typed that have not yet been expanded and are thus constrained by the principal type. By the way P_ℓ is defined, the cost of setting P_ℓ to `false` will be the minimum weight of all the location variables for the locations of x's definition and its dependencies. Thus, P_ℓ conservatively approximates all the potential minimum error sources that involve these locations.

We denote by SOLVE the procedure that given p, R, and $\mathfrak{L}$ returns some model M that is a solution of $I(p, R, \mathfrak{L})$. This procedure can be, for instance, implemented using the previously introduced algorithm for computing minimum error sources in § 4.3.

Lemma 22. SOLVE *is total.*

Lemma 22 follows from our assumption that R is defined for the root location ℓ_p of the program p. That is, $I(p, R, \mathfrak{L})$ always has some solution since $\mathcal{A}_H$ holds in any model M where $M \not\models T_{\ell_p}$.

Given a model $M = \textsc{Solve}(p, R, \mathfrak{L})$, we define L_M to be the set of locations excluded in M:

$$L_M = \{\, \ell \in \mathit{Loc}(p) \mid M \models \neg T_\ell \,\}.$$

4.5.3 Iterative Algorithm

Next, we present our iterative algorithm for computing minimum type error sources.

In order to formalize the termination condition of the algorithm, we first need

to define the set of usage locations of `let` variables in program p that are in the scope of the current expansion $\mathfrak{L}$. We denote this set by $\mathit{Scope}(p, \mathfrak{L})$. Intuitively, $\mathit{Scope}(p, \mathfrak{L})$ consists of all those usage locations of `let` variables that either occur in the body of a top-level `let` declaration or in the defining expression of some other `let` variable which has at least one expanded usage location in $\mathfrak{L}$. Formally, $\mathit{Scope}(p, \mathfrak{L})$ is the largest set of usage locations in p that satisfies the following condition: for all $\ell \in \mathsf{dom}(\mathit{Uloc}_p)$, if $\mathit{Uloc}_p(\ell) \cap \mathfrak{L} = \emptyset \wedge \mathit{Uloc}_p(\ell) \neq \emptyset$, then $\mathit{Loc}(\ell) \cap \mathit{Scope}(p, \mathfrak{L}) = \emptyset$.

For $M = \textsc{Solve}(p, R, \mathfrak{L})$, we then define $\mathit{Usages}(p, \mathfrak{L}, M)$ to be the set of all usage locations of the `let` variables in p that are in scope of the current expansions and that are marked for expansion. That is, $\ell \in \mathit{Usages}(p, \mathfrak{L}, M)$ iff

1. $\ell \in \mathit{Scope}(p, \mathfrak{L})$, and
2. $M \not\models P_{\mathit{dloc}(\ell)}$

Note that if the second condition holds, then a potentially cheaper error source exists that involves locations in the definition of the variable x used at ℓ. Hence, that usage of x should not be typed by x's principal type but by the expanded typing constraints generated from x's defining expression.

We say that a solution L_M, corresponding to the result of $\textsc{Solve}(p, R, \mathfrak{L})$, is *proper* if $\mathit{Usages}(p, \mathfrak{L}, M) \subseteq \mathfrak{L}$, i.e., L_M does not contain any usage locations of `let` variables that are in scope and still typed by unexpanded instances of principal types.

Algorithm 1 shows the iterative algorithm. It takes an ill-typed program p and a cost function R as input and returns a minimum error source. The set $\mathfrak{L}$ of locations to be expanded is initialized to $\mathfrak{L}_0$. In each iteration, the algorithm first

Algorithm 1 Iterative algorithm for computing a minimum error source

```
1: procedure IterMinError(p, R)
2:     𝔏 ← 𝔏₀
3:     loop
4:         M ← Solve(p, R, 𝔏)
5:         L_u ← Usages(p, 𝔏, M)
6:         if L_u ⊆ 𝔏 then
7:             return L_M
8:         end if
9:         𝔏 ← 𝔏 ∪ L_u
10:    end loop
11: end procedure
```

computes a minimum error source for the current expansion using the procedure Solve from the previous section. If the computed error source is proper, the algorithm terminates and returns the current solution L_M. Otherwise, all usage locations of `let` variables involved in the current minimum solution are marked for expansion and the algorithm continues.

Correctness. We devote the remainder of this section to arguing the correctness of our iterative algorithm. In a nutshell, we show by induction that the solutions computed by our algorithm are also solutions of the naive algorithm that expands all usages of `let` variables immediately as in [72].

We start with the base case of the induction where we fully expand all constraints, i.e., $\mathfrak{L} = \mathit{Loc}(p)$.

Lemma 23. *Let p be a program and R a cost function and let $M = \textsc{Solve}(p, R, \mathit{Loc}(p))$. Then $L_M \subseteq \mathit{Loc}(p)$ is a minimum error source of p subject to R.*

Lemma 23 follows from [72, Theorem 1] because if $\mathfrak{L} = \mathit{Loc}(p)$, then we obtain exactly the same reduction to WPMaxSMT as in our previous work. More precisely, in this case the A-Var-Prin rule is never used. Hence, all usages of

`let` variables are typed by the expanded typing constraints according to rule A-Var-Exp. The actual proof requires a simple induction over the derivations of the constraint typing relation defined in Figure 4.8, respectively, the constraint typing relation defined in 4.3.

We next prove that in order to achieve full expansion it is not necessary that $\mathfrak{L} = \mathit{Loc}(p)$. To this end, define the set $\mathfrak{L}_p$, which consists of $\mathfrak{L}_0$ and all usage locations of `let` variables in p:

$$\mathfrak{L}_p = \mathfrak{L}_0 \cup \bigcup_{l \in \mathsf{dom}(\mathit{Uloc}_p)} \mathit{Uloc}_p(l).$$

Then $\vdash_{\mathfrak{L}}$ generates the same constraints as $\vdash_{\mathit{Loc}(p)}$ as stated by the following lemma.

Lemma 24. *For any p, Π, Γ, α, and $\mathcal{A}$, we have $\Pi, \Gamma \vdash_{\mathfrak{L}_p} p : \alpha \mid \mathcal{A}$ iff $\Pi, \Gamma \vdash_{\mathit{Loc}(p)} p : \alpha \mid \mathcal{A}$.*

Lemma 24 can be proved using a simple induction on the derivations of $\vdash_{\mathfrak{L}_p}$, respectively, $\vdash_{\mathit{Loc}(p)}$. First, note that $\mathit{Loc}(p) \setminus \mathfrak{L}_p$ is the set of locations of well-typed `let` variable definitions in p. Hence, the derivations using $\vdash_{\mathfrak{L}_p}$ will never use the A-Let-Exp rule, only A-Let-Prin. However, the A-Let-Prin rule updates both Π and Γ, so applications of A-Var-Exp (A-Var-Prin is never used in either case) will be the same as if $\vdash_{\mathit{Loc}(p)}$ is used.

The following lemma states that if the iterative algorithm terminates, then it computes a correct result.

Lemma 25. *Let p be a program, R a cost function, and $\mathfrak{L}$ such that $\mathfrak{L}_0 \subseteq \mathfrak{L} \subseteq \mathfrak{L}_p$. Further, let $M = \textsc{Solve}(p, R, \mathfrak{L})$ such that L_M is proper. Then, L_M is a minimum error source of p subject to R.*

The proof of Lemma 25 can be found in Appendix A.7.3. For brevity, we

provide here only the high-level argument. The basic idea is to show that adding each of the remaining usage locations to $\mathfrak{L}$ results in typing constraints for which L_M is again a proper minimum error source. More precisely, we show that for each set $\mathfrak{D}$ such that $\mathfrak{L}_0 \subseteq \mathfrak{L} \subseteq \mathfrak{L} \cup \mathfrak{D} \subseteq \mathfrak{L}_p$, if M is the maximum model of $I(p, R, \mathfrak{L})$ from which L_M was computed, then M can be extended to a maximum model M' of $I(p, R, \mathfrak{L} \cup \mathfrak{D})$ such that $L_{M'} = L_M$. That is, L_M is again a proper minimum error source for $I(p, R, \mathfrak{L} \cup \mathfrak{D})$. The proof goes by induction on the cardinality of the set $\mathfrak{D}$. Therefore, by the case $\mathfrak{L} \cup \mathfrak{D} = \mathfrak{L}_p$, Lemma 23, and Lemma 24 we have that L_M is a true minimum error source for p subject to R.

Finally, note that the iterative algorithm always terminates since $\mathfrak{L}$ is bounded from above by the finite set $\mathfrak{L}_p$ and $\mathfrak{L}$ grows in each iteration. Together with Lemma 25, this proves the total correctness of the algorithm.

Theorem 6. *Let p be a program and R a cost function. Then,* IterMinError(p, R) *terminates and computes a minimum error source for p subject to R.*

4.5.4 Implementation and Evaluation

We now show the results of evaluating our improved algorithm for finding minimum type error sources on the OCaml student benchmark suite from [54] introduced in § 4.4. Additionally, we also evaluate our implementation on larger programs extracted from the GRASShopper [75] program verification tool written in OCaml.

The prototype implementation of the new algorithm was uniformly faster than the naive approach in our experiments. Most importantly, the number of generated typing constraints produced by the algorithm is almost an order of magnitude smaller than when using the naive approach. Consequently, the new algorithm also ran faster in the experiments.

We note that the new algorithm and the algorithm from § 4.3 provide the same formal guarantees. Hence, we do not repeat here the experiment on quality of type error sources.

Implementation. Our implementation of the new algorithm directly follows the definition in 4.5.3. That is, our implementation is a simple Python wrapper around the implementation of the naive algorithm.

The generation of typing constraints for each iteration in our algorithm directly follows the typing rules in Figure 4.3. In addition, we perform a simple optimization that reduces the total number of typing constraints. When typing an expression `let` $x = e_1$ `in` e_2, the `A-Let-Prin` and `A-Let-Exp` rules always add a fresh instance of the constraint $\mathcal{A}_1$ for e_1 to the whole set of constraints. This is to ensure that type errors in e_1 are not missed if x is never used in e_2. We can avoid this duplication of $\mathcal{A}_1$ in certain cases. If a principal type was successfully computed for the `let` variable beforehand, the constraints $\mathcal{A}_1$ must be consistent. If the expression e_1 refers to variables in the environment that have been bound by lambda abstraction, then not instantiating $\mathcal{A}_1$ at all could make the types of these variables under-constrained. However, if $\mathcal{A}_1$ is consistent and e_1 does not contain variables that come from lambda abstractions, then we do not need to include a fresh instance of $\mathcal{A}_1$ in `A-Let-Prin`. Similarly, if e_1 has no principal type because of a type error and the variable x is used somewhere in e_2, then the algorithm ensures that all such usages are expanded and included in the whole set of typing constraints. Therefore, we can safely omit the extra instance of $\mathcal{A}_1$ as well.

Student benchmarks. In our first experiment, we collected statistics for finding a single minimum error source in the student benchmarks with our iterative

algorithm and the naive algorithm. We measured the number of typing constraints generated (Fig. 4.9), the execution times (Fig. 4.10), and the number of expansions and iterations taken by our algorithm (Table 4.1).

Figure 4.9 shows the statistics for the total number of generated typing assertions. By typing assertions we mean logical assertions, encoding the typing constraints, that we pass to the weighted MaxSMT solver. The number of typing assertions roughly corresponds to the sum of the total number of locations, constraints attached to each location due to copying, and the number of well typed `let` definitions. Again, all 8 groups of programs are shown on the `x` axis in Figure 4.9. The numbers in parenthesis indicate the number of programs in each group. For each group and each approach (naive and iterative), we plot the maximum, minimum and average number of typing assertions. To show the general trend for how both approaches are scaling, lines have been drawn between the averages for each group. (All of the figures in this section follow this pattern.) As can be seen, our algorithm reduces the total number of generated typing assertions. This number grows exponentially with the size of the program for the naive approach. With our approach, this number seems to grow at a much slower rate since it does not expand every usage of a `let` variable unless necessary. These results make us cautiously optimistic that the number of assertions the iterative approach expands will be polynomial in practice. Note that the total number of typing assertions produced by our algorithm is the one that is generated in the last iteration of the algorithm.

The statistics for execution times are shown in Figure 4.10. The iterative algorithm is consistently faster than the naive solution. We believe this to be a direct consequence of the fact that our algorithm generates a substantially smaller num-

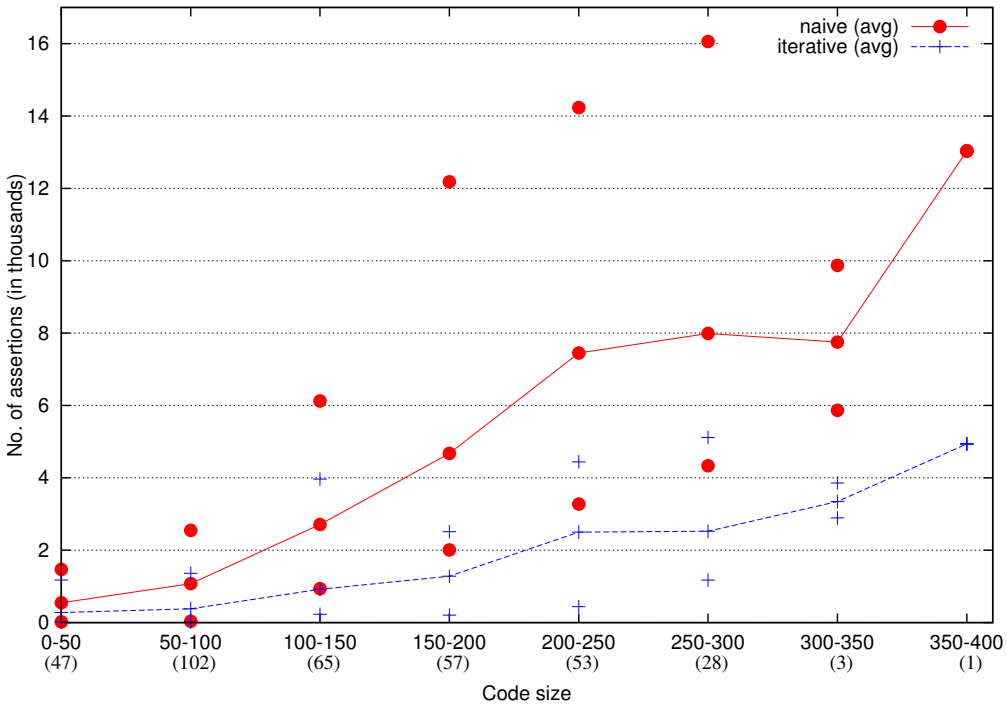

Figure 4.9: Maximum, average, and minimum number of typing assertions for computing a minimum error source by naive and iterative approach

ber of typing constraints. The difference in execution times between our algorithm and the naive approach increases with the size of the input program. Note that the total times shown are collected across all iterations.

We also measured the statistics on the number of iterations and expansions taken by our algorithm. The number of expansions corresponds to the total number of usage locations of `let` variables that have been expanded in the last iteration of our algorithm. The results, shown in Table 4.1, indicate that the total number of iterations required does not substantially change with the input size. We hypothesize that this is due to the fact that type errors are usually tied only to a small portion of the input program, whereas the rest of the program is not relevant to the error.

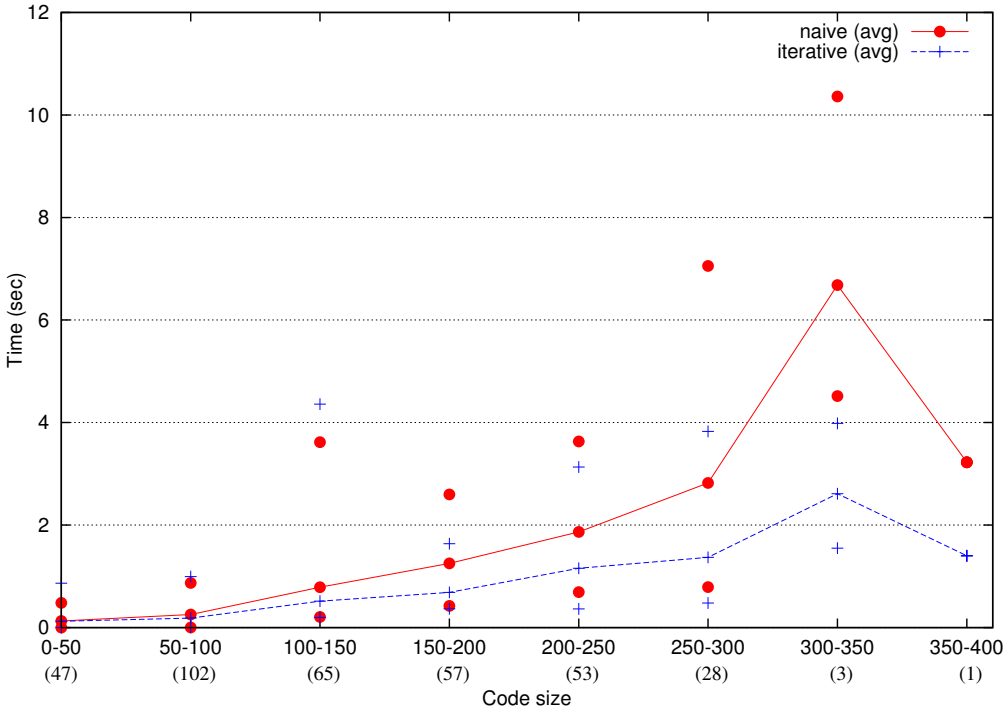

Figure 4.10: Maximum, average, and minimum execution times for computing a minimum error source by naive and iterative approach

It is worth noting that both the naive and iterative algorithm compute single error sources. The algorithms may compute different solutions for the same input since the fixed cost function does not enforce unique solutions. Both approaches are complete and would compute identical solutions for the all error sources problem [72]. The iterative algorithm does not attempt to find a minimum error source in the least number of iterations possible, but rather it expands `let` definitions on-demand as they occur in the computed error sources. This means that the algorithm sometimes continues expanding `let` definitions even though there exists a proper minimum error source for the current expansion. In our future work, we plan to consider how to enforce the search algorithm so that it first finds those minimum error sources that require less iterations and expansions.

	iterations			expansions		
	min	avg	max	min	avg	max
0-50	0	0.49	2	0	1.7	11
50-100	0	0.29	3	0	0.88	13
100-150	0	0.49	4	0	1.37	32
150-200	0	0.44	3	0	1.82	19
200-250	0	0.49	2	0	3.11	30
250-300	0	0.36	2	0	6.04	45
300-350	0	0.67	2	0	3.33	10
350-400	0	0	0	0	0	0

Table 4.1: Statistics for the number of expansions and iterations when computing a single minimum error source

GRASShopper benchmarks. Since we lacked an appropriate corpus of larger ill-typed user written programs, we generated ill-typed programs from the source code of the GRASShopper tool [75]. We chose GRASShopper because it contains non-trivial code that mostly falls into the OCaml fragment supported by Easy-OCaml. For our experiments, we took several modules from the GRASShopper source code and put them together into four programs of 1000, 1500, 2000, and 2500 lines of code, respectively. These modules include the core data structures for representing the abstract syntax trees of programs and specification logics, as well as complex utility functions that operate on these data structures. We included comments when counting the number of program lines. However, comments were generally scars. The largest program with 2500 lines comprised 282 top-level `let` definitions and 567 `let` definitions in total. We then introduced separately five distinct type errors to each program, obtaining a new benchmarks suite of 20 programs in total. We introduced common type mismatch errors such as calling a function or passing an argument with an incompatible type.

We repeated the previous experiments on the generated GRASShopper bench-

marks. The benchmarks are grouped by code size. There are four groups of five programs corresponding to programs with 1000, 1500, 2000, and 2500 lines.

Figure 4.11 shows the total number of generated typing assertions subject to the code size. This figure follows the conventions of Fig. 4.9 except that the number of constraints is given on a logarithmic scale. Note that the minimum, maximum, and average points are plotted in Figures 4.11 and 4.12 for each group and algorithm, but these are relatively close to each other and hence visually mostly indistinguishable. The total number of assertions generated by our algorithm is consistently an order of magnitude smaller than when using the naive approach. The naive approach expands all `let` defined variables where the iterative approach expands only those `let` definitions that are needed to find the minimum error source. Consequently, the times taken by our algorithm to compute a minimum error source are smaller than when using the naive one, as shown in Figure 4.12. Beside solving a larger weighed MaxSMT instance, the naive approach also has to spend more time generating typing assertions than our iterative algorithm. Finally, Table 4.2 shows the statistics on the number of iterations and expansion our algorithm made while computing the minimum error source. Again, the total number of iterations appears to be independent of the size of the input program.

	iterations			**expansions**		
	min	avg	max	min	avg	max
1000	0	0.2	1	0	0.2	1
1500	0	0.4	2	0	2.8	14
2000	0	0.6	2	0	53.8	210
2500	0	0.2	1	0	3	15

Table 4.2: Statistics for the number of expansions and iterations when computing a single minimum error source for larger programs

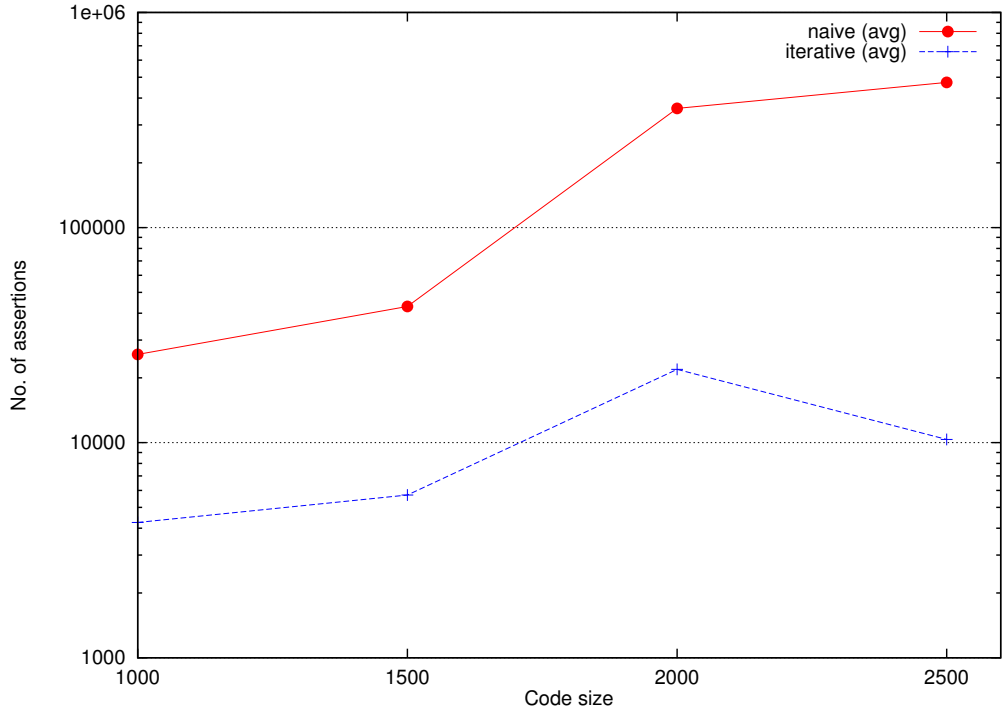

Figure 4.11: Maximum, average, and minimum number of typing assertions for computing a minimum error source by naive and iterative approach for larger programs

Comparison to other tools. Our algorithm also outperforms the approach by Myers and Zhang [97] in terms of speed on the same student benchmarks. While our algorithm ran always under 5 seconds, their algorithm took over 80 seconds for some programs. We also ran their tool SHErrLoc [82] on one of our GRASSHopper benchmark programs of 2000 lines of code. After approximately 3 minutes, their tool ran out of memory. We believe this is due to the exponential explosion in the number of typing constraints due to polymorphism. For that particular program, the total number of typing constraints their tool generated was roughly $200,000$. On the other hand, their tool shows high precision in correctly pinpointing the actual source of type errors. These results nicely exemplify the nature of type error localization. In order to solve the problem of producing high quality type

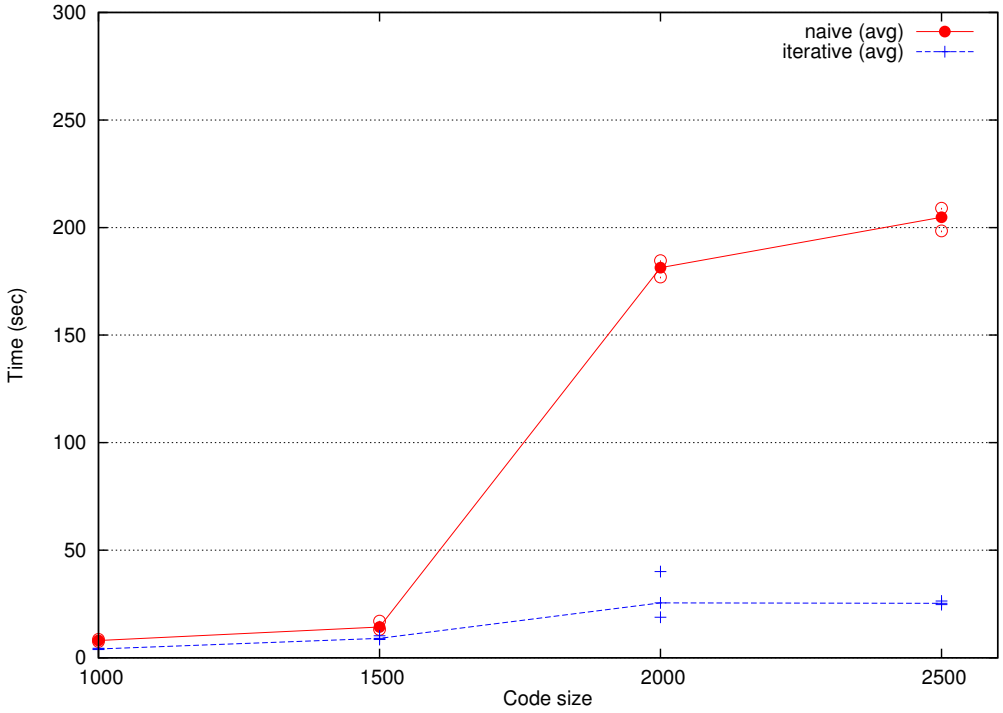

Figure 4.12: Maximum, average, and minimum execution times for computing a minimum error source by naive and iterative approach for larger programs

error reports, one needs to consider the whole typing data. However, the size of that data can be impractically large, making the generation of type error reports slow to the point of being not usable. One benefit of our approach is that these two problems can be studied independently. We mainly focused on the second problem, i.e., how to make the search for high-quality type error sources practically fast.

4.6 Related Work

Previous work on localization of type errors has mainly focused on designing concrete systems for generating quality type error messages. The approach taken in [94, 22] computes a trace of type inference, i.e., a set of decisions made by

type inference algorithm that led to a type error. Similarly, type error localization algorithms presented in [89, 30, 77] show a program slice involved in the error. The obvious difference between these approaches and the one taken in this thesis concerns the used notion of a type error source. In our work, type error sources are minimal sets of expressions that, once corrected, yield a well-typed program and are guaranteed to satisfy the compiler or user provided optimality criterion. In contrast, modeling error sources as traces and slices often results in redundant information being reported to the programmer, such as statements or type inference decisions that are related to the error but do not cause it. The works presented in [14, 13, 67] design completely new type systems that, while performing the type inference, also collect additional information that can be reported to the programmer in the case of type errors. Our work, on the other hand, does not require changes to the existing type systems yet it simply piggybacks on top of them. More closely related to our approach is the `Seminal` [54] tool, which computes several possible error sources by repeated calls to the type checker rather than an SMT solver. However, the search for error causes is based on fixed heuristics and provides no formal guarantees that all error sources are found, respectively, that they are ranked according to some criterion. Sulzmann [86, 84] presents a type error debugger for Haskell that presents to the programmer a minimal set of locations whose corresponding expressions are responsible for the type error. Our work also considers a minimal set of expressions inducing a type error as a potential error source, but we also allow incorporating heuristics for choosing which minimal set will be reported to the programmer as there can be many such sets. Zhang and Myers [97] encode typing information for Hindley-Milner type systems in terms of constraint graphs. The generated graphs are then analyzed to find most likely error

sources by using Bayesian inference. It is unclear how this approach would support more expressive type systems. Also, the constraint graph generation suffers from the exponential blow-up due to polymorphism, which our work addresses by the lazy expansion of constraints induced by typing polymorphic functions (§ 4.5). On the other hand, the proposed technique is designed for general diagnosis of static errors, whereas we focus specifically on diagnosis of type errors. In a follow up work, the authors also apply their techniques to Haskell [98].

Previous approaches based on constraint solving [33, 85] produce minimal but not minimum error sources and consider specific ranking criteria for specific type systems. Hage and Heeren [35] build a type error debugging system for Haskell by incorporating several fixed and expert designed heuristics for explaining type errors. Our work, in contrast, provides a general framework for type error debugging where the heuristic for identifying the root cause of a type error is merely an input to the system. The work by the same authors that is more related to our work can be found in [36]. Here, the authors present a framework for constraint-based type inference that allows compilers to plug in various different strategies for solving the constraints, consequently allowing more flexibility for producing type error reports. These strategies allow different constraint solving decision based on the context and can be used, for instance, to manipulate the order in which the constraints are solved. In contrast to our work, it is unclear whether their system produces the optimal solution given a constraint-solving input strategy. Seidel et al. devise a general machine learning algorithm that blames program expressions for causing type errors by offline learning from the corpus of ill-typed programs [81]. While their approach can support various type systems and ranking criteria, it does not provide a formal guarantee that the blamed program expressions indeed consti-

tute an actual error source. The problem of type error localization has also been studied in the context of Java generics [12], domain-specific languages [34], and dependently typed languages [25].

Following our work initially published in [72, 73], the MaxSMT approach to type inference has also been applied to Python [39]. Here, a MaxSMT solver is used to infer types for as large as possible portion of the given program, which is particularly useful since Python is a dynamically typed language and inferring static types for the whole program is consequently often impossible. We note that our approach is in part inspired by the `Bug-Assist` tool [47], which uses a MaxSAT procedure for fault localization in imperative programs. However, the problem we are solving in this thesis is quite different.

Chapter 5

Conclusions

The goal of this thesis is to provide a mathematically rigorous framework for the systematic development of type inference algorithms that are convenient to use by the programmers. Towards achieving that goal, we focused on two important problems surrounding type inference: (1) how to constructively design type inference algorithms that improve over the state-of-the-art and (2) how to automatically debug type errors that arise during inference. This thesis approaches these two specific problems based on the close connection between type inference and program analysis techniques. We addressed the first problem by using abstract interpretation to constructively design Liquid type inference, an advanced family of algorithms that combine classical typing disciplines and known static analyses to prove various safety properties of functional programs. By rigorously modeling Liquid type inference using the primitives of abstract interpretation, we unveiled the design space of such algorithms and generalized it in a way that allows easy construction of novel type inference algorithms that are sound by construction. Regarding the second problem, we have shown how the problem of type

error localization for type inference algorithms in the style of Hindley-Milner can be cast as an optimization problem expressed in a formal logic. We then showed how this problem can be solved using automated theorem provers. Finally, we experimentally illustrated how our approach yields type error debugging systems that outperform the state-of-the-art tools in correctly identifying the root causes of type errors while still being efficient, even for large programs.

The results presented in this thesis naturally open several interesting research directions. An exciting future work project is to use our abstract interpretation model of Liquid types to obtain and implement type inference algorithms that are more expressive than the original Liquid type inference. This could be achieved by instantiating our generalization of Liquid types with, say, the polyhedra abstract domain or automata-based domains. Another interesting research problem is to extend our framework for type error debugging to support type inference algorithms that are more expressive than the Hindley-Milner algorithm, such as Liquid types for instance. Due to the close connection between type inference and static analysis, the problem of building a general framework for debugging static analyses naturally arises. In general, we believe this thesis provides important results and generally paves the way for the systematic study and development of enhanced type analyses that are usable by the programmers.

Appendix A

Data Flow Refinement Type Inference

A.1 Example Concrete Map

Let us reconsider the example from § 3.1. The code is again given below, but this time we also annotate expressions of interest with location labels.

```
1  let dec y = y - 1 in
2  let f x g = if x >= 0 then g^o x^p else x^q
3  in f^a (f^b 1^c dec^d)^j dec^m
```

The concrete execution map M_e for this program, restricted to the execution nodes

of interest is as follows:

$$\mathtt{q}^{\mathtt{a}} \mapsto \bot \qquad \mathtt{q}^{\mathtt{b}} \mapsto \bot \qquad \mathtt{c} \mapsto 1 \qquad \mathtt{p}^{\mathtt{a}} \mapsto 0 \qquad \mathtt{p}^{\mathtt{b}} \mapsto 1$$

$$\mathtt{j} \mapsto 0 \qquad \mathtt{d} \mapsto [\mathtt{p}^{\mathtt{b}} \mapsto (1,\ 0)] \qquad \mathtt{m} \mapsto [\mathtt{p}^{\mathtt{a}} \mapsto (0,\ -1)]$$

$$\mathtt{dec} \mapsto [\mathtt{p}^{\mathtt{b}} \mapsto (1,\ 0),\ \mathtt{p}^{\mathtt{a}} \mapsto (0,\ -1)]$$

$$\mathtt{a} \mapsto [\mathtt{j} \mapsto (0,\ [\mathtt{m} \mapsto ([\mathtt{p}^{\mathtt{a}} \mapsto (0,\ -1)],\ -1)])]$$

$$\mathtt{b} \mapsto [\mathtt{c} \mapsto (1,\ [\mathtt{d} \mapsto ([\mathtt{p}^{\mathtt{b}} \mapsto (1,\ 0)],\ 0)])]$$

$$\mathtt{f} \mapsto [\mathtt{j} \mapsto (0,\ [\mathtt{m} \mapsto ([\mathtt{p}^{\mathtt{a}} \mapsto (0,\ -1)],\ -1)]),$$
$$\mathtt{c} \mapsto (1,\ [\mathtt{d} \mapsto ([\mathtt{p}^{\mathtt{b}} \mapsto (1,\ 0)],\ 0)])]$$

For this specific program, the execution will always reach the program expressions with locations $\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{j}$, and $\mathtt{m}$ with the same environment. For this reason, we identify these expression nodes with their locations. Expressions with labels $\mathtt{o}, \mathtt{p}$, and $\mathtt{q}$ will be reachable with two environments, each of which corresponds to executing the body of $\mathtt{f}$ with the value passed to it at call sites with locations $\mathtt{a}$ and $\mathtt{b}$. We use this call site information to distinguish the two evaluations of the body of $\mathtt{f}$. For instance, the expression node associated with the $\mathtt{x}^{\mathtt{q}}$ expression when performing the analysis for the call to $\mathtt{f}$ made at location $\mathtt{a}$ is denoted by $\mathtt{q}^{\mathtt{a}}$ in the above table.

A.2 Concrete Semantics Proofs

Lemma 26. prop *is increasing.*

Proof. Let $\langle v_1', v_2' \rangle = \mathsf{prop}(v_1, v_2)$. The proof goes by mutual structural induction on v_1 and v_2'.

Base cases.

- $v_1 = \bot$.

$$v_1' = \bot, v_2' = v_2 \qquad \text{def. of } \mathsf{prop}$$
$$v_1 \sqsubseteq v_1', v_2 \sqsubseteq v_2' \qquad \text{def. of } \sqsubseteq$$

- $v_1 = \omega$.

$$v_1' = v_2' = \omega \qquad \text{def. of } \mathsf{prop}$$
$$v_1 \sqsubseteq v_1', v_2 \sqsubseteq v_2' \qquad \text{def. of } \sqsubseteq$$

- $v_1 = c, v_2 = \bot$.

$$v_1' = v_2' = c \qquad \text{def. of } \mathsf{prop}$$
$$v_1 \sqsubseteq v_1', v_2 \sqsubseteq v_2' \qquad \text{def. of } \sqsubseteq$$

- $v_1 = c, v_2 = c$.

$$v_1' = v_2' = c \qquad \text{def. of } \mathsf{prop}$$
$$v_1 \sqsubseteq v_1', v_2 \sqsubseteq v_2' \qquad \text{def. of } \sqsubseteq$$

- $v_1 = c_1, v_2 = c_2, c_1 \neq c_2$.

$$v_1' = v_1, v_2' = \omega \qquad \text{def. of } \mathsf{prop}$$
$$v_1 \sqsubseteq v_1', v_2 \sqsubseteq v_2' \qquad \text{def. of } \sqsubseteq$$

- $v_1 = c_1, v_2 = \omega$.

$$v_1' = v_1, v_2' = \omega \qquad \text{def. of } \mathsf{prop}$$
$$v_1 \sqsubseteq v_1', v_2 \sqsubseteq v_2' \qquad \text{def. of } \sqsubseteq$$

- $v_1 = c, v_2 \in \mathcal{T}$.

$$v_1' = v_1, v_2' = \omega \qquad \text{def. of } \mathsf{prop}$$
$$v_1 \sqsubseteq v_1', v_2 \sqsubseteq v_2' \qquad \text{def. of } \sqsubseteq$$

- $v_1 \in \mathcal{T}, v_2 = \bot$.

$$v_1' = v_1, v_2' = T_\bot \qquad \text{def. of } \mathsf{prop}$$
$$v_1 \sqsubseteq v_1', v_2 \sqsubseteq v_2' \qquad \text{def. of } \sqsubseteq$$

- $v_1 \in \mathcal{T}, v_2 = c$.

$$v_1' = v_1, v_2' = \omega \qquad \text{def. of } \mathsf{prop}$$
$$v_1 \sqsubseteq v_1', v_2 \sqsubseteq v_2' \qquad \text{def. of } \sqsubseteq$$

- $v_1 \in \mathcal{T}, v_2 = \omega$.

$$v_1' = v_2' = \omega \qquad \text{def. of } \mathsf{prop}$$
$$v_1 \sqsubseteq v_1', v_2 \sqsubseteq v_2' \qquad \text{def. of } \sqsubseteq$$

Induction case: $v_1 = T_1, v_2 = T_2$. By definition of prop, we need to consider two cases of call-sites subject to the table T_2.

- $n_{\mathsf{cs}} \notin T_2$.

$$v_{1i} = v_{1i}', v_{2i} = v_{2i}', v_{1o} = v_{1o}', v_{20} = v_{2o}' \qquad \text{def. of } \mathsf{prop}$$
$$v_{1i} \sqsubseteq v_{1i}', v_{2i} \sqsubseteq v_{2i}', v_{1o} \sqsubseteq v_{1o}', v_{20} \sqsubseteq v_{2o}' \qquad \text{def. of } \sqsubseteq$$

- $n_{\mathsf{cs}} \in T_2$.

$$v_{1i} \sqsubseteq v_{1i}', v_{2i} \sqsubseteq v_{2i}', v_{1o} \sqsubseteq v_{1o}', v_{20} \sqsubseteq v_{2o}' \qquad \text{def. of } \mathsf{prop} \text{ and i.h.} \qquad \text{(A.1)}$$

□

Lemma 27. step *is increasing.*

Proof. By structural induction on program expressions. The result follows from the increasing property of the join $\sqcup$ and Lemma 26. □

Lemma 28. prop *is monotone.*

Proof. Let $\langle v_1', v_2' \rangle = \mathsf{prop}(v_1, v_2)$, $\langle v_3', v_4' \rangle = \mathsf{prop}(v_3, v_4)$, $v_1 \sqsubseteq v_3$, and $v_2 \sqsubseteq v_4$. We show that $v_1' \sqsubseteq v_3'$ and $v_2' \sqsubseteq v_4'$. The proof goes by mutual structural induction on v_3 and v_4.

Base cases.

- $v_3 = \omega$.

$$\begin{aligned} v_3' = \omega, v_4' = \omega && \text{def. of } \mathsf{prop} \\ v_1' \sqsubseteq v_3', v_2' \sqsubseteq v_4' && \text{def. of } \sqsubseteq \end{aligned}$$

- $v_4 = \omega$.

$$\begin{aligned} v_4' = \omega && \text{def. of } \mathsf{prop} \\ v_2' \sqsubseteq v_4' && \text{by } \sqsubseteq \end{aligned}$$

  - $v_3 = c$

$$\begin{aligned} v_1 = \bot \vee v_1 = c && \text{by } v_1 \sqsubseteq v_3 \\ v_3' = c, v_1' = \bot \vee v_1' = c && \text{def. of } \mathsf{prop} \\ v_1' \sqsubseteq v_3' && \text{by } \sqsubseteq \end{aligned}$$

  - $v_3 \in \mathcal{T}$

$$\begin{aligned} v_3' = \omega && \text{def. of } \mathsf{prop} \\ v_1' \sqsubseteq v_3' && \text{by } \sqsubseteq \end{aligned}$$

- $v_3 = \bot$.

$$\begin{aligned} v_1 = \bot && \text{by } v_1 \sqsubseteq v_3 \\ v_4' = v_4, v_2' = v_2, v_3' = v_3, v_1' = v_1 && \text{def. of } \mathsf{prop} \\ v_1' \sqsubseteq v_3', v_2' \sqsubseteq v_4' && \text{def. of } \sqsubseteq \end{aligned}$$

- $v_4 = \bot$.

$$\begin{aligned} v_2 &= \bot && \text{by } v_2 \sqsubseteq v_4 \\ v_1' = v_1, v_3' &= v_3 && \text{def. of } \mathsf{prop} \\ v_1' &\sqsubseteq v_3' && \text{def. of } \sqsubseteq \end{aligned}$$

  - $v_3 = c$

$$\begin{aligned} v_1 &= \bot \vee v_1 = c && \text{by } v_1 \sqsubseteq v_3 \\ v_4' = c, v_2' &= \bot \vee v_2' = c && \text{def. of } \mathsf{prop} \\ v_2' &\sqsubseteq v_4' && \text{by } \sqsubseteq \end{aligned}$$

  - $v_3 \in \mathcal{T}$

$$\begin{aligned} v_1 &= \bot \vee v_1 \in \mathcal{T} && \text{by } v_1 \sqsubseteq v_3 \\ v_4' = T_\bot, v_2' &= \bot \vee v_2' = T_\bot && \text{def. of } \mathsf{prop} \\ v_2' &\sqsubseteq v_4' && \text{by } \sqsubseteq \end{aligned}$$

- $v_3 = c, v_4 = c$.

$$\begin{aligned} v_4' = v_4 = v_3 = v_3' &= c && \text{def. of } \mathsf{prop} \\ v_1 = \bot \vee v_1 = c, v_2 &= \bot \vee v_2 = c && \text{by } v_1 \sqsubseteq v_3, v_2 \sqsubseteq v_4 \\ v_1' = \bot \vee v_1' = c, v_2' &= \bot \vee v_2' = c && \text{def. of } \mathsf{prop} \\ v_1' \sqsubseteq v_3', v_2' &\sqsubseteq v_4' && \text{def. of } \sqsubseteq \end{aligned}$$

- $v_3 = c_1, v_4 = c_2, c_1 \neq c_2$.

$$\begin{aligned} v_4' = \omega, v_3 = v_3' &= c_1 && \text{def. of } \mathsf{prop} \\ v_1 = \bot \vee v_1 &= c_1 && \text{by } v_1 \sqsubseteq v_3, v_2 \sqsubseteq v_4 \\ v_1' = \bot \vee v_1' &= c_1 && \text{def. of } \mathsf{prop} \\ v_1' \sqsubseteq v_3', v_2' &\sqsubseteq v_4' && \text{def. of } \sqsubseteq \end{aligned}$$

- $v_3 = c, v_4 \in \mathcal{T}$.

$$\begin{aligned} v_4' = \omega, v_3 = v_3' = c && \text{def. of } \mathsf{prop} \\ v_1 = \bot \vee v_1 = c && \text{by } v_1 \sqsubseteq v_3, v_2 \sqsubseteq v_4 \\ v_1' = \bot \vee v_1' = c && \text{def. of } \mathsf{prop} \\ v_1' \sqsubseteq v_3', v_2' \sqsubseteq v_4' && \text{def. of } \sqsubseteq \end{aligned}$$

- $v_3 \in \mathcal{T}, v_4 = c$.

$$\begin{aligned} v_4' = \omega, v_3 = v_3' && \text{def. of } \mathsf{prop} \\ v_1 = \bot \vee v_1 \in \mathcal{T}, v_2 = \bot \vee v_2 = c && \text{by } v_1 \sqsubseteq v_3, v_2 \sqsubseteq v_4 \\ v_1' = v_1 && \text{def. of } \mathsf{prop} \\ v_1' \sqsubseteq v_3', v_2' \sqsubseteq v_4' && \text{def. of } \sqsubseteq \end{aligned}$$

Induction case: $v_3 = T_3$ and $v_4 = T_4$. If $v_1 = \bot$, then $v_1' = \bot$ and $v_1' \sqsubseteq v_3'$ trivially follows. When $v_2 = \bot$, we have $v_2' = \bot \vee v_2' = T_\bot$ and the argument follows from Lemma 26.

We are hence left with the case when $v_1 = T_1$ and $v_2 = T_2$. First, we have that $n_{\mathsf{cs}} \in T_2 \implies n_{\mathsf{cs}} \in T_4$ by the $T_2 \sqsubseteq T_4$ assumption. We have three cases to consider.

- $n_{\mathsf{cs}} \notin T_2$ and $n_{\mathsf{cs}} \notin T_4$. Trivial, as there is no propagation.
- $n_{\mathsf{cs}} \notin T_2$ and $n_{\mathsf{cs}} \in T_4$. By prop, there is no propagation between the pairs $T_1(n_{\mathsf{cs}})$ and $T_2(n_{\mathsf{cs}})$. The result then follows from Lemma 26.
- $n_{\mathsf{cs}} \in T_2$ and $n_{\mathsf{cs}} \in T_4$. The result follows from the induction hypothesis.

□

Lemma 29. step *is monotone.*

Proof. Let $M' \dot{\sqsubseteq} M''$ and $k \in \mathbb{N}$. For any E and e, we show that $M'_f \dot{\sqsubseteq} M''_f$ where $M'_f = \mathsf{step}_k[\![e]\!](E)(M')$ and $M''_f = \mathsf{step}_k[\![e]\!](E)(M'')$. The proof goes by structural induction on e and natural induction on k. When $k = 0$, the argument is trivial. We thus consider the case when $k > 0$ and do the case analysis on e.

- $e = c^\ell$. $M'_f \dot{\sqsubseteq} M''_f$ follows from the monotonicity of $\sqcup$.
- $e = x^\ell$. Here, $M'_f \dot{\sqsubseteq} M''_f$ follows from the monotonicity of prop (Lemma 28).
- $e = (e_1^{\ell_1} e_2^{\ell_2})^\ell$. We do case analysis on the values computed for e_1 and e_2 by $\mathsf{step}_k[\![e]\!](E)(M')$ and $\mathsf{step}_{k+1}[\![e]\!][k](E)(M'')$.
  - $v'_1 = \bot$. Then $M'_f = M'_1$ and the result holds by Lemma 27.
  - $v''_1 = \bot$. By i.h., $M'_1 \dot{\sqsubseteq} M''_1$, hence $v'_1 = \bot$ and the argument follows from the previous case.
  - $v'_1 \neq \bot \wedge v'_1 \notin \mathcal{T}$. By i.h., $M'_1 \dot{\sqsubseteq} M''_1$ and hence $v''_1 \neq \bot \wedge v'' \notin \mathcal{T}$. Therefore, $M''_f = M_\omega$ and the result follows trivially.
  - $v'_1 \in \mathcal{T}, v''_1 \notin \mathcal{T}$. The argument is trivial as $M''_f = M_\omega$.
  - $v'_1 \in \mathcal{T}, v''_1 \in \mathcal{T}$. By i.h., $v'_1 \sqsubseteq v''_1$. The cases for v'_2 and v''_2 are handled the same way as for v'_1 and v''_1. We are thus interested in the remaining case where $v'_2, v''_2 \notin \{\bot, \omega\}$ where by i.h. $v'_2 \sqsubseteq v''_2$. The result then follows by the monotonicity of prop (Lemma 28).
- $e = (\lambda x.e)^\ell$. The argument, similar to the case for function applications, follows from Lemmas 28, 26, and basic properties of joins $\sqcup$.

□

Proof (of Lemma 2)**.**

Proof. Follows directly from Lemma 26 and Lemma 28. □

Proof (of Lemma 3)**.**

Proof. Follows directly from Lemma 27 and Lemma 29. □

We next prove that our definitions of joins and meets are indeed correct. The definition of a binary join on concrete values is raised to an arbitrary number of values in the expected way where $\sqcup\emptyset \stackrel{\text{def}}{=} \bot$ and $\sqcup\{v\} \stackrel{\text{def}}{=} v$. The meet $\sqcap$ is defined similarly. We also define a depth of a concrete data flow value in the expected way. The depth of constants, errors, and unreachable values is simply 0. The depths of tables is the maximum depth of any of the values stored for any call site, increased by 1.

Definition 3 (Depth of Values)**.**

$$\mathsf{depth}(\bot) \stackrel{\text{def}}{=} \mathsf{depth}(c) \stackrel{\text{def}}{=} \mathsf{depth}(\omega) \stackrel{\text{def}}{=} 0$$

$$\mathsf{depth}(T) \stackrel{\text{def}}{=} 1 + \mathsf{max}(\bigcup_{n_{\mathsf{cs}}} \mathsf{depth}(\pi_1(T(n_{\mathsf{cs}}))) \cup \bigcup_{n_{\mathsf{cs}}} \mathsf{depth}(\pi_1(T(n_{\mathsf{cs}}))))$$

Note that depth is well-defined since, although there is an infinite number of call sites in a table, the depth of every value stored at every call site in a table is bounded by the table depth, as tables are defined inductively.

Proof (of Lemma 1)**.**

Proof. We carry the proof by case analysis on elements of V and induction on the minimum depth thereof when V consists of tables only.

We first consider the trivial (base) cases where V is not a set of multiple tables.

- $V = \emptyset$. By $\sqsubseteq$, $lub(V) = \bot$ and $\sqcup(V) = \bot$ by $\sqcup$.
- $V = \{v\}$. Trivial.

- $c \in V, T \in V$. By $\sqsubseteq$, lub of $\{c, T\}$ is ω and since ω is the top element, $lub(V) = \omega = \sqcup V$.

- $c_1 \neq c_2, c_1 \in V, c_2 \in V$. As in the previous case.

- $\bot \in V$. By $\sqsubseteq$, $lub(V) = lub(V/\{\bot\})$ and $\sqcup(V) = \sqcup(V/\{\bot\})$ by $\sqcup$. The set $V/\{\bot\}$ either falls into one of the above cases or consists of multiple tables, which is the case we show next.

Let $V \subseteq \mathcal{T}$ and $|V| > 1$. Let d be the minimum depth of any table in V. By $\sqsubseteq$, lub of V is a table T that for every call site n_{cs} has input (resp. output) that is lub of the corresponding inputs (resp. outputs) of tables in V for n_{cs}. We focus on inputs; the same argument goes for outputs. Let $V_i = \{\pi_1(v(n_{\mathsf{cs}})) \mid n_{\mathsf{cs}} \in \mathcal{N}_e \wedge v \in V\}$. By $\sqcup$, $\pi_1(T(n_{\mathsf{cs}})) = \sqcup V_i$. We hence need to show that $\sqcup V_i = lub(V_i)$. The set V_i either falls into one of the trivial cases, from which the argument easily follows, or $V_i \subseteq \mathcal{T} \wedge |V| > 1$. In the latter case, the minimum depth of any value in V_i is smaller than d by the definition of depth and the result follows from the i.h.

The proof of correctness for the meet $\sqcap$ is similar. □

A.2.1 Additional Properties

We now introduce and prove several interesting properties related to the concrete data flow semantics.

A.2.1.1 Finiteness

We say a value v is finite $\mathsf{fin}(v)$ if it is either a ground value or a table that has seen only a finite number of inputs.

$$\mathsf{fin}(v) \stackrel{\text{def}}{=} (v \in \{\bot, \omega\} \cup \mathit{Cons}) \vee (v \in \mathcal{T} \wedge |\{n_{\mathsf{cs}} \mid n_{\mathsf{cs}} \in v\}| \in \mathbb{N} \wedge \forall n_{\mathsf{cs}} \in v.\, \mathsf{fin}(\pi_1(v(n_{\mathsf{cs}}))) \wedge \mathsf{fin}(\pi_2(v(n_{\mathsf{cs}}))))$$

It can be shown that prop does not affect the finiteness of values. The same clearly holds for joins $\sqcup$.

Lemma 30. *Let v_1, v_2, v_3, and v_4 be values. If $\mathsf{fin}(v_1)$, $\mathsf{fin}(v_2)$, and $\langle v_3, v_4\rangle = \mathsf{prop}(v_1, v_2)$, then $\mathsf{fin}(v_3)$ and $\mathsf{fin}(v_4)$.*

Proof. By structural induction on v_1 and v_2. The cases when either v_1 or v_2 are not a table are trivial. For tables, the argument follows by i.h. and the simple fact that prop does not invent new call site nodes, yet only manipulates the existing ones, whose total number is finite by the lemma assumption. □

We say a map M is finite $\mathsf{fin}(M)$ if all of its values are finite and the set of reachable nodes is finite as well $|\{n \mid M(n) \neq \bot\}| \in \mathbb{N}$. It can be shown that step preserves the finiteness of maps.

Lemma 31. *Let M and M' be maps, E an environment, e^ℓ an expression, and k a fuel. If $\mathsf{fin}(M)$, (e, E) is well-formed, and $\mathsf{step}_k[\![e^\ell]\!](E)(M) = M'$, then $\mathsf{fin}(M')$.*

Proof. The proof goes by natural induction on k and structural induction on e. The argument follows from the Lemma 30. Note that in the case e is a lambda abstraction and $M(E \diamond \ell) = T$, for some table T, the set of n_{cs} in T is finite. Hence, the set $\overline{M}$ is finite and by i.h. each $M' \in \overline{M}$ is finite as well. The result then follows from the basic properties of joins. □

A.2.1.2 Computation Range

We define an *expression range* of an expression node $E \diamond \ell$ subject to a program p to be a set of expression nodes whose location is a sublocation of ℓ in p and whose environment is an extension of E

$$\mathsf{erange}_p(E \diamond \ell) \stackrel{\text{def}}{=} \{E' \diamond \ell' \mid E \subseteq E' \wedge \ell' \in Loc(p(\ell))\}.$$

The corresponding *range* is then defined as

$$\mathsf{range}_p(E \diamond \ell) \stackrel{\text{def}}{=} \mathsf{erange}_p(E \diamond \ell) \cup \{n \mid n \in \mathsf{rng}(E') \wedge E' \diamond \_ \in \mathsf{erange}_p(E \diamond \ell))\}$$

Note that in the above definition we use the _ pattern to existentially quantify a certain object instead of explicitly writing down the quantifier. We do this to avoid clutter.

Let domain validity $\mathsf{domvalid}(\ell, p, E)$ of an environment E subject to program p and a location ℓ in p holds iff $\mathsf{dom}(E)$ consists of exactly all variables bound above ℓ in p. We use this definition to characterize the range of a computation performed by step.

Lemma 32. *Let M and M' be maps, E an environment, e^ℓ an expression of a program p, and k a fuel. If $\mathsf{domvalid}(\ell, p, E)$ and $\mathsf{step}_k[\![e^\ell]\!](E)(M) = M'$, then $\forall n. M(n) \neq M'(n) \implies n \in \mathsf{range}_p(E \diamond \ell)$.*

Proof. The proof goes by natural induction on k and structural induction on e. The domain validity assumption together with the uniqueness of locations in p guarantees that environments are always extended, i.e., their contents never get rewritten by environment updates done when analyzing the bodies of lambda abstraction, which is needed in order to ensure that updated nodes are in the erange of the current node. □

We now state few properties relating ranges. First, ranges of non-related ex-

pression do not share visited expression nodes.

Lemma 33. *Let E be an environment, p be a program, ℓ_1 and ℓ_2 two locations such that $Loc(p(\ell_1)) \cap Loc(p(\ell_2)) = \emptyset$. Then, $\mathsf{range}_p(E{\diamond}\ell_1) \cap \mathsf{range}_p(E{\diamond}\ell_2) \cap \mathcal{N}_e = \emptyset$.*

Proof. Follows directly from $Loc(p(\ell_1)) \cap Loc(p(\ell_2)) = \emptyset$. □

Similarly, the ranges of nodes whose environments are non-related do not have any common expression nodes.

Lemma 34. *Let E_1 and E_2 be two environments and ℓ a location of a program p. If $\mathsf{dom}(E_1) = \mathsf{dom}(E_2)$ and $E_1 \neq E_2$, then $\mathsf{range}_p(E{\diamond}\ell_1) \cap \mathsf{range}_p(E{\diamond}\ell_2) \cap \mathcal{N}_e = \emptyset$.*

Proof. The argument follows from the fact that the extensions of E_1 and E_2 have to be different as $\mathsf{dom}(E_1) = \mathsf{dom}(E_2)$ and environments are modeled as functions. □

A.2.1.3 Refined Range

We can also give a more refined definition of a range.

$$\mathsf{eranger}_p(E{\diamond}\ell) \stackrel{\text{def}}{=} \mathbf{lfp}^{\subseteq}_{\{E{\diamond}\ell\}} \lambda L.\, L \cup \{E'{\diamond}\ell' \mid \ell' \in Loc(p(\ell)) \wedge E \subseteq E' \wedge \forall n \in \mathsf{rng}(E' \backslash E).$$
$$\exists n' \in L.\, env(n) = env(n') \wedge loc(n) \in Loc(p(\ell))\}$$

The corresponding *range* is then defined as expected

$$\mathsf{ranger}_p(E{\diamond}\ell) \stackrel{\text{def}}{=} \mathsf{eranger}_p(E{\diamond}\ell) \cup \{n \mid n \in \mathsf{rng}(E') \wedge E'{\diamond}\_ \in \mathsf{eranger}_p(E{\diamond}\ell))\}$$

The same result about the range of computation can be shown.

Lemma 35. *Let M and M' be maps, E an environment, e^ℓ an expression of a program p, and k a fuel. If $\mathsf{domvalid}(\ell, p, E)$ and $\mathsf{step}_k[\![e^\ell]\!](E)(M) = M'$, then $\forall n. M(n) \neq M'(n) \implies n \in \mathsf{ranger}_p(E{\diamond}\ell)$.*

Proof. Similar to the proof of Lemma 32. □

However, something stronger can be said about the disjointness.

Lemma 36. *Let E be an environment, p be a program, ℓ_1 and ℓ_2 two locations such that $Loc(p(\ell_1)) \cap Loc(p(\ell_2)) = \emptyset$. Then, $\mathsf{range}_p(E \diamond \ell_1) \cap \mathsf{range}_p(E \diamond \ell_2) = \mathsf{rng}(E)$.*

Proof. Expression nodes of both ranges are clearly different by $Loc(p(\ell_1)) \cap Loc(p(\ell_2)) = \emptyset$. The environment nodes n not in $\mathsf{rng}(E)$ must have the associated variable $loc(n)$ from $Loc(p(\ell_1))$ and $Loc(p(\ell_2))$, respectively. The result then again follows from $Loc(p(\ell_1)) \cap Loc(p(\ell_2)) = \emptyset$. □

Similarly, the ranges of nodes whose environments are non-related have the same property.

Lemma 37. *Let E_1 and E_2 be two environments and ℓ a location of a program p. If $\mathsf{dom}(E_1) = \mathsf{dom}(E_2)$ and $E_1 \neq E_2$, then $\mathsf{range}_p(E \diamond \ell_1) \cap \mathsf{range}_p(E \diamond \ell_2) = \mathsf{rng}(E_1) \cap \mathsf{rng}(E_2)$.*

Proof. The environment nodes n in both ranges must have the associated environment $env(n)$ equal to the environment of some expression node in the corresponding range. These environments are by definition the extensions of E_1 and E_2, respectively, that have to be different as $\mathsf{dom}(E_1) = \mathsf{dom}(E_2)$ and environments are modeled as functions. Hence, the only shared nodes are those environment nodes that appear in both E_1 and E_2. □

A.2.1.4 Environment Consistency

We next formalize the notion of environment consistency for a map that states that no node can be visited with an environment holding a $\perp$ or ω value.

$$\mathsf{envcons}(M) \overset{\text{def}}{\iff} \forall n, n_x \in \mathsf{rng}(env(n)).\, M(n) \neq \perp \implies M(n_x) \notin \{\perp, \omega\}$$

Additionally, we also define consistency of an environment E subject to a map M

$$\mathsf{envcons}_M(E) \overset{\text{def}}{\iff} \forall n_x \in \mathsf{rng}(E).\, M(n_x) \notin \{\bot, \omega\}$$

A.2.1.5 Table IO Consistency

We next formalize the notion of input output consistency for tables.

$$\mathsf{tiocons}(v) \overset{\text{def}}{\iff} (v{\in}\{\bot,\omega\} \cup \mathit{Cons}) \vee (v{\in}\mathcal{T} \wedge \forall n_{\mathsf{cs}}{\in}v.\pi_1(v(n_{\mathsf{cs}}))=\bot \implies \pi_2(v(n_{\mathsf{cs}}))=\bot \\ \wedge\ \forall n_{\mathsf{cs}} \in v.\, \mathsf{tiocons}(\pi_1(v(n_{\mathsf{cs}}))) \wedge \mathsf{tiocons}(\pi_2(v(n_{\mathsf{cs}}))))$$

As expected, a map is input-output consistent $\mathsf{tiocons}(M)$ if all of its values are IO consistent.

A.2.1.6 Constant Consistency

We next formalize the notion of constant consistency that intuitively states that reachable nodes corresponding to constant and lambda expression must be either constants and tables, respectively, or an error value. Let p be a program. A map M is constant consistent $\mathsf{constcons}_p(M)$ if for every node n, where $M(n) = v$ and $\mathit{loc}(n) = \ell$, it must be

$$v \in \{\omega, \bot, c\} \qquad \text{if } p(\ell) = c$$

$$v \in \{\omega, \bot\} \cup \mathcal{T} \qquad \text{if } p(\ell) = \mu f.\lambda x.e$$

A.2.1.7 Error Consistency

We continue by formalizing the notion of the error absence of values that intuitively states the a given value is not an error and, if it is a table, it does not recursively

contain errors.

$$\mathsf{noerr}(v) \stackrel{\text{def}}{\iff} (v \in \{\bot\} \cup \mathit{Cons}) \vee (v \in \mathcal{T} \wedge \forall n_{\mathsf{cs}} \in v. \\ \mathsf{noerr}(\pi_1(v(n_{\mathsf{cs}}))) \wedge \mathsf{noerr}(\pi_2(v(n_{\mathsf{cs}}))))$$

A map M is error consistent $\mathsf{errcons}(M)$ if either (1) $M = M_\omega$ or (2) for every node n it must be that $\mathsf{noerr}(M(n))$.

A.2.1.8 Concrete Subtyping

Next, we introduce the notion of subtyping for our concrete values that resembles the subtyping on types.

$$v_1 <:^c v_2 \stackrel{\text{def}}{\iff} (v_2 = \bot) \vee (v_1 = v_2 \wedge v_1 \in \{\omega\} \cup \mathit{Cons}) \vee (v_1 \in \mathcal{T} \wedge v_2 \in \mathcal{T} \wedge \\ \forall n_{\mathsf{cs}} \in v_2.\, \pi_1(v_2(n_{\mathsf{cs}})) <:^c \pi_1(v_1(n_{\mathsf{cs}})) \wedge \pi_2(v_1(n_{\mathsf{cs}})) <:^c \pi_2(v_2(n_{\mathsf{cs}})))$$

A map M is subtype consistent $\mathsf{csub}_p(M)$, subject to a program p, iff for every node n, where $loc(n) = \ell$, it must be that

$$M(env(n)(x)) <:^c M(n) \qquad \textbf{if } p(\ell) = x \textbf{ and } x \in \mathsf{dom}(env(n))$$

$$M(env(n)\diamond\ell_1) <:^c [env(n)\diamond\ell_2 \mapsto \langle M(env(n)\diamond\ell_2), M(n)\rangle] \qquad \textbf{if } p(\ell) = (e_1^{\ell_1} e_2^{\ell_2}) \textbf{ and } M(env(n)\diamond\ell_1) \in \mathcal{T}$$

$$M(env(n)\diamond\ell_1) <:^c M(n) \qquad \textbf{if } p(\ell) = (e_0^{\ell_0} \,?\, e_1^{\ell_1} : e_2^{\ell_2}) \textbf{ and } M(env(n)\diamond\ell_0) = \mathit{true}$$

$$M(env(n)\diamond\ell_2) <:^c M(n) \qquad \textbf{if } p(\ell) = (e_0^{\ell_0} \,?\, e_1^{\ell_1} : e_2^{\ell_2}) \textbf{ and } M(env(n)\diamond\ell_0) = \mathit{false}$$

$$M(n) <:^c M(n_f) \wedge [n_{\mathsf{cs}} \mapsto \langle M(n_x), M(E_1\diamond\ell_1)\rangle] <:^c M(n) \qquad \textbf{if } p(\ell) = (\mu f.\lambda x.e_1^{\ell_1}),\ M(n) \in \mathcal{T}, n_{\mathsf{cs}} \in M(n),$$

$$n_x = env(n)\diamond n_{\mathsf{cs}}\diamond x,\ n_f = env(n)\diamond n_{\mathsf{cs}}\diamond f, \textbf{ and } E_1 = env(n).x : n_x.f : n_f$$

We can also show that propagation does not invalidate subtyping.

Lemma 38. *Let $v_1 \in \mathcal{V}$, $v_2 \in \mathcal{V}$, $v_3 \in \mathcal{V}$, and $v_4 \in \mathcal{V}$. If $v_1 <:^c v_2$ and $\langle v_3, v_4\rangle = \mathsf{prop}(v_1, v_2)$, then $v_3 <:^c v_4$.*

Proof. By structural induction on v_1 and v_2. □

Lemma 39. *Let $v_1 \in \mathcal{V}$, $v_2 \in \mathcal{V}$, $v_3 \in \mathcal{V}$, and $v_4 \in \mathcal{V}$. If $v_1 <:^c v_2$, $v_1 \notin \{\bot, \omega\}$, and $\langle v_3, v_4\rangle = \mathsf{prop}(v_1, v_2)$, then $v_3 \notin \{\bot, \omega\}$ and $v_4 \notin \{\bot, \omega\}$.*

Proof. Immediate from the definition of prop. □

We can even say something stronger.

Lemma 40. *Let $v_1 \in \mathcal{V}$, $v_2 \in \mathcal{V}$, $v_3 \in \mathcal{V}$, and $v_4 \in \mathcal{V}$. If $v_1 <:^c v_2$, $\mathsf{noerr}(v_1)$, $\mathsf{noerr}(v_2)$, and $\langle v_3, v_4\rangle = \mathsf{prop}(v_1, v_2)$, then $\mathsf{noerr}(v_3)$ and $\mathsf{noerr}(v_4)$.*

Proof. By structural induction on v_1 and v_2. The base cases are trivial. For the induction case where both v_1 and v_2 are tables, the argument follows by the i.h. □

A.2.1.9 Dataflow Invariant

We say a map M is data flow consistent subject to a program p if it satisfies the invariant $df_p(M)$ iff

$$\mathsf{fin}(M) \wedge \mathsf{csub}_p(M) \wedge \mathsf{errcons}(M) \wedge \mathsf{tiocons}(M) \wedge \mathsf{constcons}_p(M) \wedge \mathsf{envcons}(M)$$

A.2.1.10 Constant Compatibility

We next introduce formally the notion of constant compatibility between two concrete values that intuitively states that the join of values does not introduce error values.

$$\begin{aligned}\mathsf{ccomp}(v_1, v_2) \stackrel{\text{def}}{\iff} (v_1 \in \{\bot, \omega\}) \vee (v_2 \in \{\bot, \omega\}) \vee (v_1 \in \mathit{Cons} \wedge v \in \mathit{Cons} \wedge v_1 = v_2) \\ \vee (v_1 \in \mathcal{T} \wedge v_2 \in \mathcal{T} \wedge \forall n_{\mathsf{cs}}.\, \mathsf{ccomp}(\pi_1(v_2(n_{\mathsf{cs}})), \pi_1(v_1(n_{\mathsf{cs}}))) \wedge \\ \mathsf{ccomp}(\pi_2(v_1(n_{\mathsf{cs}})), \pi_2(v_2(n_{\mathsf{cs}}))))\end{aligned}$$

We also raise the definition of constant compatibility to a set of values V in the expected way $\mathsf{ccomp}(V) \stackrel{\text{def}}{\iff} \forall v_1, v_2 \in V.\, \mathsf{ccomp}(v_1, v_2)$. Note that constant compatibility is a reflexive relation.

Lemma 41. *Let v be a concrete value. Then,* $\mathsf{ccomp}(v, v)$.

Proof. By structural induction on v. □

Next, we say that two maps M_1 and M_2 are constant compatible $\mathsf{ccomp}(M_1, M_2)$ iff $\forall n.\, \mathsf{ccomp}(M_1(n), M_2(n))$. Given a set of maps $\hat{M}$, we define its constant compatibility $\mathsf{ccomp}(\hat{M})$ as $\forall M_1 \in \hat{M}, M_2 \in \hat{M}.\, \mathsf{ccomp}(M_1, M_2)$.

Constant compatibility preserves error consistency of maps over joins.

Lemma 42. *Let M_1 and M_2 be two concrete maps such that* $\mathsf{ccomp}(M_1, M_2)$, $\mathsf{errcons}(M_1)$, *and* $\mathsf{errcons}(M_2)$. *Then,* $\mathsf{errcons}(M_1 \dot{\sqcup} M_2)$.

Proof. By the definition of $\mathsf{ccomp}(v_1, v_2)$ on values v_1 and v_2, the join $v_1 \sqcup v_2$ cannot introduce errors. That is, if $\mathsf{noerr}(v_1)$ and $\mathsf{noerr}(v_2)$, then $\mathsf{noerr}(v_1 \sqcup v_2)$. This can be easily shown by structural induction on v_1 and v_2. The result then follows directly from the lemma assumptions. □

A similar result can be shown in the other direction.

Lemma 43. *Let M, M_1, and M_2 be concrete maps such that $M = M_1 \dot{\sqcup} M_2$. If* $\mathsf{errcons}(M)$ *and $M \neq M_\omega$, then* $\mathsf{ccomp}(M_1, M_2)$, $\mathsf{errcons}(M_1)$, *and* $\mathsf{errcons}(M_2)$.

Proof. Similar to the proof of Lemma 42. □

Clearly, the above result can be lifted to an arbitrary finite join of maps.

Lemma 44. *Let $\hat{M}$ be a finite set of concrete maps. If* $\mathsf{errcons}(\dot{\sqcup}\hat{M})$ *and $\dot{\sqcup}\hat{M} \neq M_\omega$, then* $\mathsf{ccomp}(\hat{M})$ *and* $\forall M \in \hat{M}.\, \mathsf{errcons}(M)$.

Proof. By induction on $|\hat{M}|$. The argument follows almost directly from Lemma 43. □

We next show that constant compatibility is preserved by prop of value chains.

Lemma 45. *Let $v_1 \in \mathcal{V}$, $v_2 \in \mathcal{V}$, $v_3 \in \mathcal{V}$, $v_1' \in \mathcal{V}$, $v_2' \in \mathcal{V}$, $v_2'' \in \mathcal{V}$, and $v_3' \in \mathcal{V}$. If $v_1 <:^c v_2$, $v_2 <:^c v_3$, $\langle v_1', v_2' \rangle = \mathsf{prop}(v_1, v_2)$, and $\langle v_2'', v_3' \rangle = \mathsf{prop}(v_2, v_3)$, then* $\mathsf{ccomp}(v_2', v_2'')$.

Proof. By structural induction on v_2. We first cover the base cases.

- $v_2 = \bot$. Then, $v_3 = \bot$ and hence $v_2'' = \bot$ by the def. of prop.

- $v_2 = c$. By $<:^c$, $v_1 = c$ and $v_3 = c \vee v_2 = \bot$. Therefore, $v_2' = v_2'' = c$.

- $v_2 = \omega$. By $<:^c$, $v_1 = \omega$ and $v_3 = \omega \vee v_2 = \bot$. Therefore, $v_2' = v_2'' = \omega$.

Let us now consider the case when $v_2 \in \mathcal{T}$. By $<:^c$, it must be that $v_1 \in \mathcal{T}$. If $v_3 = \bot$, then $v_2'' = v_2$ and $v_2' = v_2 \sqsubseteq v_2'$ by Lemma 26. The result then follows trivially. Hence, we are left with the case where $v_3 \in \mathcal{T}$. Let us focus on any call site n_{cs} and let $v_1(n_{\mathsf{cs}}) = \langle v_{1i}, v_{1o}\rangle$, $v_2(n_{\mathsf{cs}}) = \langle v_{2i}, v_{2o}\rangle$, and $v_3(n_{\mathsf{cs}}) = \langle v_{3i}, v_{3o}\rangle$. By the def. of prop, we have that $v_1' \in \mathcal{T}$, $v_2' \in \mathcal{T}$, $v_2'' \in \mathcal{T}$, and $v_3' \in \mathcal{T}$. Let then $v_1'(n_{\mathsf{cs}}) = \langle v_{1i}', v_{1o}'\rangle$, $v_2'(n_{\mathsf{cs}}) = \langle v_{2i}', v_{2o}'\rangle$, $v_2''(n_{\mathsf{cs}}) = \langle v_{2i}'', v_{2o}''\rangle$, and $v_3'(n_{\mathsf{cs}}) = \langle v_{3i}', v_{3o}'\rangle$. By the def. of prop, we have to only consider the cases of whether $n_{\mathsf{cs}} \in v_2$ and $n_{\mathsf{cs}} \in v_3$.

- $n_{\mathsf{cs}} \notin v_2, n_{\mathsf{cs}} \notin v_3$. By the def. of prop, we have $v_{2i}' = v_{2i}'' = v_{2i}$ and $v_{2o}' = v_{2o}'' = v_{2o}$. The argument is then straightforward.

- $n_{\mathsf{cs}} \notin v_2, n_{\mathsf{cs}} \in v_3$. By the def. of prop, we have $v_{2i}' = v_{2i}$ and $v_{2o}' = v_{2o}$. By Lemma 26, we have $v_{2i} \sqsubseteq v_{2i}''$ and $v_{2o} \sqsubseteq v_{2o}''$. The argument then follows easily.

- $n_{\mathsf{cs}} \in v_2, n_{\mathsf{cs}} \notin v_3$. By the def. of prop, we have $v_{2i}'' = v_{2i}$ and $v_{2o}'' = v_{2o}$. By Lemma 26, we have $v_{2i} \sqsubseteq v_{2i}'$ and $v_{2o} \sqsubseteq v_{2o}'$. The argument then follows easily.

- $n_{\mathsf{cs}} \in v_2, n_{\mathsf{cs}} \in v_3$. First, by $<:^c$ we have $v_{3i} <:^c v_{2i}$ and $v_{2i} <:^c v_{1i}$. Next, we also have $v_{1o} <:^c v_{2o}$ and $v_{2o} <:^c v_{3o}$. By the def. of prop, we next have $\langle v_{2i}', v_{1i}'\rangle = \mathsf{prop}(v_{2i}, v_{1i})$, $\langle v_{3i}', v_{2i}''\rangle = \mathsf{prop}(v_{3i}, v_{2i})$, $\langle v_{1o}', v_{2o}'\rangle = \mathsf{prop}(v_{1o}, v_{2o})$, and $\langle v_{2o}'', v_{3o}'\rangle = \mathsf{prop}(v_{2o}, v_{3o})$. By the i.h., we then have $\mathsf{ccomp}(v_{2i}', v_{2i}'')$ and $\mathsf{ccomp}(v_{2o}', v_{2o}'')$. The result then follows from the def. of ccomp.

□

A.2.2 Inductive Properties

Given a fixed program m, we define the family of inductive properties P, indexed by fuels k, as a set of triples consisting of locations in the program, concrete maps, and environments that is closed subject to the concrete transformer step. More precisely, given a program m and a fuel k, we say that $P_k \in \wp(\mathit{Loc} \times \mathcal{M} \times \mathcal{E})$ is an *inductive invariant* (of step) if

$$(\ell, M, E) \in P_k \implies (\ell, \mathsf{step}_k[\![e^\ell]\!](E)(M), E) \in P_k$$

where $e^\ell \in m$ and (e, E) is well-formed. We additionally require that all potential recursive calls of step for the subexpressions of e are closed subject to P, as follows:

- $e^\ell = e_1^{\ell_1} e_2^{\ell_2}$.

$$\frac{(\ell, M, E) \in P_{k+1}}{(\ell_1, M, E) \in P_k}$$

$$\frac{(\ell, M, E) \in P_{k+1} \qquad (\ell_1, M_1, E) \in P_k \qquad M_1 = \mathsf{step}_k[\![e_1^{\ell_1}]\!](E)(M) \quad M_1(E \diamond \ell_1) \notin \{\bot, \omega\}}{(\ell_2, M_1, E) \in P_k}$$

- $e^\ell = (e_0^{\ell_0}\,?\,e_1^{\ell_1} : e_2^{\ell_2})^\ell$.

$$\frac{(\ell, M, E) \in P_{k+1}}{(\ell_0, M, E) \in P_k}$$

$$\frac{(\ell, M, E) \in P_{k+1} \qquad (\ell_0, M_0, E) \in P_k \qquad M_0 = \mathsf{step}_k[\![e_0^{\ell_0}]\!](E)(M) \quad M_0(E{\diamond}\ell_0) = \mathit{true}}{(\ell_1, M, E) \in P_k}$$

$$\frac{(\ell, M, E) \in P_{k+1} \qquad (\ell_0, M_0, E) \in P_k \qquad M_0 = \mathsf{step}_k[\![e_0^{\ell_0}]\!](E)(M) \quad M_0(E{\diamond}\ell_0) = \mathit{false}}{(\ell_2, M, E) \in P_k}$$

- $e^\ell = (\mu f.\lambda x.e_1^{\ell_1})^\ell$.

$$\frac{\begin{array}{c} M(\ell{\diamond}E) = T \qquad n_{\mathsf{cs}} \in T \qquad \pi_1(T(n_{\mathsf{cs}})) \notin \{\bot, \omega\} \\ n_x = E{\diamond}n_{\mathsf{cs}}{\diamond}x \quad n_f = E{\diamond}n_{\mathsf{cs}}{\diamond}f \quad E_1 = E.x : n_x.f : n_f \\ \langle T'_x, T' \rangle = \mathsf{prop}([n_{\mathsf{cs}} \mapsto \langle M(n_x), M(E_1{\diamond}\ell_1)\rangle], T) \\ \langle T'', T'_f \rangle = \mathsf{prop}(T, M(n_f)) \qquad \langle v'_x, v'_1 \rangle = T'x(n_{\mathsf{cs}}) \\ M_1 = M[E{\diamond}\ell \mapsto T' \sqcup T'', n_x \mapsto v'_x, n_f \mapsto T'_f, E_1{\diamond}\ell_1 \mapsto v'_1] \\ (\ell, M, E) \in P_{k+1} \end{array}}{(\ell_1, M_1, E_1) \in P_k}$$

Given a program e^ℓ, we define its *strongest inductive invariant* family I as the smallest inductive property of e^ℓ such that $(\ell, M_\bot, \emptyset) \in I_k$, for every k.

A.3 Path Data Flow Semantics

In this section, we introduce an alternative definition of the concrete data flow semantics that, besides data flow values, also keeps track of paths the data flow values take. This semantics can be thought of as accumulating ghost information of the data flow semantics. We will use this ghost information to show several useful properties of the original data flow semantics.

A.3.1 Semantic Domains

We define paths $p \in \mathit{Path}$ as a sequence of nodes $\mathcal{N}^+$. The path data flow values are defined in a similar way as are the data flow values except that path information is maintained as well.

$$w \in \mathcal{V}^{\mathsf{w}} \ ::= \ \bot \mid \omega \mid \langle c, p\rangle \mid \langle T^{\mathsf{w}}, p\rangle \qquad \text{path (data flow) values}$$

$$T^{\mathsf{w}} \in \mathcal{T}^{\mathsf{w}} \ \stackrel{\text{def}}{=} \ \mathit{Path} \to \mathcal{V}^{\mathsf{w}} \times \mathcal{V}^{\mathsf{w}} \qquad \text{path tables}$$

$$M^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}} \ \stackrel{\text{def}}{=} \ \mathcal{N} \to \mathcal{V}^{\mathsf{w}} \qquad \text{path execution maps}$$

The set $\mathit{Cons}^{\mathsf{w}} \stackrel{\text{def}}{=} \mathit{Cons} \times \mathit{Path}$ denotes the set of all constant path values that also Boolean path values $\mathit{Bool}^{\mathsf{w}} \stackrel{\text{def}}{=} \{\mathit{true}, \mathit{false}\} \times \mathit{Path}$.

The computational partial ordering on path values is defined similarly to the ordering on the ordinary the data flow values. In addition, we require that corresponding paths are related by the prefix relation $\preceq$.

$$\begin{aligned} w_1 \sqsubseteq^{\mathsf{w}} w_2 \stackrel{\text{def}}{\Longleftrightarrow} \ & w_1 = \bot \ \vee \ w_2 = \omega \ \vee \\ & (w_1 = \langle c, p_1\rangle \wedge w_2 = \langle c, p_2\rangle \wedge c \in \mathit{Cons} \wedge p_1, p_2 \in \mathit{Path} \wedge p_1 \preceq p_2) \ \vee \\ & w_1{=}\langle T_1^{\mathsf{w}}, p_1\rangle \wedge w_2{=}\langle T_2^{\mathsf{w}}, p_2\rangle \wedge T_1^{\mathsf{w}}, T_2^{\mathsf{w}}{\in}\mathcal{T}^{\mathsf{w}} \wedge p_1, p_2 \in \mathit{Path} \wedge p_1 \preceq p_2 \\ & \wedge \forall n_{\mathsf{cs}}.\, T_1^{\mathsf{w}}(n_{\mathsf{cs}}) \stackrel{\cdot}{\sqsubseteq}^{\mathsf{w}} T_2^{\mathsf{w}}(n_{\mathsf{cs}}) \end{aligned}$$

The join on the path values is defined as follows.

$$\langle c, p_1\rangle \sqcup^{\mathsf{w}} \langle c, p\rangle \stackrel{\text{def}}{=} \langle c, p\rangle \qquad \text{if } p_1 \preceq p_2 \vee p_2 \preceq p_1 \text{ and } p = p_1 \preceq p_2 \,?\, p_2 : p_1$$

$$\langle T_1^{\mathsf{w}}, p_1\rangle \sqcup^{\mathsf{w}} \langle T_2^{\mathsf{w}}, p_2\rangle \stackrel{\text{def}}{=} \langle \Lambda n_{\mathsf{cs}}.\, T_1^{\mathsf{cr}}(n_{\mathsf{cs}}) \dot{\sqcup}^{\mathsf{w}} T_2^{\mathsf{cr}}(n_{\mathsf{cs}}), p\rangle \qquad \text{if } p_1 \preceq p_2 \vee p_2 \preceq p_1 \text{ and } p = p_1 \preceq p_2 \,?\, p_2 : p_1$$

$$\bot \sqcup^{\mathsf{w}} w \stackrel{\text{def}}{=} w \qquad w \sqcup^{\mathsf{w}} \bot \stackrel{\text{def}}{=} w \qquad w_1 \sqcup^{\mathsf{w}} w_2 \stackrel{\text{def}}{=} \omega \quad \text{(otherwise)}$$

The meet on path values is defined similarly, as expected.

Lemma 46. *The join $\sqcup^{\mathsf{w}}$ and $\sqcap^{\mathsf{w}}$ on path values are lub and glb operators subject to the $\sqsubseteq^{\mathsf{w}}$ ordering.*

Proof. Similar to the proof of Lemma 1. □

An empty table that maps every call site to the pair of $\bot$ values is denoted by $T_\bot^{\mathsf{w}}$. We reuse the notation for denoting singleton tables from the data flow semantics. The same applies for the notation $n_{\mathsf{cs}} \in T^{\mathsf{w}}$ denoting if some non-$\bot$ input has been seen for the call site n_{cs} in a given table T^{w}. A path value map $M^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}} \stackrel{\text{def}}{=} \mathcal{N} \to \mathcal{V}^{\mathsf{w}}$ is a function from nodes to path values, as expected. Path maps $M_\bot^{\mathsf{w}}$ and M_ω^{w} assign to each node the $\bot$ and ω value, respectively.

We also define a function paths that collects paths of a value

$$\mathsf{paths}(\bot) \stackrel{\text{def}}{=} \emptyset \qquad \mathsf{paths}(\omega) \stackrel{\text{def}}{=} \emptyset \qquad \mathsf{paths}(\langle c, p\rangle) \stackrel{\text{def}}{=} \{p\}$$

$$\mathsf{paths}(\langle T^{\mathsf{w}}, p\rangle) \stackrel{\text{def}}{=} \{p\} \cup \bigcup_{p' \in T^{\mathsf{w}}} \{p'\} \cup \mathsf{paths}(\pi_1(T^{\mathsf{w}}(p'))) \cup \mathsf{paths}(\pi_2(T^{\mathsf{w}}(p')))$$

All paths $\mathsf{paths}(M^{\mathsf{w}})$ of path map M^{w} are defined as the union of all paths of values in the map, as expected.

A.3.2 Data Propagation

The data propagation $\mathsf{prop}^{\mathsf{w}} : \mathcal{V}^{\mathsf{w}} \times \mathcal{V}^{\mathsf{w}} \to \mathcal{V}^{\mathsf{w}} \times \mathcal{V}^{\mathsf{w}}$ on path values closely resembles the data propagation on the plain data flow values. The main difference is that a meaningful propagation only happens in the case when the paths of the given values match by the $\preceq$ relation. Otherwise, the propagation just sets both values to top ω. As we shall see shortly, such cases never happen during the computation.

$$
\begin{array}{l}
\mathsf{prop}^{\mathsf{w}}(\langle T_1^{\mathsf{w}}, p_1\rangle, \langle T_2^{\mathsf{w}}, p_2\rangle) \stackrel{\text{def}}{=} \qquad \text{if } p_1 \preceq p_2 \\
\quad \textbf{let } T' = \Lambda p. \\
\qquad \textbf{if } p \notin T_2^{\mathsf{w}} \textbf{ then } \langle T_1^{\mathsf{w}}(p), T_2^{\mathsf{w}}(p)\rangle \textbf{ else} \\
\qquad \textbf{let } \langle w_{1i}, w_{1o}\rangle = T_1^{\mathsf{w}}(n_{\mathsf{cs}})\textbf{;}\ \langle w_{2i}, w_{2o}\rangle = T_2^{\mathsf{w}}(n_{\mathsf{cs}}) \\
\qquad\quad \langle w'_{2i}, w'_{1i}\rangle = \mathsf{prop}^{\mathsf{w}}(w_{2i}, w_{1i})\textbf{;}\ \langle w'_{1o}, w'_{2o}\rangle = \mathsf{prop}^{\mathsf{w}}(w_{1o}, w_{2o}) \\
\qquad \textbf{in } (\langle w'_{1i}, w'_{1o}\rangle, \langle w'_{2i}, w'_{2o}\rangle) \\
\quad \textbf{in } \langle \Lambda p.\ \pi_1(T'(p)), \Lambda p.\ \pi_2(T'(p))\rangle \\
\mathsf{prop}^{\mathsf{w}}(\langle T^{\mathsf{w}}, p\rangle, \bot) \stackrel{\text{def}}{=} (\langle T^{\mathsf{w}}, p\rangle, \langle T_\bot^{\mathsf{w}}, p\rangle) \qquad \mathsf{prop}^{\mathsf{w}}(\langle T^{\mathsf{w}}, p\rangle, \omega) \stackrel{\text{def}}{=} \langle \omega, \omega\rangle \\
\mathsf{prop}^{\mathsf{w}}(\langle T_1^{\mathsf{w}}, p_1\rangle, \langle T_2^{\mathsf{w}}, p_2\rangle) \stackrel{\text{def}}{=} \langle \omega, \omega\rangle \qquad \text{if } p_1 \not\preceq p_2 \\
\mathsf{prop}^{\mathsf{w}}(\langle c, p_1\rangle, \langle c, p_2\rangle) \stackrel{\text{def}}{=} (\langle c, p_1\rangle, \langle c, p_2\rangle) \qquad \text{if } p_1 \preceq p_2 \\
\mathsf{prop}^{\mathsf{w}}(\langle c, p_1\rangle, \langle c, p_2\rangle) \stackrel{\text{def}}{=} \langle \omega, \omega\rangle \qquad \text{if } p_1 \not\preceq p_2 \\
\mathsf{prop}^{\mathsf{w}}(w_1, w_2) \stackrel{\text{def}}{=} \langle w_1, w_1 \sqcup^{\mathsf{w}} w_2\rangle \qquad \textbf{(otherwise)}
\end{array}
$$

Lemma 47. $\mathsf{prop}^{\mathsf{w}}$ *is increasing and monotone.*

Proof. By structural induction on the arguments of $\mathsf{prop}^{\mathsf{w}}$. The argument for the increasing property is rather straightforward. Regarding monotonicity, arguably the most interesting case is that of constants. Suppose $w_1 = \langle c, p_1\rangle$, $w_2 = \langle c, p_2\rangle$,

$w_3 = \langle c, p_3 \rangle$, and $w_4 = \langle c, p_4 \rangle$. Let $w_1 \sqsubseteq^{\mathsf{w}} w_3$ and $w_2 \sqsubseteq^{\mathsf{w}} w_4$. First, if $p_1 \not\preceq p_2$ then $p_3 \not\preceq p_4$ since by the definition of $\sqsubseteq^{\mathsf{w}}$ it must be that $p_1 \preceq p_3$ and $p_2 \preceq p_4$. In this case, the argument is trivial. If $p_1 \preceq p_2$ the argument is again simple as the values w_1 and w_2 do not get updated. Assume now that $w_2 = \bot$ and $w_4 = \bot$. The argument is again straightforward. Lastly, assume that only $w_2 = \bot$ (note that we cannot have the case when only $w_4 = \bot$ since we assume for the purpose of showing monotonicity that $w_2 \sqsubseteq^{\mathsf{w}} w_4$). The propagation will result in an updated value $w_2' = w_1$. If $p_3 \not\sqsubseteq^{\mathsf{w}} p_4$, the argument is easy. Otherwise, since $p_1 \preceq p_3$ and $p_3 \preceq p_4$, then it also must be that $p_1 \preceq p_4$. The same reasoning, together with the induction hypothesis, applies to the case of tables. □

It can be also shown that the propagation does not invent new paths. That is, propagation between two values results in two values where each resulting value is either top ω or a new path values that has the same path as the original path value.

Lemma 48. *Let $\langle w_1', w_2' \rangle = \mathsf{prop}^{\mathsf{w}}(w_1, w_2)$ for some path values w_1, w_1', w_2, and w_2'. Then, if $w_1 = \langle \_, p \rangle$ for some path p, then $w_1' = \omega$ or $w_2' = \langle \_, p \rangle$. The same hold for w_2 and w_2'.*

Proof. The proof goes by structural induction on both w_1 and w_2. □

A.3.3 Transformer

The transformer $\mathsf{step}^{\mathsf{w}} : \mathbb{N} \to \lambda^d \to \mathcal{E} \to \mathcal{M}^{\mathsf{w}} \to \mathcal{M}^{\mathsf{w}}$ of the paths semantics works similarly as the transformer for the original data flow semantics. The only difference is that the path information needs to be properly handled when creating new values. For that purpose, we introduce a function for extending path information

in path values with nodes.

$$\mathsf{ext}(p, n) \stackrel{\text{def}}{=} p \qquad \text{if } \mathsf{top}(p) = n \qquad \mathsf{ext}(p, n) \stackrel{\text{def}}{=} p \cdot n \qquad \text{(otherwise)}$$

$$\mathsf{ext}(\langle T^{\mathsf{w}}, p\rangle, n) \stackrel{\text{def}}{=} \langle T^{\mathsf{w}}, \mathsf{ext}(p, n)\rangle \qquad \mathsf{ext}(\langle c, p\rangle, n) \stackrel{\text{def}}{=} \langle c, \mathsf{ext}(p, n)\rangle$$

$$\mathsf{ext}(\omega, n) \stackrel{\text{def}}{=} \omega \qquad \mathsf{ext}(\bot, n) \stackrel{\text{def}}{=} \bot$$

Here, we use top to denote the top/final node of a path. Further, $\cdot$ operation concatenates two given paths. Note that we also consider a node as a path. Finally, observe that the operation ext on paths is idempotent. Also, let path be a partial function on path values that returns a path associated with the value, if such a path exists. Note that path semantics models call sites as paths rather than nodes, but the variable nodes are created only using the top of the call site path. As we shall see later, the actual paths observed during the computation that have unique top nodes.

Lemma 49. $\mathsf{step}^{\mathsf{w}}$ *is increasing and monotone.*

Proof. The increasing property follows from the increasing property of $\mathsf{prop}^{\mathsf{w}}$ (Lemma 47). The proof of monotonicity closely follows that of Lemma 28. □

A.3.4 Additional Properties

We next introduce several useful properties related to the path semantics.

A.3.4.1 Computation Range

The results regarding the computation range are the same as for the ordinary data flow semantics.

Lemma 50. *Let* M^{w} *and* M_0^{w} *be path maps,* E *an environment,* e^{ℓ} *an expression*

$$\mathsf{step}_0^{\mathsf{w}}[\![e]\!](E)(M^{\mathsf{w}}) \stackrel{\text{def}}{=} M^{\mathsf{w}}$$

$$\mathsf{step}_{k+1}^{\mathsf{w}}[\![c^{\ell}]\!](E)(M^{\mathsf{w}}) \stackrel{\text{def}}{=} M^{\mathsf{w}}[E{\diamond}\ell \to M^{\mathsf{w}}(E{\diamond}\ell) \sqcup \langle c, E{\diamond}\ell\rangle]$$

$$\begin{array}{l}\mathsf{step}_{k+1}^{\mathsf{w}}[\![x^{\ell}]\!](E)(M^{\mathsf{w}}) \stackrel{\text{def}}{=}\\ \quad \textbf{let } \langle w'_x, w'\rangle = \mathsf{prop}^{\mathsf{w}}(M^{\mathsf{w}}(E(x)), M^{\mathsf{w}}(E{\diamond}\ell))\\ \quad \textbf{in } M^{\mathsf{w}}[E(x) \mapsto w'_x, E{\diamond}\ell \mapsto \mathsf{ext}(w', E{\diamond}\ell)]\end{array}$$

$$\begin{array}{l}\mathsf{step}_{k+1}^{\mathsf{w}}[\![(e_1^{\ell_1} e_2^{\ell_2})^{\ell}]\!](E)(M^{\mathsf{w}}) \stackrel{\text{def}}{=}\\ \quad \textbf{let } M_1^{\mathsf{w}} = \mathsf{step}_k^{\mathsf{w}}[\![e_1]\!](E)(M^{\mathsf{w}}); \; w_1 =_{M_1^{\mathsf{w}}} M_1^{\mathsf{w}}(E{\diamond}\ell_1) \textbf{ in}\\ \quad \textbf{if } w_1 \notin \mathcal{T}^{\mathsf{w}} \textbf{ then } M_{\omega}^{\mathsf{w}} \textbf{ else}\\ \quad\; \textbf{let } M_2^{\mathsf{w}} = \mathsf{step}_k^{\mathsf{w}}[\![e_2]\!](E)(M_1^{\mathsf{w}}); \; w_2 =_{M_2^{\mathsf{w}}} M_2^{\mathsf{w}}(E{\diamond}\ell_2) \textbf{ in}\\ \quad\; \textbf{let } \langle w'_1, T^{\mathsf{w}}\rangle = \mathsf{prop}^{\mathsf{w}}(w_1, [\mathsf{path}(w_1) \mapsto \langle w_2, M_2^{\mathsf{w}}(E{\diamond}\ell)\rangle])\\ \qquad \langle w'_2, w'\rangle = T^{\mathsf{w}}(\mathsf{path}(w_1))\\ \quad\; \textbf{in } M_2^{\mathsf{w}}[E{\diamond}\ell_1 \mapsto w'_1, E{\diamond}\ell_2 \mapsto w'_2, E{\diamond}\ell \mapsto \mathsf{ext}(w', E{\diamond}\ell)]\end{array}$$

$$\begin{array}{l}\mathsf{step}_{k+1}^{\mathsf{w}}[\![(\mu f.\lambda x.e_1^{\ell_1})^{\ell}]\!](E)(M^{\mathsf{w}}) \stackrel{\text{def}}{=}\\ \quad \textbf{let } T_w =_{M^{\mathsf{w}}} \langle T_{\perp}^{\mathsf{w}}, E{\diamond}\ell\rangle \sqcup^{\mathsf{w}} M^{\mathsf{w}}(E{\diamond}\ell) \textbf{ in}\\ \quad \textbf{let } \overline{M^{\mathsf{w}}} = \Lambda p \in T_w. \textbf{ if } \pi_1(T^{\mathsf{w}}(n_{\mathsf{cs}})) = \omega \textbf{ then } M_{\omega}^{\mathsf{w}} \textbf{ else}\\ \quad\; \textbf{let } n_x{=}E{\diamond}\mathsf{top}(p){\diamond}x; \; n_f{=}E{\diamond}\mathsf{top}(p){\diamond}f; \; E_1{=}E[x{\mapsto}n_x, f{\mapsto}n_f]\\ \qquad \langle T_x^{\mathsf{w}}, T_0^{\mathsf{w}}\rangle = \mathsf{prop}^{\mathsf{w}}([p \mapsto \langle M^{\mathsf{w}}(n_x), M^{\mathsf{w}}(E_1{\diamond}\ell_1)\rangle], T_w)\\ \qquad \langle T_2^{\mathsf{w}}, T_f^{\mathsf{w}}\rangle = \mathsf{prop}^{\mathsf{w}}(T_w, M^{\mathsf{w}}(n_f)); \; \langle w'_x, w'_1\rangle = T_x^{\mathsf{w}}(p)\\ \qquad M_1^{\mathsf{w}} = M^{\mathsf{w}}[E{\diamond}\ell{\mapsto}T_1^{\mathsf{w}} \sqcup^{\mathsf{w}} T_2^{\mathsf{w}}, n_f{\mapsto}\mathsf{ext}(T_f^{\mathsf{w}}, n_f), n_x{\mapsto}\mathsf{ext}(w'_x, n_x), E_1{\diamond}\ell_1{\mapsto}w'_1]\\ \quad\; \textbf{in } \mathsf{step}_k^{\mathsf{w}}[\![e_1]\!](E_1)(M_1^{\mathsf{w}})\\ \quad \textbf{in } M^{\mathsf{w}}[E{\diamond}\ell \mapsto T^{\mathsf{w}}] \dot{\sqcup}^{\mathsf{w}} \dot{\bigsqcup}_{p \in T^{\mathsf{w}}}^{\mathsf{w}} \overline{M}(p)\end{array}$$

$$\begin{array}{l}\mathsf{step}_{k+1}^{\mathsf{w}}[\![(e_0^{\ell_0} \,?\, e_1^{\ell_1} : e_2^{\ell_2})^{\ell}]\!](E)(M^{\mathsf{w}}) \stackrel{\text{def}}{=}\\ \quad \textbf{let } M_0^{\mathsf{w}} = \mathsf{step}_k^{\mathsf{w}}[\![e_0]\!](E)(M^{\mathsf{w}}); \; w_0 =_{M_0^{\mathsf{w}}} M_0^{\mathsf{w}}(E{\diamond}\ell_0) \textbf{ in}\\ \quad \textbf{if } w_0 \notin \mathit{Bool}^{\mathsf{w}} \textbf{ then } M_{\omega}^{\mathsf{w}} \textbf{ else}\\ \quad\; \textbf{let } b = \textbf{if } w_0 = \langle \mathit{true}, \_\rangle \textbf{ then } 1 \textbf{ else } 2 \textbf{ in}\\ \quad\; \textbf{let } M_b^{\mathsf{w}} = \mathsf{step}_k^{\mathsf{w}}[\![e_b]\!](E)(M_0^{\mathsf{w}}); \; w_b =_{M_b^{\mathsf{w}}} M_b^{\mathsf{w}}(E{\diamond}\ell_b) \textbf{ in}\\ \quad\; \textbf{let } \langle w'_b, w'\rangle = \mathsf{prop}^{\mathsf{w}}(w_b, M_b^{\mathsf{w}}(E{\diamond}\ell))\\ \quad\; \textbf{in } M_b^{\mathsf{w}}[E{\diamond}\ell \mapsto \mathsf{ext}(w', E{\diamond}\ell), E{\diamond}\ell_b \mapsto w'_b]\end{array}$$

of a program e' where (e, E) is well-formed, and k a fuel. If $\mathsf{domvalid}(\ell, e', E)$ *and* $\mathsf{step}_k^{\mathsf{w}}[\![e^{\ell}]\!](E)(M^{\mathsf{w}}) = M_0^{\mathsf{w}}$ *then* $\forall n. M^{\mathsf{w}}(n) \neq M_0^{\mathsf{w}}(n) \implies n \in \mathsf{ranger}_{e'}(E{\diamond}\ell)$.

Proof. Similar to the proof of Lemma 35. □

Lemma 51. *Let E be an environment, e' be a program, ℓ_1 and ℓ_2 two locations such that $Loc(e'(\ell_1)) \cap Loc(e'(\ell_2)) = \emptyset$. Then, $\mathsf{range}_{e'}(E{\diamond}\ell_1) \cap \mathsf{range}_{e'}(E{\diamond}\ell_2) = \mathsf{rng}(E)$.*

Proof. Similar to the proof of Lemma 36. □

Lemma 52. *Let E_1 and E_2 be two environments and ℓ a location of a program e'. If $\mathsf{dom}(E_1) = \mathsf{dom}(E_2)$ and $E_1 \neq E_2$, then $\mathsf{range}_{e'}(E{\diamond}\ell_1) \cap \mathsf{range}_{e'}(E{\diamond}\ell_2) = \mathsf{rng}(E_1) \cap \mathsf{rng}(E_2)$.*

Proof. Similar to the proof of Lemma 37. □

A.3.4.2 Finiteness

We say a value w is finite $\mathsf{fin}(w)$ if it is either a ground value or a table that has seen only a finite number of inputs, just as in the case of regular data flow semantics.

$$\mathsf{fin}(w) \stackrel{\text{def}}{=} (w \in \{\bot, \omega\} \cup \mathit{Cons}^{\mathsf{w}}) \vee (\exists T^{\mathsf{w}}.\, w = \langle T^{\mathsf{w}}, \rangle \in \mathcal{T}^{\mathsf{w}} \wedge |\{p \mid p \in T^{\mathsf{w}}\}| \in \mathbb{N} \wedge \forall p \in T^{\mathsf{w}}.\, \mathsf{fin}(\pi_1(T^{\mathsf{w}}(p))) \wedge \mathsf{fin}(\pi_2(T^{\mathsf{w}}(p))))$$

As expected, $\mathsf{prop}^{\mathsf{w}}$ does not affect the finiteness of values. The same clearly holds for joins $\sqcup$.

Lemma 53. *Let w_1, w_2, w_3, and w_4 be path values. If $\mathsf{fin}(w_1)$, $\mathsf{fin}(w_2)$, and $\langle w_3, w_4 \rangle = \mathsf{prop}^{\mathsf{w}}(w_1, w_2)$, then $\mathsf{fin}(w_3)$ and $\mathsf{fin}(w_4)$.*

Proof. Similar to the proof of Lemma 30. □

We say a path map M^{w} is finite $\mathsf{fin}(M^{\mathsf{w}})$ if all of its values are finite and the set of reachable nodes is finite as well $|\{n \mid M^{\mathsf{w}}(n) \neq \bot\}| \in \mathbb{N}$. It can be shown that $\mathsf{step}^{\mathsf{w}}$ preserves the finiteness of maps.

Lemma 54. *Let* M^{w} *and* M_0^{w} *be maps,* E *an environment,* e^{ℓ} *an expression where* (e, E) *is well-formed, and* k *a fuel. If* $\mathsf{fin}(M^{\mathsf{w}})$ *and* $\mathsf{step}_k^{\mathsf{w}}[\![e^{\ell}]\!](E)(M^{\mathsf{w}}) = M_0^{\mathsf{w}}$*, then* $\mathsf{fin}(M_0^{\mathsf{w}})$.

Proof. Similar to the proof of Lemma 31. □

A.3.4.3 Top of the Path Consistency

We say a path map M^{w} is consistent subject to the top of the paths $\mathsf{tpcons}(M^{\mathsf{w}})$ iff for every node n it is the case that $M^{\mathsf{w}}(n) \in \{\bot, \omega\}$ or $M^{\mathsf{w}}(n) = \langle \_, p \rangle$ for some path p where $\mathsf{path}(p) = n$.

The consistency of maps subject to the top of paths is an preserved by the path semantics transformer.

Lemma 55. *Let* $M^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}}$ *where* $\mathsf{tpcons}(M^{\mathsf{w}})$*,* $M_0^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}}$*,* e^{ℓ} *an expression,* $k \in \mathbb{N}$*, and* E *an environment where* (e, E) *is well-formed. If* $\mathsf{step}_k^{\mathsf{w}}[\![e^{\ell}]\!](E)(M^{\mathsf{w}}) = M_0^{\mathsf{w}}$*, then* $\mathsf{tpcons}(M_0^{\mathsf{w}})$.

Proof. The proof goes by induction on k and e. The argument follows from the basic properties of the join on path values, Lemma 48, and the idempotency of ext operation. □

Top of the path consistency can be used to establish an interesting and useful property of the concrete transformer $\mathsf{step}^{\mathsf{w}}$. Once path has been inferred for some value in the map, then after applying the transformer, the updated value is either top ω or a new path value but with the same associated path.

Lemma 56. *Let* $n \in \mathcal{N}$*,* $M^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}}$ *where* $\mathsf{tpcons}(M^{\mathsf{w}})$*,* $M_0^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}}$*,* e^{ℓ} *an expression,* $k \in \mathbb{N}$*, and* E *an environment where* (e, E) *is well-formed. Suppose*

$\mathsf{step}^{\mathsf{w}}_k[\![e^\ell]\!](E)(M^{\mathsf{w}}) = M^{\mathsf{w}}_0$. *If* $M^{\mathsf{w}}(n) = \langle \_, p\rangle$ *for some path* p*, then* $M^{\mathsf{w}}(n_0) = \langle \_, p\rangle$ *or* $M^{\mathsf{w}}_0(n) = \omega$.

Proof. The proof goes by induction on k and e. The argument follows from the basic properties of the join on path values and Lemma 48. The top of the path consistency property together with Lemma 55 is needed to prove that the calls to ext on existing paths do not create new paths. □

A.3.4.4 Call Site Path consistency

We next define the notion of call site path consistency intuitively capturing the idea that a table can be called at call site path that refers to the future points reached by the table, i.e., call site paths must be an extension of the path associated with the table. A path value w is call site path consistent $\mathsf{pcscons}(w)$ iff

$$(w \in \{\bot, \omega\} \cup \mathit{Cons}^{\mathsf{w}}) \vee$$

$$\exists T^{\mathsf{w}}, p.w = \langle T^{\mathsf{w}}, p\rangle \wedge \forall p' \in T^{\mathsf{w}}.\, p \preceq p' \wedge \mathsf{pcscons}(\pi_1(T^{\mathsf{w}}(p'))) \wedge \mathsf{pcscons}(\pi_2(T^{\mathsf{w}}(p')))$$

We raise the definition of call site path consistency to the maps. That is, a path map M^{w} is said to be call site path consistent $\mathsf{pcscons}(M^{\mathsf{w}})$ iff for every node n we have $\mathsf{pcscons}(M^{\mathsf{w}}(n))$.

It can be shown that the data propagation does not invalidate the call site path consistency.

Lemma 57. *Let* $\langle w'_1, w'_2\rangle = \mathsf{prop}^{\mathsf{w}}(w_1, w_2)$ *for path values* w_1, w'_1, w_2*, and* w'_2*. If* $\mathsf{pcscons}(w_1)$ *and* $\mathsf{pcscons}(w_2)$*, then* $\mathsf{pcscons}(w'_1)$ *and* $\mathsf{pcscons}(w'_2)$.

Proof. The proof goes by structural induction on both w_1 and w_2. The base cases are trivial and so is the the induction case when both w_1 and w_2 are path tables but the path associated with w_1 is not a prefix of the one associated with w_2.

Otherwise, the argument follows from the i.h. and the fact that any call site path seen in w_2 has to also be an extension of the path associated with w_1. □

The call site path consistency is preserved by the transformer of the path semantics.

Lemma 58. *Let* $M^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}}$ *where* $\mathsf{pcscons}(M^{\mathsf{w}})$, $M_0^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}}$, e^{ℓ} *an expression,* $k \in \mathbb{N}$, *and* E *an environment where* (e, E) *is well-formed. If* $\mathsf{step}_k^{\mathsf{w}}[\![e^{\ell}]\!](E)(M^{\mathsf{w}}) = M_0^{\mathsf{w}}$, *then* $\mathsf{pcscons}(M_0^{\mathsf{w}})$.

Proof. The proof goes by induction on k and e. The argument follows from the basic properties of the join on path values, Lemma 48, Lemma 57, and Lemma 56. □

A.3.4.5 Root Path Consistency

A path p is called to be root consistent $\mathsf{rootcons}_e(p)$, subject to some program e, iff its root node corresponds to a constant or lambda expression of the program.

$$\mathsf{rootcons}_e(p) \overset{\text{def}}{\iff} \exists c.e(loc(\mathsf{root}(p))) = c \vee \exists f, x, e'.\, e(loc(\mathsf{root}(p))) = \mu f.\lambda x.e'$$

Note that we use $\mathsf{root}(p)$ to denote the starting/root node of a path p. A map M^{w} is root consistent if all of its paths in $\mathsf{paths}(M^{\mathsf{w}})$ are root consistent.

Lemma 59. *Let* e *be a program,* $\langle w_1', w_2' \rangle = \mathsf{prop}^{\mathsf{w}}(w_1, w_2)$ *for path values* w_1, w_1', w_2, *and* w_2'. *If* $\mathsf{rootcons}_e(w_1)$ *and* $\mathsf{rootcons}_e(w_2)$, *then* $\mathsf{rootcons}(w_1')$ *and* $\mathsf{rootcons}(w_2')$.

Proof. The proof goes by structural induction on both w_1 and w_2. □

The root path consistency is preserved by the transformer of the path semantics.

Lemma 60. *Let* $M^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}}$ *where* $\mathsf{pcscons}(M^{\mathsf{w}})$, $M_0^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}}$, e^{ℓ} *an expression,* $k \in \mathbb{N}$, *and* E *an environment where* (e, E) *is well-formed. If* $\mathsf{step}_k^{\mathsf{w}}[\![e^{\ell}]\!](E)(M^{\mathsf{w}}) = M_0^{\mathsf{w}}$, *then* $\mathsf{pcscons}(M_0^{\mathsf{w}})$.

Proof. The proof goes by induction on k and e. The argument follows from the basic properties of the join on path values, Lemma 48, and Lemma 59. □

A.3.4.6 Table IO Consistency

We next formalize the notion of input-output consistency for tables that is the same as for the standard data flow semantics.

$$\begin{aligned}\mathsf{tiocons}(w) \overset{\text{def}}{\iff} (w \in \{\bot, \omega\} \cup \mathit{Cons}^{\mathsf{w}}) \vee (\exists T^{\mathsf{w}}.\, w = \langle T^{\mathsf{w}}, \_\rangle \wedge \forall p \in T^{\mathsf{w}}.\\ \pi_1(T^{\mathsf{w}}(p)) = \bot \implies \pi_2(T^{\mathsf{w}}(p)) = \bot \wedge\\ \mathsf{tiocons}(\pi_1(T^{\mathsf{w}}(p))) \wedge \mathsf{tiocons}(\pi_2(T^{\mathsf{w}}(p))))\end{aligned}$$

As expected, a map is input-output consistent $\mathsf{tiocons}_e(M)$, subject to a program e, iff all of its values are IO consistent and for every node n

$$\begin{aligned}&M^{\mathsf{w}}(\mathit{env}(n)\diamond\ell_1)=\bot \implies M^{\mathsf{w}}(\mathit{env}(n)\diamond\ell_2)=\bot \wedge\\ &\quad M^{\mathsf{w}}(\mathit{env}(n)\diamond\ell_2)=\bot \implies M^{\mathsf{w}}(\mathit{env}(n)\diamond\ell)=\bot\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{if } \exists e_1, e_2, \ell_1, \ell_2.\, e(\mathit{loc}(n)) = e_1^{\ell_1} e_2^{\ell_2}\\ &\qquad \pi_1(M^{\mathsf{w}}(n)(p))=\bot \implies M^{\mathsf{w}}(n_f)=\bot \wedge\\ &\qquad \pi_1(M^{\mathsf{w}}(n)(p))=\bot \implies M^{\mathsf{w}}(n_x)=\bot \wedge\\ &\qquad \pi_1(M^{\mathsf{w}}(n_x))=\bot \implies M^{\mathsf{w}}(E_1\diamond\ell_1)=\bot\\ &\textbf{if } M(n) \in \mathcal{T}^{\mathsf{w}} \textbf{ and } \exists x, f, e_1, \ell_1, n_x, n_f, p. \quad p \in M^{\mathsf{w}}(n) \wedge n_x{=}\mathit{env}(n)\diamond\mathsf{top}(p)\diamond x\\ &\wedge n_f{=}\mathit{env}(n)\diamond\mathsf{top}(p)\diamond f \wedge E_1 = E.x{:}n_x.f{:}n_f\end{aligned}$$

The input-output consistency of values is not violated by data propagation.

Lemma 61. *Let e be a program, $\langle w_1', w_2'\rangle = \mathsf{prop}^{\mathsf{w}}(w_1, w_2)$ for path values w_1, w_1', w_2, and w_2'. If $\mathsf{tiocons}_e(w_1)$ and $\mathsf{tiocons}_e(w_2)$, then $\mathsf{tiocons}(w_1')$ and $\mathsf{tiocons}(w_2')$.*

Proof. The proof goes by structural induction on both w_1 and w_2. □

The input-output consistency is preserved by $\mathsf{step}^{\mathsf{w}}$.

Lemma 62. *Let* $M^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}}$ *where* $\mathsf{tiocons}(M^{\mathsf{w}})$, $M_0^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}}$, e^{ℓ} *an expression,* $k \in \mathbb{N}$, *and* E *an environment where* (e, E) *is well-formed. If* $\mathsf{step}_k^{\mathsf{w}}[\![e^{\ell}]\!](E)(M^{\mathsf{w}}) = M_0^{\mathsf{w}}$, *then* $\mathsf{tiocons}(M_0^{\mathsf{w}})$.

Proof. The proof goes by induction on k and e. The argument follows from the fact that the join on path values does not invalidate input output consistency and Lemma 61. □

A.3.4.7 Table Input Consistency

A path value w is input consistent $\mathsf{inpcons}_e(w)$, subject to some program e, iff the path associated with every input of a table has to come from a function application expression in e.

$$\begin{aligned}\mathsf{inpcons}_e(w) \overset{\text{def}}{\iff} (w \in \{\bot, \omega\} \cup \mathit{Cons}^{\mathsf{w}}) \vee \exists T^{\mathsf{w}}.\, w = \langle T^{\mathsf{w}}, \_\rangle \wedge \forall p \in T^{\mathsf{w}}.\\ \mathsf{inpcons}_e(\pi_1(T^{\mathsf{w}}(p))) \wedge \mathsf{inpcons}_e(\pi_2(T^{\mathsf{w}}(p))) \wedge \exists e_1, e_2, \ell_2, p'.\\ e_1^{\mathit{loc}(\mathsf{top}(p))} e_2^{\ell_2} \in e \wedge (\pi_1(T^{\mathsf{w}}(p)) \in \{\bot, \omega\} \vee \pi_1(T^{\mathsf{w}}(p)) = \langle \_, p'\rangle\\ \wedge\, \mathsf{top}(p') = \mathit{env}(\mathsf{top}(p)) \diamond \ell_2)\end{aligned}$$

A map M^{w} is input consistent $\mathsf{inpcons}_e(M^{\mathsf{w}})$ iff all of its values are input consistent.

Lemma 63. *Let* e *be a program,* $\langle w_1', w_2'\rangle = \mathsf{prop}^{\mathsf{w}}(w_1, w_2)$ *for path values* w_1, w_1', w_2, *and* w_2'. *If* $\mathsf{inpcons}_e(w_1)$ *and* $\mathsf{inpcons}_e(w_2)$, *then* $\mathsf{inpcons}(w_1')$ *and* $\mathsf{inpcons}(w_2')$.

Proof. The proof goes by structural induction on both w_1 and w_2 and it relies on Lemma 48. □

The input consistency of path maps is preserved by $\mathsf{step}^{\mathsf{w}}$.

Lemma 64. *Let* $M^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}}$ *where* $\mathsf{inpcons}(M^{\mathsf{w}})$, $M_0^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}}$, e^{ℓ} *an expression,* $k \in \mathbb{N}$, *and* E *an environment where* (e, E) *is well-formed. If* $\mathsf{step}_k^{\mathsf{w}}[\![e^{\ell}]\!](E)(M^{\mathsf{w}}) = M_0^{\mathsf{w}}$, *then* $\mathsf{inpcons}(M_0^{\mathsf{w}})$.

Proof. The proof goes by induction on k and e. The argument follows from the fact that join on paths values does not invalidate input consistency and Lemma 63. □

A.3.4.8 Table Output Consistency

A path value w is output consistent $\mathsf{outcons}_e(w)$, subject to some program e, iff the path associated with every input of a table has to come from a function application expression in e.

$$\begin{aligned}\mathsf{outcons}_e(w) \overset{\text{def}}{\iff} (w \in \{\bot, \omega\} \cup \mathit{Cons}^{\mathsf{w}}) \vee \exists T^{\mathsf{w}}, p, f, x, e_1, \ell_1.\, w = \langle T^{\mathsf{w}}, p\rangle \wedge \\ e(\mathit{loc}(\mathsf{root}(p))) = \mu f.\lambda x.\, e_1^{\ell_1} \wedge \forall p' \in T^{\mathsf{w}}.\, \mathsf{outcons}_e(\pi_1(T^{\mathsf{w}}(p'))) \wedge \\ \mathsf{outcons}_e(\pi_2(T^{\mathsf{w}}(p'))) \wedge (\exists E, p''.\, \pi_2(T^{\mathsf{w}}(p')) \in \{\bot, \omega\} \vee \\ \pi_2(T^{\mathsf{w}}(p')) = \langle \_, p''\rangle \wedge \mathsf{top}(p'') = E \diamond \ell_1 \wedge E = \\ \mathit{env}(\mathsf{root}(p)).x : \mathit{env}(\mathsf{root}(p)) \diamond \mathsf{top}(p') \diamond x.f : \mathit{env}(\mathsf{root}(p)) \diamond \mathsf{top}(p') \diamond f)\end{aligned}$$

A map M^{w} is output consistent $\mathsf{outcons}_e(M^{\mathsf{w}})$ iff all of its values are output consistent.

Lemma 65. *Let* e *be a program,* $\langle w_1', w_2'\rangle = \mathsf{prop}^{\mathsf{w}}(w_1, w_2)$ *for path values* w_1, w_1', w_2, *and* w_2'. *If* $\mathsf{outcons}_e(w_1)$ *and* $\mathsf{outcons}_e(w_2)$, *then* $\mathsf{outcons}(w_1')$ *and* $\mathsf{outcons}(w_2')$.

Proof. The proof goes by structural induction on both w_1 and w_2. □

The output consistency of path maps is preserved by $\mathsf{step}^{\mathsf{w}}$.

Lemma 66. *Let* $M^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}}$ *where* $\mathsf{outcons}(M^{\mathsf{w}})$, $M_0^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}}$, e^{ℓ} *an expression,* $k \in \mathbb{N}$, *and* E *an environment where* (e, E) *is well-formed. If* $\mathsf{step}_k^{\mathsf{w}}[\![e^{\ell}]\!](E)(M^{\mathsf{w}}) = M_0^{\mathsf{w}}$, *then* $\mathsf{outcons}(M_0^{\mathsf{w}})$.

Proof. The proof goes by induction on k and e. The argument follows from the fact that join on paths values does not invalidate input consistency and Lemma 63. □

A.3.4.9 Constant Consistency

We next formalize the notion of constant consistency for path maps that is identical to the notion of constant consistency for regular maps. A path map M^{w} is constant consistent $\mathsf{constcons}_e(M^{\mathsf{w}})$, subject to a program e, iff for every node n, where $M^{\mathsf{w}}(n) = w$ for some path value w and $loc(n) = \ell$ for some location ℓ, it must be

$$w \in \{\omega, \bot, \langle c, n\rangle\} \qquad \text{if } e(\ell) = c$$

$$w \in \{\omega, \bot\} \cup \{\langle T^{\mathsf{w}}, E\diamond\ell\rangle \mid T^{\mathsf{w}} \in \mathcal{T}^{\mathsf{w}}\} \quad \text{if } e(\ell) = \mu f.\lambda x.e_1 \text{ for some } x,\ f, \text{ and } e_1$$

A.3.4.10 Concrete Path Subtyping

The notion of subtyping for path values resembles the one on the regular data flow values. Additionally, we also require that a table cannot see a call site path (coming from the future) unless the future table has seen it as well.

$$\begin{aligned} w_1 <:^{\mathsf{w}} w_2 \overset{\text{def}}{\iff} & (w_2 = \bot) \vee (w_1 = \langle c, p_1\rangle \wedge w_2 = \langle c, p_2\rangle \wedge c \in \mathit{Cons} \wedge p_1, p_2 \in \mathit{Path}) \\ & \vee (w_1 = w_2 = \omega) \vee (w_1 = \langle T_1^{\mathsf{w}}, p_1\rangle \wedge w_2 = \langle T_2^{\mathsf{w}}, p_2\rangle \wedge T_1^{\mathsf{w}}, T_2^{\mathsf{w}} \in \mathcal{T}^{\mathsf{w}} \\ & \wedge\ p_1, p_2 \in \mathit{Path} \wedge p_1 \preceq p_2 \wedge (\forall p \in T_1^{\mathsf{w}}.\, p_2 \preceq p \Longrightarrow p \in T_2^{\mathsf{w}}) \wedge \\ & \forall p \in T_2^{\mathsf{w}}.\pi_1(T_2^{\mathsf{w}}(p)) <:^{\mathsf{w}} \pi_1(T_1^{\mathsf{w}}(p)) \wedge \pi_2(T_1^{\mathsf{w}}(p)) <:^{\mathsf{w}} \pi_2(T_2^{\mathsf{w}}(p))) \end{aligned}$$

Unlike the subtyping relation on the ordinary data flow values, the above subtyping relation for path values is transitive, given certain simple assumptions on the values.

Lemma 67. *Let* $w_1 \in \mathcal{V}^{\mathsf{w}}$, $w_2 \in \mathcal{V}^{\mathsf{w}}$, *and* $w_3 \in \mathcal{V}^{\mathsf{w}}$ *such that* $\mathsf{tiocons}(w_1)$, $\mathsf{tiocons}(w_2)$, $\mathsf{tiocons}(w_3)$, $\mathsf{pcscons}(w_1)$, $\mathsf{pcscons}(w_2)$, *and* $\mathsf{pcscons}(w_3)$. *If* $w_1 <:^{\mathsf{w}} w_2$

and $w_2 <:^{\mathsf{w}} w_3$*, then* $w_1 <:^{\mathsf{w}} w_3$.

Proof. By structural induction on w_2. We first cover the base cases.

- $w_2 = \bot$

$$w_3 = \bot \qquad \text{by def. of } <:^{\mathsf{w}}$$
$$w_1 <:^{\mathsf{w}} \bot$$

- $w_2 = \omega$

$$w_3 = \bot \vee w_3 = \bot \qquad \text{by def. of } <:^{\mathsf{w}}$$
$$w_1 = \omega \qquad \text{by def. of } <:^{\mathsf{w}}$$
$$\omega <:^{\mathsf{w}} \omega \wedge \omega <:^{\mathsf{w}} \bot$$

- $w_2 = \langle c, p_2 \rangle$

$$w_1 = \langle c, p_1 \rangle, p_1 \preceq p_2 \qquad \text{by def. of } <:^{\mathsf{w}}$$
$$w_3 = \bot \vee w_3 = \langle c, p_3 \rangle \wedge p_2 \preceq p_3 \qquad \text{by def. of } <:^{\mathsf{w}}$$
$$\langle c, p_1 \rangle <:^{\mathsf{w}} \bot \wedge \langle c, p_1 \rangle <:^{\mathsf{w}} \langle c, p_3 \rangle \qquad \text{by trans. of } \preceq$$

For the induction case, suppose $w_2 = \langle T_2^{\mathsf{w}}, p_2 \rangle$ for some T_2^{w} and p_2. If $w_3 = \bot$, the argument is trivial. We are thus left with the case where $w_1 = \langle T_1^{\mathsf{w}}, p_1 \rangle$ and $w_3 = \langle T_3^{\mathsf{w}}, p_3 \rangle$ for some T_1^{w}, T_3^{w}, p_1, and p_3 where $p_1 \preceq p_2$ and $p_2 \preceq p_3$. Hence, $p_1 \preceq p_3$. Let us first argue that $\forall p \in T_1^{\mathsf{w}}. p_3 \preceq p \Longrightarrow p \in T_3^{\mathsf{w}}$. Suppose not, i.e., there exists a path $p \in T_1^{\mathsf{w}}$ such that $p_3 \preceq p$ and $p \notin T_3^{\mathsf{w}}$. By $p_2 \preceq p_3$, we have that $p_2 \preceq p$ and hence $p \in T_2^{\mathsf{w}}$ by $w_1 <:^{\mathsf{w}} w_2$. Therefore, it also must be that $p \in T_3^{\mathsf{w}}$ by $w_2 <:^{\mathsf{w}} w_3$.

Consider next any call site path p. Let $T_1^{\mathsf{w}}(p) = \langle w_{1i}, w_{1o} \rangle$, $T_2^{\mathsf{w}}(p) = \langle w_{2i}, w_{2o} \rangle$, and $T_3^{\mathsf{w}}(p) = \langle w_{3i}, w_{3o} \rangle$. If $p \notin T_3^{\mathsf{w}}$, the argument is trivial. Hence, we can only focus on the case where $p \in T_3^{\mathsf{w}}$. We branch on the occurrence of p in T_2^{w}.

Suppose $p \notin T_2^{\mathsf{w}}$. Hence, $w_{2i} = \bot$ and by $\mathsf{tiocons}(w_2)$ it must be that $w_{2o} = \bot$. Next, it cannot be the case where both $p \in T_1^{\mathsf{w}}$ and $p \in T_3^{\mathsf{w}}$ when $p \notin T_2^{\mathsf{w}}$. To see this, suppose this is not the case, i.e., $p \in T_1^{\mathsf{w}}$. By $\mathsf{pcscons}(w_3)$, we have that $p_3 \preceq p$. Since $p_2 \preceq p_3$, then also $p_2 \preceq p$. Then, $p_2 \in T_2^{\mathsf{w}}$ by $w_1 <:^{\mathsf{w}} w_2$, a contradiction. Hence, $p \notin T_1^{\mathsf{w}}$. By $\mathsf{tiocons}(w_1)$ we have $w_{1i} = w_{1o} = \bot$. The result then follows by $T_1^{\mathsf{w}}(p) = T_2^{\mathsf{w}}(p)$.

Suppose now $p \in T_2^{\mathsf{w}}$. Then, we have $w_{3i} <:^{\mathsf{w}} w_{2i}$ and $w_{2i} <:^{\mathsf{w}} w_{1i}$ as well as $w_{1o} <:^{\mathsf{w}} w_{2o}$ and $w_{2o} <:^{\mathsf{w}} w_{3o}$ and the result follows by i.h. □

We next prove several results stating how the propagation between values affect the subtyping relation between the values. We start with the expected result that the propagation preserves subtyping between path values.

Lemma 68. *Let $w_1 \in \mathcal{V}^{\mathsf{w}}$, $w_2 \in \mathcal{V}^{\mathsf{w}}$, $w_3 \in \mathcal{V}^{\mathsf{w}}$, and $w_4 \in \mathcal{V}^{\mathsf{w}}$. If $w_1 <:^{\mathsf{w}} w_2$ and $\langle w_3, w_4 \rangle = \mathsf{prop}^{\mathsf{w}}(w_1, w_2)$, then $w_3 <:^{\mathsf{w}} w_4$.*

Proof. By structural induction on w_1 and w_2. The proof argument is similar to the one of Lemma 38. □

The propagation between two values reserves the subtyping relation not only between the two values, but also on the whole subtyping chain containing these two values.

Lemma 69. *Let $w_1 \in \mathcal{V}^{\mathsf{w}}$, $w_2 \in \mathcal{V}^{\mathsf{w}}$, $w_3 \in \mathcal{V}^{\mathsf{w}}$, $w_1' \in \mathcal{V}^{\mathsf{w}}$, $w_2' \in \mathcal{V}^{\mathsf{w}}$, and $w_3' \in \mathcal{V}^{\mathsf{w}}$. Suppose $w_1 <:^{\mathsf{w}} w_2$, $w_2 <:^{\mathsf{w}} w_3$, $\mathsf{tiocons}(w_1)$, $\mathsf{tiocons}(w_2)$, $\mathsf{tiocons}(w_3)$, $\mathsf{pcscons}(w_1)$, $\mathsf{pcscons}(w_2)$, and $\mathsf{pcscons}(w_3)$. If $\langle w_1', w_2' \rangle = \mathsf{prop}^{\mathsf{w}}(w_1, w_2)$ then $w_2' <:^{\mathsf{w}} w_3$. Similarly, if $\langle w_2', w_3' \rangle = \mathsf{prop}^{\mathsf{w}}(w_2, w_3)$ then $w_1 <:^{\mathsf{w}} w_2'$.*

Proof. The proof goes by structural induction on w_2 and follows the structure of the proof of Lemma 67. Also, the proof relies on Lemma 61 and Lemma 57 to trigger the induction hypothesis. □

It can also be shown that updated versions, by means of propagation, of value lying on the subtyping chain can be joined while retaining the subtyping property of the chain.

Lemma 70. *Let* $w_1 \in \mathcal{V}^{\mathsf{w}}$, $w_2 \in \mathcal{V}^{\mathsf{w}}$, $w_3 \in \mathcal{V}^{\mathsf{w}}$, $w_1' \in \mathcal{V}^{\mathsf{w}}$, $w_2' \in \mathcal{V}^{\mathsf{w}}$, $w_2'' \in \mathcal{V}^{\mathsf{w}}$, *and* $w_3'' \in \mathcal{V}^{\mathsf{w}}$. *Suppose* $w_1 <:^{\mathsf{w}} w_2$, $w_2 <:^{\mathsf{w}} w_3$, $\mathsf{tiocons}(w_1)$, $\mathsf{tiocons}(w_2)$, $\mathsf{tiocons}(w_3)$, $\mathsf{pcscons}(w_1)$, $\mathsf{pcscons}(w_2)$, *and* $\mathsf{pcscons}(w_3)$. *Next, suppose* $\langle w_1', w_2' \rangle = \mathsf{prop}^{\mathsf{w}}(w_1, w_2)$ *and* $\langle w_2'', w_3'' \rangle = \mathsf{prop}^{\mathsf{w}}(w_2, w_3)$ *such that* $w_1' <:^{\mathsf{w}} w_2' <:^{\mathsf{w}} w_3$ *and* $w_1 <:^{\mathsf{w}} w_2'' <:^{\mathsf{w}} w_3''$. *Then,* $w_1' <:^{\mathsf{w}} w_2 \sqcup^{\mathsf{w}} w_2'' <:^{\mathsf{w}} w_3'$.

Proof. By structural induction on w_1, w_2, and w_3. □

We now finally define the subtyping consistency of path maps. A map M^{w} is subtyping consistent $\mathsf{csub}_e(M^{\mathsf{w}})$, subject to a program e, iff for every node n, where $loc(n) = \ell$, it must be that

$$M^{\mathsf{w}}(env(n) \diamond \ell_2) <:^{\mathsf{w}} \pi_1(M^{\mathsf{w}}(env(n) \diamond \ell_1)(p)) \wedge \pi_2(M^{\mathsf{w}}(env(n) \diamond \ell_1)(p)) <:^{\mathsf{w}} M^{\mathsf{w}}(n)$$

$$\textbf{if } \exists e_1, e_2, \ell_1, \ell_2, T^{\mathsf{w}}, p.\, e(\ell) = (e_1^{\ell_1} e_2^{\ell_2}) \textbf{ and } M^{\mathsf{w}}(env(n) \diamond \ell_1) = \langle T^{\mathsf{w}}, p \rangle$$

$$\pi_1(M^{\mathsf{w}}(n)(p)) <:^{\mathsf{w}} M^{\mathsf{w}}(n_x) \wedge M^{\mathsf{w}}(E_1 \diamond \ell_1) <:^{\mathsf{w}} \pi_2(M^{\mathsf{w}}(n)(p))$$

$$\textbf{if } \exists x, f, e_1, \ell_1, T^{\mathsf{w}}, p, n_x, n_f.\, e(\ell) = (\mu f.\lambda x.e_1^{\ell_1}),\ M^{\mathsf{w}}(n) = \langle T^{\mathsf{w}}, \_ \rangle, p \in M^{\mathsf{w}}(n),$$

$$n_x = env(n) \diamond \mathsf{top}(p) \diamond x,\ n_f = env(n) \diamond \mathsf{top}(p) \diamond f, \textbf{ and } E_1 = env(n).x : n_x.f : n_f$$

and $\forall p \in \mathsf{paths}(M^{\mathsf{w}}), \langle n_1, n_2 \rangle \in p.\, M^{\mathsf{w}}(n_1) <:^{\mathsf{w}} M^{\mathsf{w}}(n_2)$.

A.3.4.11 Path Consistency

We next introduce a path consistency, a property of maps similar to subtyping but providing more information about paths of values related to subtyping. A path map M^{w} is called path consistent $\mathsf{pvcons}(M^{\mathsf{w}})$, subject to a program e, iff $\forall p \in \mathsf{paths}(M^{\mathsf{w}}).\, M^{\mathsf{w}}(\mathsf{top}(p)) = p \wedge \forall \langle n_1, n_2 \rangle \in p.\, \mathsf{agree}_{n_2}(M^{\mathsf{w}}(n_1), M^{\mathsf{w}}(n_2))$ where

$$\mathsf{agree}_n(w_1, w_2) \overset{\text{def}}{\iff} \exists p_1, p_2.\, p_2 = p_1 \cdot n \wedge w_1 = \langle \_, p_1 \rangle \wedge w_2 = \langle \_, p_2 \rangle.$$

That is, the values associated with neighbouring nodes cannot be bottom $\perp$ or top ω and associated paths have to be an extension of each other.

We next prove several interesting properties of path consistent maps. We use $\mathsf{prefix}(p, i)$ to denote a prefix of p up to and including the node at position i in p. We also use $n \in^i p$ to denote that n appears in p at the position i.

Lemma 71. *Let M^{w} be a path map, $p \in \mathsf{paths}(M^{\mathsf{w}})$, and $n \in^i p$. If $\mathsf{pvcons}(M^{\mathsf{w}})$, then $M^{\mathsf{w}}(n) = \langle \_, \mathsf{prefix}(p, i) \rangle$.*

Proof. Follows from the definition of pvcons and agree. □

If two values agree, the data propagation between these two values is preserved.

Lemma 72. *Let e be a program, n a node, and $\langle w_3, w_4 \rangle = \mathsf{prop}^{\mathsf{w}}(w_1, w_2)$ for path values w_1, w_2, w_3, and w_4. If $\mathsf{agree}_n(w_1, w_2)$, then $\mathsf{agree}_n(w_3, w_4)$.*

Proof. The argument follows from Lemma 48. □

A.3.4.12 Path Determinism

We next introduce another useful property of paths taken by data flow values in our semantics. That is, every taken path is deterministic in the sense that for every node in a path there is always a unique predecessor node. For this purpose,

we define the predecessor relation $\mathsf{pred}^{e}_{M^{\mathsf{w}}}(n_1, n_2)$ to hold between nodes n_1 and n_2, subject to a path map M^{w} and program e, iff one of the following six cases hold

1.

$$n_1 \in \mathcal{N}_x \wedge n_2 \in \mathcal{N}_e \wedge \exists x.\, e(loc(n_1)) = x \wedge x \in \mathsf{dom}(env(n_1)) \wedge n_2 = env(n_1)(x)$$

2.

$$n_1 \in \mathcal{N}_e \wedge n_2 \in \mathcal{N}_x \wedge \exists n, E_0, e_1, e_2, \ell_1, \ell_2, x, T^{\mathsf{w}}, p.\, n_2{=}E_0{\diamond}n{\diamond}x \wedge n_1{=}env(n){\diamond}\ell_2$$
$$\wedge\, e_1^{loc(n)} e_2^{\ell_2} \in e \wedge M^{\mathsf{w}}(env(n){\diamond}\ell_2) \notin \{\bot, \omega\} \wedge M^{\mathsf{w}}(n) = \langle T^{\mathsf{w}}, p\rangle \wedge$$
$$env(\mathsf{root}(p)) = E_0 \wedge e(loc(\mathsf{root}(p))) = \mu\_.\lambda x.\_$$

3.

$$n_1 \in \mathcal{N}_e \wedge n_2 \in \mathcal{N}_x \wedge \exists n, E_0, e_1, e_2, \ell_1, \ell_2, f, T^{\mathsf{w}}, p.\, n_2{=}E_0{\diamond}n{\diamond}f \wedge n_1{=}\mathsf{root}(p)$$
$$\wedge\, e_1^{loc(n)} e_2^{\ell_2} \in e \wedge M^{\mathsf{w}}(env(n){\diamond}\ell_2) \notin \{\bot, \omega\} \wedge M^{\mathsf{w}}(n) = \langle T^{\mathsf{w}}, p\rangle \wedge$$
$$env(\mathsf{root}(p)) = E_0 \wedge e(loc(\mathsf{root}(p))) = \mu f.\lambda\_.\_$$

4.

$$n_1 \in \mathcal{N}_e \wedge n_2 \in \mathcal{N}_e \wedge \exists n, E_0, E_b, e_1, e_2, e_b, \ell_1, \ell_2, \ell_b, f, x, T^{\mathsf{w}}, p.$$
$$env(n_2){=}env(n) \wedge n_1{=}E_b{\diamond}\ell_b \wedge e(loc(n_2)) = e_1^{loc(n)} e_2^{\ell_2} \wedge M^{\mathsf{w}}(n) = \langle T^{\mathsf{w}}, p\rangle \wedge$$
$$M^{\mathsf{w}}(env(n){\diamond}\ell_2) \notin \{\bot, \omega\} \wedge env(\mathsf{root}(p)) = E_0 \wedge e(loc(\mathsf{root}(p))) = \mu f.\lambda x.e_b^{\ell_b} \wedge$$
$$E_b = E_0.x : E_0{\diamond}n{\diamond}x.f : E_0{\diamond}n{\diamond}f$$

5.

$$n_1 \in \mathcal{N}_e \wedge n_2 \in \mathcal{N}_e \wedge \exists e_0, e_1, e_2, \ell_0.$$
$$e(loc(n_2)) = e_0^{\ell_0}\, ?\, e_1^{loc(n_1)} : e_2 \wedge M^{\mathsf{w}}(env(n_2){\diamond}\ell_0) = \langle true, \_\rangle \wedge env(n_1) = env(n_2)$$

6.

$$n_1 \in \mathcal{N}_e \wedge n_2 \in \mathcal{N}_e \wedge \exists e_0, e_1, e_2, \ell_0.$$
$$e(loc(n_2)) = e_0^{\ell_0} \,?\, e_1 : e_2^{loc(n_1)} \wedge M^{\mathsf{w}}(env(n_2) \diamond \ell_0) = \langle \mathit{false}, \_ \rangle \wedge env(n_1) = env(n_2)$$

Note that we again use the _ pattern to quantify out certain mathematical objects to avoid clutter.

The first interesting property of the predecessor relation is that there is no predecessor of a node that corresponds to the constant expression or lambda expression.

Lemma 73. *Let n_2 be a node, M^{w} a path map, and e a program. If $e(loc(n_2))$ is a constant or a lambda expression, then there does not exist a node n_1 such that $\mathsf{pred}^e_{M^{\mathsf{w}}}(n_1, n_2)$.*

Proof. Follows from the definition of the pred relation. □

A node cannot be its predecessor, i.e., the pred relation is not reflexive.

Lemma 74. *Let n be a node, M^{w} a path map, and e a program. Then, it cannot be the case that $\mathsf{pred}^e_{M^{\mathsf{w}}}(n, n)$.*

Proof. Follows from the definition of the pred relation. □

Next, the predecessor relation is deterministic.

Lemma 75. *Let n, n_1, and n_2 be nodes, M^{w} a path map, and e a program. If $\mathsf{pred}^e_{M^{\mathsf{w}}}(n, n_1)$ and $\mathsf{pred}^e_{M^{\mathsf{w}}}(n, n_2)$, then $n_1 = n_2$.*

Proof. Follows from the definition of the pred relation, the fact that environments and path maps are modeled as functions, and uniqueness of locations in e. □

Although the predecessor relation is deterministic, two different nodes can share their predecessor.

Lemma 76. *Let n, n_1, and n_2 be nodes where $n_1 \neq n_2$, M^{w} a path map, and e a program. If $\mathsf{pred}^e_{M^{\mathsf{w}}}(n, n_1)$ and $\mathsf{pred}^e_{M^{\mathsf{w}}}(n, n_2)$, then $n \in \mathcal{N}_x$ or $e(loc(n))$ is a lambda expression.*

Proof. Follows from the definition of the pred relation. □

We say that path p is deterministic $\mathsf{det}^e_{M^{\mathsf{w}}}(p)$, subject to a program e and path map M^{w}, iff $\forall \langle n_1, n_2 \rangle \in p.\, \mathsf{pred}^e_{M^{\mathsf{w}}}(n_1, n_2)$. A map M^{w} is deterministic $\mathsf{det}^e(M^{\mathsf{w}})$ subject to a program e iff $\forall p \in \mathsf{paths}(M^{\mathsf{w}}).\, \mathsf{det}^e_{M^{\mathsf{w}}}(p)$.

A.3.4.13 Error Consistency

Similar to data flow semantics, we introduce the notion of the error absence of values that intuitively states the a given value is not an error and, if it is a table, it does not recursively contain errors.

$$\mathsf{noerr}(w) \overset{\text{def}}{\iff} (w \in \{\bot\} \cup \mathit{Cons}^{\mathsf{w}}) \vee (\exists T^{\mathsf{w}}.\, w = \langle T^{\mathsf{w}}, \_ \rangle \wedge \forall p \in T^{\mathsf{w}}.\, \mathsf{noerr}(\pi_1(T^{\mathsf{w}}(p))) \wedge \mathsf{noerr}(\pi_2(T^{\mathsf{w}}(p))))$$

A path map M^{w} is error consistent $\mathsf{errcons}(M^{\mathsf{w}})$ if either (1) $M^{\mathsf{w}} = M^{\mathsf{w}}_{\omega}$ or (2) for every node n it must be that $\mathsf{noerr}(M^{\mathsf{w}}(n))$.

Data propagation does not invalidate the absence of error property of path values, given that these values are related by the concrete subtyping relation on path values.

Lemma 77. *Let $w_1 \in \mathcal{V}^{\mathsf{w}}$, $w_2 \in \mathcal{V}^{\mathsf{w}}$, $w_3 \in \mathcal{V}^{\mathsf{w}}$, and $w_4 \in \mathcal{V}^{\mathsf{w}}$. If $w_1 <:^{\mathsf{w}} w_2$, $\mathsf{noerr}(w_1)$, $\mathsf{noerr}(w_2)$, and $\langle w_3, w_4 \rangle = \mathsf{prop}^{\mathsf{w}}(w_1, w_2)$, then $\mathsf{noerr}(w_3)$ and $\mathsf{noerr}(w_4)$.*

Proof. By structural induction on w_1 and w_2. The proof argument is almost identical to the one of Lemma 40. □

A.3.4.14 Environment Consistency

The notion of environment consistency for a path map is identical to the one for regular data flow maps.

$$\mathsf{envcons}(M^{\mathsf{w}}) \overset{\text{def}}{\iff} \forall n, n_x \in \mathsf{rng}(env(n)).\, M^{\mathsf{w}}(n) \neq \bot \implies M^{\mathsf{w}}(n_x) \notin \{\bot, \omega\}.$$

Consistency of an environment E subject to a path map M^{w} is defined as follows

$$\mathsf{envcons}^{\mathsf{w}}_{M}(E) \overset{\text{def}}{\iff} \forall n_x \in \mathsf{rng}(E).\, M^{\mathsf{w}}(n_x) \notin \{\bot, \omega\}$$

A.3.4.15 Consistency

We can now define a general consistency $\mathsf{cons}_e(M^{\mathsf{w}})$ of a path map M^{w} subject to a program e as follows

$$\begin{aligned}\mathsf{cons}_e(M^{\mathsf{w}}) \overset{\text{def}}{\iff} \mathsf{tpcons}(M^{\mathsf{w}}) \wedge \mathsf{pcscons}(M^{\mathsf{w}}) \wedge \mathsf{rootcons}_e(M^{\mathsf{w}}) \wedge \mathsf{inpcons}_e(M^{\mathsf{w}}) \wedge \\ \mathsf{outcons}_e(M^{\mathsf{w}}) \wedge \mathsf{det}_e(M^{\mathsf{w}}) \wedge \mathsf{csub}_e(M^{\mathsf{w}}) \wedge \mathsf{tiocons}_e(M^{\mathsf{w}}) \wedge \mathsf{fin}(M^{\mathsf{w}}) \\ \wedge\ \mathsf{constcons}_e(M^{\mathsf{w}}) \wedge \mathsf{errcons}(M^{\mathsf{w}}) \wedge \mathsf{envcons}(M^{\mathsf{w}}) \wedge \mathsf{pvcons}(M^{\mathsf{w}})\end{aligned}$$

We now prove several interesting properties of general consistency of maps.

Lemma 78. *Let M^{w} be a consistent map $\mathsf{cons}_e(M^{\mathsf{w}})$ subject to a program e. Then, $\forall p \in \mathsf{paths}(M^{\mathsf{w}}), n \in p.\, M^{\mathsf{w}}(n) \notin \{\bot, \omega\}$.*

Proof. Follows from the definition of pvcons. □

The next result states that the observed paths in a consistent map can be uniquely identified by their top nodes.

Lemma 79. *Let M^{w} be a consistent map $\mathsf{cons}_e(M^{\mathsf{w}})$ subject to a program e. Then, $\forall p_1, p_2 \in \mathsf{paths}(M^{\mathsf{w}}).p_1 \neq p_2 \implies \mathsf{top}(p_1) \neq \mathsf{top}(p_2)$.*

Proof. By the induction on both $|p_1|$ and $|p_2|$. By $\mathsf{pvcons}(M^{\mathsf{w}})$, prefixes of both p_1 and p_2 are in $\mathsf{paths}(M^{\mathsf{w}})$. The argument then follows from $\mathsf{det}_e(M^{\mathsf{w}})$, $\mathsf{rootcons}_e(M^{\mathsf{w}})$, Lemma 75, and Lemma 73. □

Further, a node cannot appear at two different location in any path of a consistent map. That is, individual paths do not constitute cycles.

Lemma 80. *Let M^{w} be a consistent map $\mathsf{cons}_e(M^{\mathsf{w}})$ subject to a program e and $p \in \mathsf{paths}(M^{\mathsf{w}})$. For any given node n, suppose $n \in^i p$ and $n \in^j \in p$. Then, it must be that $i = j$.*

Proof. Suppose not, $i \neq j$. Let $p_i = \mathsf{prefix}(p, i)$ and $p_j = \mathsf{prefix}(p, j)$. Clearly, $p_i \neq p_j$. However, by Lemma 71 it must be that $M^{\mathsf{w}}(n) = \langle \_, p_i \rangle = \langle \_, p_j \rangle$. Hence, $p_i = p_j$. Contradiction. □

Moreover, a node is always visited from the same predecessor node.

Lemma 81. *Let M^{w} be a consistent map $\mathsf{cons}_e(M^{\mathsf{w}})$ subject to a program e. Also, let n_1 and n_2 be nodes such that $\mathsf{pred}^e_{M^{\mathsf{w}}}(n_1, n_2)$. Then, $M^{\mathsf{w}}(n_2) = \bot \vee M^{\mathsf{w}}(n_2) = \omega \vee \mathsf{agree}_{n_2}(M^{\mathsf{w}}(n_1), M^{\mathsf{w}}(n_2))$.*

Proof. Let $M^{\mathsf{w}}(n_2) = \langle \_, p \rangle$ for some p. First, consider the case $|p \models 1$.

$$p \in \mathsf{paths}(M^{\mathsf{w}}) \qquad \text{by def. of } \mathsf{paths}$$
$$e(loc(n_2)) \text{ is a constant or lambda} \qquad \text{by def. of } \mathsf{rootcons}$$
$$\neg \mathsf{pred}^e_{M^{\mathsf{w}}}(n_1, n_2) \qquad \text{by Lemma 73}$$

Next, consider the case $|p| > 1$.

$$p = p' \cdot n_2 \qquad \text{by } \mathsf{tpcons}_e(M^{\mathsf{w}}) \text{ for some } p'$$
$$M^{\mathsf{w}}(n_1) = \langle \_, p' \rangle \qquad \text{by Lemma 71 and Lemma 80}$$

□

We next show that propagation among values from a consistent map cannot invalidate subtyping on branching points in the subtyping chains.

Lemma 82. *Let M^{w} be a consistent map $\mathsf{cons}_e(M^{\mathsf{w}})$ subject to a program e. Also, let n, n_1, and n_2 be nodes such that $n_1 \neq n_2$, $\mathsf{pred}^e_{M^{\mathsf{w}}}(n, n_2)$, and $\mathsf{pred}^e_{M^{\mathsf{w}}}(n, n_2)$. Suppose $M^{\mathsf{w}}(n) <:^{\mathsf{w}} M^{\mathsf{w}}(n_1)$, $M^{\mathsf{w}}(n) <:^{\mathsf{w}} M^{\mathsf{w}}(n_2)$, and $\langle w', w'_1 \rangle = \mathsf{prop}^{\mathsf{w}}(M^{\mathsf{w}}(n), M^{\mathsf{w}}(n_1))$. Then, $w' <:^{\mathsf{w}} M^{\mathsf{w}}(n_2)$.*

Proof. Let $M^{\mathsf{w}}(n) = w$, $M^{\mathsf{w}}(n_1) = w_1$, and $M^{\mathsf{w}}(n_2) = w_2$. First, observe that if $w_1 = \bot$ then $w' = w$ and the result follows immediately. Otherwise, the proof goes by a case analysis on $M^{\mathsf{w}}(n_2)$.

- $w_2 = \bot$. Trivial, by the def. of $<:^{\mathsf{w}}$.
- $w_2 = \langle c, p_2 \rangle$. Then $w = \langle c, p \rangle$ so that $p \preceq p_2$ and $w_1 = \bot$ or $w_1 = \langle c, p_1 \rangle$ where $p \preceq p_1$. Then, by the def. of $\mathsf{prop}^{\mathsf{w}}$ it follows that $w' = \langle c, p \rangle$.
- $w_2 = \omega$. Then, $w = w' = w'_1 = w_2 = \omega$.

Next, suppose $w = \langle T_2^{\mathsf{w}}, p_2 \rangle$ for some T_2^{w} and p_2. Then it must be that $w_1 = \langle T_1^{\mathsf{w}}, p_1 \rangle$ for some T_1^{w} and p_1. Also, it is the case that $w = \langle T^{\mathsf{w}}, p \rangle$ for some T^{w} and p. First, observe that

$$\mathsf{top}(p) = n, \mathsf{top}(p_1) = n_1, \mathsf{top}(p_2) = n_2 \qquad \text{by } \mathsf{tpcons}(M^{\mathsf{w}})$$

$$p_1 \neq p_2$$

We next show that $p_1 \not\preceq p_2$; the case $p_2 \not\preceq p_1$ is handled similarly. Suppose $p_1 \preceq p_2$. By $\mathsf{pvcons}(M^{\mathsf{w}})$ and $\mathsf{det}_e(M^{\mathsf{w}})$, we know that $p_2 = p{\cdot}n_2$ and $p_1 = p{\cdot}n_1$. Hence, there exist i and j such that $i \neq j$, $n \in^i p_2$, $n \in^j \in p_2$. However, this is a contradiction by Lemma 80. Thus, $p_1 \neq p_2$, $p_1 \not\preceq p_2$, and $p_2 \not\preceq p_1$. By $\mathsf{pcscons}(M^{\mathsf{w}})$, the

set of observed call site paths in T_1^{w} and T_2^{w} is different. The result then follows immediately from the definition of $\mathsf{prop}^{\mathsf{w}}$. □

We next show that output values in tables must agree, via concrete subtyping, with the value of the lambda body used to initially populate the table.

Lemma 83. *Let M^{w} be a consistent map $\mathsf{cons}_e(M^{\mathsf{w}})$ subject to a program e, w a path value, n a node, and p_c as well as p' paths. Suppose $\langle T^{\mathsf{w}}, p\rangle = M^{\mathsf{w}}(n)$, for some T^{w} and p, and let $\pi_2(T^{\mathsf{w}}(p_c)) = w$. If $w = \langle \_, p'\rangle$, then $M^{\mathsf{w}}(\mathsf{top}(p')) <:^{\mathsf{w}} w$ and $M^{\mathsf{w}}(\mathsf{top}(p')) = \langle \_, p'\rangle$.*

Proof.

$p, p_c, p' \in \mathsf{paths}(M^{\mathsf{w}})$	by def. of paths
$e(loc(\mathsf{root}(p))) = \mu f.\lambda x.e_1^{\ell_1}$	for some f, x, e_1, ℓ_1 by $\mathsf{rootcons}_e(M^{\mathsf{w}})$
$n_x = env(\mathsf{root}p)\diamond\mathsf{top}(p_c)\diamond x,$	assume
$n_f = env(\mathsf{root}p)\diamond\mathsf{top}(p_c)\diamond f,$	
$E_1 = env(\mathsf{root}(p)).x : n_x.f : n_f$	
$\mathsf{top}(p') = E_1\diamond\ell_1$	by $\mathsf{outcons}_e(M^{\mathsf{w}})$
$M^{\mathsf{w}}(\mathsf{root}(p)) <:^{\mathsf{w}} M^{\mathsf{w}}(n)$	by $\mathsf{csub}_e(M^{\mathsf{w}}), \mathsf{tiocons}M^{\mathsf{w}}, \mathsf{pcscons}_e(M^{\mathsf{w}})$, and Lemma 67
$M^{\mathsf{w}}(\mathsf{root}(p)) = \langle T_r^{\mathsf{w}}, \_\rangle$	for some T_r^{w} by def. of $<:^{\mathsf{w}}$
$M^{\mathsf{w}}(E_1\diamond\ell_1) <:^{\mathsf{w}} \pi_2(T_r^{\mathsf{w}}(p_c))$	by $\mathsf{csub}_e(M^{\mathsf{w}})$
$M^{\mathsf{w}}(E_1\diamond\ell_1) <:^{\mathsf{w}} w$	by $\mathsf{csub}_e(M^{\mathsf{w}}), \mathsf{tiocons}M^{\mathsf{w}}, \mathsf{pcscons}_e(M^{\mathsf{w}})$, and Lemma 67
$M^{\mathsf{w}}(E_1\diamond\ell_1) = \langle \_, p'\rangle$	by $\mathsf{tpcons}(M^{\mathsf{w}})$ and Lemma 79

□

A similar result holds for inputs of tables that are pushed to tables at the execution points corresponding to function application expressions.

Lemma 84. *Let M^{w} be a consistent map $\mathsf{cons}_e(M^{\mathsf{w}})$ subject to a program e, w a path value, n a node, and p_c as well as p' paths. Suppose $\langle T^{\mathsf{w}}, p\rangle = M^{\mathsf{w}}(n)$, for some T^{w} and p, and let $\pi_1(T^{\mathsf{w}}(p_c)) = w$. If $w = \langle \_, p'\rangle$, then $M^{\mathsf{w}}(\mathsf{top}(p')) <:^{\mathsf{w}} w$ and $M^{\mathsf{w}}(\mathsf{top}(p')) = \langle \_, p'\rangle$.*

Proof.

$$p, p_c, p' \in \mathsf{paths}(M^{\mathsf{w}}) \qquad \text{by def. of } \mathsf{paths}$$

$$p \preceq p_c \qquad \text{by } \mathsf{pcscons}(M^{\mathsf{w}})$$

$$e_1^{loc(\mathsf{top}(p_c))} e_2^{loc(\mathsf{top}(p'))} \in e, \qquad \text{for some } e_1, e_2 \text{ by } \mathsf{inpcons}_e(M^{\mathsf{w}})$$

$$env(\mathsf{top}(p_c)) = env(\mathsf{top}(p'))$$

$$M^{\mathsf{w}}(n) <:^{\mathsf{w}} M^{\mathsf{w}}(\mathsf{top}(p_c)) \qquad \text{by } \mathsf{csub}_e(M^{\mathsf{w}}), \mathsf{tiocons}M^{\mathsf{w}}, \mathsf{pcscons}_e(M^{\mathsf{w}}), \text{ and Lemma 67}$$

$$M^{\mathsf{w}}(\mathsf{top}(p_c)) = \langle T_c^{\mathsf{w}}, \_\rangle \qquad \text{by def. of } <:^{\mathsf{w}} \text{ and } \mathsf{pvcons}(M^{\mathsf{w}})$$

$$\pi_1(T_c^{\mathsf{w}}(p_c)) <:^{\mathsf{w}} w \qquad \text{by def. of } <:^{\mathsf{w}}$$

$$M^{\mathsf{w}}(\mathsf{top}(p')) <:^{\mathsf{w}} \pi_1(T_c^{\mathsf{w}}(p_c)) \qquad \text{by } \mathsf{csub}_e(M^{\mathsf{w}})$$

$$M^{\mathsf{w}}(\mathsf{top}(p')) <:^{\mathsf{w}} w \qquad \text{by } \mathsf{csub}_e(M^{\mathsf{w}}), \mathsf{tiocons}M^{\mathsf{w}}, \mathsf{pcscons}_e(M^{\mathsf{w}}), \text{ and Lemma 67}$$

$$M^{\mathsf{w}}(\mathsf{top}(p')) = \langle \_, p'\rangle \qquad \text{by } \mathsf{tpcons}(M^{\mathsf{w}}) \text{ and Lemma 79}$$

□

We next show that the concrete transformer does not change the structure of the paths, i.e., the pred relation.

Lemma 85. *Let* M^{w}, M_1^{w}, *and* M_2^{w} *be path maps such that* $\mathsf{cons}_e(M_1^{\mathsf{w}})$, $\mathsf{cons}_e(M_2^{\mathsf{w}})$, *and* $\mathsf{cons}_e(M^{\mathsf{w}})$ *for some program* e. *Given an expression* $e_0^\ell \in e$, *fuel* k, *and two environments* E_1 *and* E_2 *such that* $E_1 \neq E_2$, $\mathsf{domvalid}(\ell, e, E_1,$, $\mathsf{domvalid}(\ell, e, E_2)$, *and* $\mathsf{dom}(E_1) = \mathsf{dom}(E_2)$, *assume that* $\mathsf{step}_k^{\mathsf{w}}[\![e_0^\ell]\!](E_1)(M^{\mathsf{w}}) = M_1^{\mathsf{w}}$ *and* $\mathsf{step}_k^{\mathsf{w}}[\![e_0^\ell]\!](E_2)(M^{\mathsf{w}}) = M_2^{\mathsf{w}}$. *Then,* $\forall p_1, p_2 \in \mathsf{paths}(M_1^{\mathsf{w}}) \cup \mathsf{paths}(M_2^{\mathsf{w}}), i, j, n.\, n \in^i p_1 \wedge n \in^j p_2 \implies \mathsf{prefix}(p_1, i) = \mathsf{prefix}(p_2, j)$.

Proof. Let $p' = \mathsf{prefix}(p_1, i)$ and $p'' = \mathsf{prefix}(p_2, j)$. Clearly, $\mathsf{top}(p') = n = \mathsf{top}(p'')$ by the definition of prefix. First, if $p_1 \in \mathsf{paths}(M_1^{\mathsf{w}})$ and $p_2 \in \mathsf{paths}(M_1^{\mathsf{w}})$, then by Lemma 71 we have that $p', p'' \in \mathsf{paths}(M_1^{\mathsf{w}})$ and the result thus follows from Lemma 79. The same argument applies for the case when $p_1 \in \mathsf{paths}(M_2^{\mathsf{w}})$ and $p_2 \in \mathsf{paths}(M_2^{\mathsf{w}})$.

We are therefore left with the case when $p_1 \in \mathsf{paths}(M_1^{\mathsf{w}})$ and $p_2 \in \mathsf{paths}(M_2^{\mathsf{w}})$ but $p_1 \notin \mathsf{paths}(M_2^{\mathsf{w}})$ and $p_2 \notin \mathsf{paths}(M_1^{\mathsf{w}})$. (The same argument that follows also applies for the case where we invert the memberships of p_1 and p_2.) By Lemma 71, we have that $p' \in \mathsf{paths}(M_1^{\mathsf{w}})$ and $p'' \in \mathsf{paths}(M_2^{\mathsf{w}})$. We thus have $\mathsf{det}^e_{M_1^{\mathsf{w}}}(p')$ and $\mathsf{det}^e_{M_2^{\mathsf{w}}}(p'')$. The argument follows by a mutual induction on $|p'|$ and $|p''|$. If $|p'| = |p''| = 1$, the argument is trivial. Next, observe that by $\mathsf{rootcons}_e(M_1^{\mathsf{w}})$ and $\mathsf{rootcons}_e(M_2^{\mathsf{w}})$ and Lemma 73 it cannot be the case that $|p'| > 1$ and $|p''| = 1$ (and similarly $|p''| > 1$ and $|p'| = 1$). Hence, both p' and p'' have a predecessor of a top node n; let those predecessor node be n' and n'', respectively. We next show that $n' = n''$. The argument then inductively follows by Lemma 71. We consider all the possible cases of n by inspecting the definition of pred. Observe that we have $\mathsf{pred}^e_{M_1^{\mathsf{w}}}(n', n)$ and $\mathsf{pred}^e_{M_2^{\mathsf{w}}}(n'', n)$.

- $e(loc(n)) = x$.

$$x \in \mathsf{dom}(env(n)) \qquad \text{by def. of } \mathsf{pred}$$

$$\forall M^{\mathsf{w}}, n_0.\, \mathsf{pred}^e_{M^{\mathsf{w}}}(n_0, n) \Longrightarrow n_0 = env(n)(x) \qquad \text{by def. of } \mathsf{pred}$$

$$\mathsf{pred}^e_{M_1^{\mathsf{w}}}(env(n)(x), n), \mathsf{pred}^e_{M_2^{\mathsf{w}}}(env(n)(x), n)$$

$$n' = n'' = env(n)(x)$$

- $n = E{\diamond}n_c{\diamond}x$ for some E, n_c, and x.
  - $n_c \notin \mathsf{ranger}(E_1{\diamond}\ell), n_c \notin \mathsf{ranger}(E_2{\diamond}\ell)$ **assume**

    $M^{\mathsf{w}}(n_c) = M_1^{\mathsf{w}}(n_c) = M_2^{\mathsf{w}}(n_c)$ by Lemma 50 and Lemma 51

    $n' = n''$ by def. of pred
  - $n_c \in \mathsf{ranger}(E_1{\diamond}\ell), n_c \in \mathsf{ranger}(E_2{\diamond}\ell)$ **assume**

    impossible by Lemma 50 and Lemma 51
  - $n_c \in \mathsf{ranger}(E_1{\diamond}\ell), n_c \notin \mathsf{ranger}(E_2{\diamond}\ell)$ **assume**

    $M_2^{\mathsf{w}}(n_c) = M^{\mathsf{w}}(n_c)$ by Lemma 50 and Lemma 51

    $M^{\mathsf{w}}(n_c) \neq \bot$ by def. of pred

    $M_1^{\mathsf{w}}(n_c) \neq \omega$ by $\mathsf{errcons}(M_1^{\mathsf{w}})$

    $M^{\mathsf{w}}(n_c) = \langle \_, p_c \rangle, M_1^{\mathsf{w}}(n_c) = \langle \_, p_c \rangle,$ for some p_c by Lemma 56

    $M_2^{\mathsf{w}}(n_c) = \langle \_, p_c \rangle$

    $n' = n''$ by def. of pred
  - $n_c \notin \mathsf{ranger}(E_1{\diamond}\ell), n_c \in \mathsf{ranger}(E_2{\diamond}\ell)$ **assume**

    similar to above
- Similar for other cases of n.

□

We also show that the join of two consistent path maps, obtained by parallel applications of the concrete transformer, is subtype consistent.

Lemma 86. *Let* M^{w}, M_1^{w}, *and* M_2^{w} *be path maps such that* $\mathsf{cons}_e(M_1^{\mathsf{w}})$, $\mathsf{cons}_e(M_2^{\mathsf{w}})$, *and* $\mathsf{cons}_e(M^{\mathsf{w}})$ *for some program* e. *Assume we are also given an expression* $e_0^\ell \in e$, *fuel* k, *and two environments* E_1 *and* E_2 *such that* $E_1 \neq E_2$, $\mathsf{domvalid}(\ell, e, E_1)$, $\mathsf{domvalid}(\ell, e, E_2)$, *and* $\mathsf{dom}(E_1) = \mathsf{dom}(E_2)$. *If* $\mathsf{step}_k^{\mathsf{w}}[\![e_0^\ell]\!](E_1)(M^{\mathsf{w}}) = M_1^{\mathsf{w}}$ *and* $\mathsf{step}_k^{\mathsf{w}}[\![e_0^\ell]\!](E_2)(M^{\mathsf{w}}) = M_2^{\mathsf{w}}$, *then* $\mathsf{csub}_e(M_1^{\mathsf{w}} \dot{\sqcup}^{\mathsf{w}} M_2^{\mathsf{w}})$.

Proof. Let $p \in \mathsf{paths}(M_1^{\mathsf{w}} \dot{\sqcup}^{\mathsf{w}} M_2^{\mathsf{w}})$ and $\langle n_1, n_2 \rangle$. We show that

$$M_1^{\mathsf{w}}(n_1) \sqcup^{\mathsf{w}} M_2^{\mathsf{w}}(n_1) <:^{\mathsf{w}} M_2^{\mathsf{w}}(n_1) \sqcup^{\mathsf{w}} M_1^{\mathsf{w}}(n_2)$$

For the remainder of the definition of csub_e, the argument follows from Lemma 50 and Lemma 51. Let $w_1 = M_1^{\mathsf{w}}(n_1) \sqcup^{\mathsf{w}} M_2^{\mathsf{w}}(n_1)$ and $w_2 = M_1^{\mathsf{w}}(n_2) \sqcup^{\mathsf{w}} M_2^{\mathsf{w}}(n_2)$.

First, observe that $p \in \mathsf{paths}(M_1^{\mathsf{w}})$ or $p \in \mathsf{paths}(M_2^{\mathsf{w}})$ by the definition of $\dot{\sqcup}^{\mathsf{w}}$. Also, note that $M_1^{\mathsf{w}}(n_1) \neq \omega, M_2^{\mathsf{w}}(n_1) \neq \omega, M_2^{\mathsf{w}}(n_1) \neq \omega$, and $M_1^{\mathsf{w}}(n_2) \neq \omega$. Suppose not. By $\mathsf{errcons}(M_1^{\mathsf{w}})$ and $\mathsf{errcons}(M_2^{\mathsf{w}})$ we have that $M_1^{\mathsf{w}} = M_\omega^{\mathsf{w}}$ or $M_2^{\mathsf{w}} = M_\omega^{\mathsf{w}}$, in which case $M_1^{\mathsf{w}} \dot{\sqcup}^{\mathsf{w}} M_2^{\mathsf{w}} = M_\omega^{\mathsf{w}}$ and thus $\mathsf{paths}(M_1^{\mathsf{w}} \dot{\sqcup}^{\mathsf{w}} M_2^{\mathsf{w}}) = \emptyset$, a contradiction. Further, note that $w_1 \neq \bot$ and $w_2 \neq \bot$ by $\mathsf{cons}_e(M_1^{\mathsf{w}})$, $\mathsf{cons}_e(M_2^{\mathsf{w}})$, and Lemma 78.

We proceed with the proof by structural induction on w_1 and w_2 where we focus on the cases where $w_1 \neq \bot$ and $w_2 \neq \bot$. Suppose first $w_1 = \omega$. There are two cases possible. First, $M_1^{\mathsf{w}}(n_1) = \langle c_1, p_1 \rangle$ and $M_2^{\mathsf{w}}(n_1) = \langle c_2, p_2 \rangle$ where either

$c_1 \neq c_2$ or $p_1 \not\preceq p_2 \wedge p_2 \not\preceq p_1$.

$p_1 = p_2$ by Lemma 85

$M_1^{\mathsf{w}}(\mathsf{root}(p_1)) = \langle c, \_\rangle, M_2^{\mathsf{w}}(\mathsf{root}(p_2)) = \langle c, \_\rangle$ by $\mathsf{rootcons}_e(M_1^{\mathsf{w}}), \mathsf{rootcons}_e(M_2^{\mathsf{w}})$, $\mathsf{constcons}(M_1^{\mathsf{w}})$, and $\mathsf{constcons}(M_2^{\mathsf{w}})$

$M_1^{\mathsf{w}}(\mathsf{top}(p_1)) = \langle c, \_\rangle, M_2^{\mathsf{w}}(\mathsf{top}(p_2)) = \langle c, \_\rangle$ by $\mathsf{csub}_e(M_1^{\mathsf{w}}), \mathsf{csub}_e(M_2^{\mathsf{w}})$, Lemma 67, and def. of $<:^{\mathsf{w}}$

contradiction

The other case is when $M_1^{\mathsf{w}}(n_1)$ is a constant path value and $M_2^{\mathsf{w}}(n_1)$ is table path value, or vice versa. The argument is then exactly the same. Moreover, the same reasoning steps are sufficient to show that $w_2 \neq \omega$. In conclusion, $w_1 \neq \omega$ and $w_2 \neq \omega$.

Next, suppose that $w_1 = \langle c_1, p_1\rangle$ and $w_2 = \langle c_2, p_2\rangle$ where either $c_1 \neq c_2$ or $p_1 \not\preceq p_2 \wedge p_2 \not\preceq p_1$.

$(M_1^{\mathsf{w}}(n_1) = \langle c_1, p_1\rangle \wedge M_1^{\mathsf{w}}(n_2) = \langle c_1, p_2\rangle) \vee (M_2^{\mathsf{w}}(n_1) = \langle c_1, p_1\rangle \wedge M_2^{\mathsf{w}}(n_2) = \langle c_1, p_2\rangle)$ by $\mathsf{cons}_e(M_1^{\mathsf{w}}), \mathsf{cons}_e(M_2^{\mathsf{w}})$, and Lemma 78

$p_1 \preceq p_2$ by $\mathsf{csub}_e(M_1^{\mathsf{w}})$ and $\mathsf{csub}_e(M_2^{\mathsf{w}})$

$c_1 = c_2$ by $\mathsf{rootcons}_e(M_1^{\mathsf{w}}), \mathsf{rootcons}_e(M_2^{\mathsf{w}})$, $\mathsf{constcons}(M_1^{\mathsf{w}}), \mathsf{constcons}(M_2^{\mathsf{w}})$, $\mathsf{csub}_e(M_1^{\mathsf{w}}), \mathsf{csub}_e(M_2^{\mathsf{w}})$, Lemma 67, and def. of $<:^{\mathsf{w}}$

contradiction

The same reasoning steps apply for the other base cases of $w_1 \in \mathit{Cons}^{\mathsf{w}} \wedge w_2 \in \mathcal{T}^{\mathsf{w}}$ and $w_1 \in \mathcal{T}^{\mathsf{w}} \wedge w_2 \in \mathit{Cons}^{\mathsf{w}}$. When $w_1 \in \mathcal{T}^{\mathsf{w}}$ and $w_2 \in \mathcal{T}^{\mathsf{w}}$, for every call site

path p the inputs of w_1 and w_2, and outputs respectively, at p must agree on the top nodes by $\mathsf{inpcons}_e(M_1^{\mathsf{w}})$, $\mathsf{inpcons}_e(M_2^{\mathsf{w}})$, $\mathsf{outcons}_e(M_1^{\mathsf{w}})$, and $\mathsf{outcons}_e(M_2^{\mathsf{w}})$. The same reasoning as above then inductively applies by Lemma 84, Lemma 83, and the definition of $<:^{\mathsf{w}}$. □

Lemma 87. *Let M^{w}, M_1^{w}, and M_2^{w} be path maps such that $\mathsf{cons}_e(M_1^{\mathsf{w}})$, $\mathsf{cons}_e(M_2^{\mathsf{w}})$, and $\mathsf{cons}_e(M^{\mathsf{w}})$ for some program e. Given an expression $e_0^\ell \in e$, fuel k, and two environments E_1 and E_2 such that $E_1 \neq E_2$, $\mathsf{domvalid}(\ell, e, E_1)$, $\mathsf{domvalid}(\ell, e, E_2)$, and $\mathsf{dom}(E_1) = \mathsf{dom}(E_2)$, assume that $\mathsf{step}_k^{\mathsf{w}}[\![e_0^\ell]\!](E_1)(M^{\mathsf{w}}) = M_1^{\mathsf{w}}$ and $\mathsf{step}_k^{\mathsf{w}}[\![e_0^\ell]\!](E_2)(M^{\mathsf{w}}) = M_2^{\mathsf{w}}$. Then, $\mathsf{pvcons}(M_1^{\mathsf{w}} \dot{\sqcup}^{\mathsf{w}} M_2^{\mathsf{w}})$.*

Proof. Similar to the proof of Lemma 86. □

Lemma 88. *Let M^{w}, M_1^{w}, and M_2^{w} be path maps such that $\mathsf{cons}_e(M_1^{\mathsf{w}})$, $\mathsf{cons}_e(M_2^{\mathsf{w}})$, and $\mathsf{cons}_e(M^{\mathsf{w}})$ for some program e. Given an expression $e_0^\ell \in e$, fuel k, and two environments E_1 and E_2 such that $E_1 \neq E_2$, $\mathsf{domvalid}(\ell, e, E_1)$, $\mathsf{domvalid}(\ell, e, E_2)$, and $\mathsf{dom}(E_1) = \mathsf{dom}(E_2)$, assume that $\mathsf{step}_k^{\mathsf{w}}[\![e_0^\ell]\!](E_1)(M^{\mathsf{w}}) = M_1^{\mathsf{w}}$ and $\mathsf{step}_k^{\mathsf{w}}[\![e_0^\ell]\!](E_2)(M^{\mathsf{w}}) = M_2^{\mathsf{w}}$. Then, $\mathsf{errcons}(M_1^{\mathsf{w}} \dot{\sqcup}^{\mathsf{w}} M_2^{\mathsf{w}})$.*

Proof. Similar to the proof of Lemma 86. □

The above results are used to show that the general consistency of path maps is preserved by $\mathsf{step}^{\mathsf{w}}$.

Theorem 7. *Let $M^{\mathsf{w}}, M_f^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}}$, $M_f^{\mathsf{w}} \in \mathcal{M}^{\mathsf{w}}$, e^ℓ an expression of a program m, $k \in \mathbb{N}$, and E an environment where (e, E) is well-formed, $\mathsf{domvalid}(\ell, m, E)$, and $\mathsf{envcons}_M^{\mathsf{w}}(E)$. If $\mathsf{cons}_m(M^{\mathsf{w}})$ and $\mathsf{step}_k^{\mathsf{w}}[\![e^\ell]\!](E)(M^{\mathsf{w}}) = M_f^{\mathsf{w}}$, then $\mathsf{cons}_m(M_f^{\mathsf{w}})$ and $\mathsf{envcons}_{M_f^{\mathsf{w}}}(E)$.*

Proof. The proof goes by induction on k and structural induction on e. We need to show that $\mathsf{envcons}(M^{\mathsf{w}})$, $\mathsf{csub}_m(M_f^{\mathsf{w}})$, $\mathsf{pvcons}_m(M_f^{\mathsf{w}})$, $\mathsf{errcons}(M_f^{\mathsf{w}})$, $\mathsf{constcons}_m(M_f^{\mathsf{w}})$, $\mathsf{det}_m(M_f^{\mathsf{w}})$. When $k = 0$, the argument is trivial. We consider the cases for e when $k > 1$.

- $e = c$. Let $M^{\mathsf{w}}(E{\diamond}\ell) = w$. By $\mathsf{constcons}_m(M_f^{\mathsf{w}})$, we have that $w \in \{\bot, \omega, \langle c, \ell\rangle\}$. If $w = \langle c, \ell\rangle$, then $M^{\mathsf{w}} = M_f^{\mathsf{w}}$ and the argument is trivial. If $w = \omega$, then by $\mathsf{errcons}(M^{\mathsf{w}})$, we have $M^{\mathsf{w}} = M_\omega^{\mathsf{w}}$ and hence $M_f^{\mathsf{w}} = M_\omega^{\mathsf{w}} = M^{\mathsf{w}}$, from which the argument directly follows. If $w = \bot$, then $M_f^{\mathsf{w}} = M^{\mathsf{w}}[E{\diamond}\ell \mapsto \langle c, E{\diamond}\ell\rangle]$, from which $\mathsf{det}_e(M_f^{\mathsf{w}})$, $\mathsf{errcons}(M_f^{\mathsf{w}})$, $\mathsf{envcons}(M_f^{\mathsf{w}})$, and $\mathsf{constcons}_m(M_f^{\mathsf{w}})$ immediately follow. Lastly, $\mathsf{pvcons}(M_f^{\mathsf{w}})$ and $\mathsf{csub}_m(M_f^{\mathsf{w}})$ hold by Lemma 78. Note that $\mathsf{envcons}_{M_f^{\mathsf{w}}}(E)$ holds trivially.

- $e = x$. Let $w_x = M^{\mathsf{w}}(E(x))$ and $w = M^{\mathsf{w}}(E{\diamond}\ell)$. If $w_x = \omega \vee w = \omega$, then $M^{\mathsf{w}} = M_\omega^{\mathsf{w}}$ and by Lemma 47 it follows that $M_f^{\mathsf{w}} = M_\omega^{\mathsf{w}}$ and $\mathsf{cons}_m(M_f^{\mathsf{w}})$ trivially. Also note that $\mathsf{constcons}_m(M_f^{\mathsf{w}})$ holds trivially.

  Next, note that by the def. of pred, we have $\mathsf{pred}^m_{M^{\mathsf{w}}}(E(x), E{\diamond}\ell)$. By Lemma 81, it is the case that $w = \bot$ or $\mathsf{agree}_{E{\diamond}\ell}(w_x, w)$. In case $w = \bot$, $\mathsf{det}^m(M_f^{\mathsf{w}})$ follows by $\mathsf{pred}^m_{M^{\mathsf{w}}}(E(x), E{\diamond}\ell)$ and Lemma 78 whereas $\mathsf{pvcons}(M_f^{\mathsf{w}})$ follows by the def. of $\mathsf{prop}^{\mathsf{w}}$ and ext. In the other case, $\mathsf{det}^m(M_f^{\mathsf{w}})$ and $\mathsf{pvcons}(M_f^{\mathsf{w}})$ follow by Lemma 56.

  Since $w = \bot$ or $\mathsf{agree}_{E{\diamond}\ell}(w_x, w)$, by $\mathsf{cons}_m(M^{\mathsf{w}})$ and the definition of $<:^{\mathsf{w}}$, we have that $w_x <:^{\mathsf{w}} w$. Then by Lemma 77 it follows that $\mathsf{errcons}(M_f^{\mathsf{w}})$. This lemma and Lemma 47 imply $\mathsf{envcons}(M_f^{\mathsf{w}})$ and $\mathsf{envcons}_{M_f^{\mathsf{w}}}(E)$.

  We now turn to proving that $\mathsf{csub}_m(M_f^{\mathsf{w}})$. By Lemma 68, we have that $w'_x <:^{\mathsf{w}} w'$. For any path at which the pair $\langle E(x), E{\diamond}\ell\rangle$ lies, the subtyping

across the whole path is not invalidated by Lemma 69 and $\mathsf{cons}_m(M_f^{\mathsf{w}})$. As paths can branch on variable nodes by Lemma 76, the subtyping between $E(x)$ and successive nodes in other paths on which $E(x)$ lies holds by Lemma 82. Next, suppose $e_1^{\ell_1} x^{\ell} \in m$ for some e_1 and ℓ_1. If $M^{\mathsf{w}}(E \diamond \ell_1) = \langle T^{\mathsf{w}}, p_1 \rangle$, for some $T^{\mathsf{w}} \in \mathcal{T}^{\mathsf{w}}$ and path p_1, then we need to show that $w' <:^{\mathsf{w}} \pi_2(T^{\mathsf{w}}(p_1))$. We know by the def. of csub_m that $w <:^{\mathsf{w}} \pi_1(T^{\mathsf{w}}(p_1))$ and, from earlier, that $w <:^{\mathsf{w}} w'_x$. Then, the argument follows by Lemma 69. The similar reasoning applies for other places where x can appear in m.

- $e = e_1^{\ell_1} e_2^{\ell_2}$. Clearly (e_1, E) is well-formed and $\mathsf{domvalid}(\ell_1, m, E)$ so by the i.h., we have that $\mathsf{cons}_m(M_1^{\mathsf{w}})$ and $\mathsf{envcons}_{M_1^{\mathsf{w}}}(E)$. Also, (e_2, E) is well-formed and $\mathsf{domvalid}(\ell_2, m, E)$ so we have by the i.h. that $\mathsf{cons}_m(M_2^{\mathsf{w}})$ and $\mathsf{envcons}_{M_2^{\mathsf{w}}}(E)$. By Lemma 50 and Lemma 51 we have that $w_1 = M_2^{\mathsf{w}}(E \diamond \ell_1)$. Hence, we have that $w_1 <:^{\mathsf{w}} [\mathsf{path}(w_1) \mapsto \langle w_2, w \rangle]$, where $w = M_2^{\mathsf{w}}(E \diamond \ell)$, by the def. of $\mathsf{csub}_m(M_2^{\mathsf{w}})$. Then, $\mathsf{envcons}(M_f^{\mathsf{w}})$ and $\mathsf{envcons}_{M_f^{\mathsf{w}}}(E)$ hold trivially, $\mathsf{errcons}(M_f^{\mathsf{w}})$ follows by Lemma 77, and $\mathsf{constcons}(M_f^{\mathsf{w}})$ holds by the def. of $\mathsf{prop}^{\mathsf{w}}$. That is, suppose $m(\ell_1)$ is a lambda expression. Then by Lemma 47, $w'_1 \in \mathcal{T}^{\mathsf{w}} \cup \{\omega\}$. However, $w'_1 \neq \omega$ by Lemma 77.

  We next show that $\mathsf{csub}_m(M_f^{\mathsf{w}})$. First, let $w_o = \pi_2(T^{\mathsf{w}}(\mathsf{path}(w_1)))$. We know that $w_o <:^{\mathsf{w}} w$. If $w_o = \bot$, then $w = \bot$. A more interesting case is when $w_o = \langle \_, p_o \rangle$. By $\mathsf{outcons}_m(M_2^{\mathsf{w}})$, we have that $\mathsf{top}(p_o) = E_b \diamond \ell_b$ where $E_b = E_0.x : E_0 \diamond n_c \diamond x.f : E_0 \diamond n_c \diamond f$, $E_0 = env(\mathsf{root}(\mathsf{path}(w_1)))$, $E_c = \mathsf{top}(\mathsf{path}(w_1))$, and $m(loc(\mathsf{root}(\mathsf{path}(w_1)))) = \mu f.\lambda x.e_b^{\ell_b}$ for some e_b, f, x, and ℓ_b. Hence, $\mathsf{pred}^m_{M_2^{\mathsf{w}}}(E_b \diamond \ell_b, E \diamond \ell)$ by the def. of pred. By Lemma 81, it is the case that $w = \bot$ or $\mathsf{agree}_{E \diamond \ell}(M_2^{\mathsf{w}}(E_b \diamond \ell_b), w)$.

We next need to show that $M_f^{\mathsf{w}}(E_b \diamond \ell_b) <:^{\mathsf{w}} w'$. Note that $M_f^{\mathsf{w}}(E_b \diamond \ell_b) = M_2^{\mathsf{w}}(E_b \diamond \ell_b)$. We know that $w_o <:^{\mathsf{w}} w$ by $\mathsf{csub}_m(M_2^{\mathsf{w}})$ and hence $w'_o <:^{\mathsf{w}} w'$ by Lemma 68. By Lemma 83, we have that $M_2^{\mathsf{w}}(E_b \diamond \ell_b) <:^{\mathsf{w}} w_o$. Therefore, by Lemma 69 we have $M_2^{\mathsf{w}}(E_b \diamond \ell_b) <:^{\mathsf{w}} w'_o$ and thus by Lemma 67 it must be that $M_2^{\mathsf{w}}(E_b \diamond \ell_b) <:^{\mathsf{w}} w'$. The same argument applies for the updated input of w'_1. The remaining argument for $\mathsf{csub}_m(M_f^{\mathsf{w}})$ as well as $\mathsf{det}_m(M_f^{\mathsf{w}})$, $\mathsf{pvcons}(M_f^{\mathsf{w}})$, is the same as in the case of program variables.

- $e = \mu f.\lambda x.\, e_1^{\ell_1}$. First, observe that for any $p_1, p_2 \in T^{\mathsf{w}}$ where $p_1 \neq p_2$, we have that $\mathsf{top}(p_1) \neq \mathsf{top}(p_2)$ by Lemma 79. Hence, for any $p \in T^{\mathsf{w}}$, we have unique n_x, n_f, and E_1. By $\mathsf{csub}_m(M^{\mathsf{w}})$, we know that $T^{\mathsf{w}} <:^{\mathsf{w}} M^{\mathsf{w}}(n_f)$ and $[p \mapsto \langle M^{\mathsf{w}}(n_x), M^{\mathsf{w}}(E_1 \diamond \ell_1)\rangle] <:^{\mathsf{w}} T^{\mathsf{w}}$. Since $\pi_1(T^{\mathsf{w}}(p)) \neq \bot$, we have by the definition of pred and $\mathsf{inpcons}_m(M^{\mathsf{w}})$ that $\mathsf{pred}^m_{M^{\mathsf{w}}}(\mathsf{top}(\mathsf{path}(\pi_1(T^{\mathsf{w}}(p)))), n_x)$ and by the def. of pred we also have $\mathsf{pred}^m_{M^{\mathsf{w}}}(E \diamond \ell, n_f)$. The argument then goes as usual to show that $\mathsf{cons}_m(M_1^{\mathsf{w}})$. Note that in order to show that $\mathsf{csub}_m(M_1^{\mathsf{w}})$ we also additionally use Lemma 70 to show that the join $T_1^{\mathsf{w}} \sqcup^{\mathsf{w}} T_2^{\mathsf{w}}$ does not invalidate the subtyping on the paths at which n_f appears.

  Finally, we switch to proving that the join of all such M_1^{w}, resulting in M_f^{w}, is consistent. First, $\mathsf{errcons}(M_f^{\mathsf{w}})$ follows by Lemma 54 and Lemma 88. Next, $\mathsf{envcons}(M_f^{\mathsf{w}})$ and $\mathsf{envcons}_{M_f^{\mathsf{w}}}(E)$ simply follows by the def. of $\dot{\sqcup}^{\mathsf{w}}$. Similarly, $\mathsf{det}_m(M_f^{\mathsf{w}})$ follows from the def. of $\dot{\sqcup}^{\mathsf{w}}$. Finally, $\mathsf{csub}_m(M_f^{\mathsf{w}})$ and $\mathsf{pvcons}(M_f^{\mathsf{w}})$ follows by Lemma 54, Lemma 86, and Lemma 87.

- $e = e_0^{\ell_0}\, ?\, e_1^{\ell_1} : e_2^{\ell_2}$. Similar to the earlier cases.

□

A.3.5 Connection to Data Flow Semantics

In this subsection, we make the formal connection between path data flow semantics and the regular data flow semantics without the path information. To that end, we first introduce the notion on uniqueness $\mathsf{tpcsuniq}(w)$ of a value w in terms of the uniqueness of paths subject to their top nodes.

$$\mathsf{tpcsuniq}(w) \overset{\text{def}}{\iff} (w \in \{\bot, \omega\} \cup \mathit{Cons}^{\mathsf{w}}) \vee (\exists T^{\mathsf{w}}.\, w = \langle T^{\mathsf{w}}, \_\rangle \wedge (\forall p_1, p_2 \in T^{\mathsf{w}}.$$
$$\mathsf{top}(p_1) = \mathsf{top}(p_2) \implies p_1 = p_2) \wedge \forall p \in T^{\mathsf{w}}.$$
$$\mathsf{tpcsuniq}(\pi_1(T^{\mathsf{w}}(p))) \wedge \mathsf{tpcsuniq}(\pi_2(T^{\mathsf{w}}(p))))$$

We first prove the results stating that two paths drawn from a set of root consistent and deterministic paths must have unique top nodes.

Lemma 89. *Let P be a set of paths such that for every $p \in P$ we have $\mathsf{rootcons}_e(p)$ and $\mathsf{det}^e_{M^{\mathsf{w}}}(p)$ for some program e and a path map M^{w}. Then, for any $p_1, p_2 \in P$, $\mathsf{top}(p_1) = \mathsf{top}(p_2)$ implies $p_1 = p_2$.*

Proof. Similar to the proof of Lemma 79. □

Data propagation does not invalidate the tpcsuniq property.

Lemma 90. *Let w_1 and w_2 be path values s.t. $\mathsf{tpcsuniq}(w_1)$, $\mathsf{tpcsuniq}(w_2)$, and $P = \mathsf{paths}(w_1) \cup \mathsf{paths}(w_2)$ where for every $p \in P$ we have $\mathsf{rootcons}_e(p)$ and $\mathsf{det}^e_{M^{\mathsf{w}}}(p)$ for some program e and a path map M^{w}. Further, let w_3 and w_4 be values such that $\langle w_3, w_4\rangle = \mathsf{prop}^{\mathsf{w}}(w_1, w_2)$. Then, $\mathsf{tpcsuniq}(w_3)$ and $\mathsf{tpcsuniq}(w_4)$.*

Proof. By structural induction on w_1 and w_2. The base cases are trivial. For the inductive case, let $w_1 = \langle T_1^{\mathsf{w}}, p_1\rangle$ and $w_2 = \langle T_2^{\mathsf{w}}, p_2\rangle$ where $p_1 \preceq p_2$. Let $p \in T_2^{\mathsf{w}}$ for some p. By Lemma 89, there is no different $p' \in T_2^{\mathsf{w}}$ such that $\mathsf{top}(p) = \mathsf{top}(p')$. Also, for every $p' \in T_1^{\mathsf{w}}$ s.t. $\mathsf{top}(p') = \mathsf{top}(p)$, we have by Lemma 89 that $p = p'$. In

other words, by Lemma 89 the call site paths both in w_1 and w_2 can be uniquely and consistently identified with their top nodes. The argument then follows by i.h. $\square$

Next, we present the conversion function $\mathsf{strip} : \mathcal{V}^{\mathsf{w}} \rightharpoonup \mathcal{V}$ from path data flow values to regular data flow values that simply strips the path information.

$$\mathsf{strip}(\bot) \stackrel{\text{def}}{=} \bot \qquad \mathsf{strip}(\omega) \stackrel{\text{def}}{=} \omega \qquad \mathsf{strip}(\langle c, p\rangle) \stackrel{\text{def}}{=} c$$

$$\mathsf{strip}(\langle T^{\mathsf{w}}, p\rangle) \stackrel{\text{def}}{=} \Lambda n_{\mathsf{cs}}.\, \textbf{if } \exists p' \in T^{\mathsf{w}}.\, \mathsf{top}(p') = n_{\mathsf{cs}} \quad \text{if } \mathsf{tpcsuniq}(\langle T^{\mathsf{w}}, p\rangle)$$
$$\textbf{then } \langle \mathsf{strip}(\pi_1(T^{\mathsf{w}}(p'))), \mathsf{strip}(\pi_2(T^{\mathsf{w}}(p')))\rangle$$
$$\textbf{else } \langle \bot, \bot\rangle$$

The strip function on maps $\mathsf{strip} : \mathcal{M}^{\mathsf{w}} \rightharpoonup \mathcal{M}$ is defined in the expected way.

Using the strip function, we next connect the subtyping relation between path values and plain data flow values.

Lemma 91. *Let w_1 and w_2 be path values and $P = \mathsf{paths}(w_1) \cup \mathsf{paths}(w_2)$ where for every $p \in P$ we have $\mathsf{rootcons}_e(p)$ and $\mathsf{det}^e_{M^{\mathsf{w}}}(p)$ for some program e and a path map M^{w}. Further, assume that $\mathsf{tpcsuniq}(w_1)$ and $\mathsf{tpcsuniq}(w_2)$. Then, $w_1 <:^{\mathsf{w}} w_2$ implies $\mathsf{strip}(w_1) <:^c \mathsf{strip}(w_2)$.*

Proof. First, note that $\mathsf{tpcsuniq}(w_1)$ implies $w_1 \in \mathsf{dom}(\mathsf{strip})$. The same holds for w_2. The proof goes by structural induction on w_1 and w_2. We first cover the base cases.

- $w_1 = \bot$, $w_1 \in \mathcal{V}^{\mathsf{w}}$. The result immediately follows by the def. of $<:^c$ since $\mathsf{strip}(\bot) = \bot$.

- $w_1 = \omega$.

$$w_2 = \omega \qquad \text{by def. of } <:^{\mathsf{w}}$$

$$\mathsf{strip}(w_1) = \mathsf{strip}(w_2) = \omega \qquad \text{by def. of } \mathsf{strip}$$

- $w_1 = \langle c, p_1 \rangle, w_2 = \langle c, p_2 \rangle, p_1 \preceq p_2$.

$$\mathsf{strip}(w_1) = \mathsf{strip}(w_2) = c \qquad \text{by def. of } \mathsf{strip}$$

For the inductive case, let $w_1 = \langle T_1^{\mathsf{w}}, p_1 \rangle$ and $w_2 = \langle T_2^{\mathsf{w}}, p_2 \rangle$ where $p_1 \preceq p_2$. Let $p \in T_2^{\mathsf{w}}$ for some p. By Lemma 89, there is no different $p' \in T_2^{\mathsf{w}}$ such that $\mathsf{top}(p) = \mathsf{top}(p')$. Also, for every $p' \in T_1^{\mathsf{w}}$ s.t. $\mathsf{top}(p') = \mathsf{top}(p)$, we have by Lemma 89 that $p = p'$. The argument then follows by i.h. □

We next show that the propagation between path values and their stripped counterparts in the data flow semantics yields values that are again related by the strip function.

Lemma 92. *Let w_1 and w_2 be path values such that $w_1 <:^{\mathsf{w}} w_2$ and $P = \mathsf{paths}(w_1) \cup \mathsf{paths}(w_2)$ where for every $p \in P$ we have $\mathsf{rootcons}_e(p)$ and $\mathsf{det}^e_{M^{\mathsf{w}}}(p)$ for some program e and a path map M^{w}. Further, assume that $\mathsf{tpcsuniq}(w_1)$, $\mathsf{tpcsuniq}(w_2)$, and $\langle w_1', w_2' \rangle = \mathsf{prop}^{\mathsf{w}}(w_1, w_2)$ for some path values w_1' and w_2'. Lastly, let $\langle v_1', v_2' \rangle = \mathsf{prop}(\mathsf{strip}(w_1), \mathsf{strip}(w_2))$. Then, $w_1' \in \mathsf{dom}(\mathsf{strip}), w_2' \in \mathsf{dom}(\mathsf{strip}), \mathsf{strip}(w_1') = v_1'$, and $\mathsf{strip}(w_2') = v_2'$.*

Proof. By Lemma 90, we have $\mathsf{tpcsuniq}(w_1')$ and $\mathsf{tpcsuniq}(w_2')$. By the def. of strip, it must also be that $w_1 \in \mathsf{dom}(\mathsf{strip})$ and $w_1 \in \mathsf{dom}(\mathsf{strip})$. The proof continues by the structural induction on w_1 and w_2. We cover the base cases first.

- $w_1 = \bot, w_2 = \bot$.

$$w_1' = w_2' = \bot \qquad \text{by def. of } \mathsf{prop}^{\mathsf{w}}$$
$$\mathsf{strip}(w_1) = \bot, \mathsf{strip}(w_2) = \bot \qquad \text{by def. of } \mathsf{strip}$$
$$v_1' = v_2' = \bot \qquad \text{by def. of } \mathsf{prop}$$

- $w_1 = \langle c, p\rangle, w_2 = \bot$.

$$w_1' = \langle c, p\rangle, w_2' = \langle c, p\rangle \qquad \text{by def. of } \mathsf{prop}^{\mathsf{w}}$$
$$\mathsf{strip}(w_1) = c, \mathsf{strip}(w_2) = \bot \qquad \text{by def. of } \mathsf{strip}$$
$$v_1' = c, v_2' = c \qquad \text{by def. of } \mathsf{prop}$$

- $w_1 = \langle T^{\mathsf{w}}, p\rangle, w_2 = \bot$.

$$w_1' = w_1, w_2' = \langle T_\bot^{\mathsf{w}}, p\rangle \qquad \text{by def. of } \mathsf{prop}^{\mathsf{w}}$$
$$v_1' = \mathsf{strip}(w_1), v_2' = T_\bot \qquad \text{by def. of } \mathsf{prop}$$

- $w_1 = \langle c, p_1\rangle, w_2 = \langle c, p_2\rangle, p_1 \preceq p_2$.

$$w_1' = \langle c, p_1\rangle, w_2' = \langle c, p_2\rangle \qquad \text{by def. of } \mathsf{prop}^{\mathsf{w}}$$
$$\mathsf{strip}(w_1) = c, \mathsf{strip}(w_2) = c \qquad \text{by def. of } \mathsf{strip}$$
$$v_1' = c, v_2' = c \qquad \text{by def. of } \mathsf{prop}$$

- $w_1 = \omega, w_2 = \omega$.

$$w_1' = \omega, \omega \qquad \text{by def. of } \mathsf{prop}^{\mathsf{w}}$$
$$\mathsf{strip}(w_1) = \omega, \mathsf{strip}(w_2) = \omega \qquad \text{by def. of } \mathsf{strip}$$
$$v_1' = \omega, v_2' = \omega \qquad \text{by def. of } \mathsf{prop}$$

For the inductive case, let $w_1 = \langle T_1^{\mathsf{w}}, p_1\rangle$ and $w_2 = \langle T_2^{\mathsf{w}}, p_2\rangle$ where $p_1 \preceq p_2$. Let $p \in T_2^{\mathsf{w}}$ for some p. By Lemma 89, there is no different $p' \in T_2^{\mathsf{w}}$ such that $\mathsf{top}(p) = \mathsf{top}(p')$. Also, for every $p' \in T_1^{\mathsf{w}}$ s.t. $\mathsf{top}(p') = \mathsf{top}(p)$, we have by Lemma 89 that

$p = p'$. Hence, the call site paths seen in T_1^{w} and T_2^{w} are uniquely identified by their top nodes. The argument then follows by def. of paths, $\mathsf{tpcsuniq}$, and i.h. $\square$

Lemma 93. *Let w_1 and w_2 be path values such that $P = \mathsf{paths}(w_1) \cup \mathsf{paths}(w_2)$ where for every $p_1 p_2 \in P$ we have $\mathsf{top}(p_1) = \mathsf{top}(p_2) \implies p_1 = p_2$. Further, assume that $\mathsf{tpcsuniq}(w_1)$, $\mathsf{inpcons}_e(w_1)$, $\mathsf{outcons}_e(w_1)$, $\mathsf{tiocons}(w_1)$, $\mathsf{tpcsuniq}(w_2)$, $\mathsf{inpcons}_e(w_2)$, $\mathsf{outcons}_e(w_2)$, and $\mathsf{tiocons}(w_2)$. Further, assume $w_1 \in \mathsf{dom}(\mathsf{strip})$, $w_2 \in \mathsf{dom}(\mathsf{strip})$, $\mathsf{strip}(w_1) = v_1$, and $\mathsf{strip}(w_2) = v_2$. Lastly, we assume that w_1 and w_2 agree on the paths. Then, $w_1 \sqcup^{\mathsf{w}} w_2 \in \mathsf{dom}(\mathsf{strip})$ and $\mathsf{strip}(w_1 \sqcup^{\mathsf{w}} w_2) = v_1 \sqcup v_2$.*

Proof. We carry the proof by structural induction on w_1 and w_2. We first cover the base cases.

- $w_1 = \bot$.

$$\begin{aligned} w_1 \sqcup^{\mathsf{w}} w_2 &= w_2 && \text{by def. of } \sqcup^{\mathsf{w}} \\ v_1 &= \bot && \text{by def. of } \mathsf{strip} \\ v_1 \sqcup v_2 = v_2 = \mathsf{strip}(w_2) &= \mathsf{strip}(w_1 \sqcup^{\mathsf{w}} w_2) && \text{by def. of } \sqcup \end{aligned}$$

- $w_2 = \bot$. Similar to the previous case.

- $w_1 = \omega$.

$$\begin{aligned} w_1 \sqcup^{\mathsf{w}} w_2 &= \omega && \text{by def. of } \sqcup^{\mathsf{w}} \\ v_1 &= \omega && \text{by def. of } \sqcup \\ v_1 \sqcup v_2 &= \omega && \text{by def. of } \sqcup \\ \mathsf{strip}(\omega) &= \omega && \text{by def. of } \mathsf{strip} \end{aligned}$$

- $w_1 = \langle c_1, p\rangle, w_2 = \langle c_2, p\rangle, c_1 \neq c_2$.

$$\begin{aligned} w_1 \sqcup^{\mathsf{w}} w_2 &= \omega && \text{by def. of } \sqcup^{\mathsf{w}} \\ v_1 = c_1, v_2 &= c_2 && \text{by def. of } \mathsf{strip} \\ v_1 \sqcup v_2 &= \omega && \text{by def. of } \sqcup \\ \mathsf{strip}(\omega) &= \omega && \text{by def. of } \mathsf{strip} \end{aligned}$$

- $w_1 = \langle c, p\rangle, w_2 = \langle c, p\rangle$.

$$\begin{aligned} w_1 \sqcup^{\mathsf{w}} w_2 &= \langle c, p\rangle && \text{by def. of } \sqcup^{\mathsf{w}} \\ v_1 = c, v_2 &= c && \text{by def. of } \mathsf{strip} \\ v_1 \sqcup v_2 &= c && \text{by def. of } \sqcup \\ \mathsf{strip}(\langle c, p\rangle) &= c && \text{by def. of } \mathsf{strip} \end{aligned}$$

- $w_1 = \langle c, p\rangle, w_2 = \langle T^{\mathsf{w}}, p\rangle$.

$$\begin{aligned} w_1 \sqcup^{\mathsf{w}} w_2 &= \omega && \text{by def. of } \sqcup^{\mathsf{w}} \\ v_1 = c, v_2 &\in \mathcal{T} && \text{by def. of } \mathsf{strip} \\ v_1 \sqcup v_2 &= \omega && \text{by def. of } \sqcup \\ \mathsf{strip}(\omega) &= \omega && \text{by def. of } \mathsf{strip} \end{aligned}$$

- *Other.* Similar for the other cases.

For the inductive case, let $w_1 = \langle T_1^{\mathsf{w}}, p_1\rangle$ and $w_2 = \langle T_2^{\mathsf{w}}, p_2\rangle$ where $p_1 \preceq p_2$. Let $p \in T_2^{\mathsf{w}}$ for some p. By Lemma 89, there is no different $p' \in T_2^{\mathsf{w}}$ such that $\mathsf{top}(p) = \mathsf{top}(p')$. Also, for every $p' \in T_1^{\mathsf{w}}$ s.t. $\mathsf{top}(p') = \mathsf{top}(p)$, we have by Lemma 89 that $p = p'$. Hence, the call site paths seen in T_1^{w} and T_2^{w} are uniquely identified by their top nodes.

- $p \in T_1^{\mathsf{w}}, p \in T_2^{\mathsf{w}}$. Let $T_1^{\mathsf{w}}(p) = \langle w_{1i}, w_{1o}\rangle$ and $T_2^{\mathsf{w}}(p) = \langle w_{2i}, w_{2o}\rangle$. If $w_{1i}, w_{2i} \notin \{\bot, \omega\}$, then $\mathsf{path}(w_{2i}) = \mathsf{path}(w_{1i})$ by $\mathsf{path}(w_{2i}), \mathsf{path}(w_{1i}) \in P$ (by def. of

paths), Lemma 89, and $\mathsf{top}(\mathsf{path}(w_{2i})) = \mathsf{top}(\mathsf{path}(w_{1i}))$ which follows by the def. of $\mathsf{inpcons}$. The same property holds for the outputs by the same reasoning and def. of $\mathsf{outcons}$. The argument then follows from the i.h.

- $p \in T_1^{\mathsf{w}}, p \notin T_2^{\mathsf{w}}$. By $\mathsf{tiocons}(T_2^{\mathsf{w}})$, we then have that $T_2^{\mathsf{w}}(p) = \langle \bot, \bot \rangle$. The argument then follows trivially since $\mathsf{strip}(w_2)(\mathsf{top}(p)) = \langle \bot, \bot \rangle$.

- *Other cases.* Handled similarly.

□

We can now establish a result connecting the path based data flow semantics and plain data flow semantics. For that purpose, we first state an almost obvious result.

Lemma 94. *Let M^{w} be a consistent map $\mathsf{cons}_e(M^{\mathsf{w}})$ subject to a program e. Then, $M^{\mathsf{w}} \in \mathsf{dom}(\mathsf{strip})$.*

Proof. The result follows from Lemma 79. □

We can now state our main theorem.

Theorem 8. *Let e^ℓ be an expression of a program m, M^{w} a consistent map $\mathsf{cons}_m(M^{\mathsf{w}})$, E an environment where $\mathsf{domvalid}(\ell, m, E)$, $\mathsf{envcons}_{M^{\mathsf{w}}}(E)$, and (e, E) is well-formed. Further, let M be a concrete map such that $M = \mathsf{strip}(M^{\mathsf{w}})$ and k a step index fuel. If $\mathsf{step}_k^{\mathsf{w}}[\![e^\ell]\!](E)(M^{\mathsf{w}}) = M_f^{\mathsf{w}}$ and $\mathsf{step}_k[\![e^\ell]\!](E)(M) = M_f$, for some maps M_f^{w} and M_f, then $M_f^{\mathsf{w}} \in \mathsf{dom}(\mathsf{strip})$ and $\mathsf{strip}(M_f^{\mathsf{w}}) = M_f$.*

Proof. The proof goes by natural induction on k and structural induction on e. When $k = 0$, the argument is trivial. Otherwise, the proof branches on e.

- $e = c$. Let $w = M^{\mathsf{w}}(E{\diamond}\ell)$, $w_f = M_f^{\mathsf{w}}(E{\diamond}\ell)$, $v = M(E{\diamond}\ell)$, and $v_f = M_f(E{\diamond}\ell)$. By $\mathsf{cons}_m(M^{\mathsf{w}})$, $w \in \{\bot, \omega, \langle c, E{\diamond}\ell\rangle\}$. If $w = \bot$, then $w_f = \langle c, E{\diamond}\ell\rangle$, $v = \bot$, $v_f = c$, and the result follows from the definition of strip. If $w = \langle c, E{\diamond}\ell\rangle$, then $w_f = w$ and $v_f = v$, from which the result follows immediately. Lastly, if $w = \omega$, then by $\mathsf{errcons}(M^{\mathsf{w}})$ (implied by $\mathsf{cons}_e(M^{\mathsf{w}})$) we have $M^{\mathsf{w}} = M_\omega^{\mathsf{w}}$ and hence $M_f^{\mathsf{w}} = M_\omega^{\mathsf{w}}$, $M = M_\omega$, as well as $M_f = M_\omega$. The result then follows simply from the definition of strip.

- $e = x$. Let $w_x = M^{\mathsf{w}}(E(x))$, $w = M^{\mathsf{w}}(E{\diamond}\ell)$, $v_x = M(E(x))$, and $v = M(E{\diamond}\ell)$. By $\mathsf{csub}_m(M^{\mathsf{w}})$ (implied by $\mathsf{cons}_m(M^{\mathsf{w}})$) and Lemma 81, we have $w_x <:^{\mathsf{w}} w$. Further, by $\mathsf{cons}_m(M^{\mathsf{w}})$, we have $\mathsf{tpcsuniq}(w_1)$, $\mathsf{tpcsuniq}(w_2)$, and that all paths in $\mathsf{paths}(w_1) \cup \mathsf{paths}(w_2)$ are deterministic and root consistent. The result then follows from Lemma 92.

- $e = e_1^{\ell_1} e_2^{\ell_2}$. Clearly, we have $\mathsf{domvalid}(\ell_1, m, E)$ and so by i.h. $M_1^{\mathsf{w}} \in \mathsf{dom}(\mathsf{strip})$ and $M_1 = \mathsf{strip}(M_1^{\mathsf{w}})$. By Theorem 7 it is the case that $\mathsf{envcons}_{M_1^{\mathsf{w}}}(E)$ and $\mathsf{cons}_m(M_1^{\mathsf{w}})$. As $\mathsf{domvalid}(\ell_2, m, E)$ trivially, we have by i.h. that $M_2^{\mathsf{w}} \in \mathsf{dom}(\mathsf{strip})$ and $M_2 = \mathsf{strip}(M_2^{\mathsf{w}})$. Also, again by Theorem 7 we have $\mathsf{envcons}_{M_2^{\mathsf{w}}}(E)$ and $\mathsf{cons}_m(M_2^{\mathsf{w}})$.

  By Lemmas 50 and 51, we have $w_1 = M_2^{\mathsf{w}}(E{\diamond}\ell_1)$ and so by $\mathsf{csub}_m(M_2^{\mathsf{w}})$ (implied by $\mathsf{cons}_m(M_2^{\mathsf{w}})$) it must be $w_1 <:^{\mathsf{w}} w'$ where $w' = \langle[\mathsf{path}(w_1) \mapsto \langle w_2, M_2^{\mathsf{w}}(E{\diamond}\ell_2)\rangle], \mathsf{path}(w_1)\rangle$. Note that by the def. of strip and $M_2 = \mathsf{strip}(M_2^{\mathsf{w}})$, we have $\mathsf{strip}(w') = [\mathsf{top}(\mathsf{path}(w_1)) \mapsto \langle v_2, M_2(E{\diamond}\ell_2)\rangle]$. Further, by $\mathsf{cons}_m(M^{\mathsf{w}})$, we have $\mathsf{tpcsuniq}(w_1)$, $\mathsf{tpcsuniq}(w')$, and that all paths in $\mathsf{paths}(w_1) \cup \mathsf{paths}(w')$ are deterministic and root consistent. The result then follows from Lemma 92.

- $e = \mu f.\lambda x.e_1^{\ell_1}$. Let $w = M^{\mathsf{w}}(E{\diamond}\ell)$ and $v = M(E{\diamond}\ell)$. By $\mathsf{cons}_m(M^{\mathsf{w}})$,

$w \in \{\bot, \omega, \langle T^{\mathsf{w}}, E{\diamond}\ell\rangle\}$ for some $T^{\mathsf{w}} \in \mathcal{T}^{\mathsf{w}}$. If $w = \bot$, then $T_w = \langle T^{\mathsf{w}}_{wbot}, E{\diamond}\ell\rangle$, $v = \bot, T = T_\bot$. In that case, $M^{\mathsf{w}}_f = M^{\mathsf{w}}[E{\diamond}\ell \mapsto T_w]$, $M_f = M[E{\diamond}\ell \mapsto T]$, the result follows from the definition of strip. If $w = \omega$, then by $\mathsf{errcons}(M^{\mathsf{w}})$ (implied by $\mathsf{cons}_e(M^{\mathsf{w}})$) we have $M^{\mathsf{w}} = M^{\mathsf{w}}_\omega$ and hence $M^{\mathsf{w}}_f = M^{\mathsf{w}}_\omega$ by Lemma 49, $M = M_\omega$ and thus $M_f = M_\omega$ by Lemma 27. The result then follows simply from the definition of strip. Lastly, if $w = \langle T^{\mathsf{w}}, E{\diamond}\ell\rangle$, then $T_w = w$ and $T = v$. The argument then goes as follows.

By Lemma 79 and def. of strip, we have $\{n_{\mathsf{cs}} \mid n_{\mathsf{cs}} \in T\} = \{\mathsf{top}(p) \mid p \in T^{\mathsf{w}}\}$. That is, for every $p \in T^{\mathsf{w}}$ there is a unique $n_{\mathsf{cs}} \in T$ such that $\mathsf{top}(p) = n_{\mathsf{cs}}$, and vice versa. Then, by $\mathsf{csub}_m(M^{\mathsf{w}})$ (implied by $\mathsf{cons}_m(M^{\mathsf{w}})$) it is that $\langle[p \mapsto \langle M^{\mathsf{w}}(n_x), M^{\mathsf{w}}(E_1{\diamond}\ell_1)\rangle], p\rangle <:^{\mathsf{w}} T_w$. Similarly, by $\mathsf{csub}_m(M^{\mathsf{w}})$ and Lemma 81 we have $T_w <:^{\mathsf{w}} M^{\mathsf{w}}(n_f)$. Also, by $\mathsf{cons}_m(M^{\mathsf{w}})$ we have $\mathsf{tpcsuniq}(M^{\mathsf{w}}(n_x))$, $\mathsf{tpcsuniq}(M^{\mathsf{w}}(E_1{\diamond}\ell_1))$, $\mathsf{tpcsuniq}(T_w)$, $\mathsf{tpcsuniq}(M^{\mathsf{w}}(n_f))$, and all paths in $\mathsf{paths}(M^{\mathsf{w}}(n_x)) \cup \mathsf{paths}(M^{\mathsf{w}}(n_f)) \cup \mathsf{paths}(T_w) \cup \mathsf{paths}M^{\mathsf{w}}(E_1{\diamond}\ell_1)$ are deterministic and root consistent. Then by Lemma 92 and Lemma 93 we have $M^{\mathsf{w}}_1 \in \mathsf{dom}(\mathsf{strip})$ and $\mathsf{strip}(M^{\mathsf{w}}_1) = M_1$. By Lemmas 47 and 77 as well as $\mathsf{envcons}_{M^{\mathsf{w}}}(E)$ we have $\mathsf{envcons}_{M^{\mathsf{w}}_1}(E_1)$. Also, $\mathsf{domvalid}(\ell_1, m, E_1)$ holds by the definition of E_1 and $\mathsf{domvalid}(\ell, m, E)$. Then by the i.h., $\mathsf{step}^{\mathsf{w}}_k[\![e_1]\!](E_1)(M^{\mathsf{w}}_1) \in \mathsf{dom}(\mathsf{strip})$ and $\mathsf{strip}(\mathsf{step}^{\mathsf{w}}_k[\![e_1]\!](E_1)(M^{\mathsf{w}}_1)) = \mathsf{step}_k[\![e_1]\!](E_1)(M_1)$.

Finally, observe that for each two different $M^{\mathsf{w}}_1, M^{\mathsf{w}}_2 \in \vec{M}^{\mathsf{w}}$, M^{w}_1 and M^{w}_2 agree on all nodes except possibly those in $\mathsf{rng}(E)$ by Lemma 52. By $\mathsf{envcons}_{M^{\mathsf{w}}}(E)$, for every node n in $\mathsf{rng}(E)$ we have $M^{\mathsf{w}}(n) \notin \{\bot, \omega\}$. Then by Lemma 56 we have $\mathsf{path}(M^{\mathsf{w}}(n)) = \mathsf{path}(M^{\mathsf{w}}_1(n))$, $\mathsf{path}(M^{\mathsf{w}}(n)) = \mathsf{path}(M^{\mathsf{w}}_2(n))$, for every node n in $\mathsf{rng}(E)$, which clearly implies $\mathsf{path}(M^{\mathsf{w}}_2(n)) = \mathsf{path}(M^{\mathsf{w}}_1(n))$. The result then follows from $\mathsf{fin}(M^{\mathsf{w}})$, which implies $|\{\mathsf{top}(p) \mid p \in T^{\mathsf{w}}\}|$ is finite,

and Lemma 93.

- $e = e_0^{\ell_0} \,?\, e_1^{\ell_1} : e_2^{\ell_2}$. Similar to the earlier cases.

□

We additionally show that consistency of a path map implies that its stripped plain data flow map has the data flow invariant property.

Lemma 95. *Let M^{w} be consistent map $\mathsf{cons}_e(M^{\mathsf{w}})$ subject to a program e. Then, $df(\mathsf{strip}(M^{\mathsf{w}}))$.*

Proof. First, observe that by Lemma 94 we have $M^{\mathsf{w}} \in \mathsf{dom}(\mathsf{strip})$. Next, let $M = \mathsf{strip}(M^{\mathsf{w}})$. Then, $\mathsf{errcons}(M)$ follows by $\mathsf{errcons}(M^{\mathsf{w}})$ (implied by $\mathsf{cons}_e(M^{\mathsf{w}})$) and the definition of strip. The same argument can be given to show $\mathsf{fin}(M)$, $\mathsf{tiocons}(M)$, $\mathsf{constcons}_e(M)$, and $\mathsf{envcons}(M)$. Finally, $\mathsf{csub}_e(M)$ follows from $\mathsf{csub}_e(M^{\mathsf{w}})$, $\mathsf{rootcons}_e(M^{\mathsf{w}})$, $\mathsf{tpcons}(M^{\mathsf{w}})$, and $\mathsf{det}_e(M^{\mathsf{w}})$. To see this in more detail, first observe that the set of reachable (non-$\bot$) nodes of M^{w} and M is the same by the def. of strip. For every reachable node n that corresponds to a non-leaf expression of e, we have that $M^{\mathsf{w}}(n) = \langle \_, p \rangle$ for some path p. By $\mathsf{tpcons}(M^{\mathsf{w}})$, $\mathsf{top}(p) = n$ and by $\mathsf{rootcons}_e(M^{\mathsf{w}})$ as well as Lemma 73 we have that $|p| > 1$. Then, for the predecessor n' of n in p, we have that $M^{\mathsf{w}}(n') <:^{\mathsf{w}} M^{\mathsf{w}}(n)$ by $\mathsf{csub}_e(M^{\mathsf{w}})$ and $\mathsf{pred}^e_{M^{\mathsf{w}}}(n', n)$ by $\mathsf{det}_e(M^{\mathsf{w}})$. The result then follows from the definition of pred, Lemma 91, and the fact that $\mathsf{cons}_e(M^{\mathsf{w}})$ implies $\mathsf{tpcsuniq}(M^{\mathsf{w}}(n')) \wedge \mathsf{tpcsuniq}(M^{\mathsf{w}}(n))$. □

A.4 Relational Semantics Proofs

We first prove that our definitions of meets and joins over relational values are correct.

Proof (Proof of Lemma 4)**.**

Proof. The proof follows that of Lemma 1.

□

We next prove the monotonicity of the concretization function for relational values.

Lemma 96. *Let* $r_1 \in \mathcal{V}^{\mathsf{r}}_N$*,* $r_2 \in \mathcal{V}^{\mathsf{r}}_N$*, and* $r_1 \sqsubseteq^{\mathsf{r}} r_2$*. Then,* $\gamma^{\mathsf{r}}_N(r_1) \subseteq \gamma^{\mathsf{r}}_N(r_2)$*.*

Proof. The proof goes by structural induction on r_2. We first cover the base cases.

- $r_2 = \bot^{\mathsf{r}}$. Then $r_1 = \bot^{\mathsf{r}}$ and the result follows trivially.
- $r_2 = \top^{\mathsf{r}}$. Trivially by the def. of $\gamma^{\mathsf{r}}_N(\top^{\mathsf{r}})$.
- $r_2 = R^{\mathsf{r}}_2$. Then either $r_1 = \bot^{\mathsf{r}}$ or $r_1 = R^{\mathsf{r}}_1$ s.t. $R^{\mathsf{r}}_1 \subseteq R^{\mathsf{r}}_2$. Since the former case is trivial due to $\forall r.\, \gamma^{\mathsf{r}}_N(\bot^{\mathsf{r}}) \subseteq \gamma^{\mathsf{r}}_N(r)$ by the def. of γ^{r}_N, we focus on the latter case. Here,

$$R^{\mathsf{r}}_1 \subseteq R^{\mathsf{r}}_2 \implies \{\langle M, c\rangle \mid D^{\mathsf{r}} \in R^{\mathsf{r}}_1 \wedge D^{\mathsf{r}}(\nu) = c\} \subseteq \{\langle M, c\rangle \mid D^{\mathsf{r}} \in R^{\mathsf{r}}_2 \wedge D^{\mathsf{r}}(\nu) = c\} \implies$$
$$\{\langle M, c\rangle \mid D^{\mathsf{r}} \in R^{\mathsf{r}}_1 \wedge D^{\mathsf{r}}(\nu) = c \wedge \forall n \in N.M(n) \in \gamma^{\mathsf{d}}(M(n))\} \subseteq$$
$$\{\langle M, c\rangle \mid D^{\mathsf{r}} \in R^{\mathsf{r}}_2 \wedge D^{\mathsf{r}}(\nu) = c \wedge \forall n \in N.M(n) \in \gamma^{\mathsf{d}}(M(n))\} \implies$$
$$\gamma^{\mathsf{r}}_N(r_1) \subseteq \gamma^{\mathsf{r}}_N(r_2).$$

For the induction step $r = T^{\mathsf{r}}_2$, if $r_1 = \bot^{\mathsf{r}}$ the argument is trivial. We hence focus on the case where $r_1 = T^{\mathsf{r}}_1$. Let $T^{\mathsf{r}}_1(n_{\mathsf{cs}}) = \langle r_{1i}, r_{1o}\rangle$ and $T^{\mathsf{r}}_2(n_{\mathsf{cs}}) = \langle r_{2i}, r_{2o}\rangle$ for any call site n_{cs}. By $T^{\mathsf{r}}_1 \sqsubseteq^{\mathsf{r}} T^{\mathsf{r}}_2$, we have $r_{1i} \sqsubseteq^{\mathsf{r}} r_{2i}$ and $r_{1o} \sqsubseteq^{\mathsf{r}} r_{2o}$. By i.h.

on r_{2i} and r_{2o}, it follows $\gamma^{\mathsf{r}}_N(r_{1i}) \subseteq \gamma^{\mathsf{r}}_N(r_{2i})$ and $\gamma^{\mathsf{r}}_N(r_{1o}) \subseteq \gamma^{\mathsf{r}}_N(r_{2o})$, which implies $\gamma^{\mathsf{r}}_N(T^{\mathsf{r}}_1) \subseteq \gamma^{\mathsf{r}}_N(T^{\mathsf{r}}_2)$ by the def. of γ^{r}_N. □

It can be also shown that the concretization of relational values does not impose any constraints on the values for nodes not in the given scope.

Lemma 97. *For any $n \notin N$, n is unconstrained in $\gamma^{\mathsf{r}}_N(r_N)$ for any relational value r. That is, $\forall n \notin N, v, v', M.\langle M, v\rangle \in \gamma^{\mathsf{r}}_N(r_N) \Longrightarrow \langle M[n \to v'], v\rangle \in \gamma^{\mathsf{r}}_N(r_N)$.*

Proof. The proof goes by structural induction on r. The base cases are trivial. The induction case for tables follows from the i.h. and the fact that for any table T^{r} and call site n_{cs}, n_{cs} is not in the scope of $\pi_i(T^{\mathsf{r}}(n_{\mathsf{cs}}))$. □

The next proof shows that the concretization function satisfies the meet-morphism requirement.

Theorem 9 (Complete Meet Morphism on Rel. Values)**.** *Let $V^r \in \wp(\mathcal{V}^{\mathsf{r}}_N)$. Then $\gamma^{\mathsf{r}}_N(\sqcap^{\mathsf{r}} V^r) = \bigcap_{r \in V^r} \gamma^{\mathsf{r}}_N(r)$.*

Proof. We carry the proof by case analysis on elements of V^r and induction on the minimum depth thereof when V^r consists of tables only.

We first consider the trivial (base) cases where V^r is not a set of multiple relational tables.

- $V^r = \emptyset$. Here, $\gamma^{\mathsf{r}}_N(\sqcap^{\mathsf{r}}\emptyset) = \gamma^{\mathsf{r}}_N(\top^{\mathsf{r}}) = \mathcal{M} \times \mathcal{V} = \bigcap \emptyset$.
- $V^r = \{r\}$. Trivial.
- $R^{\mathsf{r}} \in V^r, T^{\mathsf{r}} \in V^r$. We have $\sqcap^{\mathsf{r}}\{R^{\mathsf{r}}, T^{\mathsf{r}}\} = \bot^{\mathsf{r}}$ and since $\bot^{\mathsf{r}}$ is the bottom element, $\sqcap^{\mathsf{r}} V^r = \bot^{\mathsf{r}}$. Similarly, $\gamma^{\mathsf{r}}_N(R^{\mathsf{r}}) \cap \gamma^{\mathsf{r}}_N(T^{\mathsf{r}}) = \gamma^{\mathsf{r}}_N(\bot^{\mathsf{r}})$ and since $\forall r. \gamma^{\mathsf{r}}_N(\bot^{\mathsf{r}}) \subseteq \gamma^{\mathsf{r}}_N(r)$, it follows $\bigcap_{r \in V^r} \gamma^{\mathsf{r}}_N(r) = \gamma^{\mathsf{r}}_N(\bot^{\mathsf{r}}) = \gamma^{\mathsf{r}}_N(\sqcap^{\mathsf{r}} V^r)$.

- $V^r \subseteq \mathcal{R}^{\mathsf{r}}_N$. Here,

$$\gamma^{\mathsf{r}}_N(\sqcap^{\mathsf{r}} V^r) = \gamma^{\mathsf{r}}_N(\bigcap_{R^{\mathsf{r}} \in V^r} R^{\mathsf{r}})$$

$$= \{\langle M, c\rangle \mid D^{\mathsf{r}} \in (\bigcap_{R^{\mathsf{r}} \in V^r} R^{\mathsf{r}})) \wedge D^{\mathsf{r}}(\nu) = c \wedge \forall n \in N.M(n) \in \gamma^{\mathsf{d}}(M(n))\} \cup \gamma^{\mathsf{r}}_N(\bot^{\mathsf{r}})$$

[by def. of γ^{r}_N]

$$= \bigcap_{R^{\mathsf{r}} \in V^r} (\{\langle M, c\rangle \mid D^{\mathsf{r}} \in R^{\mathsf{r}} \wedge D^{\mathsf{r}}(\nu) = c \wedge \forall n \in N.M(n) \in \gamma^{\mathsf{d}}(M(n))\} \cup \gamma^{\mathsf{r}}_N(\bot^{\mathsf{r}}))$$

[by uniqueness/non-overlapping of γ^{d}]

$$= \bigcap_{R^{\mathsf{r}} \in V^r} \gamma^{\mathsf{r}}_N(R^{\mathsf{r}})$$ **[by def. of γ^{r}_N]**

- $\top^{\mathsf{r}} \in V^r$. Here, $\sqcap^{\mathsf{r}} V^r = \sqcap^{\mathsf{r}}(V^r/\{\top^{\mathsf{r}}\})$ and $\bigcap_{r \in V^r} \gamma^{\mathsf{r}}_N(r) = \bigcap_{r \in V^r, r \neq \top^{\mathsf{r}}} \gamma^{\mathsf{r}}_N(r)$ since $\gamma^{\mathsf{r}}_N(\top^{\mathsf{r}}) = \mathcal{M} \times \mathcal{V}$. The set $V^r/\{\top^{\mathsf{r}}\}$ either falls into one of the above cases or consists of multiple tables, which is the case we show next.

Let $V^r \subseteq \mathcal{T}^{\mathsf{r}}_N$ and $|V^r| > 1$. Let d be the minimum depth of any table in V^r.

$$\bigcap_{T^{\mathsf{r}} \in V^r} \gamma^{\mathsf{r}}_N(T^{\mathsf{r}}) = \{\langle M, T\rangle \mid \forall n_{\mathsf{cs}}.\, T(n_{\mathsf{cs}}) = \langle v_i, v_o\rangle \wedge \langle M_i, v_i\rangle \in \bigcap_{T^{\mathsf{r}} \in V^r} \gamma^{\mathsf{r}}(\pi_1(T^{\mathsf{r}}(n_{\mathsf{cs}}))) \wedge$$

$$\langle M_o, v_o\rangle \in \bigcap_{T^{\mathsf{r}} \in V^r} \gamma^{\mathsf{r}}(\pi_2(T^{\mathsf{r}}(n_{\mathsf{cs}}))) \wedge M_i =_{\setminus\{n_{\mathsf{cs}}\}} M \wedge M_o = M_i[n_{\mathsf{cs}} \mapsto v_i]\} \cup \gamma^{\mathsf{r}}_N(\bot^{\mathsf{r}})$$

[by def. of γ^{r}_N and n_{cs} not in the scope of $\pi_1(T^{\mathsf{r}}(n_{\mathsf{cs}}))$]

$$= \{\langle M, T\rangle \mid \forall n_{\mathsf{cs}}.\, T(n_{\mathsf{cs}}) = \langle v_i, v_o\rangle \wedge \langle M_i, v_i\rangle \in \gamma^{\mathsf{r}}_N(\sqcap^{\mathsf{r}}_{T^{\mathsf{r}} \in V^r} \pi_1(T^{\mathsf{r}}(n_{\mathsf{cs}}))) \wedge$$

$$\langle M_o, v_o\rangle \in \gamma^{\mathsf{r}}_N(\sqcap^{\mathsf{r}}_{T^{\mathsf{r}} \in V^r} \pi_2(T^{\mathsf{r}}(n_{\mathsf{cs}}))) \wedge M_i =_{\setminus\{n_{\mathsf{cs}}\}} M \wedge M_o = M_i[n_{\mathsf{cs}} \mapsto v_i]\} \cup \gamma^{\mathsf{r}}_N(\bot^{\mathsf{r}})$$

[by i.h. on the min. depth of $\{\pi_1(T^{\mathsf{r}}(n_{\mathsf{cs}})) \mid T^{\mathsf{r}} \in V^r\}$ and $\{\pi_2(T^{\mathsf{r}}(n_{\mathsf{cs}})) \mid T^{\mathsf{r}} \in V^r\}$]

$$= \gamma^{\mathsf{r}}_N(\sqcap^{\mathsf{r}}_{T^{\mathsf{r}} \in V^r} T^{\mathsf{r}})$$ **[by def. of γ^{r}_N]**

□

We can now state the monotonicity of abstraction function for the relational domain.

Lemma 98. *α^{r}_N is monotone.*

Proof. The result follows from the properties of Galois connections. □

The meet morphism requirement is also satisfied by the concretization function defined over relational maps.

Theorem 10 (Complete Meet Morphism on Rel. Maps)**.** *Let $m^r \in \wp(\mathcal{M}^{\mathsf{r}})$. Then $\dot{\gamma}^{\mathsf{r}}(\dot{\sqcap}^{\mathsf{r}}\, m^r) = \bigcap_{M^{\mathsf{r}} \in m^r} \dot{\gamma}^{\mathsf{r}}(M^{\mathsf{r}})$.*

Proof.

$$\begin{aligned}
\bigcap_{M^{\mathsf{r}} \in m^r} \dot{\gamma}^{\mathsf{r}}(M^{\mathsf{r}}) &= \{M \mid \forall n.\, \langle M, M(n)\rangle \in \bigcap_{M^{\mathsf{r}} \in m^r} \gamma^{\mathsf{r}}_N(M^{\mathsf{r}}(n))\} \\
&= \{M \mid \forall n.\, \langle M, M(n)\rangle \in \gamma^{\mathsf{r}}_N(\sqcap^{\mathsf{r}}_{M^{\mathsf{r}} \in m^r} M^{\mathsf{r}}(n))\} && \textbf{[by Lemma 9]} \\
&= \dot{\gamma}^{\mathsf{r}}(\dot{\sqcap}^{\mathsf{r}}\, m^r) && \textbf{[by def. of } \dot{\gamma}^{\mathsf{r}} \textbf{ and } \dot{\sqcap}^{\mathsf{r}}\textbf{]}
\end{aligned}$$

□

A.4.1 Domain Operations

In this subsection, we state the basic properties and the corresponding proofs of strengthening operations on relational values.

A.4.1.1 Strengthening

We first show that our strengthening operations indeed perform strengthening.

Lemma 99. *Let $r_1 \in \mathcal{V}^{\mathsf{r}}_N$, $r \in \mathcal{V}^{\mathsf{r}}_N$, and $n \in \mathcal{N}$. Then $r[n \leftarrow r_1] \sqsubseteq^{\mathsf{r}} r$.*

Proof. In the below calculational proof, let $S = \{\langle M'[n \mapsto v'], v\rangle \mid \langle M', v'\rangle \in$

$\gamma^{\mathsf{r}}_N(r_1)\}$.

$$\begin{aligned}
\alpha^{\mathsf{r}}_N(\gamma^{\mathsf{r}}_N(r)) \sqsubseteq^{\mathsf{r}} r && \text{by prop. of Galois conn.}\\
\alpha^{\mathsf{r}}_N(\gamma^{\mathsf{r}}_N(r) \cap S) \sqsubseteq^{\mathsf{r}} \alpha^{\mathsf{r}}_N(\gamma^{\mathsf{r}}_N(r)) && \text{by Lemma 98}\\
\alpha^{\mathsf{r}}_N(\gamma^{\mathsf{r}}_N(r) \cap S) \sqsubseteq^{\mathsf{r}} r && \text{by transitivity}\\
r[n \leftarrow r_1] \sqsubseteq^{\mathsf{r}} r && \text{by def. of } \cdot[\cdot \leftarrow \cdot]
\end{aligned}$$

□

Lemma 100. *Let $r \in \mathcal{V}^{\mathsf{r}}_N$, $n_1 \in \mathcal{N}$, and $n_2 \in \mathcal{N}$. Then $r[n_1{=}n_2] \sqsubseteq^{\mathsf{r}} r$.*

Proof. Similar to the proof of Lemma 99. □

Lemma 101. *Let $r \in \mathcal{V}^{\mathsf{r}}_N$ and $n \in \mathcal{N}$. Then $r[\nu{=}n_2] \sqsubseteq^{\mathsf{r}} r$.*

Proof. Similar to the proof of Lemma 99. □

Lemma 102. *Let $r \in \mathcal{V}^{\mathsf{r}}_N$ and $M^{\mathsf{r}} \in \mathcal{M}^{\mathsf{r}}$. Then $r[M^{\mathsf{r}}] \sqsubseteq^{\mathsf{r}} r$.*

Proof. Follows from the properties of meets and Lemma 99. □

We can also show that strengthening operations are idempotent. We show the proof of idempotency for strengthening by a relational value.

Lemma 103. *Let $r_1 \in \mathcal{V}^{\mathsf{r}}_N$, $r \in \mathcal{V}^{\mathsf{r}}_N$, and $n \in \mathcal{N}$. Then $r[n \leftarrow r_1] = r[n \leftarrow r_1][n \leftarrow r_1]$.*

Proof. In the below calculational proof, let $S = \{\langle M'[n \mapsto v'], v\rangle \mid \langle M', v'\rangle \in$

$\gamma^{\mathsf{r}}_N(r_1)\}$ and $r' = r[n \leftarrow r_1] = \alpha^{\mathsf{r}}_N(\gamma^{\mathsf{r}}_N(r) \cap S)$.

$$
\begin{aligned}
r'[n \leftarrow r_1] &= \alpha^{\mathsf{r}}_N(\gamma^{\mathsf{r}}_N(r') \cap S) && \text{by def. of } \cdot[\cdot \leftarrow \cdot] \\
\gamma^{\mathsf{r}}_N(r) \cap S &\subseteq \gamma^{\mathsf{r}}_N(r') && \text{by prop. of Galois conn.} \\
\gamma^{\mathsf{r}}_N(r) \cap S &\subseteq \gamma^{\mathsf{r}}_N(r') \cap S && \text{by def. of } \cap \\
\alpha^{\mathsf{r}}_N(\gamma^{\mathsf{r}}_N(r) \cap S) &\sqsubseteq^{\mathsf{r}} \alpha^{\mathsf{r}}_N(\gamma^{\mathsf{r}}_N(r') \cap S) && \text{by Lemma 98} \\
r[n \leftarrow r_1] &\sqsubseteq^{\mathsf{r}} r'[n \leftarrow r_1] && \text{by def. of } \cdot[\cdot \leftarrow \cdot] \\
r[n \leftarrow r_1] &\sqsubseteq^{\mathsf{r}} r[n \leftarrow r_1][n \leftarrow r_1] && \text{by def. of } r' \\
r[n \leftarrow r_1][n \leftarrow r_1] &\sqsubseteq^{\mathsf{r}} r[n \leftarrow r_1] && \text{by Lemma 99} \\
r[n \leftarrow r_1][n \leftarrow r_1] &= r[n \leftarrow r_1]
\end{aligned}
$$

□

A.4.1.2 Monotonicity

All of the strengthening operations on relational values are monotone.

Lemma 104. *Let* $r_1 \in \mathcal{V}^{\mathsf{r}}_N$, $r_2 \in \mathcal{V}^{\mathsf{r}}_N$, $r \in \mathcal{V}^{\mathsf{r}}_N$, *and* $n \in \mathcal{N}$. *Then* $r_1 \sqsubseteq^{\mathsf{r}} r_2 \implies r_1[n \leftarrow r] \sqsubseteq^{\mathsf{r}} r_2[n \leftarrow r]$.

Proof. In the below calculational proof, let $S = \{\langle M'[n \mapsto v'], v\rangle \mid \langle M', v'\rangle \in \gamma^{\mathsf{r}}_N(r)\}$.

$$
\begin{aligned}
r_1 \sqsubseteq^{\mathsf{r}} r_2 &\implies \gamma^{\mathsf{r}}_N(r_1) \subseteq \gamma^{\mathsf{r}}_N(r_2) && \text{by Lemma 96} \\
&\iff \gamma^{\mathsf{r}}_N(r_1) \cap S \subseteq \gamma^{\mathsf{r}}_N(r_2) \cap S \\
&\implies \alpha^{\mathsf{r}}_N(\gamma^{\mathsf{r}}_N(r_1) \cap S) \sqsubseteq^{\mathsf{r}} \alpha^{\mathsf{r}}_N(\gamma^{\mathsf{r}}_N(r_2) \cap S) && \text{by Lemma 98} \\
&\iff r_1[n \leftarrow r] \sqsubseteq^{\mathsf{r}} r_2[n \leftarrow r] && \text{by def. of } \cdot[\cdot \leftarrow \cdot]
\end{aligned}
$$

□

Lemma 105. *Let* $r_1 \in \mathcal{V}^{\mathsf{r}}_N$, $r_2 \in \mathcal{V}^{\mathsf{r}}_N$, $r'_1 \in \mathcal{V}^{\mathsf{r}}_{N_0}$, $r'_2 \in \mathcal{V}^{\mathsf{r}}_{N_0}$, *and* $n \in \mathcal{N}$. *Then* $r_1 \sqsubseteq^{\mathsf{r}} r_2 \wedge r'_1 \sqsubseteq^{\mathsf{r}} r'_2 \implies r_1[n \leftarrow r'_1] \sqsubseteq^{\mathsf{r}} r_2[n \leftarrow r'_2]$.

Proof. In the below calculational proof, let $S_1 = \{\langle M'[n \mapsto v'], v\rangle \mid \langle M', v'\rangle \in \gamma^{\mathsf{r}}_N(r'_1)\}$ and $S_2 = \{\langle M'[n \mapsto v'], v\rangle \mid \langle M', v'\rangle \in \gamma^{\mathsf{r}}_N(r'_2)\}$.

First note that $r'_1 \sqsubseteq^{\mathsf{r}} r'_2$ implies by Lemma 96 that $\gamma^{\mathsf{r}}_N(r'_1) \subseteq \gamma^{\mathsf{r}}_N(r'_2)$. Hence, $S_1 \subseteq S_2$.

$$
\begin{aligned}
r_1 \sqsubseteq^{\mathsf{r}} r_2 &\implies \gamma^{\mathsf{r}}_N(r_1) \subseteq \gamma^{\mathsf{r}}_N(r_2) && \text{by Lemma 96} \\
&\iff \gamma^{\mathsf{r}}_N(r_1) \cap S_1 \subseteq \gamma^{\mathsf{r}}_N(r_2) \cap S_2 && \text{by } S_1 \subseteq S_2 \\
&\implies \alpha^{\mathsf{r}}_N(\gamma^{\mathsf{r}}_N(r_1) \cap S_1) \sqsubseteq^{\mathsf{r}} \alpha^{\mathsf{r}}_N(\gamma^{\mathsf{r}}_N(r_2) \cap S_2) && \text{by mono. of } \alpha^{\mathsf{r}}_N \\
&\iff r_1[n \leftarrow r] \sqsubseteq^{\mathsf{r}} r_2[n \leftarrow r] && \text{by def. of } \cdot[\cdot \leftarrow \cdot]
\end{aligned}
$$

□

Lemma 106. *Let* $r_1 \in \mathcal{V}^{\mathsf{r}}_N$, $r_2 \in \mathcal{V}^{\mathsf{r}}_N$, $n_1 \in \mathcal{N}$, *and* $n_2 \in \mathcal{N}$. *Then* $r_1 \sqsubseteq^{\mathsf{r}} r_2 \implies r_1[n_1{=}n_2] \sqsubseteq^{\mathsf{r}} r_2[n_1{=}n_2]$.

Proof. Similar as for Lemma 104. □

Lemma 107. *Let* $r_1 \in \mathcal{V}^{\mathsf{r}}_N$, $r_2 \in \mathcal{V}^{\mathsf{r}}_N$, *and* $n \in \mathcal{N}$. *Then* $r_1 \sqsubseteq^{\mathsf{r}} r_2 \implies r_1[\nu{=}n] \sqsubseteq^{\mathsf{r}} r_2[\nu{=}n]$.

Proof. Similar as for Lemma 104. □

Lemma 108. *Let* $r_1 \in \mathcal{V}^{\mathsf{r}}_N$, $r_2 \in \mathcal{V}^{\mathsf{r}}_N$, *and* $M^{\mathsf{r}} \in \mathcal{M}^{\mathsf{r}}$. *Then* $r_1 \sqsubseteq^{\mathsf{r}} r_2 \implies r_1[M^{\mathsf{r}}] \sqsubseteq^{\mathsf{r}} r_2[M^{\mathsf{r}}]$.

Proof. The result follows from Lemma 104 and the monotonicity of meets. □

Lemma 109. *Let* $r_1 \in \mathcal{V}^{\mathsf{r}}_N$, $r_2 \in \mathcal{V}^{\mathsf{r}}_N$, $M^{\mathsf{r}}_1 \in \mathcal{M}^{\mathsf{r}}$, *and* $M^{\mathsf{r}}_2 \in \mathcal{M}^{\mathsf{r}}$. *Then* $r_1 \sqsubseteq^{\mathsf{r}} r_2 \wedge M^{\mathsf{r}}_1 \mathrel{\dot{\sqsubseteq}}^{\mathsf{r}} M^{\mathsf{r}}_2 \implies r_1[M^{\mathsf{r}}_1] \sqsubseteq^{\mathsf{r}} r_2[M^{\mathsf{r}}_2]$.

Proof. The result follows from Lemma 105 and the monotonicity of meets. □

Lemma 110. *Let* $r_1 \in \mathcal{V}^{\mathsf{r}}_N$, $r_2 \in \mathcal{V}^{\mathsf{r}}_N$, *and* N' *be a scope. Then* $r_1 \sqsubseteq^{\mathsf{r}} r_2 \implies r_1[N'] \sqsubseteq^{\mathsf{r}} r_2[N']$.

Proof. Similar as for Lemma 104. □

A.4.1.3 Structural Strengthening

We also here provide a structural definition of strengthening and argue its soundness. First, whenever $n \notin N$, then $r_{\hat{N}}\lceil n \leftarrow r \rceil \stackrel{\text{def}}{=} r$. Otherwise,

$$
\begin{array}{lll}
r\lceil n \leftarrow \bot^{\mathsf{r}} \rceil & \stackrel{\text{def}}{=} \bot^{\mathsf{r}} & \\
r\lceil n \leftarrow r' \rceil & \stackrel{\text{def}}{=} r & \text{if } r \in \{\bot^{\mathsf{r}}, \top^{\mathsf{r}}\} \text{ or } r' \in \{\top^{\mathsf{r}}\} \cup \mathcal{T}^{\mathsf{r}} \\
R^{\mathsf{r}}_1\lceil n \leftarrow R^{\mathsf{r}}_2 \rceil & \stackrel{\text{def}}{=} R^{\mathsf{r}}_1[n \leftarrow R^{\mathsf{r}}_2] & \\
T^{\mathsf{r}}\lceil n \leftarrow r \rceil & \stackrel{\text{def}}{=} \Lambda n_{\mathsf{cs}}.\langle \pi_1(T^{\mathsf{r}}(n_{\mathsf{cs}}))\lceil n \leftarrow r \rceil, \pi_2(T^{\mathsf{r}}(n_{\mathsf{cs}}))\lceil n \leftarrow \exists n_{\mathsf{cs}}.\, r \rceil \rangle & r \in \mathcal{R}^{\mathsf{r}}_N
\end{array}
$$

It can be shown that the structural strengthening is an overapproximation of the ordinary one.

Lemma 111. *Let* $r \in \mathcal{V}^{\mathsf{r}}$, $r' \in \mathcal{V}^{\mathsf{r}}$, *and* $n \in \mathcal{N}$. *Then,* $r[n \leftarrow r'] \sqsubseteq^{\mathsf{r}} r\lceil n \leftarrow r' \rceil$.

Proof. When $n \notin N$, the argument is trivial by Lemma 99. Otherwise, if $r' = \bot^{\mathsf{r}}$, the result follows by the fact monotonicity of α^{r}_N and by the definition of γ^{r}_N. The proof is carried by structural induction on r. First, for the cases where $r' \in \{\top^{\mathsf{r}}\} \cup \mathcal{T}^{\mathsf{r}}$, the result follows immediately by Lemma 99. For the rest of the proof, we thus assume that $r' \in \mathcal{R}^{\mathsf{r}}_N$.

For the base cases when $r \in \{\bot^{\mathsf{r}}, \top^{\mathsf{r}}\}$ the result again follows easily from Lemma 99. For the base case $r \in \mathcal{R}^{\mathsf{r}}_N$, the argument is trivial. For the induction case when $r \in \mathcal{T}^{\mathsf{r}}_N$, we have the following:

$$\begin{aligned}
&\{\langle M, T\rangle \mid \forall n_{\mathsf{cs}}.\, T(n_{\mathsf{cs}}){=}\langle v_i, v_o\rangle \wedge T^{\mathsf{r}}(n_{\mathsf{cs}}){=}\langle r_i, r_o\rangle \wedge \langle M_i, v_i\rangle \in \gamma^{\mathsf{r}}(r_i) \\
&\qquad \wedge \langle M_o, v_o\rangle \in \gamma^{\mathsf{r}}(r_o) \wedge M_i =_{\setminus\{n_{\mathsf{cs}}\}} M \wedge M_o{=}M_i[n_{\mathsf{cs}} \mapsto v_i]\} \cap \mathcal{M} \\
&\qquad \textbf{where } \mathcal{M} = \{\langle M'[n \mapsto v'], v\rangle \mid \langle M', v'\rangle \in \gamma^{\mathsf{r}}(r')\} \\
&= \{\langle M, T\rangle \mid \forall n_{\mathsf{cs}}.\, T(n_{\mathsf{cs}}){=}\langle v_i, v_o\rangle \wedge T^{\mathsf{r}}(n_{\mathsf{cs}}){=}\langle r_i, r_o\rangle \wedge \langle M_i, v_i\rangle \in \gamma^{\mathsf{r}}(r_i) \\
&\qquad \wedge \langle M_o, v_o\rangle \in \gamma^{\mathsf{r}}(r_o) \wedge M_i =_{\setminus\{n_{\mathsf{cs}}\}} M \wedge M_o{=}M_i[n_{\mathsf{cs}} \mapsto v_i] \wedge \langle M, T\rangle \in \mathcal{M}\} \\
&= \{\langle M, T\rangle \mid \forall n_{\mathsf{cs}}.\, T(n_{\mathsf{cs}}){=}\langle v_i, v_o\rangle \wedge T^{\mathsf{r}}(n_{\mathsf{cs}}){=}\langle r_i, r_o\rangle \wedge \langle M_i, v_i\rangle \in (\gamma^{\mathsf{r}}(r_i) \cap \mathcal{M}) \\
&\qquad \wedge \langle M_o, v_o\rangle \in \gamma^{\mathsf{r}}(r_o) \wedge M_i =_{\setminus\{n_{\mathsf{cs}}\}} M \wedge M_o{=}M_i[n_{\mathsf{cs}} \mapsto v_i] \wedge M \in \mathcal{M}\} \\
&\qquad \textbf{by } n_{\mathsf{cs}} \textbf{ not in the scope of } r_i \textbf{ and def. of } \mathcal{M} \textbf{ (values)} \\
&= \{\langle M, T\rangle \mid \forall n_{\mathsf{cs}}.\, T(n_{\mathsf{cs}}){=}\langle v_i, v_o\rangle \wedge T^{\mathsf{r}}(n_{\mathsf{cs}}){=}\langle r_i, r_o\rangle \wedge \langle M_i, v_i\rangle \in (\gamma^{\mathsf{r}}(r_i) \cap \mathcal{M}) \\
&\qquad \wedge \langle M_o, v_o\rangle \in (\gamma^{\mathsf{r}}(r_o) \cap \mathcal{M}^{n_{\mathsf{cs}}}) \wedge M_i =_{\setminus\{n_{\mathsf{cs}}\}} M \wedge M_o{=}M_i[n_{\mathsf{cs}} \mapsto v_i]\} \\
&\qquad \textbf{where } \mathcal{M}^{n_{\mathsf{cs}}} = \{\langle M[n_{\mathsf{cs}} \mapsto v'], v\rangle \mid \langle M, v\rangle \in \mathcal{M} \wedge v' \in \mathcal{V}\} \\
&\subseteq \{\langle M, T\rangle \mid \forall n_{\mathsf{cs}}.\, T(n_{\mathsf{cs}}){=}\langle v_i, v_o\rangle \wedge T^{\mathsf{r}}(n_{\mathsf{cs}}){=}\langle r_i, r_o\rangle \wedge \langle M_i, v_i\rangle \in \gamma^{\mathsf{r}}(r_i[n \leftarrow r']) \\
&\qquad \wedge \langle M_o, v_o\rangle \in \gamma^{\mathsf{r}}(r_o[n \leftarrow \exists n_{\mathsf{cs}}.\, r']) \wedge M_i =_{\setminus\{n_{\mathsf{cs}}\}} M \wedge M_o{=}M_i[n_{\mathsf{cs}} \mapsto v_i]\} \\
&\qquad \textbf{by the def. of } \cdot[\cdot \leftarrow \cdot]\textbf{, def. of } \exists \cdot\,.\,\cdot\textbf{, and Proposition 1} \\
&= \{\langle M, T\rangle \mid \forall n_{\mathsf{cs}}.\, T(n_{\mathsf{cs}}){=}\langle v_i, v_o\rangle \wedge T^{\mathsf{r}}(n_{\mathsf{cs}}){=}\langle r_i, r_o\rangle \wedge \langle M_i, v_i\rangle \in \gamma^{\mathsf{r}}(r_i\lceil n \leftarrow r'\rceil) \\
&\qquad \wedge \langle M_o, v_o\rangle \in \gamma^{\mathsf{r}}(r_o\lceil n \leftarrow \exists n_{\mathsf{cs}}.\, r'\rceil) \wedge M_i =_{\setminus\{n_{\mathsf{cs}}\}} M \wedge M_o{=}M_i[n_{\mathsf{cs}} \mapsto v_i]\} \\
&\qquad \textbf{by i.h. and Proposition 1} \\
&= M'
\end{aligned}$$

We then have that $r[n \leftarrow r'] \sqsubseteq^{\mathsf{r}} \alpha^{\mathsf{r}}(M' \cup \gamma^{\mathsf{r}}_N(\bot^{\mathsf{r}})) = \alpha^{\mathsf{r}}_N(\gamma^{\mathsf{r}}_N(r\lceil n \leftarrow r'\rceil)) \sqsubseteq^{\mathsf{r}} r\lceil n \leftarrow r'\rceil$ by Proposition 1. □

A.4.2 Propagation and Transformer

Proof (of Lemma 6)**.**

Proof. For the increasing property, the proof goes by structural induction on both relational arguments of $\mathsf{prop}^{\mathsf{r}}$ and it closely follows the proof of Lemma 26. Regarding monotonicity, the proof goes by structural induction on the relational arguments of $\mathsf{prop}^{\mathsf{r}}$, closely follows the proof of Lemma 28, and relies on Lemma 105 (introduced later). □

Proof (of Lemma 7)**.**

Proof. The increasing property of $\mathsf{Step}^{\mathsf{r}}_{k+1}[\![\cdot]\!]$ is obvious. Regarding monotonicity, the proof goes by structural induction on program expressions and it closely follows the proof of Lemma 29. The proof relies on Lemma 6 as well as Lemma 109, 107, 110, and 106. □

A.4.2.1 Range of Computation

Lemma 112. *Let M^{r} and M^{r}_1 be relational maps, E an environment, e^{ℓ} an expression of a program p, and k a fuel. If* $\mathsf{domvalid}(\ell, p, E)$ *and* $\mathsf{Step}^{\mathsf{r}}_k[\![e^{\ell}]\!](E)(M^{\mathsf{r}}) = M^{\mathsf{r}}_1$*, then* $\forall n. M^{\mathsf{r}}(n) \neq M^{\mathsf{r}}_1(n) \implies n \in \mathsf{range}_p(E \diamond \ell)$.

Proof. The same as in the proof of Lemma 32. □

The same result about the range of computation can be shown.

Lemma 113. *Let M^{r} and M^{r}_1 be relational maps, E an environment, e^{ℓ} an expression of a program p, and k a fuel. If* $\mathsf{domvalid}(\ell, p, E)$ *and* $\mathsf{Step}^{\mathsf{r}}_k[\![e^{\ell}]\!](E)(M) = M^{\mathsf{r}}_1$*, then* $\forall n. M^{\mathsf{r}}(n) \neq M^{\mathsf{r}}_1(n) \implies n \in \mathsf{ranger}_p(E \diamond \ell)$.

Proof. Similar to the proof of Lemma 35. □

A.4.3 Soundness

We first provide several useful results used to argue soundness of relational semantics.

Lemma 114. *Let $M \in \mathcal{M}$, $v \in \mathcal{V}$, $n \in \mathcal{N}$, and $r \in \mathcal{V}^{\mathsf{r}}_N$. Suppose $\langle M, v\rangle \in \gamma^{\mathsf{r}}_N(r)$ and $M(n) \in \mathcal{T}$. Then, for any $T \in \mathcal{T}$, $\langle M[n \mapsto T], v\rangle \in \gamma^{\mathsf{r}}_N(r)$.*

Proof. When $n \notin N$, the argument follows by Lemma 97. If $r \in \{\bot^{\mathsf{r}}, \top^{\mathsf{r}}\}$, the argument is trivial. For other cases, the result follows by the definition of γ^{r}_N, specifically γ^{d} ($\gamma^{\mathsf{d}}(\mathsf{F}) = \mathcal{T}$). □

Lemma 115. *Let $M \in \mathcal{M}$, $M' \in \mathcal{M}$, $v \in \mathcal{V}$, and $r \in \mathcal{V}^{\mathsf{r}}_N$. Suppose $M \dot{\sqsubseteq} M'$, $\langle M, v\rangle \in \gamma^{\mathsf{r}}_N(r)$, and $\forall n \in N.\, M(n) \neq \omega \implies M'(n) \neq \omega$. Then $\langle M', v\rangle \in \gamma^{\mathsf{r}}_N(r)$.*

Proof. The argument goes by structural induction on r. By Lemma 97, the proof analysis can focus only on the nodes N of M'. The cases when $r \in \{\bot^{\mathsf{r}}, \top^{\mathsf{r}}\}$ are trivial. The other cases follow by the definition of γ^{r}. For instance, suppose $n \in N$ and $M(n) = c$. Then, $M'(n) = c$. Otherwise, suppose $M(n) \in \mathcal{T}$. The argument then follows from Lemma 114. □

A.4.3.1 Data Propagation

We start proving the soundness of the relational semantics by formalizing and showing that relational propagation abstracts the concrete one. We begin by introducing the notion of scope consistency that closely resembles the environment consistency.

$$\mathsf{scopecons}_N(M) \overset{\text{def}}{\iff} \forall n \in N.\, M(n) \notin \{\bot, \omega\}$$

Lemma 116. *Let v_1, v_2, v_3, and v_4 be values, M_1 and M_2 maps, $r_1 \in \mathcal{V}^{\mathsf{r}}_N$, $r_2 \in \mathcal{V}^{\mathsf{r}}_N$, $r_3 \in \mathcal{V}^{\mathsf{r}}_N$, and $r_4 \in \mathcal{V}^{\mathsf{r}}_N$. Let $v_1 <:^c v_2$, $\mathsf{noerr}(v_1)$, $\mathsf{noerr}(v_2)$, $\langle M_1, v_1\rangle \in \gamma^{\mathsf{r}}_N(r_1)$, $\langle M_2, v_2\rangle \in \gamma^{\mathsf{r}}_N(r_2)$, $M_1 \dot{\sqsubseteq} M_2$, $\mathsf{scopecons}_N(M_1)$, and $\mathsf{scopecons}_N(M_2)$. If $\langle v_3, v_4\rangle = \mathsf{prop}(v_1, v_2)$ and $\langle r_3, r_4\rangle = \mathsf{prop}^{\mathsf{r}}(r_1, r_2)$, then $\langle M_1, v_3\rangle \in \gamma^{\mathsf{r}}_N(r_3)$ and $\langle M_2, v_4\rangle \in \gamma^{\mathsf{r}}_N(r_4)$.*

Proof. By structural induction on both v_1 and v_2.

- $v_1 = \bot, v_2 = \bot$.

$$\begin{aligned} v_3 = \bot, v_4 = \bot && \text{by def. of } \mathsf{prop} \\ \forall r.\, \mathcal{M} \times \{\bot\} \in \gamma^{\mathsf{r}}_N(r) && \text{by def. of } \gamma^{\mathsf{r}}_N \end{aligned}$$

- $v_1 = c, v_2 = \bot$.

$$\begin{aligned} v_3 = c, v_4 = c && \text{by def. of } \mathsf{prop} \\ \langle M_2, v_4\rangle \in \gamma^{\mathsf{r}}_N(r_1) && \text{by } \mathsf{scopecons}_{M_2}(N) \text{ and Lemma 115} \\ r_1 \in \mathcal{R}^{\mathsf{r}}_N \cup \{\top^{\mathsf{r}}\} && \text{by def. of } \gamma^{\mathsf{r}}_N \\ r_3 = r_1, r_1 \sqsubseteq^{\mathsf{r}} r_4 && \text{by def. of } \mathsf{prop}^{\mathsf{r}} \end{aligned}$$

- $v_1 = T, v_2 = \bot$, $r_2 \notin \mathcal{T}^{\mathsf{r}}_N$.

$$\begin{aligned} v_3 = v_1, v_4 = T_\bot && \text{by def. of } \mathsf{prop} \\ r_1 \in \mathcal{T}^{\mathsf{r}}_N \cup \{\top^{\mathsf{r}}\} && \text{by def. of } \gamma^{\mathsf{r}}_N \\ r_3 = r_1 \vee r_3 = \top^{\mathsf{r}}, r_4 = T^{\mathsf{r}}_{\bot^{\mathsf{r}}} \vee r_4 = \top^{\mathsf{r}} && \text{by def. of } \mathsf{prop}^{\mathsf{r}} \\ \langle M_2, T_\bot\rangle \in \gamma^{\mathsf{r}}_N(r_4) && \text{by def. of } \gamma^{\mathsf{r}}_N \end{aligned}$$

- $v_1 = T, v_2 = \bot$, $r_2 \in \mathcal{T}^{\mathsf{r}}_N$.

$$
\begin{aligned}
v_3 = v_1, v_4 = T_\bot && \text{by def. of } \mathsf{prop} \\
r_1 \in \mathcal{T}^{\mathsf{r}}_N \cup \{\top^{\mathsf{r}}\} && \text{by def. of } \gamma^{\mathsf{r}}_N \\
r_3 = r_1 \vee r_3 = \top^{\mathsf{r}}, r_2 \sqsubseteq^{\mathsf{r}} r_4 && \text{by def. of } \mathsf{prop}^{\mathsf{r}} \text{ and Lemma 6} \\
\gamma^{\mathsf{r}}_N(T^{\mathsf{r}}_{\bot^{\mathsf{r}}}) \subseteq \gamma^{\mathsf{r}}_N(r_4) && \text{by def. of } \gamma^{\mathsf{r}}_N \\
\langle M_2, T_\bot \rangle \in \gamma^{\mathsf{r}}_N(T^{\mathsf{r}}_{\bot^{\mathsf{r}}}) && \text{by def. of } \gamma^{\mathsf{r}}_N \\
\langle M_2, T_\bot \rangle \in \gamma^{\mathsf{r}}_N(r_4)
\end{aligned}
$$

- $v_1 = c, v_2 = c$.

$$
\begin{aligned}
v_3 = c, v_4 = c && \text{by def. of } \mathsf{prop} \\
r_1 \sqsubseteq^{\mathsf{r}} r_3, r_2 \sqsubseteq^{\mathsf{r}} r_4 && \text{by def. of } \mathsf{prop}^{\mathsf{r}} \text{ and Lemma 6}
\end{aligned}
$$

For the induction case, let $v_1 = T_1$ and $v_2 = T_2$. By definition of γ^{r}_N, it must be that $r_1 \in \mathcal{T}^{\mathsf{r}}_N \cup \{\top^{\mathsf{r}}\}$ and $r_2 \in \mathcal{T}^{\mathsf{r}}_N \cup \{\top^{\mathsf{r}}\}$. If $r_1 = \top^{\mathsf{r}}$ or $r_2 = \top^{\mathsf{r}}$, the argument is trivial as $r_3 = r_4 = \top^{\mathsf{r}}$ by the def. of $\mathsf{prop}^{\mathsf{r}}$. We thus focus on the case when $r_1 = T^{\mathsf{r}}_1$ and $r_2 = T^{\mathsf{r}}_2$. First, observe that $n_{\mathsf{cs}} \in T_2 \implies n_{\mathsf{cs}} \in T^{\mathsf{r}}_2$ by the def. of γ^{r}_N. If $n_{\mathsf{cs}} \in T^{\mathsf{r}}_2$ but $n_{\mathsf{cs}} \notin T_2$, the argument follows from Lemma 6.

We are left with the case where $n_{\mathsf{cs}} \in T^{\mathsf{r}}_2$ and $n_{\mathsf{cs}} \in T_2$. Let $\langle v_{1i}, v_{1o} \rangle = T_1(n_{\mathsf{cs}})$, $\langle v_{2i}, v_{2o} \rangle = T_1(n_{\mathsf{cs}})$, $\langle r_{1i}, r_{1o} \rangle = T^{\mathsf{r}}_1(n_{\mathsf{cs}})$, and $\langle r_{2i}, r_{2o} \rangle = T^{\mathsf{r}}_2(n_{\mathsf{cs}})$. By definition of $<:^c$, we have that $v_{2i} <:^c v_{1i}$ and $v_{1o} <:^c v_{2o}$.

Suppose first that $\langle v'_{2i}, v'_{1i} \rangle = \mathsf{prop}(v_{2i}, v_{1i})$ and $\langle r'_{2i}, r'_{1i} \rangle = \mathsf{prop}^{\mathsf{r}}(r_{2i}, r_{1i})$. We have $\mathsf{noerr}(v_{1i})$, $\mathsf{noerr}(v_{2i})$, $\mathsf{noerr}(v_{1o})$, $\mathsf{noerr}(v_{2o})$ by the definition of noerr. Also, it trivially follows that $\mathsf{scopecons}_{M_1}(N \setminus \{n_{\mathsf{cs}}\})$ and $\mathsf{scopecons}_{M_2}(N \setminus \{n_{\mathsf{cs}}\})$. By definition of γ^{r}, the fact that n_{cs} is not in the scope of v_{2i}, v_{1i}, r_{2i}, r_{1i}, and Lemma 97, we have $\langle M_1, v_{1i} \rangle \in \gamma^{\mathsf{r}}_{N \setminus \{n_{\mathsf{cs}}\}}(r_{1i})$ and $\langle M_2, v_{2i} \rangle \in \gamma^{\mathsf{r}}_{N \setminus \{n_{\mathsf{cs}}\}}(r_{2i})$. It then follows from induction hypothesis that $\langle M_1, v'_{1i} \rangle \in \gamma^{\mathsf{r}}_{N \setminus \{n_{\mathsf{cs}}\}}(r'_{1i})$ and $\langle M_2, v'_{2i} \rangle \in \gamma^{\mathsf{r}}_{N \setminus \{n_{\mathsf{cs}}\}}(r'_{2i})$.

Next, $\langle M_1[n_{\mathsf{cs}} \mapsto v_{1i}], v_{1o}\rangle \in \gamma^{\mathsf{r}}_{N\cup\{n_{\mathsf{cs}}\}}(r_{1o})$ and $\langle M_2[n_{\mathsf{cs}} \mapsto v_{2i}], v_{2o}\rangle \in \gamma^{\mathsf{r}}_{N\cup\{n_{\mathsf{cs}}\}}(r_{2o})$. Since $n_{\mathsf{cs}} \in v_2$, we know $v_{2i} \neq \bot$. Also, we know by $\mathsf{noerr}(v_{2i})$ that $v_{2i} \neq \omega$. Hence, if $v_{2i} = c$, then by $v_{2i} <:^c v_{1i}$ we have that $v_{1i} = c$. Further, if $v_{2i} \in \mathcal{T}$, then $\langle M_1[n_{\mathsf{cs}} \mapsto v_{2i}], v_{1o}\rangle$ by Lemma 114. It also holds that $\langle M_1[n_{\mathsf{cs}} \mapsto v_{2i}], v_{1o}\rangle \in \gamma^{\mathsf{r}}_{N\cup\{n_{\mathsf{cs}}\}}(r_{1o})$ and $\langle M_2[n_{\mathsf{cs}} \mapsto v_{2i}], v_{2o}\rangle \in \gamma^{\mathsf{r}}_{N\cup\{n_{\mathsf{cs}}\}}(r_{2o})$. By definition of strengthening, one then also has $\langle M_1[n_{\mathsf{cs}} \mapsto v_{2i}], v_{1o}\rangle \in \gamma^{\mathsf{r}}_{N\cup\{n_{\mathsf{cs}}\}}(r_{1o}[n_{\mathsf{cs}} \leftarrow r_{2i}])$ and $\langle M_2[n_{\mathsf{cs}} \mapsto v_{2i}], v_{2o}\rangle \in \gamma^{\mathsf{r}}_{N\cup\{n_{\mathsf{cs}}\}}(r_{2o}[n_{\mathsf{cs}} \leftarrow r_{2i}])$.

Let $\langle v'_{1o}, v'_{2o}\rangle = \mathsf{prop}(v_{1o}, v_{2o})$ and $\langle r'_{1o}, r'_{2o}\rangle = \mathsf{prop}^{\mathsf{r}}(r_{1o}[n_{\mathsf{cs}} \leftarrow r_{2i}], r_{2o}[n_{\mathsf{cs}} \leftarrow r_{2i}])$. As $M_1[n_{\mathsf{cs}} \mapsto v_{2i}] \mathrel{\dot{\sqsubseteq}^{\mathsf{r}}} M_2[n_{\mathsf{cs}} \mapsto v_{2i}]$, $\mathsf{scopecons}_{M_1[n_{\mathsf{cs}}\mapsto v_{2i}]}(N \cup \{n_{\mathsf{cs}}\})$, and $\mathsf{scopecons}_{M_2[n_{\mathsf{cs}}\mapsto v_{2i}]}(N \cup \{n_{\mathsf{cs}}\})$, we have by induction hypothesis that $\langle M_1[n_{\mathsf{cs}} \mapsto v_{2i}], v'_{1o}\rangle \in \gamma^{\mathsf{r}}_{N\cup\{n_{\mathsf{cs}}\}}(r'_{1o})$ and $\langle M_2[n_{\mathsf{cs}} \mapsto v_{2i}], v'_{2o}\rangle \in \gamma^{\mathsf{r}}_{N\cup\{n_{\mathsf{cs}}\}}(r'_{2o})$.

By Lemma 38 and Lemma 39, we have $v'_{2i} <:^c v'_{1i}$ and, if $v_{2i} = c$, $v_{2i} = v'_{2i} = v_{1i} = v'_{1i} = c$. In that case, $\langle M_1[n_{\mathsf{cs}} \mapsto v'_{1i}], v'_{1o}\rangle \in \gamma^{\mathsf{r}}_{N\cup\{n_{\mathsf{cs}}\}}(r'_{1o})$ and $\langle M_1[n_{\mathsf{cs}} \mapsto v'_{2i}], v'_{2o}\rangle \in \gamma^{\mathsf{r}}_{N\cup\{n_{\mathsf{cs}}\}}(r'_{2o})$. In the case $v'_{2i} \in \mathcal{T}$, then also $v'_{1i} \in \mathcal{T}$ and the same result follows by Lemma 114. □

A.4.3.2 Joins

We next show several results showing the soundness of the relational join subject to to the concrete one.

Lemma 117. *Let v_1, v_2, be values, M a concrete map, $r_1 \in \mathcal{V}^{\mathsf{r}}_N$, and $r_2 \in \mathcal{V}^{\mathsf{r}}_N$. If $\mathsf{noerr}(v_1)$, $\mathsf{noerr}(v_2)$, $\mathsf{ccomp}(v_1, v_2)$, $\langle M, v_1\rangle \in \gamma^{\mathsf{r}}_N(r_1)$, $\langle M, v_2\rangle \in \gamma^{\mathsf{r}}_N(r_2)$, then $\langle M, v_1 \sqcup v_2\rangle \in \gamma^{\mathsf{r}}_N(r_1 \sqcup^{\mathsf{r}} r_2)$.*

Proof. By structural induction on both v_1 and v_2. Let $v = v_1 \sqcup v_2$ and $r = r_1 \sqcup^{\mathsf{r}} r_2$.

- $v_1 = \bot$ or $v_1 = \bot$. Trivial, as $v = c$ and the argument then follows by monotonicity of γ^{r}_N.

- $v_1 = c, v_2 = c$. Again, by monotonicity of γ^{r}_N.

For the induction case, let $v_1 = T_1$ and $v_2 = T_2$. By definition of γ^{r}_N, it must be that $r_1 \in \mathcal{T}^{\mathsf{r}}_N \cup \{\top^{\mathsf{r}}\}$ and $r_2 \in \mathcal{T}^{\mathsf{r}}_N \cup \{\top^{\mathsf{r}}\}$. If $r_1 = \top^{\mathsf{r}}$ or $r_2 = \top^{\mathsf{r}}$, the argument is trivial as $r_3 = \top^{\mathsf{r}}$ by the def. of $\sqcup^{\mathsf{r}}$. We thus focus on the case when $r_1 = T^{\mathsf{r}}_1$ and $r_2 = T^{\mathsf{r}}_2$. Consider any call site n_{cs}. Let $\langle v_{1i}, v_{1o}\rangle = T_1(n_{\mathsf{cs}})$, $\langle v_{2i}, v_{2o}\rangle = T_1(n_{\mathsf{cs}})$, $\langle r_{1i}, r_{1o}\rangle = T^{\mathsf{r}}_1(n_{\mathsf{cs}})$, and $\langle r_{2i}, r_{2o}\rangle = T^{\mathsf{r}}_2(n_{\mathsf{cs}})$. Let us first consider the case for inputs. We have $\langle M, v_{1i}\rangle \in \gamma^{\mathsf{r}}_N(r_{1i})$, $\langle M, v_{2i}\rangle \in \gamma^{\mathsf{r}}_N(r_{2i})$, $\mathsf{noerr}(v_{2i})$, $\mathsf{noerr}(v_{1i})$, and $\mathsf{ccomp}(v_{1i}, v_{2i})$. By i.h., we then have $\langle M, v_{1i} \sqcup v_{2i}\rangle \in \gamma^{\mathsf{r}}(r_{1i} \sqcup^{\mathsf{r}} r_{2i})$.

Next, we have that $\langle M[n_{\mathsf{cs}} \mapsto v_{1i}], v_{1o}\rangle \in \gamma^{\mathsf{r}}_N(r_{1o})$, $\langle M[n_{\mathsf{cs}} \mapsto v_{2i}], v_{2o}\rangle \in \gamma^{\mathsf{r}}_N(r_{2o})$, $\mathsf{noerr}(v_{2o})$, $\mathsf{noerr}(v_{1o})$, and $\mathsf{ccomp}(v_{1o}, v_{2o})$. Now, if $v_{1i} = \bot$, then $v_{1o} = \bot$ by the def. of γ^{r} and $\mathsf{noerr}(v_{1o})$. Hence, $\langle M[n_{\mathsf{cs}} \mapsto v_{2i}], v_{1o}\rangle \in \gamma^{\mathsf{r}}_N(r_{1o})$ as well again by the definition of γ^{r}_N. The same holds in the other direction. For the remaining cases, if $v_{1i} = c$ then by $\mathsf{ccomp}(v_{1i}, v_{2i})$ we have $v_{1i} = v_{2i}$, hence clearly $\langle M[n_{\mathsf{cs}} \mapsto v_{2i}], v_{1o}\rangle \in \gamma^{\mathsf{r}}_N(r_{1o})$. Lastly, if $v_{1i} \in \mathcal{T}$ then by $\mathsf{ccomp}(v_{1i}, v_{2i})$ we have $v_{2i} \in \mathcal{T}$, and by Lemma 114 $\langle M[n_{\mathsf{cs}} \mapsto v_{2i}], v_{1o}\rangle \in \gamma^{\mathsf{r}}_N(r_{1o})$. By the induction hypothesis, we then have $\langle M[n_{\mathsf{cs}} \mapsto v_{2i}], v_{2o} \sqcup v_{1o}\rangle \in \gamma^{\mathsf{r}}_N(r_{1o} \sqcup^{\mathsf{r}} r_{2o})$. Again, if $v_{2i} \in \mathcal{T}$ then by $\mathsf{ccomp}(v_{1i}, v_{2i})$ and Lemma 114 $\langle M[n_{\mathsf{cs}} \mapsto v_{2i} \sqcup v_{1i}], v_{2o} \sqcup v_{1o}\rangle \in \gamma^{\mathsf{r}}_N(r_{1o} \sqcup^{\mathsf{r}} r_{2o})$. Similar reasoning applies for the cases when $v_{2i} \in \{\bot\} \cup \mathit{Cons}$. □

Lemma 118. *Let $K \geq 2$, v_1 to v_K be values where $\mathsf{ccomp}(\{v_1, \ldots, v_K\})$ and $\forall 1 \leq i \leq K. \mathsf{noerr}(v_i)$, r_1 to r_K relational values in $\mathcal{V}^{\mathsf{r}}_N$, and $\forall 1 \leq i \leq K. \langle M, v_i\rangle \in \gamma^{\mathsf{r}}_N(r_i)$. Then, $\langle M, \sqcup_{1 \leq i \leq K} v_i\rangle \in \gamma^{\mathsf{r}}_N(\sqcup^{\mathsf{r}}_{1 \leq i \leq K} r_i)$.*

Proof. The argument goes by induction on K. The result follows from the Lemma 117 and the fact that join $\sqcup$ does not invalidate the constant compatibility of values. □

Lemma 119. *Let $K \geq 2$, $\hat{M} = \{M_1, \ldots, M_K\}$ be a set of concrete maps where*

$\mathsf{ccomp}(\hat{M})$ *and* $\forall M \in \hat{M}.\, \mathsf{errcons}(M)$, $\hat{M}_r = \{M_1^{\mathsf{r}}, \ldots, M_K^{\mathsf{r}}\}$ *a set of relational maps, and* $\forall 1 \leq i \leq K.\, M_i \in \dot{\gamma}^{\mathsf{r}}(M_i^{\mathsf{r}})$. *Then,* $\dot{\sqcup}\hat{M} \in \dot{\gamma}^{\mathsf{r}}(\dot{\sqcup}^{\mathsf{r}}\, \hat{M}_r)$.

Proof. First, let

$$M_u = \dot{\sqcup}\hat{M} = \Lambda n.\, \sqcup \{M(n) \mid M \in \hat{M}\}$$

and

$$M_u^{\mathsf{r}} = \dot{\sqcup}^{\mathsf{r}}\, \hat{M} = \Lambda n.\, \sqcup^{\mathsf{r}} \{M^{\mathsf{r}}(n) \mid M^{\mathsf{r}} \in \hat{M}_r\}$$

First, suppose there exists a map $M_i \in \hat{M}$ s.t. exists a node n where it is not the case that $\mathsf{noerr}(M_i(n))$. By $\mathsf{errcons}(M_i)$, it must be that $M_i = M_\omega$, in which case $M_u = M_\omega$. Also, it then must be that $M_i^{\mathsf{r}} = M_{\top^{\mathsf{r}}}^{\mathsf{r}}$ and clearly $M_u^{\mathsf{r}} = M_{\top^{\mathsf{r}}}^{\mathsf{r}}$. The result then trivially follows by the definition of $\dot{\gamma}^{\mathsf{r}}$.

Otherwise, we have the following. As a reminder, by the definition of $\dot{\gamma}^{\mathsf{r}}$ one has that $M \in \dot{\gamma}^{\mathsf{r}}(M^{\mathsf{r}})$ implies that $\forall n.\langle M, M(n)\rangle \in \gamma^{\mathsf{r}}(M^{\mathsf{r}}(n))$. Also, by Lemma 42 and $M_u \neq M_\omega$, we have $\mathsf{errcons}(M_u)$. Hence,

$$\begin{aligned}
\forall 1 \leq i \leq K.\, \forall n.\langle M_i, M_i(n)\rangle \in \gamma^{\mathsf{r}}(M_i^{\mathsf{r}}(n)) && \text{by def. of } \dot{\gamma}^{\mathsf{r}} \\
\forall 1 \leq i \leq K.\, \forall n.\langle M_u, M_i(n)\rangle \in \gamma^{\mathsf{r}}(M_i^{\mathsf{r}}(n)) && \text{by Lemma 115} \\
\forall n.\langle M_u, \sqcup\{M_i(n) \mid M_i \in \hat{M}\}\rangle \in \gamma^{\mathsf{r}}(\sqcup^{\mathsf{r}}\{M_i^{\mathsf{r}}(n) \mid M_i^{\mathsf{r}} \in \hat{M}_r\}) && \text{by Lemma 118} \\
\forall n.\langle M_u, M_u(n)\rangle \in \gamma^{\mathsf{r}}(M_u^{\mathsf{r}}(n)) && \text{by def. of } \dot{\sqcup}^{\mathsf{r}} \text{ and } \dot{\sqcup} \\
M_u \in \dot{\gamma}^{\mathsf{r}}(M_u^{\mathsf{r}}) && \text{by def. of } \dot{\gamma}^{\mathsf{r}}
\end{aligned}$$

□

A.4.3.3 Constants

We now prove the soundness of relational abstraction for constants.

Lemma 120. *Let* $M \in \mathcal{M}$, N *a scope, and* $c \in \mathit{Cons}$. *If* $\mathsf{scopecons}_M(N)$, *then*

$\langle M, c\rangle \in \gamma^{\mathsf{r}}_N(c^{\mathsf{r}})$.

Proof. By definition, we have $c^{\mathsf{r}}_N\{\, D^{\mathsf{r}} \in \mathcal{D}^{\mathsf{r}}_N \mid D^{\mathsf{r}}(\nu) = c \,\}$. By γ^{r}_N, we also have

$$\gamma^{\mathsf{r}}_N(c^{\mathsf{r}}) = \gamma^{\mathsf{r}}_N(\{\, D^{\mathsf{r}} \in \mathcal{D}^{\mathsf{r}}_N \mid D^{\mathsf{r}}(\nu) = c \,\}) =$$

$$\{\langle M, c\rangle \mid D^{\mathsf{r}} \in \mathcal{D}^{\mathsf{r}}_N \wedge D^{\mathsf{r}}(\nu) = c \wedge \forall n \in N.\ M(n) \in \gamma^{\mathsf{d}}(D^{\mathsf{r}}(n))\} \cup \gamma^{\mathsf{r}}(\bot^{\mathsf{r}}) =$$

$$\{\langle M, c\rangle \mid D^{\mathsf{r}}(\nu) = c \wedge \forall n \in N.\ M(n) \notin \{\bot, \omega\}\} \cup \gamma^{\mathsf{r}}(\bot^{\mathsf{r}})$$

The result then follows from the definition of $\mathsf{scopecons}$. □

A.4.3.4 Strengthening

We now prove the soundness of several strengthening operations.

Lemma 121. *Let $M \in \mathcal{M}$, $v_1 \in \mathcal{V}$, $n \in \mathcal{N}$, $r \in \mathcal{V}^{\mathsf{r}}_N$, and $r' \in \mathcal{V}^{\mathsf{r}}$. If $\langle M, v_1\rangle \in \gamma^{\mathsf{r}}_N(r)$ and $\langle M, M(n)\rangle \in \gamma^{\mathsf{r}}(r')$, then $\langle M, v_1\rangle \in \gamma^{\mathsf{r}}_N(r[n \leftarrow r'])$.*

Proof.

$$\gamma^{\mathsf{r}}_N(r_N[n \leftarrow r']) = \gamma^{\mathsf{r}}_N(\alpha^{\mathsf{r}}_N(\gamma^{\mathsf{r}}(r_N) \cap \{\langle M'[n \mapsto v'], v\rangle \mid \langle M', v'\rangle \in \gamma^{\mathsf{r}}(r')\})) \supseteq$$

$$\gamma^{\mathsf{r}}(r_N) \cap \{\langle M'[n \mapsto v'], v\rangle \mid \langle M', v'\rangle \in \gamma^{\mathsf{r}}(r')\} \supseteq$$

$$\{\langle M, v_1\rangle\} \cap \{\langle M'[n \mapsto v'], v\rangle \mid \langle M', v'\rangle \in \gamma^{\mathsf{r}}(r')\} \supseteq$$

$$\{\langle M, v_1\rangle\} \cap \{\langle M'[n \mapsto v'], v\rangle \mid \langle M, v'\rangle \in \{\langle M, M(n)\rangle\}\} =$$

$$\{\langle M, v_1\rangle\} \cap \{\langle M[n \mapsto M(n)], v\rangle \mid v \in \mathcal{V}\} =$$

$$\{\langle M, v_1\rangle\} \cap \{\langle M, v\rangle \mid v \in \mathcal{V}\} =$$

$$\{\langle M, v_1\rangle\}$$

□

We can generalize the above result.

Lemma 122. *Let $M \in \mathcal{M}$, $v \in \mathcal{V}$, N a scope, $M^{\mathsf{r}} \in \mathcal{M}^{\mathsf{r}}$, $r \in \mathcal{V}^{\mathsf{r}}_N$. If $\langle M, v\rangle \in \gamma^{\mathsf{r}}_N(r)$ and $M \in \dot{\gamma}^{\mathsf{r}}(M^{\mathsf{r}})$, then $\langle M, v\rangle \in r[M^{\mathsf{r}}]$.*

Proof. Consider any $n \in N$. By $M \in \dot{\gamma}^{\mathsf{r}}(M^{\mathsf{r}})$, we have that $\langle M, M(n)\rangle \in \gamma^{\mathsf{r}}(M^{\mathsf{r}}(n))$. Then by Lemma 121, $\langle M, v\rangle \in \gamma^{\mathsf{r}}_N(r[n \leftarrow M^{\mathsf{r}}(n)])$. Hence, $\langle M, v\rangle \in \bigcap_{n\in N} \gamma^{\mathsf{r}}_N(r[n \leftarrow M^{\mathsf{r}}(n)])$. Then by Lemma 9 we have $\langle M, v\rangle \in \gamma^{\mathsf{r}}_N(\sqcap^{\mathsf{r}}_{n\in N} r[n \leftarrow M^{\mathsf{r}}(n)])$. □

Next, we present a results showing that updates to a relational map with sound information yields a sound map.

Lemma 123. *Let $M \in \mathcal{M}$, $M^{\mathsf{r}} \in \mathcal{M}^{\mathsf{r}}$, $n \in \mathcal{N}$, $v \in \mathcal{V}$, and $r \in \mathcal{V}^{\mathsf{r}}$. If $\langle M, v\rangle \in \gamma^{\mathsf{r}}(r)$ and $M \in \dot{\gamma}^{\mathsf{r}}(M^{\mathsf{r}})$, then $M \in \dot{\gamma}^{\mathsf{r}}(M^{\mathsf{r}}[n \mapsto r])$.*

Proof. Clearly, as $\forall n'.\langle M, M(n')\rangle \in \gamma^{\mathsf{r}}(M^{\mathsf{r}}[n \mapsto r](n'))$. □

Our rescoping operations are also sound.

Lemma 124. *Let $M \in \mathcal{M}$, $v_1 \in \mathcal{V}$, $r \in \mathcal{V}^{\mathsf{r}}_N$, and $N' \subseteq \wp(\mathcal{N})$ such that $\mathsf{scopecons}_M(N')$. If $\langle M, v_1\rangle \in \gamma^{\mathsf{r}}_N(r)$, then $\langle M, v_1\rangle \in \gamma^{\mathsf{r}}_{N\cup N'}(r[N \cup N'])$*

Proof.

$$\begin{aligned} r_N[N \cup N'] &= \alpha^{\mathsf{r}}_{N\cup N'}(\gamma^{\mathsf{r}}(r_N) \cap \{\langle M, v\rangle \mid \forall n \in N'\backslash N.M(n) \notin \{\bot, \omega\}\}) \\ &\sqsupseteq^{\mathsf{r}} \alpha^{\mathsf{r}}_{N\cup N'}(\{\langle M, v_1\rangle)\} \cap \{\langle M, v\rangle \mid \forall n \in N'\backslash N.M(n) \notin \{\bot, \omega\}\}) \\ &= \alpha^{\mathsf{r}}_{N\cup N'}(\{\langle M, v_1\rangle)\}) \end{aligned}$$

□

Lemma 125. *Let $M \in \mathcal{M}$, $v_1 \in \mathcal{V}$, $r \in \mathcal{V}^{\mathsf{r}}_N$, and $N' \subseteq N$. If $\langle M, v_1\rangle \in \gamma^{\mathsf{r}}_N(r)$, then $\langle M, v_1\rangle \in \gamma^{\mathsf{r}}_{N'}(r[N'])$*

Proof. Straightforward as $N' \setminus N = \emptyset$. □

Strengthening of a relational value with the information not in the scope is sound as well.

A.4.3.5 Maps

We next show the soundness of updates to the relational maps subject to their concretizations.

Lemma 126. *Let $M \in \mathcal{M}$, $M^{\mathsf{r}} \in \mathcal{M}^{\mathsf{r}}$, $n \in \mathcal{N}$, $v \in \mathcal{V}$, and $r \in \mathcal{V}^{\mathsf{r}}$. If $\langle M, v\rangle \in \gamma^{\mathsf{r}}(r)$, $M \in \dot{\gamma}^{\mathsf{r}}(M^{\mathsf{r}})$, $M(n) \sqsubseteq v$, and $v \neq \omega$, then $M[n \mapsto v] \in \dot{\gamma}^{\mathsf{r}}(M^{\mathsf{r}}[n \mapsto r])$.*

Proof. Suppose not. Let $M_0 = M[n \mapsto v]$ and $M_0^{\mathsf{r}} = M^{\mathsf{r}}[n \mapsto r]$. There must be a node n' such that $\langle M_0, M_0(n')\rangle \notin \gamma^{\mathsf{r}}(M_0^{\mathsf{r}}(n'))$. We branch on several cases:

- $n \notin N_{n'}$. If $n = n'$, the argument is trivial. Otherwise we know that $\langle M, M(n')\rangle \in \gamma^{\mathsf{r}}(M^{\mathsf{r}}(n'))$, $M^{\mathsf{r}}(n') = M_0^{\mathsf{r}}(n')$, and $M(n') = M_0(n')$. Then by Lemma 97 also $\langle M_0, M_0(n')\rangle \in \gamma^{\mathsf{r}}(M_0^{\mathsf{r}}(n'))$.

- $n \in N_{n'}$. Again, we know that $\langle M, M(n')\rangle \in \gamma^{\mathsf{r}}(M^{\mathsf{r}}(n'))$, $M^{\mathsf{r}}(n') = M_0^{\mathsf{r}}(n')$, and $M(n') = M_0(n')$. Since $M \dot{\sqsubseteq} M'$, $M =_{\setminus\{n\}} M'$, and $M'(n) \neq \omega$, then by Lemma 115 $\langle M_0, M_0(n')\rangle \in \gamma^{\mathsf{r}}(M^{\mathsf{r}}(n'))$ and hence $\langle M_0, M_0(n')\rangle \in \gamma^{\mathsf{r}}(M_0^{\mathsf{r}}(n'))$.

□

A.4.3.6 Transformer

We next show that the relational transformer abstracts the concrete transformer given certain invariants on the concrete maps.

Lemma 127. *First, assume that P is an inductive invariant family of a program m where $\forall k, \ell, E.\, (\ell, M, E) \in P_k \implies df_m(M) \wedge \mathsf{envcons}_M(E)$. Now, let e^{ℓ} be an expression of m, $M_f \in \mathcal{M}$, $M \in \mathcal{M}$, $k \in \mathbb{N}$, and E an environment where (e, E) is well-formed, $\mathsf{domvalid}(\ell, m, E)$, and $(\ell, M, E) \in P_k$. Also, let $M_f^{\mathsf{r}} \in \mathcal{M}^{\mathsf{r}}$ and $M^{\mathsf{r}} \in$*

$\mathcal{M}^{\mathsf{r}}$ *such that* $M \in \dot{\gamma}^{\mathsf{r}}(M^{\mathsf{r}})$. *If* $\mathsf{step}_k[\![e^\ell]\!](E)(M) = M_f$ *and* $\mathsf{Step}^{\mathsf{r}}_k[\![e^\ell]\!](E)(M^{\mathsf{r}}) = M^{\mathsf{r}}_f$, *then* $M_f \in \dot{\gamma}^{\mathsf{r}}(M^{\mathsf{r}}_f)$.

Proof. We carry the proof by induction on k and e. The case when $k = 0$ is trivial. We note that $\mathsf{Step}^{\mathsf{r}}_k[\![e^\ell]\!](E)(M^{\mathsf{r}}) = M^{\mathsf{r}} \dot{\sqcup}^{\mathsf{r}} \mathsf{step}^{\mathsf{r}}_k[\![e^\ell]\!](E)(M^{\mathsf{r}})$, hence we focus on $\mathsf{step}^{\mathsf{r}}_k[\![e^\ell]\!]$ and consider all the cases for e. Also observe that $df(M)$ and $\mathsf{envcons}_M(E)$ by P and $(\ell, M, E) \in P_k$.

$e = c^\ell$. Let $v = M(E{\diamond}\ell)$ and $r = M^{\mathsf{r}}(E{\diamond}\ell)$. By $df_m(M)$, we have that $\mathsf{constcons}_m(M)$ which implies $v \in \{\bot, \omega, c\}$. If $v = \omega$ then by $df_n(M)$ we have $\mathsf{errcons}(M)$ and thus $M = M_\omega$. Then it must be that $M^{\mathsf{r}} = M^{\mathsf{r}}_{\top_{\mathsf{r}}}$ and the argument is trivial.

Otherwise, $v \in \{\bot, c\}$. By $\mathsf{envcons}_M(E)$ we have $\mathsf{scopecons}_M(N)$ where $N = N_{E{\diamond}\ell}$. Then by Lemma 120 it follows $\langle M, c\rangle \in \gamma^{\mathsf{r}}_N(c^{\mathsf{r}})$ and by Lemma 122 it must be that $\langle M, c\rangle \in \gamma^{\mathsf{r}}_N(c^{\mathsf{r}}[M^{\mathsf{r}}])$. Since $\mathsf{noerr}(c)$, $\mathsf{noerr}(v)$, $\mathsf{ccomp}(v, c)$, and $\langle M, v\rangle \in \gamma^{\mathsf{r}}(r)$, by Lemma 117 it must be that $\langle M, v \sqcup c\rangle \in \gamma^{\mathsf{r}}_N(r \sqcup^{\mathsf{r}} c^{\mathsf{r}}[M^{\mathsf{r}}])$. The result then follows by Lemma 126.

$e = x^\ell$. Let $M(E{\diamond}\ell) = v$, $M(E(x)) = v_x$, $M^{\mathsf{r}}(E{\diamond}\ell) = r$, and $M^{\mathsf{r}}(E(x)) = r_x$. If $v_x = \omega$ or $v = \omega$, then by $\mathsf{errcons}(M)$ it must be that $M = M_\omega$ and hence $M^{\mathsf{r}} = M^{\mathsf{r}}_{\top_{\mathsf{r}}}$. In this case, the argument is trivial.

Otherwise, we know that $\langle M, v\rangle \in \gamma^{\mathsf{r}}(r)$, and $\langle M, v_x\rangle \in \gamma^{\mathsf{r}}(r_x)$. Let r_0 be r_x implicitly rescoped with $N_{E(x)} \cup N_{E{\diamond}\ell}$. Let also r_1 be r be rescoped with $N_{E(x)} \cup N_{E{\diamond}\ell}$. By $\mathsf{envcons}_M(E)$ and $\mathsf{envcons}(M)$ and Lemma 124, we have $\langle M, v_x\rangle \in \gamma^{\mathsf{r}}(r_0)$. $\langle M, v\rangle \in \gamma^{\mathsf{r}}(r_1)$. As $E(x)$ was not initially in the scope of $E(x)$, then also $\langle M, v_x\rangle \in \gamma^{\mathsf{r}}(r_0[E(x){=}\nu])$.

Next, by $df_m(M)$ we have that $v_x <:^c v$. As by $\mathsf{errcons}(M)$ we have $\mathsf{noerr}(v)$, $\mathsf{noerr}(v_x)$, and $\mathsf{scopecons}_M(N)$ (via $\mathsf{envcons}_M(E)$), then by Lemma 116 we have

that $\langle M, v'_x\rangle \in \gamma^{\mathsf{r}}(r'_x)$ and $\langle M, v'\rangle \in \gamma^{\mathsf{r}}(r')$. As r'_x is rescoped back to $N_{E(x)}$ as r_2 and r' is rescoped back to $N_{E\diamond\ell}$ as r_3, by Lemma 125 we have $\langle M, v'_x\rangle \in \gamma^{\mathsf{r}}(r_2)$ and $\langle M, v'\rangle \in \gamma^{\mathsf{r}}(r_3)$. By Lemma 122, we have $\langle M, v'\rangle \in \gamma^{\mathsf{r}}(r_3[M^{\mathsf{r}}])$.

Let $M_1 = M[E(x) \mapsto v'_x]$, $M_2 = M_1[E\diamond\ell \mapsto v']$, $M_1^{\mathsf{r}} = M^{\mathsf{r}}[E(x) \mapsto r_2]$, and $M_2^{\mathsf{r}} = M_1^{\mathsf{r}}[E\diamond\ell \mapsto r_3]$. By Lemma 126, $M_1 \in \gamma_N^{\mathsf{r}}(M_1^{\mathsf{r}})$. By the increasing property of prop (Lemma 6), we have $M \dot{\sqsubseteq} M_1$. Also, $M =_{\setminus\{E(x)\}} M_1$ and $v'_x \neq \omega$ by $v_x <:^c v$ and Lemma 39. Hence, $\mathsf{envcons}_{M_1}(E)$ and $\mathsf{scopecons}_{M_1}(N)$. Then by Lemma 115 we have $\langle M_1, v'\rangle \in \gamma_N^{\mathsf{r}}(r_3)$. Finally, we then have by Lemma 126 that $M_2 \in \dot{\gamma}^{\mathsf{r}}(M_2^{\mathsf{r}})$.

$e = (e_1^{\ell_1} e_2^{\ell_2})^{\ell}$**.** By i.h., $M_1 \in \dot{\gamma}^{\mathsf{r}}(M_1^{\mathsf{r}})$. By the inductiveness of P, we have $df_m(M_1)$ and $\mathsf{envcons}_{M_1}(E)$. If $v_1 = \bot$, then $M_f = M_1$ and $M_1^{\mathsf{r}} \dot{\sqsubseteq}^{\mathsf{r}} M_f^{\mathsf{r}}$ by the obvious increasing property of $\mathsf{Step}_{k+1}^{\mathsf{r}}[\![\cdot]\!]$ and $\mathsf{prop}^{\mathsf{r}}$ (Lemma 6); the result thus follows by the monotonicity of $\dot{\gamma}^{\mathsf{r}}$. If $v_1 = \omega$, then by $\mathsf{errcons}(M_1)$ it must be that $M_1 = M_\omega$ and hence $M^{\mathsf{r}} = M^{\mathsf{r}}_{\top^{\mathsf{r}}}$, in which case the result follows immediately. If $v_1 \notin \mathcal{T}$, then by the definition of γ^{r} also $r_1 \notin \mathcal{T}_N^{\mathsf{r}}$. In that case, $M_f = M_\omega$ and $M^{\mathsf{r}} = M^{\mathsf{r}}_{\top^{\mathsf{r}}}$ and the argument is straightforward. If $v_1 \in \mathcal{T}$, but $r_1 \notin \mathcal{T}_N^{\mathsf{r}}$, then $r_1 = \top^{\mathsf{r}}$ and hence $M_f^{\mathsf{r}} = M^{\mathsf{r}}_{\top^{\mathsf{r}}}$. Otherwise, by i.h. $M_2 \in \dot{\gamma}^{\mathsf{r}}(M_2^{\mathsf{r}})$. Again by the inductive property of P, we have $df_m(M_2)$ and $\mathsf{envcons}_{M_2}(E)$. The cases when $v_2 \in \{\bot, \omega\}$ are handled identically as for v_1.

By $\mathsf{domvalid}(\ell, m, E)$, $\mathsf{domvalid}(\ell_1, m, E)$, and $\mathsf{domvalid}(\ell_2, m, E)$, Lemma 32, and Lemma 33, we have that $v_1 = M_2(E\diamond\ell_1)$. Similarly, by Lemma 112 it also holds $r_1 = M_2^{\mathsf{r}}(E\diamond\ell_1)$. Let $v = M_2(E\diamond\ell)$ and $r = M_2^{\mathsf{r}}(E\diamond\ell)$. Then by $M_2 \in \dot{\gamma}^{\mathsf{r}}(M_2^{\mathsf{r}})$, we have $\langle M_2, v_1\rangle \in \gamma^{\mathsf{r}}(r_1)$, $\langle M_2, v_2\rangle \in \gamma^{\mathsf{r}}(r_2)$, and $\langle M_2, v\rangle \in \gamma^{\mathsf{r}}(r)$. As r is implicitly rescoped with $N_{E\diamond\ell} \cup \{E\diamond\ell_1\}$ to r_0, then by Lemma 124, $\langle M_2, v_2\rangle \in \gamma^{\mathsf{r}}(r_0)$.

If any of v_1, v_2, or v_3 is ω, then $M_2 = M_\omega$ and the argument goes as usual. Oth-

erwise, we have $\mathsf{noerr}(v_1)$, $\mathsf{noerr}(v_2)$, $\mathsf{noerr}(v_3)$ and by $df_m(M_2)$ also $v_1 <:^c [E\diamond\ell_1 \mapsto \langle v_2, v\rangle]$. Since $\langle M_2, [E\diamond\ell_1 \mapsto \langle v_2, v_2\rangle]\rangle \in \gamma^{\mathsf{r}}_N([E\diamond\ell_1 \mapsto \langle r_2, r_0\rangle])$ by def. of γ^{r} and $\mathsf{envcons}_{M_2}(E)$, we have by Lemma 116 that $\langle M_2, v'_1\rangle \in \gamma^{\mathsf{r}}_N(r'_1)$ and $\langle M_2, [E\diamond\ell_1 \mapsto \langle v'_2, v'\rangle]\rangle \in \gamma^{\mathsf{r}}_N([E\diamond\ell_1 \mapsto \langle r'_2, r'_0\rangle])$ and hence by def. of γ^{r} we have $\langle M_2, v'_1\rangle \in \gamma^{\mathsf{r}}(r'_1)$, $\langle M_2, v'_2\rangle \in \gamma^{\mathsf{r}}(r'_2)$, and $\langle M_2, v'\rangle \in \gamma^{\mathsf{r}}(r'_0)$. As r'_0 is implicitly rescoped back with $N_{E\diamond\ell}$ to r''_0, by Lemma 125 we have $\langle M_2, v'\rangle \in \gamma^{\mathsf{r}}(r''_0)$. The argument for storing the results to maps M_2 and M^{r}_2 then follows by Lemma 39, 115, and 126 as earlier.

$(e^{\ell_0}_0 \,?\, e^{\ell_1}_1 : e^{\ell_2}_2)^{\ell}$. By i.h., we have that $M_0 \in \gamma^{\mathsf{r}}_N(M^{\mathsf{r}}_0)$. By the inductiveness of P, we have $df_m(M_0)$ and $\mathsf{envcons}_{M_0}(E)$. The cases when $v_0 \in \{\bot, \omega\}$ are handled as expected. If $v_0 \in \mathit{Bool}$ and $r_0 \notin \mathit{Bool}^{\mathsf{r}}$, then $r_0 = \top^{\mathsf{r}}$ by the definition of γ^{r} and $M^{\mathsf{r}}_f = M^{\mathsf{r}}_{\top^{\mathsf{r}}}$.

Otherwise, the argument is as follows. Suppose $v_0 = \mathit{true}$; the argument for the case when $v_0 = \mathit{false}$ is identical. Since $\mathsf{envcons}_{M_0}(E)$ implies $\mathsf{scopecons}_{M_0}(N_{E\diamond\ell})$, by Lemma 120 we have that $\langle M_0, \mathit{true}\rangle \in \gamma^{\mathsf{r}}(\mathit{true}^{\mathsf{r}})$. Since $\langle M_0, \mathit{true}\rangle \in \gamma^{\mathsf{r}}(r_0)$ by $M_0 \in \gamma^{\mathsf{r}}_N(M^{\mathsf{r}}_0)$, we have that $\langle M_0, \mathit{true}\rangle \in (\gamma^{\mathsf{r}}(\mathit{true}^{\mathsf{r}}) \cap \gamma^{\mathsf{r}}(r_0))$. Then by Lemma 9, it also holds $\langle M_0, \mathit{true}\rangle \in \gamma^{\mathsf{r}}(\mathit{true}^{\mathsf{r}} \sqcap^{\mathsf{r}} r_0)$. Now, let $n' \in N_{E\diamond\ell_0}$. By $M_0 \in \dot{\gamma}^{\mathsf{r}}(M^{\mathsf{r}}_0)$, we have that $\langle M_0, M_0(n')\rangle \in \gamma^{\mathsf{r}}(M^{\mathsf{r}}_0(n'))$. As $\langle M_0, \mathit{true}\rangle = \langle M_0, M_0(E\diamond\ell_0)\rangle$, we have that $\langle M_0, M_0(n')\rangle \in \gamma^{\mathsf{r}}(M^{\mathsf{r}}_0(n')[E\diamond\ell_0 \leftarrow \mathit{true}^{\mathsf{r}} \sqcap^{\mathsf{r}} r_0])$ by Lemma 121. Hence, by Lemma 123 it is the case that $M_0 \in \dot{\gamma}^{\mathsf{r}}(M^{\mathsf{r}}_0[n' \mapsto M^{\mathsf{r}}_0(n')[E\diamond\ell_0 \leftarrow \mathit{true}^{\mathsf{r}} \sqcap^{\mathsf{r}} r_0]])$. Further, by Lemma 10 it is also the case that $M_0 \in \dot{\gamma}^{\mathsf{r}}(\dot{\sqcap}^{\mathsf{r}}_{n'\in N_{E\diamond\ell_0}} M^{\mathsf{r}}_0[n' \mapsto M^{\mathsf{r}}_0(n')[E\diamond\ell_0 \leftarrow \mathit{true}^{\mathsf{r}} \sqcap^{\mathsf{r}} r_0]])$. Therefore, $M_0 \in \dot{\gamma}^{\mathsf{r}}(M^{\mathsf{r}}_0[E\diamond\ell_0 \leftarrow \mathit{true}^{\mathsf{r}} \sqcap^{\mathsf{r}} r_0])$.

It follows then by i.h. that $M_1 \in \gamma^{\mathsf{r}}_N(M^{\mathsf{r}}_1)$ and by the inductiveness of P, we have $df_m(M_1)$ and $\mathsf{envcons}_{M_1}(E)$. The argument continues as usual by Lemma 39, 115, and 126 as before.

$e = (\mu f.\lambda x.e_1^{\ell_1})^{\ell}$. Let $v = M(E\diamond\ell)$. By $df_m(M)$ it follows that $\mathsf{constcons}_m(M)$ which implies that $v \in \{\bot, \omega\} \cup \mathcal{T}$. If $v = \bot$, then $T = T_\bot$ and $M_f = M[E\diamond\ell \mapsto T_\bot]$. By the definition of γ_N^{r}, then either (1) $r = \bot^{\mathsf{r}}$ in which case $M_f^{\mathsf{r}} = M^{\mathsf{r}}[E\diamond\ell \mapsto T_{\bot^{\mathsf{r}}}^{\mathsf{r}}]$, (2) $r \in \mathcal{R}_N^{\mathsf{r}}$ in which case $M_f^{\mathsf{r}} = M_{\top^{\mathsf{r}}}^{\mathsf{r}}$, (3) $r \in \mathcal{T}^{\mathsf{r}}$ and the result follows by the obvious increasing property of $\mathsf{Step}_{k+1}^{\mathsf{r}}[\![\cdot]\!]$ and $\mathsf{prop}^{\mathsf{r}}$ (Lemma 6), or (4) $r = \top^{\mathsf{r}}$ in which case $M_f^{\mathsf{r}} = M_{\top^{\mathsf{r}}}^{\mathsf{r}}$. If $v = \omega$ then by $\mathsf{errcons}(M)$ it must be that $M = M_f = M_\omega$ and hence $M^{\mathsf{r}} = M_f^{\mathsf{r}} = M_{\top^{\mathsf{r}}}^{\mathsf{r}}$. If $v \in \mathcal{T}$ but $r \notin \mathcal{T}^{\mathsf{r}}$, then $r = \top^{\mathsf{r}}$ and hence $M_f^{\mathsf{r}} = M_{\top^{\mathsf{r}}}^{\mathsf{r}}$.

We are left with the case where $T \in \mathcal{T}$ and $T^{\mathsf{r}} \in \mathcal{T}^{\mathsf{r}}$ where $v = T$ and $r = T^{\mathsf{r}}$. By γ^{r}, $n_{\mathsf{cs}} \in T$ implies $n_{\mathsf{cs}} \in T^{\mathsf{r}}$. For the case where $n_{\mathsf{cs}} \notin T$ and $n_{\mathsf{cs}} \notin T^{\mathsf{r}}$, the argument is trivial. When $n_{\mathsf{cs}} \notin T$ but $n_{\mathsf{cs}} \in T^{\mathsf{r}}$, the result follows by the obvious increasing property of $\mathsf{Step}_{k+1}^{\mathsf{r}}[\![\cdot]\!]$ and $\mathsf{prop}^{\mathsf{r}}$. We are thus left with the case when $n_{\mathsf{cs}} \in T$ and $n_{\mathsf{cs}} \in T^{\mathsf{r}}$. The case when $\pi_1(T(n_{\mathsf{cs}})) \in \{\bot, \omega\}$ are handled as expected.

We thus continue assuming $\pi_1(T(n_{\mathsf{cs}})) \notin \{\bot, \omega\}$. First, we have by $df_m(M)$ that $[n_{\mathsf{cs}} \mapsto \langle M(n_x), M(E_1\diamond\ell_1)\rangle] <:^c T$ and $T <:^c M(n_f)$. By the def. of $<:^c$, $M(n_x) \notin \{\bot, \omega\}$. Next, $\langle M, M(E_1\diamond\ell_1)\rangle \in \gamma^{\mathsf{r}}(M^{\mathsf{r}}(E_1\diamond\ell_1))$ by $M \in \gamma_N^{\mathsf{r}}(M^{\mathsf{r}})$. Since $M^{\mathsf{r}}(E_1\diamond\ell_1)$ is first implicitly rescoped with $N_{E\diamond\ell_1} \cup \{n_{\mathsf{cs}}\}$ to r_0 and $M(n_x) \notin \{\bot, \omega\}$, by Lemma 124 we have $\langle M, M(E_1\diamond\ell_1)\rangle \in \gamma^{\mathsf{r}}(r_0)$. As $n_{\mathsf{cs}} \notin N_{E\diamond\ell_1}$, then it must be that $\langle M, M(E_1\diamond\ell_1)\rangle \in \gamma^{\mathsf{r}}(r_0[n_{\mathsf{cs}}{=}n_x])$. Then, r_0 is implicitly rescoped to the same scope but without n_x and then by Lemma 125 we have $\langle M, M(E_1\diamond\ell_1)\rangle \in \gamma^{\mathsf{r}}(r_1)$. We hence have $[n_{\mathsf{cs}} \mapsto \langle M(n_x), M(E_1\diamond\ell_1)\rangle] \in \gamma^{\mathsf{r}}[n_{\mathsf{cs}} \mapsto \langle M^{\mathsf{r}}(n_x), r_1\rangle$, $T \in \gamma^{\mathsf{r}}(T^{\mathsf{r}})$, and $M(n_f) \in \gamma_N^{\mathsf{r}}(M^{\mathsf{r}}(n_f))$. As $\mathsf{scopecons}_M(N_{E\diamond\ell})$ by $\mathsf{envcons}_M(E)$, $\mathsf{noerr}(T)$, $\mathsf{noerr}(M(n_x))$, $\mathsf{noerr}(M(n_f))$, and $\mathsf{noerr}(M(E_1\diamond\ell_1))$, by Lemma 116 we have that $\langle M, T'\rangle \in \gamma_N^{\mathsf{r}}(T_1^{\mathsf{r}})$, $\langle M, T''\rangle \in \gamma_N^{\mathsf{r}}(T_2^{\mathsf{r}})$, $\langle M, T_x'\rangle \in \gamma_N^{\mathsf{r}}(T_x^{\mathsf{r}})$, and $\langle M, T_f'\rangle \in \gamma^{\mathsf{r}}(T_f^{\mathsf{r}})$. By Lemma 39, we have $\mathsf{noerr}(T'')$ and $\mathsf{noerr}(T')$. By Lemma 45 we have that

$\mathsf{ccomp}(T', T'')$ and thus $\langle M, T' \sqcup T'' \rangle \in \gamma^{\mathsf{r}}(T_1^{\mathsf{r}} \sqcup^{\mathsf{r}} T_2^{\mathsf{r}})$ by Lemma 117.

Note that by Lemma 40 it must be that $\mathsf{noerr}(T'_x)$ and hence $v'_x \neq \omega$. By this and $\mathsf{envcons}_M(E)$ it must be that $\mathsf{envcons}_{M_1}(E_1)$. It follows then by i.h. that $M_1 \in \gamma_N^{\mathsf{r}}(M_1^{\mathsf{r}})$ and by inductiveness of P we have $df_m(M_1)$ and $\mathsf{envcons}_{M_1}(E_1)$. The argument continues as usual by Lemma 39, 115, and 126 as before. Also, the soundness argument for implicit rescoping and strengthening of r'_1 goes as usual and relies on the fact that n_x is not in the scope of r'_1.

Remember that for every $n_{\mathsf{cs}} \in \mathsf{dom}(\bar{M})$ we have $n_{\mathsf{cs}} \in \mathsf{dom}(\bar{M}^{\mathsf{r}})$ and $\bar{M}(n_{\mathsf{cs}}) \in \dot{\gamma}^{\mathsf{r}}(\bar{(M^{\mathsf{r}})}(n_{\mathsf{cs}}))$. Given $\mathsf{fin}(M)$ implied by $df_m(M)$, we have $|\mathsf{dom}(\bar{M})| = K$ where $K \in \mathbb{N}$. By the inductiveness of P, we have that $df_m(M_f)$ and hence $\mathsf{errcons}(M_f)$. If $M_f = M_\omega$ then there exists n_{cs} such that $\bar{M}(n_{\mathsf{cs}}) = M_\omega$ in which case $\bar{M}^{\mathsf{r}}(n_{\mathsf{cs}}) = M^{\mathsf{r}}_{\top^{\mathsf{r}}}$ and hence $M_f^{\mathsf{r}} = M^{\mathsf{r}}_{\top^{\mathsf{r}}}$. Otherwise, by the definition of $\dot{\sqcup}$ we have that $\forall M \in \bar{M}.\, M \neq M_\omega$. By Lemma 44 we have $\mathsf{ccomp}(\bar{M})$ and $\forall M \in \bar{M}.\, \mathsf{errcons}(M)$. The result then follows from Lemma 118. □

We can now prove that relational semantics abstract the concrete one.

Proof (of Theorem 1)**.**

Proof. We assume a fixed program e^{ℓ_m}. We first define a set $P_k \in \wp(Loc \times \mathcal{M} \times \mathcal{E})$ for every fuel k as the smallest set satisfying the following. Initially, $(\ell_m, M_\perp, \emptyset) \in P_k$. Second, $(\ell, M, E) \in P_k$ if (1) $\ell \in Loc(e)$, (2) $(e(\ell), E)$ is well-formed, (3) $\mathsf{domvalid}(\ell, e, E)$, (4) there exists a path map M^{w} such that $\mathsf{cons}_m(M^{\mathsf{w}})$, $M^{\mathsf{w}} \in \mathsf{dom}(\mathsf{strip})$, $M = \mathsf{strip}(M^{\mathsf{w}})$, and $\mathsf{envcons}_{M^{\mathsf{w}}}(E)$. Lastly, for every $(\ell, M, E) \in P_k$, if $\mathsf{step}_k^{\mathsf{w}}[\![e(\ell)]\!](E)(M^{\mathsf{w}}) = M_1^{\mathsf{w}}$, $\mathsf{cons}_m(M^{\mathsf{w}})$, $\mathsf{cons}_m(M_1^{\mathsf{w}})$, $M^{\mathsf{w}} \in \mathsf{dom}(\mathsf{strip})$, $M_1^{\mathsf{w}} \in \mathsf{dom}(\mathsf{strip})$, $M = \mathsf{strip}(M^{\mathsf{w}})$, and $M_1 = \mathsf{strip}(M_1^{\mathsf{w}})$, then $(\ell, M_1, E) \in P_k$.

By Theorem 7, Theorem 8, $\mathsf{cons}_m(M_\perp^{\mathsf{w}})$, $M_\perp^{\mathsf{w}} \in \mathsf{dom}(\mathsf{strip})$, and $M_\perp = \mathsf{strip}(M_\perp^{\mathsf{w}})$,

we have that P is an inductive invariant family of m. By Lemma 95 and the definition of $\mathsf{envcons}$ for path maps, we have that $(\ell, M, E) \in P_k$, for every k, implies $df_m(M)$ and $\mathsf{envcons}_M(E)$. Since $(\ell_m, M_\perp, \emptyset) \in P_k$, we have that the strongest inductive invariant I of m is subsumed by P, i.e, $S_k \subseteq P_k$. The result then follows from the definition of $\mathbf{C}[\![\cdot]\!]$ and Lemma 127. $\square$

A.5 Collapsed Semantics Proofs

We first prove that our definitions of collapsed meet and join are correct. A depth of a collapsed value is defined in the similar way as for the concrete and relational values. We hence reuse the function depth to denote the depth of collapsed values.

Proof (of Lemma 8)**.**

Proof. Similar to the proof of Lemma 4, the proof is carried by case analysis on elements of U and induction on the minimum depth thereof when U consists of collapsed tables only. □

The next proof shows that the concretization function for collapsed values is a complete meet-morphism.

Theorem 11 (Complete Meet Morphism of $\gamma^{\mathsf{cr}}_{\hat{N}}$)**.** *Let $U \in \wp(\mathcal{V}^{\mathsf{cr}}_{\hat{N}})$ and let N be a concrete scope. Then $\gamma^{\mathsf{cr}}_{\delta}(\sqcap^{\mathsf{cr}} U) = \sqcap^{\mathsf{r}}_{u \in U} \gamma^{\mathsf{cr}}_{\delta}(u)$ for $\delta \in \Delta(\hat{N}, N)$.*

Proof. We carry the proof by case analysis on elements of U and induction on the minimum depth thereof when U consists of tables only.

We first consider the trivial (base) cases where U is not a set of multiple collapsed tables.

- $U = \emptyset$. Here, $\gamma^{\mathsf{cr}}_{\delta}(\sqcap^{\mathsf{cr}}\emptyset) = \gamma^{\mathsf{cr}}_{\delta}(\top^{\mathsf{cr}}) = \top^{\mathsf{cr}} = \sqcap^{\mathsf{r}}\emptyset$.
- $U = \{u\}$. Trivial.
- $R^{\mathsf{cr}} \in U, T^{\mathsf{cr}} \in U$. We have $\sqcap^{\mathsf{cr}}\{R^{\mathsf{cr}}, T^{\mathsf{cr}}\} = \bot^{\mathsf{cr}}$ and since $\bot^{\mathsf{cr}}$ is the bottom element, $\sqcap^{\mathsf{cr}} U = \bot^{\mathsf{cr}}$. Similarly, $\gamma^{\mathsf{cr}}_{\delta}(R^{\mathsf{cr}}) \sqcap^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(T^{\mathsf{cr}}) = \bot^{\mathsf{r}} = \gamma^{\mathsf{cr}}_{\hat{N}}(\bot^{\mathsf{cr}})$ so by the definition of $\sqcap^{\mathsf{r}}$ we have $\sqcap^{\mathsf{r}}_{u \in U} \gamma^{\mathsf{cr}}_{\delta}(u) = \bot^{\mathsf{r}}$.

- $U \subseteq \mathcal{R}^{\mathsf{cr}}_{\hat{N}}$. Here,

$$\gamma^{\mathsf{cr}}_{\delta}(\sqcap^{\mathsf{cr}} U) = \gamma^{\mathsf{cr}}_{\delta}(\sqcap^{\mathsf{cr}}\{R^{\mathsf{cr}}_{\hat{N}} \mid R^{\mathsf{cr}} \in U\}) = \gamma^{\mathsf{cr}}_{\delta}(\bigcap_{R^{\mathsf{cr}} \in U} R^{\mathsf{cr}})$$

[by def. of $\sqcap^{\mathsf{cr}}$]

$$= \{D^{\mathsf{r}} \mid D^{\mathsf{cr}} \in (\bigcap_{R^{\mathsf{cr}} \in U} R^{\mathsf{cr}})) \wedge D^{\mathsf{cr}}(\nu) = D^{\mathsf{r}}(\nu) \wedge \forall \hat{n} \in \hat{N}.\, D^{\mathsf{r}}(\delta(\hat{n})) = D^{\mathsf{cr}}(\hat{n})\}$$

[by def. of $\gamma^{\mathsf{cr}}_{\delta}$]

$$= \bigcap_{R^{\mathsf{cr}} \in U} \{D^{\mathsf{r}} \mid D^{\mathsf{cr}} \in R^{\mathsf{cr}} \wedge D^{\mathsf{cr}}(\nu) = D^{\mathsf{r}}(\nu) \wedge \forall \hat{n} \in \hat{N}.\, D^{\mathsf{r}}(\delta(\hat{n})) = D^{\mathsf{cr}}(\hat{n})\}$$

$$= \bigcap_{R^{\mathsf{cr}} \in U} \gamma^{\mathsf{cr}}_{\delta}(R^{\mathsf{cr}}) = \sqcap^{\mathsf{r}}_{R^{\mathsf{cr}} \in U} \gamma^{\mathsf{cr}}_{\delta}(R^{\mathsf{cr}})$$ **[by def. of $\gamma^{\mathsf{cr}}_{\hat{N}}$ and $\sqcap^{\mathsf{r}}$]**

- $\top^{\mathsf{cr}} \in U$. Here, $\sqcap^{\mathsf{cr}} U = \sqcap^{\mathsf{cr}}(U/\{\top^{\mathsf{cr}}\})$ and $\sqcap^{\mathsf{r}}_{u \in U} \gamma^{\mathsf{cr}}_{\delta}(u) = \sqcap^{\mathsf{r}}_{u \in U, u \neq \top^{\mathsf{cr}}} \gamma^{\mathsf{cr}}_{\delta}(u)$ since $\gamma^{\mathsf{cr}}_{\hat{N}}(\top^{\mathsf{cr}}) = \top^{\mathsf{r}}$. The set $U/\{\top^{\mathsf{cr}}\}$ either falls into one of the above cases or consists of multiple tables, which is the case we show next.

Let $U \subseteq \mathcal{T}^{\mathsf{cr}}_{\hat{N}}$ and $|U| > 1$. Let d be the minimum depth of any table in U.

$$\gamma^{\mathsf{cr}}_{\delta}(\sqcap^{\mathsf{cr}} U) = \gamma^{\mathsf{cr}}_{\delta}(\langle z, \sqcap^{\mathsf{cr}}\{\pi_1(io(T^{\mathsf{cr}})) \mid T^{\mathsf{cr}} \in U\}, \sqcap^{\mathsf{cr}}_{\hat{N}}\{\pi_2(io(T^{\mathsf{cr}})) \mid T^{\mathsf{cr}} \in U\}\rangle)$$

[by def. of $\sqcap^{\mathsf{cr}}$]

$$= \lambda n_{\mathsf{cs}}.\langle \gamma^{\mathsf{cr}}_{\delta}(\sqcap^{\mathsf{cr}}\{\pi_1(io(T^{\mathsf{cr}})) \mid T^{\mathsf{cr}} \in U\}), \gamma^{\mathsf{cr}}_{\delta[z \mapsto n_{\mathsf{cs}}]}(\sqcap^{\mathsf{cr}}\{\pi_2(io(T^{\mathsf{cr}})) \mid T^{\mathsf{cr}} \in U\})\rangle$$

[by def. of γ^{cr}]

$$= \lambda n_{\mathsf{cs}}.\langle \sqcap^{\mathsf{r}}\{\gamma^{\mathsf{cr}}_{\delta}(\pi_1(io(T^{\mathsf{cr}}))) \mid T^{\mathsf{cr}} \in U\}, \sqcap^{\mathsf{r}}\{\gamma^{\mathsf{cr}}_{\delta[z \mapsto n_{\mathsf{cs}}]}(\pi_2(io(T^{\mathsf{cr}}))) \mid T^{\mathsf{cr}} \in U\}\rangle$$

[by i.h. on the min. depth of $\{\pi_1(io(T^{\mathsf{cr}}))) \mid T^{\mathsf{cr}} \in U\}$ and $\{\pi_2(io(T^{\mathsf{cr}}))) \mid T^{\mathsf{cr}} \in U\}$]

$$= \sqcap^{\mathsf{r}}_{u \in U} \gamma^{\mathsf{cr}}_{\delta}(u)$$ **[by def. of $\sqcap^{\mathsf{r}}$]**

□

We can now state the monotonicity of functions constituting the Galois connection between collapsed and relational values.

Lemma 128. *For any abstract scope $\hat{N}$, concrete scope N, and $\delta \in \Delta(\hat{N}, N)$, both $\alpha^{\mathsf{cr}}_\delta$ and $\gamma^{\mathsf{cr}}_\delta$ are monotone.*

Proof. The argument follows from the fact that $\alpha^{\mathsf{cr}}_\delta$ and $\gamma^{\mathsf{cr}}_\delta$ constitute a Galois connection (Prop.1), as evidenced by Theorem 11. □

The meet morphism requirement is also satisfied by the concretization function defined over collapsed maps.

Theorem 12 (Complete Meet Morphism of $\dot{\gamma}^{\mathsf{cr}}$)**.** *Let $m^{\mathsf{cr}} \in \wp(\mathcal{M}^{\mathsf{cr}})$. Then $\dot{\gamma}^{\mathsf{cr}}(\dot{\sqcap}^{\mathsf{cr}} m^{\mathsf{cr}}) = \dot{\sqcap}^{\mathsf{r}}_{M^{\mathsf{cr}} \in m^{\mathsf{cr}}} \dot{\gamma}^{\mathsf{cr}}(M^{\mathsf{cr}})$.*

Proof.

$$\dot{\gamma}^{\mathsf{cr}}(\dot{\sqcap}^{\mathsf{cr}} m^{\mathsf{cr}}) = \Lambda n.\ \sqcap^{\mathsf{r}} \{\gamma^{\mathsf{cr}}_\delta(\sqcap^{\mathsf{cr}}\{M^{\mathsf{cr}}(\rho(n)) \mid M^{\mathsf{cr}} \in m^{\mathsf{cr}}\} \mid \delta \in \Delta(N_{\rho(n)}, \hat{N}_n)\}$$

[by def. of $\sqcap^{\mathsf{cr}}$ and $\dot{\gamma}^{\mathsf{cr}}$]

$$= \Lambda n.\ \sqcap^{\mathsf{r}} \{\sqcap^{\mathsf{r}}\{\gamma^{\mathsf{cr}}_\delta(M^{\mathsf{cr}}(\rho(n))) \mid M^{\mathsf{cr}} \in m^{\mathsf{cr}}\} \mid \delta \in \Delta(N_{\rho(n)}, \hat{N}_n)\}$$

[by Theorem 11]

$$= \Lambda n.\ \sqcap^{\mathsf{r}} \{\sqcap^{\mathsf{r}}\{\gamma^{\mathsf{cr}}_\delta(M^{\mathsf{cr}}(\rho(n))) \mid \delta \in \Delta(N_{\rho(n)}, \hat{N}_n)\} \mid M^{\mathsf{cr}} \in m^{\mathsf{cr}}\}$$

[by assoc. of meets ($\sqcap^{\mathsf{cr}}$)]

$$= \dot{\sqcap}^{\mathsf{r}}_{M^{\mathsf{cr}} \in m^{\mathsf{cr}}} \dot{\gamma}^{\mathsf{cr}}(M^{\mathsf{cr}})$$

[by def. of $\sqcap^{\mathsf{cr}}$ and $\dot{\gamma}^{\mathsf{cr}}$]

□

Moreover, $\alpha^{\mathsf{cr}}_\delta$ is surjective and $\gamma^{\mathsf{cr}}_\delta$ is injective for any $\delta \in \Delta(\hat{N}, N)$ given a concrete scope N. Hence, the Galois connection on collapsed values is in fact a Galois insertion.

Lemma 129. *Let $\delta \in \Delta(\hat{N}, N)$ for some abstract scope $\hat{N}$ and concrete scope N. Then, $\alpha^{\mathsf{cr}}_\delta$ is surjective and $\gamma^{\mathsf{cr}}_\delta$ is injective.*

Proof. We show the injectivity of $\gamma^{\mathsf{cr}}_{\delta}$. The surjective nature of $\alpha^{\mathsf{cr}}_{\delta}$ then follows from Proposition 2. Suppose $\gamma^{\mathsf{cr}}_{\delta}$ is not surjective. Then, there exist $u_1 \in \mathcal{V}^{\mathsf{cr}}_{\hat{N}}$ and $u_2 \in \mathcal{V}^{\mathsf{cr}}_{\hat{N}}$ s.t. $u_1 \neq u_2$ (modulo variable renaming) and $\gamma^{\mathsf{cr}}_{\delta}(u_1) = \gamma^{\mathsf{cr}}_{\delta}(u_2)$. We arrive at contradiction by structural induction on u_1 and u_2. Let $\gamma^{\mathsf{cr}}_{\delta}(u_1) = r_1$ and $\gamma^{\mathsf{cr}}_{\delta}(u_2) = r_2$.

Base cases.

- $u_1 = \bot^{\mathsf{cr}}$ or $u_2 = \bot^{\mathsf{cr}}$. Assume $u_1 = \bot^{\mathsf{cr}}$ w.l.g. Then, $r_1 = \bot^{\mathsf{r}}$. By def. of $\gamma^{\mathsf{cr}}_{\delta}$, $r_2 = \bot^{\mathsf{r}}$ iff $u_2 = \bot^{\mathsf{cr}}$, contradiction.

- $u_1 = \top^{\mathsf{cr}}$ or $u_2 = \top^{\mathsf{cr}}$. Assume $u_1 = \top^{\mathsf{cr}}$ w.l.g. Then, $r_1 = \top^{\mathsf{r}}$. By def. of $\gamma^{\mathsf{cr}}_{\delta}$, $r_2 = \top^{\mathsf{r}}$ iff $u_2 = \top^{\mathsf{cr}}$, contradiction.

- $u_1 \in \mathcal{R}^{\mathsf{cr}}_{\hat{N}}, u_2 \in \mathcal{T}^{\mathsf{cr}}_{\hat{N}}$ or $u_1 \in \mathcal{T}^{\mathsf{cr}}_{\hat{N}}, u_1 \in \mathcal{R}^{\mathsf{cr}}_{\hat{N}}$. Assume $u_1 = \mathcal{R}^{\mathsf{cr}}_{\hat{N}}$ and $u_2 \in \mathcal{T}^{\mathsf{cr}}_{\hat{N}}$ w.l.g. Then, $r_1 \in \mathcal{R}^{\mathsf{r}}_{N}$ and $r_2 \in \mathcal{T}^{\mathsf{r}}_{N}$. Hence, $r_1 \neq r_2$.

- $u_1 \in \mathcal{R}^{\mathsf{cr}}_{\hat{N}}, u_2 \in \mathcal{R}^{\mathsf{cr}}_{\hat{N}}$. By def. of $\gamma^{\mathsf{cr}}_{\delta}$ and δ being a function it follows $r_1 \neq r_2$. Essentially, $\gamma^{\mathsf{cr}}_{\delta}$ simply *renames* dependency variables in dep. vectors using δ.

Induction case. Let $u_1 = \langle z, u_{1i}, u_{1o} \rangle$ and $u_2 = \langle z, u_{2i}, u_{2o} \rangle$. By $u_1 \neq u_2$, we have $u_{1i} \neq u_{2i}$ or $u_{1o} \neq u_{2o}$. The result then follows from the i.h. □

A.5.1 Domain Operations

In this subsection, we state the basic properties and the corresponding proofs of strengthening operations on collapsed values.

A.5.1.1 Soundness, Strengthening, Idempotency, and Monotonicity

We start by showing soundness of the domain operations of the collapsed semantics subject to the relational semantics. We first show the proof for strengthening by

imposing equality between two abstract nodes.

Lemma 130. *Let $\delta_1 \in \Delta(\hat{N}, N)$ for some abstract scope $\hat{N}$ and concrete scope N, $r \in \mathcal{V}^{\mathsf{r}}_{N}$, $u \in \mathcal{V}^{\mathsf{cr}}_{\hat{N}}$, $\hat{n}_1 \in \hat{\mathcal{N}}$, and $\hat{n}_2 \in \hat{\mathcal{N}}$. If $r \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta_1}(u)$, then $r[\delta_1(\hat{n}_1){=}\delta_1(\hat{n}_2)] \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta_1}(u[\hat{n}_1{=}\hat{n}_2])$.*

Proof. We recall $u[\hat{n}_1{=}\hat{n}_2] = \bigsqcup^{\mathsf{cr}} \{\, \alpha^{\mathsf{cr}}_{\delta}(\gamma^{\mathsf{cr}}_{\delta}(u)[\delta(\hat{n}_1){=}\delta(\hat{n}_2)]) \mid N \subseteq \mathcal{N} \wedge \delta \in \Delta(\hat{N}, N) \,\}$ for a given concrete scope N.

$$
\begin{aligned}
r[\delta_1(\hat{n}_1){=}\delta_1(\hat{n}_2)] &\sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u)[\delta_1(\hat{n}_1){=}\delta_1(\hat{n}_2)] && \text{by Lemma 100}\\
\alpha^{\mathsf{cr}}_{\delta_1}(r[\delta_1(\hat{n}_1){=}\delta_1(\hat{n}_2)]) &\sqsubseteq^{\mathsf{r}} \alpha^{\mathsf{cr}}_{\delta_1}(\gamma^{\mathsf{cr}}_{\delta}(u)[\delta_1(\hat{n}_1){=}\delta_1(\hat{n}_2)]) && \text{by Lemma 128}\\
\alpha^{\mathsf{cr}}_{\delta_1}(r[\delta_1(\hat{n}_1){=}\delta_1(\hat{n}_2)]) &\sqsubseteq^{\mathsf{r}} u[\delta_1(\hat{n}_1){=}\delta_1(\hat{n}_2)] && \text{by def. of } u[\delta_1(\hat{n}_1){=}\delta_1(\hat{n}_2)]\\
r[\delta_1(\hat{n}_1){=}\delta_1(\hat{n}_2)] &\sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta_1}(u[\delta_1(\hat{n}_1){=}\delta_1(\hat{n}_2)]) && \text{by Prop. 1}
\end{aligned}
$$

□

The above result can be made more general.

Lemma 131. *Strengthening operations on collapsed values are sound.*

Proof. The proof closely follows that of Lemma 130: the proof relies on the properties of monotonicity of abstraction and concretization functions and monotonicity of the corresponding strengthening operation defined over relational values. □

We next show that the strengthening operations on collapsed values indeed perform strengthening.

Lemma 132. *Let $u_1 \in \mathcal{V}^{\mathsf{cr}}_{\hat{N}}$, $u \in \mathcal{V}^{\mathsf{cr}}_{\hat{N}}$, $\hat{n}_1 \in \hat{\mathcal{N}}$ and $\hat{n}_2 \in \hat{\mathcal{N}}$. Then $u[\hat{n}_1{=}\hat{n}_2] \sqsubseteq^{\mathsf{cr}}_{\hat{N}} u$.*

Proof. Again, recall $u[\hat{n}_1{=}\hat{n}_2] = \bigsqcup^{\mathsf{cr}} \{\, \alpha^{\mathsf{cr}}_{\delta}(\gamma^{\mathsf{cr}}_{\delta}(u)[\delta(\hat{n}_1){=}\delta(\hat{n}_2)]) \mid N \subseteq \mathcal{N} \wedge \delta \in \Delta(\hat{N}, N) \,\}$ for a given concrete scope N. Let $\delta_1 \in \Delta(\hat{N}, N)$ for any concrete scope

N.

$$\gamma^{\mathsf{cr}}_{\delta_1}(u)[\delta_1(\hat{n}_1){=}\delta_1(\hat{n}_2)] \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta_1}(u) \qquad \text{by Lemma 100}$$

$$\alpha^{\mathsf{cr}}_{\delta_1}(\gamma^{\mathsf{cr}}_{\delta_1}(u)[\delta_1(\hat{n}_1){=}\delta_1(\hat{n}_2)]) \sqsubseteq^{\mathsf{r}} u \qquad \text{by prop. of Galois conn.}$$

The result then follows from the glb property of the join $\sqcup^{\mathsf{cr}}$ operator (Theorem 8). □

The above result and the proof can be applied to all strengthening operations on collapsed values.

Lemma 133. *Strengthening operations on collapsed values are indeed strengthening.*

Proof. The argument closely follows that of Lemma 132. □

As expected, strengthening operations are also monotone.

Lemma 134. *Strengthening operation on collapsed values are monotone.*

Proof. The argument follows directly from the monotonicity of strengthening operations on relational values and monotonicity of the join operator $\sqcup^{\mathsf{cr}}$ on collapsed values. □

In order to prove the idempotency of a strengthening of a collapsed value with another collapsed value, we first prove the following result.

Lemma 135. *Let $\hat{N}$ and $\hat{N}_1$ be abstract scopes, N and N_1 concrete scopes, $\hat{n} \in \hat{\mathcal{N}}$, $u \in \mathcal{V}^{\mathsf{cr}}_{\hat{N}}$, $u_1 \in \mathcal{V}^{\mathsf{cr}}_{\hat{N}_1}$, $r \in \mathcal{V}^{\mathsf{r}}_{N}$, $\delta \in \Delta(\hat{N}, N)$, and $\delta_1 \in \Delta(\hat{N}_1, N_1)$. Then, $\alpha^{\mathsf{cr}}_{\delta}(r[\delta(\hat{n}) \leftarrow \gamma^{\mathsf{cr}}_{\delta_1}(u_1)]) \sqsubseteq^{\mathsf{cr}} \alpha^{\mathsf{cr}}_{\delta}(r)[\hat{n} \leftarrow u_1]$.*

Proof. We recall

$$\alpha^{\mathsf{cr}}_{\delta}(r)[\hat{n} \leftarrow u_1] = \bigsqcup\nolimits^{\mathsf{cr}} \{\, \alpha^{\mathsf{cr}}_{\delta}(\gamma^{\mathsf{cr}}_{\delta}(\alpha^{\mathsf{cr}}_{\delta}(r))[\delta_1(\hat{n}) \leftarrow \gamma^{\mathsf{cr}}_{\delta_1}(u_1)]) \mid \delta \in \Delta(\hat{N}, N), \delta_1 \in \Delta(\hat{N}_1, N_1) \,\}$$

By basic properties of Galois connections, monotonicity of joins and strengthening operations on relational values (Theorem 104), $\alpha^{\mathsf{cr}}_{\delta}(r)[\hat{n} \leftarrow u_1]$ is larger than

$$u' = \bigsqcup\nolimits^{\mathsf{cr}} \{\, \alpha^{\mathsf{cr}}_{\delta}(r[\delta_1(\hat{n}) \leftarrow \gamma^{\mathsf{cr}}_{\delta_1}(u_1)]) \mid \delta \in \Delta(\hat{N}, N), \delta_1 \in \Delta(\hat{N}_1, N_1) \,\}$$

By definition of u', we have $\alpha^{\mathsf{cr}}_{\delta}(r[\delta(\hat{n}) \leftarrow \gamma^{\mathsf{cr}}_{\delta_1}(u_1)]) \sqsubseteq^{\mathsf{cr}} u' \sqsubseteq^{\mathsf{cr}} \alpha^{\mathsf{cr}}_{\delta}(r)[\hat{n} \leftarrow u_1]$. □

We can now prove the idempotency of strengthening using a value.

Lemma 136. *Let $\hat{N}$ and $\hat{N}_1$ be abstract scopes, N and N_1 concrete scopes, $\hat{n} \in \hat{\mathcal{N}}$, $u \in \mathcal{V}^{\mathsf{cr}}_{\hat{N}}$, $u_1 \in \mathcal{V}^{\mathsf{cr}}_{\hat{N}_1}$. Then, $u[\hat{n} \leftarrow u_1] = u[\hat{n} \leftarrow u_1][\hat{n} \leftarrow u_1]$.*

Proof. We first recall

$$u[\hat{n} \leftarrow u_1] = \bigsqcup\nolimits^{\mathsf{cr}} \{\, \alpha^{\mathsf{cr}}_{\delta}(\gamma^{\mathsf{cr}}_{\delta}(u)[\delta_1(\hat{n}) \leftarrow \gamma^{\mathsf{cr}}_{\delta_1}(u_1)]) \mid \delta \in \Delta(\hat{N}, N), \delta_1 \in \Delta(\hat{N}_1, N_1) \,\}$$

By Theorem 133, we know $u[\hat{n} \leftarrow u_1][\hat{n} \leftarrow u_1] \sqsubseteq^{\mathsf{r}} u[\hat{n} \leftarrow u_1]$. We thus need to show $u[\hat{n} \leftarrow u_1] \sqsubseteq^{\mathsf{r}} u[\hat{n} \leftarrow u_1][\hat{n} \leftarrow u_1]$.

$$u[\hat{n} \leftarrow u_1] =$$

[by def of $\cdot[\cdot \leftarrow \cdot]$]

$$\bigsqcup^{\mathsf{cr}} \{\, \alpha^{\mathsf{cr}}_{\delta}(\gamma^{\mathsf{cr}}_{\delta}(u)[\delta_1(\hat{n}) \leftarrow \gamma^{\mathsf{cr}}_{\delta_1}(u_1)]) \mid \delta \in \Delta(\hat{N}, N), \delta_1 \in \Delta(\hat{N}_1, N_1) \,\} =$$

[by Lemma 103]

$$\bigsqcup^{\mathsf{cr}} \{\, \alpha^{\mathsf{cr}}_{\delta}(\gamma^{\mathsf{cr}}_{\delta}(u)[\delta_1(\hat{n}) \leftarrow \gamma^{\mathsf{cr}}_{\delta_1}(u_1)][\delta_1(\hat{n}) \leftarrow \gamma^{\mathsf{cr}}_{\delta_1}(u_1)]) \mid \delta \in \Delta(\hat{N}, N), \delta_1 \in \Delta(\hat{N}_1, N_1) \,\} \sqsubseteq^{\mathsf{r}}$$

[by Lemma 135]

$$\bigsqcup^{\mathsf{cr}} \{\, \alpha^{\mathsf{cr}}_{\delta}(\gamma^{\mathsf{cr}}_{\delta}(u)[\delta_1(\hat{n}) \leftarrow \gamma^{\mathsf{cr}}_{\delta_1}(u_1)])[\hat{n} \leftarrow u_1] \mid \delta \in \Delta(\hat{N}, N), \delta_1 \in \Delta(\hat{N}_1, N_1) \,\} \sqsubseteq^{\mathsf{r}}$$

[by glb property of $\sqcup^{\mathsf{cr}}$]

$$(\bigsqcup^{\mathsf{cr}} \{\, \alpha^{\mathsf{cr}}_{\delta}(\gamma^{\mathsf{cr}}_{\delta}(u)[\delta_1(\hat{n}) \leftarrow \gamma^{\mathsf{cr}}_{\delta_1}(u_1)]) \mid \delta \in \Delta(\hat{N}, N), \delta_1 \in \Delta(\hat{N}_1, N_1) \,\})[\hat{n} \leftarrow u_1] =$$

[by def of $\cdot[\cdot \leftarrow \cdot]$]

$$u[\hat{n} \leftarrow u_1][\hat{n} \leftarrow u_1]$$

□

A.5.1.2 Structural Strengthening

We also here provide a structural definition of strengthening on collapsed values and argue its soundness. First, whenever $z \notin \hat{N}$, then $u_{\hat{N}}\lceil z \leftarrow u \rceil \stackrel{\text{def}}{=} u$. Otherwise,

$$\begin{aligned}
u\lceil \hat{n} \leftarrow \bot^{\mathsf{cr}} \rceil &\stackrel{\text{def}}{=} \bot^{\mathsf{cr}} \\
u\lceil \hat{n} \leftarrow u' \rceil &\stackrel{\text{def}}{=} u && \text{if } u \in \{\bot^{\mathsf{cr}}, \top^{\mathsf{cr}}\} \text{ or } u' \in \{\top^{\mathsf{cr}}\} \cup \mathcal{T}^{\mathsf{cr}} \\
R^{\mathsf{cr}}_1\lceil \hat{n} \leftarrow R^{\mathsf{cr}}_2 \rceil &\stackrel{\text{def}}{=} R^{\mathsf{cr}}_1[\hat{n} \leftarrow R^{\mathsf{cr}}_2] \\
\langle z, u_i, u_o \rangle\lceil \hat{n} \leftarrow u \rceil &\stackrel{\text{def}}{=} \langle z, u_i\lceil \hat{n} \leftarrow u \rceil, u_o\lceil n \leftarrow \exists z.\, u \rceil \rangle && u \in \mathcal{R}^{\mathsf{cr}}
\end{aligned}$$

It can be shown that the structural strengthening is sound.

Lemma 137. *Structural strengthening is sound. Let* $u \in \mathcal{V}^{\mathsf{cr}}_{\hat{N}}$, $u' \in \mathcal{V}^{\mathsf{cr}}_{\hat{N}_1}$, $\hat{n} \in \hat{\mathcal{N}}$,

$r \in \mathcal{V}^{\mathsf{r}}_{N}$, $r' \in \mathcal{V}^{\mathsf{r}}_{N_1}$, $\delta \in \Delta(\hat{N}, N)$, *and* $\delta_1 \in \Delta(\hat{N}_1, N_1)$. *If* $r \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u)$ *and* $r' \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta_1}(u')$, *then* $r[\delta(\hat{n}) \leftarrow r'] \sqsubseteq^{\mathsf{r}} u\lceil \hat{n} \leftarrow u' \rceil$.

Proof. The proof is carried by structural induction on u. The proof in fact shows that structural strengthening on collapsed values overapproximates the structural strengthening on relational values. The argument then follows from Lemma 111 and Lemma 131. □

A.5.2 Propagation and Transformer

Proof (of Lemma 10)**.**

Proof. For the increasing property, the proof goes by structural induction on both collapsed value arguments of $\mathsf{prop}^{\mathsf{cr}}$ and it closely follows the proof of Theorem 26. Regarding monotonicity, the proof again goes by structural induction on the collapsed arguments of $\mathsf{prop}^{\mathsf{cr}}$, closely follows the proof of Theorem 28, and relies on Lemma 134 (introduced later). □

We next show that propagation in collapsed semantics is an abstraction of the propagation in relational semantics.

Theorem 13. *Let* $u_1 \in \mathcal{V}^{\mathsf{cr}}_{\hat{N}}$, $u_2 \in \mathcal{V}^{\mathsf{cr}}_{\hat{N}}$, $\hat{N}$ *an abstract scope,* N *a concrete scope, and* $\delta \in \Delta(\hat{N}, N)$. *Further, let* $\langle r'_1, r'_2 \rangle = \mathsf{prop}^{\mathsf{r}}(r_1, r_2)$ *where* $r_1 \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u_1)$ *and* $r_2 \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u_2)$. *Lastly, let* $\langle u'_1, u'_2 \rangle = \mathsf{prop}^{\mathsf{cr}}(u_1, u_2)$. *Then,* $r'_1 \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u'_1)$ *and* $r'_2 \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u'_2)$.

Proof. The proof is carried by structural induction on u_1 and u_2.

Base cases.

- $u_1 = \bot^{\mathsf{cr}}$.

$$u_1' = \bot^{\mathsf{cr}}, u_2' = u_2 \qquad \text{by } \mathsf{prop}^{\mathsf{cr}}$$
$$r_1 = \bot^{\mathsf{r}} \qquad \text{by } \gamma_\delta^{\mathsf{cr}}$$
$$r_1' = \bot^{\mathsf{r}}, r_2' = r_2 \qquad \text{by } \mathsf{prop}^{\mathsf{r}}$$
$$\gamma_\delta^{\mathsf{cr}}(u_1') = \bot^{\mathsf{r}}, \gamma_\delta^{\mathsf{cr}}(u_2') = r_2' \qquad \text{by } \gamma_\delta^{\mathsf{cr}}$$
$$r_1' \sqsubseteq^{\mathsf{r}} \gamma_\delta^{\mathsf{cr}}(u_1'), r_2' \sqsubseteq^{\mathsf{r}} \gamma_\delta^{\mathsf{cr}}(u_2')$$

- $u_1 \in \mathcal{R}_{\hat{N}}^{\mathsf{cr}}$ and $u_2 = \bot^{\mathsf{cr}}$.

$$u_1' = u_1, u_2' = u_1 \qquad \text{by } \mathsf{prop}^{\mathsf{cr}}$$
$$r_2' = \bot^{\mathsf{r}} \qquad \text{by } \gamma_\delta^{\mathsf{cr}}$$
$$r_1' = r_1, r_2' = r_1 \qquad \text{by } \mathsf{prop}^{\mathsf{r}}$$
$$r_1' \sqsubseteq^{\mathsf{r}} \gamma_\delta^{\mathsf{cr}}(u_1'), r_2' \sqsubseteq^{\mathsf{r}} \gamma_\delta^{\mathsf{cr}}(u_2')$$

- $u_1 \in \mathcal{T}_{\hat{N}}^{\mathsf{cr}}$ and $u_2 = \bot^{\mathsf{cr}}$.

$$u_1' = u_1, u_2' = T_{\bot^{\mathsf{cr}}}^{\mathsf{cr}} \qquad \text{by } \mathsf{prop}^{\mathsf{cr}}$$
$$r_1' = r_1, r_2' = T_{\bot^{\mathsf{r}}}^{\mathsf{r}} \vee r_2' = \bot^{\mathsf{r}} \qquad \text{by } \mathsf{prop}^{\mathsf{r}}$$
$$T_{\bot^{\mathsf{r}}}^{\mathsf{r}} = \gamma_\delta^{\mathsf{cr}}(T_{\bot^{\mathsf{cr}}}^{\mathsf{cr}}) \qquad \text{by } \gamma_\delta^{\mathsf{cr}}$$
$$r_1' \sqsubseteq^{\mathsf{r}} \gamma_\delta^{\mathsf{cr}}(u_1'), r_2' \sqsubseteq^{\mathsf{r}} \gamma_\delta^{\mathsf{cr}}(u_2')$$

- $u_1 = \top^{\mathsf{cr}}$.

$$u_1' = \top^{\mathsf{cr}}, u_2' = \top^{\mathsf{cr}} \qquad \text{by } \mathsf{prop}^{\mathsf{cr}}$$
$$\gamma_\delta^{\mathsf{cr}}(u_1') = \top^{\mathsf{cr}}, \gamma_\delta^{\mathsf{cr}}(u_2') = \top^{\mathsf{cr}} \qquad \text{by } \gamma_\delta^{\mathsf{cr}}$$
$$r_1' \sqsubseteq^{\mathsf{r}} \gamma_\delta^{\mathsf{cr}}(u_1'), r_2' \sqsubseteq^{\mathsf{r}} \gamma_\delta^{\mathsf{cr}}(u_2')$$

- $u_1 \in \mathcal{R}^{\mathsf{cr}}_{\hat{N}}$ and $u_2 = \top^{\mathsf{cr}}$.

$$u'_1 = u_1, u'_2 = u_2 \qquad \text{by } \mathsf{prop}^{\mathsf{cr}}$$
$$r'_1 = r_1 \qquad \text{by } \mathsf{prop}^{\mathsf{r}}$$
$$r'_1 \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u'_1), r'_2 \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u'_2)$$

- $u_1 \in \mathcal{T}^{\mathsf{cr}}_{\hat{N}}$ and $u_2 = \top^{\mathsf{cr}}$.

$$u'_1 = \top^{\mathsf{cr}}, u'_2 = u_2 = \top^{\mathsf{cr}} \qquad \text{by } \mathsf{prop}^{\mathsf{cr}}$$
$$\gamma^{\mathsf{cr}}_{\delta}(u'_1) = \top^{\mathsf{r}}, \gamma^{\mathsf{cr}}_{\delta}(u'_2) = \top^{\mathsf{r}} \qquad \text{by } \gamma^{\mathsf{cr}}_{\delta}$$
$$r'_1 \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u'_1), r'_2 \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u'_2)$$

- $u_1 \in \mathcal{T}^{\mathsf{cr}}_{\hat{N}}$ and $u_2 \in \mathcal{R}^{\mathsf{cr}}_{\hat{N}}$.

$$u'_1 = u_1, u'_2 = \top^{\mathsf{cr}} \qquad \text{by } \mathsf{prop}^{\mathsf{cr}}$$
$$r'_1 = r_1 \qquad \text{by } \mathsf{prop}^{\mathsf{r}}$$
$$r'_1 \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u'_1), r'_2 \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u'_2)$$

- $u_1 \in \mathcal{R}^{\mathsf{cr}}_{\hat{N}}$ and $u_2 \in \mathcal{T}^{\mathsf{cr}}_{\hat{N}}$. Similar to the previous case.

For the induction case, let $u_1 = \langle z, u_{1i}, u_{1o} \rangle$ and $u_1 = \langle z, u_{2i}, u_{2o} \rangle$. By $\gamma^{\mathsf{cr}}_{\delta}$, the only interesting case is when $r_1 \in \mathcal{T}^{\mathsf{r}}_{N}$ and $r_2 \in \mathcal{T}^{\mathsf{r}}_{N}$ per Theorem 10. For any n_{cs}, let $r_1(n_{\mathsf{cs}}) = \langle r_{1i}, r_{1o} \rangle$ and $r_2(n_{\mathsf{cs}}) = \langle r_{2i}, r_{2o} \rangle$. By $r_1 \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u_1)$ and $r_2 \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u_2)$ we have $r_{1i} \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u_{1i})$, $r_{1o} \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta[z \mapsto n_{\mathsf{cs}}]}(u_{1o})$, $r_{2i} \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u_{2i})$, and $r_{2o} \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta[z \mapsto n_{\mathsf{cs}}]}(u_{2o})$. By i.h., we have $r'_{1i} \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u'_{1i})$ and $r'_{2i} \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u'_{2i})$. By monotonicity (Lemmas 99 and 134) and Lemma 135, $r_{1o}[n_{\mathsf{cs}} \leftarrow r_{2i}] \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta[z \mapsto n_{\mathsf{cs}}]}(u_{1o}[z \leftarrow u_{2i}])$ and $r_{2o}[n_{\mathsf{cs}} \leftarrow r_{2i}] \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta[z \mapsto n_{\mathsf{cs}}]}(u_{2o}[z \leftarrow u_{2i}])$. By i.h. and monotonicity of joins, we have $r'_{1o} \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u'_{1o})$ and $r'_{2o} \sqsubseteq^{\mathsf{r}} \gamma^{\mathsf{cr}}_{\delta}(u'_{2o})$. Since the resulting relational tables r'_1 and r'_2 can be expressed as respective joins $\sqcup^{\mathsf{r}}$ of singleton relational tables $[n_{\mathsf{cs}} \mapsto \langle r'_{1i}, r'_{1o} \rangle]$ and $[n_{\mathsf{cs}} \mapsto \langle r'_{2i}, r'_{1o} \rangle]$ for every call site n_{cs}, the result follows from the fact that the join $\sqcup^{\mathsf{r}}$ over relational values is lub (Lemma 4). $\square$

Proof (of Lemma 11)**.**

Proof. The increasing property of $\mathsf{Step}^{\mathsf{cr}}_{k+1}[\![\cdot]\!]$ is obvious. The proof goes by structural induction on program expressions and it closely follows the proof of Theorem 29. The proof relies on Theorem 10 as well as Lemma 134. □

Theorem 14. *For all programs e,* $\mathbf{C}^{\mathsf{r}}[\![e]\!] \dot{\sqsubseteq}^{\mathsf{r}} \dot{\gamma}^{\mathsf{cr}}(\mathbf{C}[\![e]\!])$.

Proof. First, observe that $M^{\mathsf{r}}_{\perp_{\mathsf{r}}} \dot{\sqsubseteq}^{\mathsf{r}} \dot{\gamma}^{\mathsf{cr}}(M^{\mathsf{cr}}_{\perp_{\mathsf{cr}}})$. The remainder of the proof is carried by structural induction on e showing that if $M^{\mathsf{r}} \dot{\sqsubseteq}^{\mathsf{r}} \dot{\gamma}^{\mathsf{cr}}(M^{\mathsf{cr}})$ then $M^{\mathsf{r}}_1 \dot{\sqsubseteq}^{\mathsf{r}} \dot{\gamma}^{\mathsf{cr}}(M^{\mathsf{cr}}_1)$ where $M^{\mathsf{r}}_1 = \mathsf{step}^{\mathsf{r}}_{k+1}[\![e]\!](E)(M^{\mathsf{r}})$, $M^{\mathsf{cr}}_1 = \mathsf{step}^{\mathsf{cr}}_{k+1}[\![e]\!](\rho(E))(M^{\mathsf{cr}})$, and ρ function is raised to environments in the expected way. We focus on the lambda abstraction rule; other rules are even more trivial. For any n_{cs}, let $M^{\mathsf{r}}_{n_{\mathsf{cs}}} = \overline{M^{\mathsf{r}}}(n_{\mathsf{cs}})$. It follows in an almost straightforward fashion by Theorems 131 and 13 that $M^{\mathsf{r}}_{n_{\mathsf{cs}}} \dot{\sqsubseteq}^{\mathsf{r}} \dot{\gamma}^{\mathsf{cr}}(M^{\mathsf{cr}}_1)$. Since $\dot{\sqcup}^{\mathsf{r}}$ is a glb (by def. of $\dot{\sqcup}^{\mathsf{r}}$ and Theorem 4), $\dot{\sqcup}^{\mathsf{r}}_{n_{\mathsf{cs}} \in T^{\mathsf{r}}} \overline{M^{\mathsf{r}}}(n_{\mathsf{cs}}) \dot{\sqsubseteq}^{\mathsf{r}} \dot{\gamma}^{\mathsf{r}}(M^{\mathsf{r}}_1)$. □

We now switch to proving Lemma 12 that states that only a finite number of nodes is reachable by the collapsed semantics. Given a program e and a location ℓ in e, let $\mathsf{scope\_vars}_e(l)$ denote the sequence of program variable pairs (f, x) that are (1) bound by some lambda abstraction strictly above l in the abstract syntax tree (AST) of e and (2) ordered top-bottom based on their binding appearance in e. As an example, given a program $e = \mu f_1.\lambda x_1.(\mu f_2.\lambda x_2.e^{\ell})^{\ell_1}$, we have for instance $\mathsf{scope\_vars}_e(\ell) = \langle (f_1, x_1), (f_2, x_2) \rangle$ and $\mathsf{scope\_vars}_e(\ell_1) = \langle (f_1, x_1) \rangle$. The function $\mathsf{scope\_vars}_e$ can be simply computed based on the abstract syntax of e. Given an environment $\hat{E}$ and pair of variables (f, x), let $\hat{E}[f, x] = \hat{E}.x{:}\hat{E} \diamond x.f{:}\hat{E} \diamond f$. Given a sequence s of variable pairs $(f_1, x_1) \cdots (f_j, x_j)$, let $\mathsf{aenv}(s) = \emptyset[f_1, x_1] \cdots [f_j, x_j]$. We can thus define an abstract environment for a location ℓ in a program $e \in \lambda^d$ as

$\mathsf{aenv}_e(\ell) \stackrel{\text{def}}{=} \mathsf{aenv}(\mathsf{scope\_vars}_e(\ell))$. Given a program e, we denote its abstract range $\mathsf{arange}(e) = \{\hat{E} \diamond \ell \mid \ell \in \mathit{Loc}(e) \wedge s = \mathsf{scope\_vars}_e(l) \wedge \hat{E} = \mathsf{aenv}(s)\}$.

Proof (of Lemma 12)**.**

Proof. The proof follows from a straightforward inductive argument on e. Either $\mathbf{C}^{\mathsf{cr}}[\![e]\!](k)(\hat{n}) = M^{\mathsf{cr}}_{\top_{\mathsf{cr}}}$ or the nodes updated by $\mathsf{step}^{\mathsf{cr}}_k[\![e]\!]$ belong to the set $\mathsf{arange}(e) \cup \{\hat{n} \mid \hat{n}' \in \mathsf{arange}(e) \wedge \hat{n} \in \mathsf{rng}(\mathit{env}(\hat{n}'))\}$ that is finite since (1) there are only a finite number of locations and lambda expressions in e, (2) program variables are unique, (3) and environments are finite partial functions. □

A.6 Data Flow Refinement Type Semantics Proofs

As usual, we first prove that our definitions of refinement type meet and join are correct. A depth depth of a refinement type value is defined in the similar way as for the values of the collapsed semantics.

Proof (of Lemma 13)**.**

Proof. Similar to the proof of Lemma 4, the proof is carried by case analysis on elements of U and induction on the minimum depth thereof when U consists of refinement function types only. □

Proof (of Lemma 14 (Complete Meet Morphism of $\gamma^{\mathsf{t}}_{\hat{N}}$))**.**

Let $U \in \wp(\mathcal{V}^{\mathsf{t}}_{\hat{N}})$ and let $\hat{N}$ be an abstract scope. We need to show that $\gamma^{\mathsf{t}}(\sqcap^{\mathsf{t}}_{\hat{N}} U) = \sqcap^{\mathsf{cr}}_{t \in U} \gamma^{\mathsf{t}}(t)$.

Proof. We carry the proof by case analysis on elements of U and induction on the minimum depth thereof when U consists of tables only.

We first consider the trivial (base) cases where U is not a set of multiple refinement function types.

- $U = \emptyset$. Here, $\gamma^{\mathsf{t}}(\sqcap^{\mathsf{t}}\emptyset) = \gamma^{\mathsf{t}}(\top^{\mathsf{t}}) = \top^{\mathsf{cr}} = \sqcap^{\mathsf{cr}}\emptyset$.
- $U = \{t\}$. Trivial.
- $\{\nu \colon \cdot \mid \cdot\} \in U, T^{\mathsf{t}} \in U$. We have $\sqcap^{\mathsf{t}}\{\{\nu \colon \cdot \mid \cdot\}, T^{\mathsf{t}}\} = \bot^{\mathsf{t}}$ and since $\bot^{\mathsf{t}}$ is the bottom element, $\sqcap^{\mathsf{t}} U = \bot^{\mathsf{t}}$. Similarly, $\gamma^{\mathsf{t}}(\{\nu \colon \cdot \mid \cdot\}) \sqcap^{\mathsf{cr}} \gamma^{\mathsf{t}}(T^{\mathsf{t}}) = \bot^{\mathsf{cr}} = \gamma^{\mathsf{t}}(\bot^{\mathsf{t}})$ so by the definition of $\sqcap^{\mathsf{cr}}$ we have $\sqcap^{\mathsf{cr}}_{t \in U} \gamma^{\mathsf{t}}(t) = \bot^{\mathsf{r}}$.
- $\{\nu \colon \mathsf{b}_1 \mid \cdot\} \in U, \{\nu \colon \mathsf{b}_2 \mid \cdot\} \in U$, and $\mathsf{b}_1 \neq \mathsf{b}_2$. Similar to the previous case.

- $\exists \mathsf{b}.\, \forall t \in U.\, t = \{\nu\colon \mathsf{b} \mid \cdot\}$. Here,

$$\gamma^{\mathsf{t}}(\sqcap^{\mathsf{t}} U) = \gamma^{\mathsf{t}}(\sqcap^{\mathsf{t}}_{R^{\mathsf{t}} \in U} R^{\mathsf{t}})$$

$$= \{D^{\mathsf{cr}} \mid D^{\mathsf{cr}} \in \gamma^{\mathsf{a}}(\sqcap^{\mathsf{a}}\{a \mid \{\nu\colon \mathsf{b} \mid a\} \in U\}) \wedge D^{\mathsf{cr}}(\nu) \in \gamma_B(\mathsf{b})\}$$

[by def. of γ^{t} and $\sqcap^{\mathsf{t}}$]

$$= \bigcap_{\{\nu \mathsf{b} \mid a\} \in U} \{D^{\mathsf{cr}} \mid D^{\mathsf{cr}} \in \gamma^{\mathsf{a}}(a) \wedge D^{\mathsf{cr}}(\nu) \in \gamma_B(\mathsf{b})\}$$

[by the complete meet-morphism of γ^{a}]

$$= \bigcap_{t \in U} \gamma^{\mathsf{t}}(t) = \sqcap^{\mathsf{cr}}_{t \in U} \gamma^{\mathsf{t}}(t) \qquad \textbf{[by def. of } \gamma^{\mathsf{t}} \textbf{ and } \sqcap^{\mathsf{cr}}\textbf{]}$$

- $\top^{\mathsf{t}} \in U$. Here, $\sqcap^{\mathsf{t}} U = \sqcap^{\mathsf{t}}(U/\{\top^{\mathsf{t}}\})$ and $\sqcap^{\mathsf{cr}}_{t \in U} \gamma^{\mathsf{t}}(t) = \sqcap^{\mathsf{cr}}_{t \in U, t \neq \top^{\mathsf{t}}} \gamma^{\mathsf{t}}(t)$ since $\gamma^{\mathsf{t}}(\top^{\mathsf{t}}) = \top^{\mathsf{cr}}$. The set $U/\{\top^{\mathsf{t}}\}$ either falls into one of the above cases or consists of multiple tables, which is the case we show next.

Let $U \subseteq \mathcal{T}^{\mathsf{t}}_{\hat{N}}$ and $|U| > 1$. Let d be the minimum depth of any table in U.

$$\gamma^{\mathsf{t}}(\sqcap^{\mathsf{t}} U) = \gamma^{\mathsf{t}}(z\colon \sqcap^{\mathsf{t}}\{\pi_1(io(T^{\mathsf{t}})) \mid T^{\mathsf{t}} \in U\} \to \sqcap^{\mathsf{t}}\{\pi_2(io(T^{\mathsf{t}})) \mid T^{\mathsf{t}} \in U\})$$

[by def. of $\sqcap^{\mathsf{t}}$]

$$= \langle z, \gamma^{\mathsf{t}}(\sqcap^{\mathsf{t}}\{\pi_1(io(T^{\mathsf{t}})) \mid T^{\mathsf{t}} \in U\}), \gamma^{\mathsf{t}}(\sqcap^{\mathsf{t}}\{\pi_2(io(T^{\mathsf{t}})) \mid T^{\mathsf{t}} \in U\})\rangle$$

[by def. of γ^{t}]

$$= \langle z, \sqcap^{\mathsf{cr}}\{\gamma^{\mathsf{t}}(\pi_1(io(T^{\mathsf{t}}))) \mid T^{\mathsf{t}} \in U\}, \sqcap^{\mathsf{cr}}\{\gamma^{\mathsf{t}}(\pi_2(io(T^{\mathsf{t}}))) \mid T^{\mathsf{t}} \in U\}\rangle$$

[by i.h. on the min. depth of $\{\pi_1(io(T^{\mathsf{t}}))) \mid T^{\mathsf{t}} \in U\}$ and $\{\pi_2(io(T^{\mathsf{t}}))) \mid T^{\mathsf{t}} \in U\}$]

$$= \sqcap^{\mathsf{cr}}_{t \in U} \gamma^{\mathsf{t}}(t) \qquad \textbf{[by def. of } \sqcap^{\mathsf{cr}}\textbf{]}$$

□

A.6.1 Domain Operations

Proof (of Lemma 15)**.**

Proof. The proof goes by structural induction on the type being strengthened. The proof shows that structural strengthening on types overapproximates the structural strengthening on collapsed values. The argument then follows from Lemma 137. □

It can be also shown that the structural strengthening indeed performs a strengthening given certain assumptions.

Lemma 138. *Let the set of types $\mathcal{V}^{\mathsf{t}}_{\hat{N}}$ be instantiated with an abstract domain of type refinements $\mathcal{A}_{\hat{N}}$ with the strengthening property $\forall a \in \mathcal{A}_{\hat{N}}, a_1 \in \mathcal{A}_{\hat{N}}. a[z \leftarrow a_1] \sqsubseteq^{\mathsf{a}}_{\hat{N}} a$. Let $t \in \mathcal{V}^{\mathsf{t}}_{\hat{N}}$ and $t_1 \in \mathcal{V}^{\mathsf{t}}_{\hat{N}}$. Then, $t[z \leftarrow t_1] \sqsubseteq^{\mathsf{t}}_{\hat{N}} t$ where strengthening $\cdot[\cdot \leftarrow]$ is defined structurally as shown in Figure 3.10.*

Proof. The proof goes by structural induction on t and case analysis of t_1. For the cases where $t \in \{\bot^{\mathsf{t}}, \top^{\mathsf{t}}\}$ or $t_1 \in \{\bot^{\mathsf{t}}, \top^{\mathsf{t}}\} \cup \mathcal{T}^{\mathsf{t}}$, the argument is trivial. For the case when both t and t_1 are basic refinement types, the argument follows from the strengthening property of abstract domain elements. The case when $t \in \mathcal{T}^{\mathsf{t}}$ and $t_1 \in \mathcal{R}^{\mathsf{t}}$ follows from the i.h. □

We can also show that strengthening is monotone assuming that refinement domain strengthening and rescoping operations are monotone.

Lemma 139. *Let $t \in \mathcal{V}^{\mathsf{t}}_{\hat{N}}$, $t' \in \mathcal{V}^{\mathsf{t}}_{\hat{N}}$, $t_1 \in \mathcal{V}^{\mathsf{t}}_{\hat{N}}$, $t_2 \in \mathcal{V}^{\mathsf{t}}_{\hat{N}}$, and $z \in DVar$. Assuming that strengthening operations on refinement domain $\mathcal{A}_{\hat{N}}$ are monotone, then $t_1 \sqsubseteq^{\mathsf{t}} t_2 \wedge t \sqsubseteq^{\mathsf{t}} t' \implies t_1[z \leftarrow t] \sqsubseteq^{\mathsf{t}} t_2[z \leftarrow t']$.*

Proof. When $z \notin \hat{N}$, the argument is straightforward. The proof goes by structural induction on t_1 and t_2. But, let us first discuss the cases of t and t'. When $t = \bot^{\mathsf{t}}$, the argument is trivial. The same applies when $t \in \mathcal{T}^{\mathsf{t}}_{\hat{N}}$ or $t = \top^{\mathsf{t}}$. Whenever $t' = \top^{\mathsf{t}}$

or $t' \in \mathcal{T}^{\mathsf{t}}_{\hat{N}}$, the argument goes by Lemma 138. We can thus carry the induction argument by just considering the cases where $t, t' \in \mathcal{R}^{\mathsf{t}}$. Let $t = \{\nu\colon \mathsf{b}_0 \mid a_0\}$ and $t' = \{\nu\colon \mathsf{b}_0 \mid a_0'\}$. Let us cover the base cases first.

- $t_1 = \bot^{\mathsf{t}}$. Here, $t_1[z \leftarrow t] = \bot^{\mathsf{t}}$ and the result follows trivially.
- $t_2 = \bot^{\mathsf{t}}$. Then it must be that $t_1 = \bot^{\mathsf{t}}$ and the result follows from the previously analyzed case.
- $t_2 = \top^{\mathsf{t}}$. We have $t_2[z \leftarrow t] = \top^{\mathsf{t}}$ and the result holds trivially.
- $t_1 = \top^{\mathsf{t}}$. Then, $t_2 = \top^{\mathsf{t}}$ which is the case already covered.
- $t_1 = \{\nu\colon \mathsf{b} \mid a_1\}, t_2 = \{\nu\colon \mathsf{b} \mid a_2\}$, where $a_1 \sqsubseteq^{\mathsf{a}} a_2$. We have $t_1[z \leftarrow t] = \{\nu\colon \mathsf{b} \mid a_1[z \leftarrow a_0]\}$ and $t_2[z \leftarrow t] = \{\nu\colon \mathsf{b} \mid a_2[z \leftarrow a_0']\}$. The result then follows from the monotonicity of the refinement strengthening.

For the induction case where $t_1 \in \mathcal{T}^{\mathsf{t}}$ and $t_2 \in \mathcal{T}^{\mathsf{t}}$, the argument follows straightforwardly from the i.h. and monotonicity of the refinement domain rescoping operation. □

A.6.2 Abstract Propagation and Transformer

Proof (of Lemma 16)**.**

Proof. The increasing property follows easily by the structural induction on both type arguments of $\mathsf{prop}^{\mathsf{t}}$. Regarding monotonicity, the argument goes on structural induction on the argument to $\mathsf{prop}^{\mathsf{t}}$ by closely following that of Lemma 28 and it relies on Lemma 139. □

Proof (of Lemma 17)**.**

Proof. The increasing property follows trivially. The argument for monotonicity follows that of Lemma 29 and it relies on the monotonicity of $\mathsf{prop}^{\mathsf{t}}$ established by Lemma 16. □

Lemma 140. *For all programs e and $k \in \mathbb{N}$, either $\mathbf{C}^{\mathsf{t}}[\![e]\!](k)(\hat{n}) = M^{\mathsf{t}}_{\top_{\mathsf{t}}}$ or the set $\{\, \hat{n} \in \hat{\mathcal{N}} \mid \mathbf{C}^{\mathsf{t}}[\![e]\!](k)(\hat{n}) \neq \bot^{\mathsf{t}} \,\}$ is finite.*

Proof. The proof is the same as the proof of Lemma 12. □

A.6.3 Widenings

Lemma 141. *Let $t_1 \in \mathcal{V}^{\mathsf{t}}_{\hat{N}}$, $t_2 \in \mathcal{V}^{\mathsf{t}}_{\hat{N}}$ such that $t_1 \sqsubseteq^{\mathsf{t}}_{\hat{N}} t_2$. Then, $\mathsf{sh}(t_1) \sqsubseteq^{\mathsf{t}}_{\hat{N}} \mathsf{sh}(t_2)$.*

Proof. The proof easily follows by structural induction on both t_1 and t_2. □

Lemma 142. *Let $\triangledown^{\mathsf{a}}_{\hat{N}}$ be a widening operator for the domain of abstract refinements $\mathcal{A}_{\hat{N}}$ used to instantiate the set of types $\mathcal{V}^{\mathsf{t}}_{\hat{N}}$ and the widening operator $\triangledown^{\mathsf{ra}}_{\hat{N}}$. Also, let $t_1 \in \mathcal{V}^{\mathsf{t}}_{\hat{N}}$, $t_2 \in \mathcal{V}^{\mathsf{t}}_{\hat{N}}$, and $t = t_1 \triangledown^{\mathsf{ra}}_{\hat{N}} t_2$. Then, $\mathsf{depth}(t) \leq \mathsf{max}(\mathsf{depth}(t_1), \mathsf{depth}(t_2))$.*

Proof. The proof goes by structural induction on both t_1 and t_2 and it relies on the properties of the sh function. □

Proof (of Lemma 18)**.**

Proof. The upper bound property of $\triangledown^{\mathsf{t}}_{\hat{N}}$ directly follows from the upper bound properties of $\triangledown^{\mathsf{ra}}_{\hat{N}}$ and $\triangledown^{\mathsf{sh}}_{\hat{N}}$. Let $t_0 \sqsubseteq^{\mathsf{t}}_{\hat{N}} t_1 \sqsubseteq^{\mathsf{t}}_{\hat{N}} \ldots$ be an increasing sequence of types. Let y^i be the sequence of types defined by $y_0 = t_0$ and $y_i = y_{i-1} \triangledown^{\mathsf{t}}_{\hat{N}} t_{i-1}$ for all $i > 0$. By $\triangledown^{\mathsf{t}}_{\hat{N}}$ being an upper bound operator it follows that $y_0 \sqsubseteq^{\mathsf{t}}_{\hat{N}} y_1 \sqsubseteq^{\mathsf{t}}_{\hat{N}} \ldots$. By Lemma 141, it is then the case that $\mathsf{sh}(y_0) \sqsubseteq^{\mathsf{t}}_{\hat{N}} \mathsf{sh}(y_1) \sqsubseteq^{\mathsf{t}}_{\hat{N}} \ldots$. Now, $\triangledown^{\mathsf{ra}}_{\hat{N}}$ increases

the shape of the two argument types by being increasing and by Lemma 141, but only by introducing $\top^{\mathsf{t}}$ (Lemma 142). Hence, either $\triangledown^{\mathsf{ra}}_{\hat{N}}$ eventually produces $\top^{\mathsf{t}}$ in the sequence y^i, thus stabilizing it, or it eventually stops affecting the shape of types in y^i. In that case, the shape of y^i eventually stabilizes by the widening properties of $\triangledown^{\mathsf{sh}}_{\hat{N}}$. The whole y^i then stabilizes by the widening operator property of $\triangledown^{\mathsf{a}}_{\hat{N}}$ and the definition of $\triangledown^{\mathsf{ra}}_{\hat{N}}$. □

A.7 Liquid Types Semantics Proofs

For the rest of this section, we assume the set of Liquid types $\mathcal{V}^{\mathsf{t}}_{\hat{N}}$ is defined as described in § 3.6.

A.7.1 Domain Operations

Lemma 143. *The strengthening operations on Liquid types are sound and strengthening.*

Proof. The results follow from the soundness and strengthening properties of the refinement domain (follow from the semantics of logical conjunction), Lemma 138, and Lemma 15. □

We can also show that strengthening operations are idempotent. Here, we assume that the strengthening of the liquid types refinement domain is idempotent. This is true in practice as strengthening amounts to adding equality conjuncts.

$$a[z \leftarrow a_1] = a[z \leftarrow a_1][z \leftarrow a_1] \qquad \text{idempotency}$$

Lemma 144. *Let $t \in \mathcal{V}^{\mathsf{t}}_{\hat{N}}$, $t_1 \in \mathcal{V}^{\mathsf{t}}_{\hat{N}}$, and $z \in DVar$. Then, $t[z \leftarrow t_1] = t[z \leftarrow t_1][z \leftarrow t_1]$.*

Proof. The proof goes by the structural induction on t. First, observe that if $z \notin \hat{N}$, the argument is trivial. Otherwise, if $t_1 = \bot^{\mathsf{t}}$ then $t[z \leftarrow \bot^{\mathsf{t}}] = \bot^{\mathsf{t}} = t[z \leftarrow \bot^{\mathsf{t}}][z \leftarrow \bot^{\mathsf{t}}]$. Next, if $t_1 \in \{\top^{\mathsf{t}}\} \cup \mathcal{T}^{\mathsf{t}}$, then $t[z \leftarrow t_1] = t = t[z \leftarrow t_1][z \leftarrow t_1]$. We can thus just focus on the case where $t \in \mathcal{R}^{\mathsf{t}}$. Let $t = \{\nu \colon \mathsf{b}_1 \mid a_1\}$. We cover the base cases first.

- $t = \bot^{\mathsf{t}}$. Here, $\bot^{\mathsf{t}}[z \leftarrow t_1] = \bot^{\mathsf{t}} = \bot^{\mathsf{t}}[z \leftarrow t_1][z \leftarrow t_1]$.

- $t = \top^{\mathsf{t}}$. Similar to the above case, $\top^{\mathsf{t}}[z \leftarrow t_1] = \top^{\mathsf{t}} = \top^{\mathsf{t}}[z \leftarrow t_1][z \leftarrow t_1]$.

- $t = \{\nu : \mathsf{b} \mid a\}$. In this case, $\{\nu : \mathsf{b} \mid a\}[z \leftarrow t_1] = \{\nu : \mathsf{b} \mid a[z \leftarrow a_1]\}$ and $\{\nu : \mathsf{b} \mid a\}[z \leftarrow t_1][z \leftarrow t_1] = \{\nu : \mathsf{b} \mid a[z \leftarrow a_1]\}[z \leftarrow t_1] = \{\nu : \mathsf{b} \mid a[z \leftarrow a_1][z \leftarrow a_1]\}$. The result then follows from the idempotency of the refinement strengthening.

For the induction case where $t \in \mathcal{T}^{\mathsf{t}}$, the result holds by the i.h. □

Lemma 145. *Strengthening operations on the liquid abstract domain of refinement are idempotent.*

Proof. Follows the proof of Lemma 144. □

A.7.2 Data Propagation

We can now prove that the sutbyping between types implies fixpoint in data propagation established between those types.

Proof (of Lemma 20)**.**

Proof. The proof goes by an induction on the maximum depth of types $d = max(\mathsf{depth}(t_1), \mathsf{depth}(t_2))$. We first cover the base cases where $d = 1$. We only need to consider few cases based on the definition of $<:^q$.

- $t_1 = \{\nu : \mathsf{b} \mid a_1\}, t_2 = \{\nu : \mathsf{b} \mid a_2\}$, and $a_1 \sqsubseteq^{\mathsf{a}} a_2$. We have $\mathsf{prop}^{\mathsf{t}}(\{\nu : \mathsf{b} \mid a_1\}, \{\nu : \mathsf{b} \mid a_2\}) = \langle \{\nu : \mathsf{b} \mid a_1\}, \{\nu : \mathsf{b} \mid a_1 \sqcup^{\mathsf{a}} a_2\} \rangle = \langle \{\nu : \mathsf{b} \mid a_1\}, \{\nu : \mathsf{b} \mid a_2\} \rangle$.

Let us now consider the induction case for $d > 1$ where $t_1 \in \mathcal{T}^{\mathsf{t}}$ and $t_2 \in \mathcal{T}^{\mathsf{t}}$. Let $\mathsf{prop}^{\mathsf{t}}(t_1, t_2) = \langle t_3, t_4 \rangle$. Observe that the maximum depth of input and output types of t_1 and t_2 and their arbitrary strengthening is smaller than d by the definition of

depth and Lemma 143. Let $t_1 = z\colon t_{1i} \to t_{1o}$ and $t_2 = z\colon t_{2i} \to t_{2o}$. By definition of $<:^q$, we have that (1) $t_{2i} <:^q t_{1i}$ and (2) $t_{1o}[z \leftarrow t_{2i}] <:^q t_{2o}[z \leftarrow t_{2i}]$. By i.h., we thus have $\mathsf{prop}^{\mathsf{t}}(t_{2i}, t_{1i}) = \langle t_{2i}, t_{1i}\rangle$ and $\mathsf{prop}^{\mathsf{t}}(t_{1o}[z \leftarrow t_{2i}], t_{2o}[z \leftarrow t_{2i}]) = \langle t_{1o}[z \leftarrow t_{2i}], t_{2o}[z \leftarrow t_{2i}]\rangle$. By Lemma 143, we have $t_{2o}[z \leftarrow t_{2i}] \sqsubseteq^{\mathsf{t}} t_{2o}$ and $t_{1o}[z \leftarrow t_{2i}] \sqsubseteq^{\mathsf{t}} t_{1o}$. Therefore, by the definition of $\mathsf{prop}^{\mathsf{t}}$ it follows $t_3 = z : t_{1i} \to t_{1o} = t_1$ and $t_4 = z\colon t_{2i} \to t_{2o} = t_2$. □

A.7.3 Connection to Liquid Types

First, we define a typing relation $\vdash^{\kappa}_{\mathsf{m}}\colon \Xi^{\kappa} \times \lambda^{d} \times \mathcal{V}^{\mathsf{t}} \times \mathcal{M}^{\mathsf{t}}$ that mimics the earlier typing relation $\vdash^{\kappa}_{q}$, but also produces a type map that accumulates the results deduced in the typing derivation. To this end, given a κ and $\Gamma \in \Xi^{\kappa}$, we define $M^{\mathsf{t}}_{\kappa,\Gamma}$ to be a type map that holds $\Gamma(x)$ for all nodes $\hat{n} \in \mathsf{rng}(\kappa)$ where $\kappa(x) = \hat{n}$; otherwise, it returns $\bot^{\mathsf{t}}$.

$$\frac{x \in \mathsf{dom}(\Gamma) \qquad \Gamma(x) <:^q t \qquad t = \Gamma(x)[\nu = \kappa(x)][\Gamma]^{\kappa}}{\Gamma \vdash^{\kappa}_{\mathsf{m}} x^{\ell} : t, M^{\mathsf{t}}_{\Gamma,\kappa}[\kappa(\Gamma)\diamond\ell \mapsto t]} \qquad \frac{t = c^{\mathsf{t}}[\Gamma]^{\kappa}}{\Gamma \vdash^{\kappa}_{\mathsf{m}} c^{\ell} : t, M^{\mathsf{t}}_{\Gamma,\kappa}[\kappa(\Gamma)\diamond\ell \mapsto t]}$$

$$\frac{\Gamma \vdash^{\kappa}_{\mathsf{m}} e_1^{\ell_1} : t_1, M_1^{\mathsf{t}} \qquad \Gamma \vdash^{\kappa}_{\mathsf{m}} e_2^{\ell_2} : t_2, M_2^{\mathsf{t}} \qquad t_1 \in \mathcal{T}^{\mathsf{t}} \qquad t_1 <:^q dx(t_1)\colon t_2 \to t}{M^{\mathsf{t}}, \Gamma \vdash^{\kappa}_{\mathsf{m}} (e_1^{\ell_1} e_2^{\ell_2})^{\ell} : t, (M_1^{\mathsf{t}} \dot{\sqcup}^{\mathsf{t}} M_2^{\mathsf{t}})[\kappa(\Gamma)\diamond\ell \mapsto t]}$$

$$\frac{\Gamma_1 = \Gamma.x :t_x.f\colon t_f \qquad dx(t)\colon t_x \to t_1 <:^q t \qquad t <:^q t_f \qquad \kappa_1 = \kappa.x :\kappa(\Gamma)\diamond x.f :\kappa(\Gamma)\diamond f \qquad \Gamma_1 \vdash^{\kappa_1}_{\mathsf{m}} e_1^{\ell_1} : t_1, M_1^{\mathsf{t}}}{\Gamma \vdash^{\kappa}_{\mathsf{m}} (\mu f.\lambda x.\, e_1^{\ell_1})^{\ell} : t, M_1^{\mathsf{t}}[\kappa(\Gamma)\diamond\ell \mapsto t]}$$

$$\frac{\Gamma \vdash^{\kappa}_{\mathsf{m}} x : t_0, M_0^{\mathsf{t}} \qquad t_0 <:^q \mathit{Bool}^{\mathsf{t}} \qquad t_1 <:^q t \quad t_2 <:^q t \qquad \Gamma_1 = \Lambda x \in \mathsf{dom}(\Gamma).\, \Gamma(x)[\kappa(x) \leftarrow t_0 \sqcap^{\mathsf{t}} \mathit{true}^{\mathsf{t}}] \quad \Gamma_1 \vdash^{\kappa}_{\mathsf{m}} e_1 : t_1, M_1^{\mathsf{t}} \qquad \Gamma_2 = \Lambda x \in \mathsf{dom}(\Gamma).\, \Gamma(x)[\kappa(x) \leftarrow t_0 \sqcap^{\mathsf{t}} \mathit{false}^{\mathsf{t}}] \quad \Gamma_2 \vdash^{\kappa}_{\mathsf{m}} e_2 : t_2, M_2^{\mathsf{t}}}{\Gamma \vdash^{\kappa}_{\mathsf{m}} (x\,?\, e_1^{\ell_1} : e_2^{\ell_2})^{\ell} : t, (M_0^{\mathsf{t}} \dot{\sqcup}^{\mathsf{t}} M_1^{\mathsf{t}} \dot{\sqcup}^{\mathsf{t}} M_2^{\mathsf{t}})[\kappa(\Gamma)\diamond\ell \mapsto t]}$$

Lemma 146. *Let e be a program, κ an injective mapping from variables to abstract nodes, $t \in \mathcal{V}^{\mathsf{t}}$, and $\Gamma \in \Xi^{\kappa}$. Then, $\Gamma \vdash_{q}^{\kappa} e : t$ iff $\Gamma \vdash_{\mathsf{m}}^{\kappa} e : t, M^{\mathsf{t}}$ for some $M^{\mathsf{t}} \in \mathcal{M}^{\mathsf{t}}$.*

Proof. The proof goes by structural induction on e. The result follows directly from the fact that $\vdash_{\mathsf{m}}^{\kappa}$ simply collects a type map M^{t}, without basing the rule premises on the contents of M^{t}. □

We now show several interesting but rather obvious properties of $\vdash_{\mathsf{m}}^{\kappa}$.

Lemma 147. *Let e be a program, κ an injective mapping from variables to abstract nodes, $t \in \mathcal{V}^{\mathsf{t}}$, and $\Gamma \in \Xi^{\kappa}$. If $\Gamma \vdash_{\mathsf{m}}^{\kappa} e : t, M^{\mathsf{t}}$ for some $M^{\mathsf{t}} \in \mathcal{M}^{\mathsf{t}}$, then $M^{\mathsf{t}}(\kappa(\Gamma) \diamond \ell) = t$.*

Proof. The proof is carried trivially by structural induction on e. □

Let domain validity $\mathsf{domvalid}(\ell, p, \Gamma)$ of a type environment Γ subject to program p and a location ℓ in p holds iff $\mathsf{dom}(\Gamma)$ consists of exactly all variables bound above ℓ in p.

Lemma 148. *Let e^{ℓ} be an expression of a program p, κ and injective mapping from variables to abstract nodes, $t \in \mathcal{V}^{\mathsf{t}}$, $\Gamma \in \Xi^{\kappa}$, and $\mathsf{domvalid}(\ell, p, \Gamma)$. If $\Gamma \vdash_{\mathsf{m}}^{\kappa} e : t, M^{\mathsf{t}}$ for some $M^{\mathsf{t}} \in \mathcal{M}^{\mathsf{t}}$, then $\forall x \in \mathsf{dom}(\Gamma).\, M^{\mathsf{t}}(\kappa(x)) = \Gamma(x)$.*

Proof. The proof is carried by structural induction on e. First, we note that programs $p \in \lambda^{d}$ have unique variables. Due to this, domain validity is preserved for updates to typing environments when typing the bodies of recursive functions. Further, every typing derivation has leaves at either variable or constant expressions where the environment assumptions are directly pushed to the maps. Finally, Γ_1 and Γ_2 are smaller than Γ when analyzing conditionals, due to strengthening,

so the join of M_1^{t} and M_2^{t} with M_0^{t} ensures that the result holds since by i.h. $\forall x \in \mathsf{dom}(\Gamma).\, M_0^{\mathsf{t}}(\kappa(x)) = \Gamma(x)$. □

Lemma 149. *Let $e \in \lambda^d$, κ an injective mapping from variables to abstract nodes, $t \in \mathcal{V}^{\mathsf{t}}$, $\Gamma \in \Xi^{\kappa}$, and $M^{\mathsf{t}} \in \mathcal{M}^{\mathsf{t}}$ such that $\Gamma \vdash^{\kappa}_{\mathsf{m}} e : t, M^{\mathsf{t}}$. Then for every $\hat{n} \in \hat{\mathcal{N}}_e$, if $M^{\mathsf{t}}(\hat{n}) \neq \bot^{\mathsf{t}}$ then $loc(\hat{n}) \in Loc(e)$.*

Proof. By structural induction on e. □

Let $\mathsf{trange}(M^{\mathsf{t}}) \stackrel{\text{def}}{=} \{\hat{n} \mid M^{\mathsf{t}}(\hat{n}) \neq \bot^{\mathsf{t}}\}$ be a set of non-$\bot^{\mathsf{t}}$ nodes of M^{t}.

Lemma 150. *Let p be a program, ℓ_1 and ℓ_2 locations in p such that $Loc(p(\ell_1)) \cap Loc(p(\ell_2)) = \emptyset$, κ an injective mapping from variables to abstract nodes, $t_1 \in \mathcal{V}^{\mathsf{t}}$, $t_2 \in \mathcal{V}^{\mathsf{t}}$, $\Gamma \in \Xi^{\kappa}$, $M_1^{\mathsf{t}} \in \mathcal{M}^{\mathsf{t}}$, $M_2^{\mathsf{t}} \in \mathcal{M}^{\mathsf{t}}$, and $\mathsf{domvalid}(\ell_1, p, \Gamma)$ as well as $\mathsf{domvalid}(\ell_2, p, \Gamma)$. Suppose $\Gamma \vdash^{\kappa}_{\mathsf{m}} p(\ell_1) : t_1, M_1^{\mathsf{t}}$ and $\Gamma \vdash^{\kappa}_{\mathsf{m}} p(\ell_2) : t_2, M_2^{\mathsf{t}}$. Then, $\mathsf{trange}(M_1^{\mathsf{t}}) \cap \mathsf{trange}(M_2^{\mathsf{t}}) = \mathsf{rng}(\kappa)$.*

Proof. The argument follows from Lemma 149. Also, since p is a program, the program variables of lambda abstractions are unique. Therefore, the updates to Γ in the derivation for $p(\ell_1)$ and $p(\ell_2)$ are unique. Also, by domain validity of Γ, we have that κ and Γ are always extended (rather than being updated, i.e., some bindings being rewritten). Therefore, only the variable nodes associated with Γ, and hence in $\mathsf{rng}(\kappa)$, will and must appear in both M_1^{t} and M_2^{t}. □

Lemma 151. *Let e^{ℓ} be an expression, κ an injective mapping from variables to abstract nodes, $t \in \mathcal{V}^{\mathsf{t}}$, $\Gamma \in \Xi^{\kappa}$ a valid typing environment, $M^{\mathsf{t}} \in \mathcal{M}^{\mathsf{t}}$, and $\mathsf{domvalid}(\ell, p, \Gamma)$. Suppose $\Gamma \vdash^{\kappa}_{\mathsf{m}} e^{\ell} : t, M^{\mathsf{t}}$. Then, M^{t} is safe and $t \neq \top^{\mathsf{t}}$.*

Proof. The proof goes by structural induction on e. By Γ being valid and the definition of $<:^q$, further updates to the environment are also safe. Also, direct

updates to the maps are safe by the definition of $<:^q$. Finally, joins do not introduce errors by Lemma 150 and 148. The fact that $t \neq \top^{\mathsf{t}}$ can be established then by Lemma 147. □

We next define a typing relation $\vdash^{\kappa}_{\mathsf{c}}$: $M^{\mathsf{t}} \times \Gamma^{\kappa} \times \lambda^{d} \times t$ that checks if a given type map satisfies our Liquid typing relation. We say that two type maps M^{t}_1 and M^{t}_2

$$\frac{x \in \mathsf{dom}(\Gamma) \qquad \Gamma(x) <:^q t \qquad M^{\mathsf{t}}(\kappa(\Gamma)\diamond\ell) = t \quad t = \Gamma(x)[\nu = \kappa(x)][\Gamma]^{\kappa}}{M^{\mathsf{t}}, \Gamma \vdash^{\kappa}_{\mathsf{c}} x^{\ell} : t} \qquad \frac{t = c^{\mathsf{t}}[\Gamma]^{\kappa} \quad M^{\mathsf{t}}(\kappa(\Gamma)\diamond\ell) = t}{M^{\mathsf{t}}, \Gamma \vdash^{\kappa}_{\mathsf{c}} c^{\ell} : t}$$

$$\frac{M^{\mathsf{t}}, \Gamma \vdash^{\kappa}_{\mathsf{c}} e^{\ell_1}_1 : t_1 \qquad M^{\mathsf{t}}, \Gamma \vdash^{\kappa}_{\mathsf{c}} e^{\ell_2}_2 : t_2 \qquad M^{\mathsf{t}}(\kappa(\Gamma)\diamond\ell) = t \qquad t_1 \in \mathcal{T}^{\mathsf{t}} \qquad t_1 <:^q dx(t_1) \colon t_2 \to t}{M^{\mathsf{t}}, \Gamma \vdash^{\kappa}_{\mathsf{c}} (e^{\ell_1}_1 e^{\ell_2}_2)^{\ell} : t}$$

$$\frac{\begin{array}{c} dx(t) \colon t_x \to t_1 <:^q t \qquad t <:^q t_f \\ \Gamma_1 = \Gamma.x \colon t_x.f \colon t_f \qquad \kappa_1 = \kappa.x \colon \kappa(\Gamma)\diamond x.f \colon \kappa(\Gamma)\diamond f \qquad M^{\mathsf{t}}, \Gamma_1 \vdash^{\kappa_1}_{\mathsf{c}} e^{\ell_1} : t_1 \\ M^{\mathsf{t}}(\kappa(\Gamma)\diamond\ell) = t \quad M^{\mathsf{t}}(\kappa_1(x)) = t_x \qquad M^{\mathsf{t}}(\kappa_1(f)) = t_f \quad M^{\mathsf{t}}(\kappa_1(\Gamma_1)\diamond\ell_1) = t_1 \end{array}}{M^{\mathsf{t}}, \Gamma \vdash^{\kappa}_{\mathsf{c}} (\mu f.\lambda x.\, e^{\ell_1}_1)^{\ell} : t}$$

$$\frac{\begin{array}{c} M^{\mathsf{t}}(\kappa(\Gamma)\diamond\ell) = t \qquad M^{\mathsf{t}}, \Gamma \vdash^{\kappa}_{q} x : t_0 \quad t_0 <:^q Bool^{\mathsf{t}} \quad t_1 <:^q t \quad t_2 <:^q t \\ \Gamma_1 = \Lambda x \in \mathsf{dom}(\Gamma).\, \Gamma(x)[\kappa(x) \leftarrow t_0 \sqcap^{\mathsf{t}} true^{\mathsf{t}}] \qquad M^{\mathsf{t}}_{\Gamma_2,\kappa}, \Gamma_1 \vdash^{\kappa}_{q} e_1 : t_1 \\ \Gamma_2 = \Lambda x \in \mathsf{dom}(\Gamma).\, \Gamma(x)[\kappa(x) \leftarrow t_0 \sqcap^{\mathsf{t}} false^{\mathsf{t}}] \qquad M^{\mathsf{t}}_{\Gamma_1,\kappa}, \Gamma_2 \vdash^{\kappa}_{q} e_2 : t_2 \end{array}}{M^{\mathsf{t}}, \Gamma \vdash^{\kappa}_{q} (x^{\ell_0}\, ?\, e^{\ell_1}_1 : e^{\ell_2}_2)^{\ell} : t}$$

agree, denoted $\mathsf{tmagree}(M^{\mathsf{t}}_1, M^{\mathsf{t}}_2)$ iff $\forall \hat{n}. M^{\mathsf{t}}_1(\hat{n}) \neq \bot^{\mathsf{t}} \wedge M^{\mathsf{t}}_2(\hat{n}) \neq \bot^{\mathsf{t}} \implies M^{\mathsf{t}}_1(\hat{n}) = M^{\mathsf{t}}_2(\hat{n})$.

Lemma 152. *Let e^{ℓ} be an expression of p be a program, κ an injective mapping from variables to abstract nodes, $t \in \mathcal{V}^{\mathsf{t}}$, $\Gamma \in \Xi^{\kappa}$, $M^{\mathsf{t}} \in \mathcal{M}^{\mathsf{t}}$, $M^{\mathsf{t}}_0 \in \mathcal{M}^{\mathsf{t}}$, and $\mathsf{domvalid}(\ell, p, \Gamma)$. If $\mathsf{tmagree}(M^{\mathsf{t}}, M^{\mathsf{t}}_0)$ and $\Gamma \vdash^{\kappa}_{\mathsf{m}} e : t, M^{\mathsf{t}}$, then $M^{\mathsf{t}}_0, \Gamma \vdash^{\kappa}_{\mathsf{c}} e : t$.*

Proof. By structural induction on e.

- $e^{\ell} = c^{\ell}$. Trivial.

- $e^\ell = x^\ell$. Also straightforward.
- $e^\ell = (e_1^{\ell_1} e_2^{\ell_2})^\ell$. By Lemma 150 and 148, we have that $\mathsf{tmagree}(M_1^\mathsf{t}, M_0^\mathsf{t})$ and $\mathsf{tmagree}(M_2^\mathsf{t}, M_0^\mathsf{t})$. The result then follows from the induction hypothesis.
- $e^\ell = (\mu f.\lambda x.e_1^{\ell_1})^\ell$. Similar to the previous case.
- $e^\ell = (x^{\ell_0}\,?\,e_1^{\ell_1} : e_2^{\ell_2})^\ell$. Similar, the argument again relies on Lemma 150 and 148.

□

We can now easily show an important connection between $\vdash^\kappa_\mathsf{m}$ and $\vdash^\kappa_\mathsf{c}$.

Lemma 153. *Let e^ℓ be an expression of p be a program, κ an injective mapping from variables to abstract nodes, $t \in \mathcal{V}^\mathsf{t}$, $\Gamma \in \Xi^\kappa$, $M^\mathsf{t} \in \mathcal{M}^\mathsf{t}$, and $\mathsf{domvalid}(\ell, p, \Gamma)$. If $\Gamma \vdash^\kappa_\mathsf{m} e : t, M^\mathsf{t}$, then $M^\mathsf{t}, \Gamma \vdash^\kappa_\mathsf{c} e : t$.*

Proof. Direct corollary of Lemma 152. □

We say a triple $(\Gamma, M^\mathsf{t}, \kappa)$ *matches* if $\forall x \in \mathsf{dom}(\Gamma).\, \Gamma(x) = M^\mathsf{t}(\kappa(x))$.

Theorem 15. *Let e^ℓ be an expression, κ an injective encoding of abstract variable nodes, $\Gamma \in \Xi^\kappa$, M^t a safe type map, and $t \in \mathcal{V}^\mathsf{t}_{\hat{N}_\kappa}$. If $M^\mathsf{t}, \Gamma \vdash^\kappa_\mathsf{c} e^\ell : t$ and $(\Gamma, M^\mathsf{t}, \kappa)$ match, then $\mathsf{Step}^\mathsf{t}[\![e^\ell]\!](\kappa(\Gamma)\diamond\ell)(M^\mathsf{t}) = M^\mathsf{t}$ and $M^\mathsf{t}(\kappa(\Gamma)\diamond\ell) = t$.*

Proof. The proof goes by structural induction on e.

- $e = x^\ell$. By matching $(\Gamma, M^\mathsf{t}, \kappa)$, we have that $\kappa(\Gamma)(x) = \kappa(x)$ and $M^\mathsf{t}(\kappa(\Gamma)(x)) = \Gamma(x)$. Hence, $M^\mathsf{t}(\kappa(\Gamma)(x)) <:^q M^\mathsf{t}(\kappa(\Gamma)\diamond\ell)$. Then by the definition of $\vdash^\kappa_\mathsf{c}$, we have that $M^\mathsf{t}(\kappa(\Gamma)\diamond\ell) = M^\mathsf{t}(\kappa(\Gamma)(x))[\nu = \kappa(x)][\Gamma]^\kappa$ and hence by matching $(\Gamma, M^\mathsf{t}, \kappa)$ we have $M^\mathsf{t}(\kappa(\Gamma)\diamond\ell) = M^\mathsf{t}(\kappa(\Gamma)(x))[\nu = \kappa(\Gamma)(x)][M^\mathsf{t}]$. Therefore,

$M^{\mathsf{t}}(\kappa(\Gamma)(x))[\nu = \kappa(\Gamma)(x)] <:^{q} M^{\mathsf{t}}(\kappa(\Gamma)\diamond\ell)$ by the structural definition of strengthening. The result then follows from Lemma 20 and Lemma 145.

- $e = c^{\ell}$. The result follows from the definition of $c^{\mathsf{t}}[\Gamma]^{\kappa}$. That is, $c^{\mathsf{t}}[\Gamma]^{\kappa} = c^{\mathsf{t}}[M^{\mathsf{t}}]$ by matching $(\Gamma, M^{\mathsf{t}}, \kappa)$.

- $e = (e_1^{\ell_1} e_2^{\ell_2})^{\ell}$. By i.h., we have that $\mathsf{Step}^{\mathsf{t}}[\![e_1^{\ell_1}]\!](\kappa(\Gamma))(M^{\mathsf{t}}) = M^{\mathsf{t}}$ and $M^{\mathsf{t}}(\kappa(\Gamma)\diamond\ell_1) = t_1$ where $t_1 \in \mathcal{T}^{\mathsf{t}}$. Similarly, $\mathsf{Step}^{\mathsf{t}}[\![e_2^{\ell_2}]\!](\kappa(\Gamma))(M^{\mathsf{t}}) = M^{\mathsf{t}}$ and $M^{\mathsf{t}}(\kappa(\Gamma)\diamond\ell_2) = t_2$ where $t_2 \notin \{\bot^{\mathsf{t}}, \top^{\mathsf{t}}\}$ by the definition of $<:^{q}$. Then by the definition of $\mathsf{Step}^{\mathsf{t}}[\![\cdot]\!]$, we thus have that $M_1^{\mathsf{t}} = M_2^{\mathsf{t}} = M^{\mathsf{t}}$ and the result then follows from Lemma 20.

- $e = (\mu f.\lambda x.\, e^{\ell_1})^{\ell}$. First, by the definition of $<:^{q}$, it must be that $\pi_1(io(T^{\mathsf{t}})) \notin \{\bot^{\mathsf{t}}, \top^{\mathsf{t}}\}$. Further, by the definition of $\mathsf{Step}^{\mathsf{t}}[\![\cdot]\!]$ and Lemma 20, we have that $T^{\mathsf{t}} = T_1^{\mathsf{t}} = T_2^{\mathsf{t}}$, $t_x = t'_x$, $t_f = t'_f$, and $t_1 = t'_1[\hat{n}_x{=}dx(T^{\mathsf{t}})]$ since $dx(T^{\mathsf{t}})$ is not initially in the scope of t_1 ($\hat{n}_x$ and $dx(T^{\mathsf{t}})$ are simply used as synonym variables). Next, by $\mathsf{domvalid}(\ell, p, \Gamma)$ it follows that $(\Gamma_1, \kappa_1, M^{\mathsf{t}})$ is matching and that domain validity is preserved $\mathsf{domvalid}(\ell_1, p, \Gamma_1)$. Also, by $\mathsf{Step}^{\mathsf{t}}[\![\cdot]\!]$ we have that $\kappa(\Gamma_1) = \hat{E}_1$. Hence, $M_1^{\mathsf{t}} = M^{\mathsf{t}}$ by the induction hypothesis.

- $e = (x^{\ell_0}\, ?\, e_1^{\ell_1} : e_2^{\ell_2})^{\ell}$. First, observe that by definition of $<:^{q}$, if $t_0 <:^{q} \mathit{Bool}^{\mathsf{t}}$, then $t_0 \neq \bot^{\mathsf{t}}$ and $t_0 \sqsubseteq^{\mathsf{t}} \mathit{Bool}^{\mathsf{t}}$. Next, by i.h., we have that $M_0^{\mathsf{t}} = M^{\mathsf{t}}$ and t_0 in $\vdash_{\mathsf{c}}^{\kappa}$ and $\mathsf{Step}^{\mathsf{t}}[\![\cdot]\!]$ are the same. Now, $\mathsf{Step}^{\mathsf{t}}[\![\cdot]\!]$ strengthens the environment nodes of $\kappa(\Gamma)$ in M_0^{t}, using the node $\hat{E}\diamond\ell_0$, while the typing relation $\vdash_{\mathsf{c}}^{\kappa}$ uses $\kappa(x)$. However, by the typing rule of $\vdash_{\mathsf{c}}^{\kappa}$ for variable nodes, we know that $\Gamma(x) <:^{q} t_0$ where $t_0 = \Gamma(x)[\nu{=}\kappa(x)][M_0^{\mathsf{t}}]$. By matching $(\Gamma, \kappa, M_0^{\mathsf{t}})$ (that is implied by matching $(\Gamma, \kappa, M^{\mathsf{t}})$ and $M_0^{\mathsf{t}} = M^{\mathsf{t}}$), we have $t_0 = M_0^{\mathsf{t}}(\kappa(\Gamma)(x))[\nu{=}\kappa(\Gamma)(x)][M_0^{\mathsf{t}}]$. By the definition of the liquid abstract

domain, t_0 implies that $\kappa(\Gamma)(x) = \hat{E}\diamond\ell_0$ and thus $\kappa(x) = \hat{E}\diamond\ell_0$. Since $\hat{E}\diamond\ell_0$ is not in the scope of any node in $\hat{N}_{\hat{E}\diamond\ell_0}$, we have $M_i^{\mathsf{t}} = M_0^{\mathsf{t}}[\hat{E}\diamond\ell_0 \leftarrow \mathit{true}^{\mathsf{t}} \sqcap^{\mathsf{t}} t_0] = M_0^{\mathsf{t}}[\kappa(x) \leftarrow \mathit{true}^{\mathsf{t}} \sqcap^{\mathsf{t}} t_0] = M_{\Gamma_1,\kappa}^{\mathsf{t}}$ and $M_j^{\mathsf{t}} = M_0^{\mathsf{t}}[\hat{E}\diamond\ell_0 \leftarrow \mathit{false}^{\mathsf{t}} \sqcap^{\mathsf{t}} t_0] = M_0^{\mathsf{t}}[\kappa(x) \leftarrow \mathit{false}^{\mathsf{t}} \sqcap^{\mathsf{t}} t_0] = M_{\Gamma_2,\kappa}^{\mathsf{t}}$. Therefore, $(\Gamma_1, \kappa_1, M_i^{\mathsf{t}})$ is matching and so is $(\Gamma_2, \kappa_2, M_j^{\mathsf{t}})$. By i.h. then, we have $M_1^{\mathsf{t}} = M_i^{\mathsf{t}}$ and $M_2^{\mathsf{t}} = M_j^{\mathsf{t}}$. By Lemma 20, we thus have that $t = t' = t''$, $t_1 = t'_1$, and $t_2 = t'_2$. The result then follows from the fact that both M_i^{t} and M_j^{t}, and consequently M_1^{t} and M_2^{t}, are smaller than M_0^{t} that is equal to M^{t}.

□

Proof (of Theorem 4)**.**

Proof. Our implicit assumption is that e^ℓ is an expression of a program p such that $\mathsf{domvalid}(\ell, p, \Gamma)$. Then by Lemma 146, there exists a type map M^{t} such that $\Gamma \vdash_q^\kappa e : t$ iff $\Gamma \vdash_{\mathsf{m}}^\kappa e : t, M^{\mathsf{t}}$. Let M^{t} be such map. We know this map is safe by Lemma 151 and that it satisfies $M^{\mathsf{t}}, \Gamma \vdash_{\mathsf{c}}^\kappa e : t$. We also know by Lemma 148 that $(\Gamma, M^{\mathsf{t}}, \kappa)$ is matching. The result then follows from Theorem 15. □

Appendix B

Type Error Localization

B.1 Proof of Lemma 25

The main idea behind our proof is centered around how the set of error sources for a particular program changes if we expand some `let` usage. In particular, we show that by expanding a well-typed `let` usage the number of error sources decreases. At the same time, we show that the number of proper error sources actually increases in the same scenario. In the case of a full expansion, when for instance $\mathfrak{L} = \mathfrak{L}_p$, it follows that all error sources are in fact proper. Using this, we show that a minimum error source that is proper for some $\mathfrak{L}$ will be a minimum error source when we extend $\mathfrak{L}$ to $\mathfrak{L}_p$.

We first state and prove few lemmas that the main proofs rely on. The next lemma states that typing derivations for the same expression using the same typing environments are equivalent modulo type variables used.

Lemma 154. *Let* $\Gamma, \Pi \vdash_{\mathfrak{L}} e^{\ell} \mid \mathcal{A}_1$ *and* $\Gamma, \Pi \vdash_{\mathfrak{L}} e^{\ell} \mid \mathcal{A}_2$. *Then,* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ *are the same modulo consistent renaming of type variables not bound in* Γ *and* Π.

Proof. This follows from the fact that the only source of non-determinism in our typing rules is the choice of names for type variables. □

The purpose of the next lemma is to show that every generated set of constraints can be trivially satisfied.

Lemma 155. *Let* $\Gamma, \Pi \vdash_{\mathfrak{L}} e^{\ell} \mid \mathcal{A}$. *Then,* $\mathcal{A}$ *is of the form* $\{T_\ell \implies \mathcal{A}'\}$.

Proof. The proof goes by induction on typing derivations. By focusing on the last rule in the derivation, we can see that every rule generates a constraint with the above form. By setting T_ℓ to `false`, the generated set is then satisfied. □

We now introduce notation and definitions used in our main arguments. Given a program p, we say that typing environment Γ_1 *pointwise implies (for p)* Γ_2, written $\Gamma_1 \twoheadrightarrow_p \Gamma_2$, iff for all typing bindings $x : \forall\vec{\alpha_1}.(\mathcal{A}_1 \Rrightarrow \beta_1) \in \Gamma_1$ and $x : \forall\vec{\alpha_2}.(\mathcal{A}_2 \Rrightarrow \beta_2) \in \Gamma_2$ for the same variable x we have that $\forall\beta_1, \beta_2.\beta_1 = \beta_2 \implies (\exists\vec{\alpha_1'}.\mathcal{A}_1 \wedge \exists\vec{\alpha_2}.\mathcal{A}_2 \wedge \mathit{PDefs}(p) \implies \exists\vec{\alpha_2'}.\mathcal{A}_2)$ is valid. Here, $\vec{\alpha_1'}$ is $\vec{\alpha_1}$ without β_1 (similarly for $\vec{\alpha_2'}$). When $\Gamma_1 \twoheadrightarrow_p \Gamma_2$ and $\Gamma_2 \twoheadrightarrow_p \Gamma_1$, we simply write $\Gamma_1 \twoheadleftarrow\!\!\!\!\twoheadrightarrow_p \Gamma_2$.

We now state two facts that are important in the proofs that follow and which can be deduced from the proof of correctness of W algorithm.

- If e is a subexpression of p and $\Gamma_1 \twoheadleftarrow\!\!\!\!\twoheadrightarrow_p \Gamma_2$, then $\rho(\Gamma_1, e) = \rho(\Gamma_2, e)$.
- Let e be a subexpression of p and $\Gamma, \Pi \vdash_{\mathfrak{L}} e : \beta_1 \mid \mathcal{A}_1$ where all T and P variables in $\mathcal{A}_1$ are assumed to be `true`. Given $\rho(\Pi, e) = \forall\vec{\delta}.\tau_{prin}$ and $\mathcal{A}_2 = \{\beta_2 = \tau_{prin}\}$, then $\forall\beta_1, \beta_2.\beta_1 = \beta_2 \implies (\exists\vec{\alpha'}.\mathcal{A}_1 \wedge \exists\vec{\delta}, \beta_2.\mathcal{A}_2) \iff (\exists\vec{\delta}.\mathcal{A}_2 \wedge \exists\vec{\alpha}.\mathcal{A}_1)$ holds. Here, $\vec{\alpha}$ is a vector of all free variables in $\mathcal{A}_1$ except those bound in Γ and Π. Also, $\vec{\alpha'}$ is $\vec{\alpha}$ without β_1.

Given $\Gamma, \Pi \vdash_{\mathfrak{L}} e^{\ell} : \beta \mid \mathcal{A}$ where e is a subexpression of a program p, we define $C_{\Gamma,\Pi,p,\mathcal{A},\beta}$ to be the formula $\exists \vec{\alpha}.\mathcal{A} \wedge \mathit{PDefs}(p)$. Here, $\vec{\alpha}$ is a vector of the set $\mathit{fv}(\mathcal{A})/(\mathit{fv}(\Gamma) \cup \mathit{fv}(\Pi) \cup \{\beta\})$. When Γ, Π, p, and β are clear from the context, we simply write $C_{\mathcal{A}}$. Also for convenience, let $\mathit{pwi}_p(\Gamma_1, \Pi_2, \Gamma_2, \Pi_2)$ for a program p hold iff $\Gamma_2 \twoheadrightarrow_p \Gamma_1$, $\Gamma_2 \twoheadrightarrow_p \Pi_2$, and $\Pi_1 \twoheadleftarrow\!\!\twoheadrightarrow_p \Pi_2$.

We are now ready to prove the first main lemma. We show that the number of error sources decreases for successive expansions of well typed `let` usages.

Lemma 156. *Let e be an expression in a program p, $\mathfrak{L}$ and $\mathfrak{L}_u$ set of locations where $\mathfrak{L}_0 \subseteq \mathfrak{L} \subseteq \mathfrak{L}_u \subseteq \mathfrak{L}_p$. Also, assume $\mathfrak{L}_u = \mathfrak{L} \cup \{\ell_u\}$ such that ℓ_u is a location of a well-typed `let` variable usage in e. If $\Gamma, \Pi \vdash_{\mathfrak{L}} e^{\ell} : \alpha \mid \psi$ and $\mathit{pwi}_p(\Gamma, \Pi, \Gamma_u, \Pi_u)$, then from the typing derivation using $\mathfrak{L}$ we can create a typing derivation using $\mathfrak{L}_u$ such that the following holds:*

- $\Gamma_u, \Pi_u \vdash_{\mathfrak{L}_u} e^{\ell} : \alpha \mid \psi_u$
- $M \models C_{\psi_u} \implies M \models C_{\psi}$

Proof. We will refer to typing derivations using $\mathfrak{L}$ and $\mathfrak{L}_u$ with d and d_u, respectively, using in general subscript u for parts of d_u. We carry the proof by induction on d, focusing on the last rule in the derivation. We start by observing that if $M \models C_{\psi_u}$ where $M \not\models T_{\ell}$, then by Lemma 155 $M \models C_{\psi}$ also holds. For simplicity, we will skip this trivial analysis of such models below; we only consider case where $M \models T_{\ell}$ and assume that C_{ψ} and C_{ψ_u} are simplified accordingly.

[**A-Abs**]. Suppose $\mathit{pwi}_p(\Gamma, \Pi, \Gamma_u, \Pi_u)$. Since $x : \alpha$ binding is trivial ($x : \alpha$ is the same as $x : \forall\alpha.(\emptyset \Rrightarrow \alpha)$), we have $\mathit{pwi}_p(\Gamma.x : \alpha, \Pi.x : \alpha, \Gamma_u.x : \alpha, \Pi_u.x; \alpha)$. We then create d_u by induction hypothesis, applying the same rule, and choosing the same name for α and γ as in d by appropriate type variable renaming

(Lemma 154). Now, suppose M is a model of C_{ψ_u}. Although α and β are quantified out in C_{ψ_u}, by semantics of existential quantification there exist a valuation to α and β that satisfies the constraint when those variables are free; let those values be α' and β', respectively. Let M' then be $M[\alpha \to \alpha', \beta \to \beta']$. Then, M' is a model of $C_{\mathcal{A}_u}$ too. By induction hypothesis, it is also a model of $C_{\mathcal{A}}$, thus also being a model of C_{ψ} when α and β are made free. Then, M' is also a model of C_{ψ} and so must be M.

Similar argument can be given for the rules [A-Hole], [A-Int], and [A-Bool].

[**A-App**]. Suppose $pwi_p(\Gamma, \Pi, \Gamma_u, \Pi_u)$. Since the environments are the same for the rule assumptions, we can use the induction hypothesis. We create d_u similarly as in the previous rule. Suppose now M is a model of C_{ψ_u}. As in the previous rule, we can easily modify M into M' such that $M' \models C_{\mathcal{A}_{1_u}}, C_{\mathcal{A}_{2_u}}$; then by induction hypothesis it also must be that $M' \models C_{\mathcal{A}_1}, C_{\mathcal{A}_2}$. The result then follows from induction hypothesis on α and β as earlier.

Similar argument can be given for the rule [A-Cond].

[**A-Let-Prin**]. Suppose $pwi_p(\Gamma, \Pi, \Gamma_u, \Pi_u)$. We first apply induction hypothesis for derivation for e_1 using $\mathfrak{L}$. We create derivation using $\mathfrak{L}_u$ as before using induction hypothesis and renaming type variables where necessary. By induction hypothesis we then have $C_{\mathcal{A}_{1u}} \implies C_{\mathcal{A}_1}$. By this and semantics of existential quantification we also have that $\Gamma_u.x : \tau_{exp_u} \twoheadrightarrow_p \Gamma.x : \tau_{exp}$.

By the above first property of W algorithm, we have $\Pi.x : \tau_{prin} \twoheadleftarrow\!\!\!\twoheadrightarrow_p \Pi_u.x : \tau_{prin_u}$. Finally, we show that $\Gamma_u.x : \tau_{exp_u} \twoheadrightarrow_p \Pi_u.x : \tau_{prin_u}$. If we assume that all T and P variables of $\mathcal{A}_1$ are set to `true`, then by the *PDefs* definition P_{ℓ_1} holds, and the argument follows from the second property of W algorithm we mention above.

Otherwise, P_{ℓ_1} is set to `false` and the argument is trivial.

The proof then continues by applying the induction hypothesis on typing derivation for e_2. The final argument is made as in the previous rules using induction hypothesis on $\mathcal{A}_{2_u}$ and FOL substitution lemma together with induction hypothesis on $\mathcal{A}_{1_u}[\vec{\beta}/\vec{\alpha}]$.

Similar argument can be given for the rule [A-Let-Exp].

[**A-Var-Prin**]. We only consider the case when $\ell \neq \ell_u$. The argument is similar for the other case. Suppose $pwi_p(\Gamma, \Pi, \Gamma_u, \Pi_u)$. We then create d_u in the usual way also making sure it ends in rule [A-Var-Exp], since $\ell \in \mathfrak{L}_u$. Now, let M be a model of C_{ψ_u}. The result then directly follows from the fact that $\Gamma_u \twoheadrightarrow_p \Pi_u$ and $\Pi_u \twoheadrightarrow_p \Pi$: by definition of pointwise implication it is also the case that $\Gamma_u \twoheadrightarrow_p \Pi$.

Similar argument can be given for the rule [A-Var-Exp].

□

To avoid clutter, we now change a bit the definition of pointwise implication, needed for the remainder of the proof, instead of redefining it from scratch. The definition again states the same, except the case where $\mathcal{A}_1$ is of the form $\{P_\ell \implies \beta_1 = \tau\}$. Then we require that $\forall \beta_1, \beta_2 . \beta_1 = \beta_2 \implies (\exists \vec{\alpha_1'} . \mathcal{A}_1 \wedge \exists \vec{\alpha_2} . \mathcal{A}_2 \wedge PDefs(p) \implies (P_\ell \implies \exists \vec{\alpha_2'} . \mathcal{A}_2))$. Also, we redefine $pwi_p(\Gamma_1, \Pi_1, \Gamma_2, \Pi_2)$ to hold for a program p iff $\Gamma_2 \twoheadrightarrow_p \Gamma_1$, $\Pi_1 \twoheadrightarrow_p \Gamma_1$, and $\Pi_1 \twoheadleftarrow\!\!\!\twoheadrightarrow_p \Pi_2$.

Let M be an FOL model and Γ a typing environment. We write $M \models \Gamma$ if $M \models \exists \vec{\alpha} . \mathcal{A}$ for every $x : \forall \vec{\alpha} . (\mathcal{A} \Rrightarrow \beta) \in \Gamma$. We say $ms(M, \Gamma, \Pi, \Gamma', \Pi')$ iff $M \models \Gamma$, $M \models \Pi$, $M \models \Gamma'$, and $M \models \Pi'$.

We now show that every proper error source is still an error source after we

perform an expansion of a well-typed `let` variable. Argued later, such an error source is also proper after the expansion.

Lemma 157. *Let e be an expression in a program p and $\mathfrak{L}$ and $\mathfrak{L}_u$ set of locations where $\mathfrak{L}_0 \subseteq \mathfrak{L} \subseteq \mathfrak{L}_u \subseteq \mathfrak{L}_p$. Also, let $\mathfrak{L}_u = \mathfrak{L} \cup \{\ell_u\}$ where ℓ_u is a location of a well-typed `let` variable usage. If $\Gamma, \Pi \vdash_{\mathfrak{L}} e : \alpha \mid \psi$, M is an FOL model such that $M \models P_{dloc(\ell_u)}$, $ms(M, \Gamma, \Pi, \Gamma', \Pi')$, and $pwi_p(\Gamma', \Pi', \Gamma, \Pi)$, then from the typing derivation using $\mathfrak{L}$ we can create a typing derivation using $\mathfrak{L}_u$ such that:*

- $\Gamma_u, \Pi_u \vdash_{\mathfrak{L}_u} e : \alpha \mid \psi_u$
- $M \models C_\psi \implies M \models C_{\psi_u}$

Proof. We borrow the notation from the proof of the previous lemma. We also carry the proof using the same technique. Likewise, we skip the trivial analysis when $M \not\models T_\ell$.

[A-Abs]. Suppose $ms(M, \Gamma, \Pi, \Gamma_u, \Pi_u)$ and $pwi_p(\Gamma_u, \Pi_u, \Gamma, \Pi)$. Assume $M \models C_\psi$ and $M \models P_{dloc(\ell_u)}$. Since $x : \alpha$ binding is trivial, we have $ms(M, \Gamma.x : \alpha, \Pi.x : \alpha, \Gamma_u.x : \alpha, \Pi_u.x : \alpha)$ and $pwi_p(\Gamma_u.x : \alpha, \Pi_u.x : \alpha, \Gamma.x : \alpha, \Pi.x : \alpha)$. The argument is then same as for the same rule in Lemma 156.

Similar argument can be given for the rules [A-Hole], [A-Int], and [A-Bool].

[A-App]. Suppose $ms(M, \Gamma, \Pi, \Gamma_u, \Pi_u)$ and $pwi_p(\Gamma_u, \Pi_u, \Gamma, \Pi)$. Assume $M \models C_\psi$ and $M \models P_{dloc(\ell_u)}$. Since the environments are the same for the rule assumptions, we can use the induction hypothesis. We proceed by using the same argument as for the same rule in Lemma 156.

Similar argument can be given for the rule [A-Cond].

[**A-Let-Prin**]. Suppose $ms(M, \Gamma, \Pi, \Gamma_u, \Pi_u)$, $pwi_p(\Gamma_u, \Pi_u, \Gamma, \Pi)$, and M is a model of C_ψ where $M \models P_{dloc(\ell_u)}$. We first apply induction hypothesis for derivation for e_1 using $\mathfrak{L}$. We create derivation using $\mathfrak{L}_u$ as before using induction hypothesis renaming type variables when necessary. By induction hypothesis we then have $C_{\mathcal{A}_1} \implies C_{\mathcal{A}_{1u}}$. By this and semantics of existential quantification we also have that $\Gamma.x : \tau_{exp} \twoheadrightarrow_p \Gamma_u.x : \tau_{exp_u}$.

By the above first property of W algorithm, we have $\Pi.x : \tau_{prin} \twoheadleftarrow\!\!\!\twoheadrightarrow_p \Pi_u.x : \tau_{prin_u}$. Finally, we show that $\Pi.x : \tau_{prin} \twoheadrightarrow_p \Gamma.x : \tau_{exp}$. We first note that if P_{ℓ_1} is set to `false`, the statement trivially follows. Otherwise, the statement follows from the properties of correctness of W algorithm.

The fact that $ms(M, \Gamma.x : \tau_{exp}, \Pi.x : \tau_{prin}, \Gamma_u.x : \tau_{exp_u}, \Pi_u.x : \tau_{prin_u})$ follows from the fact that $M \models \mathcal{A}_1[\vec{\beta}/\vec{\alpha}]$ $(\mathcal{A}_{1_u}[\vec{\beta}/\vec{\alpha}])$, $C_{\mathcal{A}_1} \implies C_{\mathcal{A}_{1u}}$, and FOL substitution lemma.

Again, the argument then similarly proceeds as in Lemma 156. Similar proof can be given for the rule [A-Let-Exp].

[**A-Var-Prin**]. We only consider the case when $\ell \neq \ell_u$. The argument is similar for the other case. We create d_u in the usual way also making sure it ends in rule [A-Var-Exp], since $\ell \in \mathfrak{L}_u$. Suppose $ms(M, \Gamma, \Pi, \Gamma_u, \Pi_u)$, $pwi_p(\Gamma_u, \Pi_u, \Gamma, \Pi)$, and M is a model of C_ψ where $M \models P_{dloc(\ell_u)}$. First, P_{ℓ_1} in ψ, coming from $\mathcal{A}[\vec{\beta}/\vec{\alpha}]$, is in fact $P_{dloc(\ell_u)}$ which is set to `true` by assumption. Using this, the argument follows from $\Pi \twoheadrightarrow_p \Gamma_u$, which transitively holds from $\Pi \twoheadrightarrow_p \Gamma$ and $\Gamma \twoheadrightarrow_p \Gamma_u$.

□

Lemma 158. *Let p be a program, R a cost function, $\mathfrak{L}$ and $\mathfrak{L}_u$ set of locations where $\mathfrak{L}_0 \subseteq \mathfrak{L} \subseteq \mathfrak{L}_u \subseteq \mathfrak{L}_p$ such that $\mathfrak{L}_u = \mathfrak{L} \cup \{\ell_u\}$ and $\ell_u \in \mathit{Scope}(p, \mathfrak{L})$, where ℓ_u is a location of a well-typed* `let` *variable usage. If $M = \textsc{Solve}(p, R, \mathfrak{L})$ where L_M is minimum and proper, then $M = \textsc{Solve}(p, R, \mathfrak{L}_u)$ and L_M is minimum and proper again.*

Proof. We first note that Solve returns solutions for $\mathcal{A}$ where $\emptyset, \emptyset \vdash_{\mathfrak{L}} e : \alpha \mid \mathcal{A}$. Then, by Lemma 156, Lemma 154, and semantics of existential quantification, we have that the number of error sources reduces or stays the same when instead of using $\mathfrak{L}$ we use $\mathfrak{L}_u$. Therefore, since M is minimum among all models before expansion, it will also be a minimum model after the expansion as R is fixed. Now, since $\ell_u \notin \mathfrak{L}$ and M is proper for $\mathfrak{L}$, we have that $\ell_u \notin \mathit{Usages}(p, \mathfrak{L}, M)$. Given our assumption that $\ell_u \in \mathit{Scope}(p, \mathfrak{L})$, it must be the that $M \models P_{\mathit{dloc}(\ell_u)}$. Then we have by Lemma 157 that proper error sources are still error sources when instead of $\mathfrak{L}$ we use $\mathfrak{L}_u$. Also by the definition of *PDefs* and *Scope*, we have that $\mathit{Usages}(p, \mathfrak{L}, M) = \mathit{Usages}(p, \mathfrak{L}_u, M)$. We hence have that L_M is also proper. □

We can finally prove Lemma 25.

Proof. From Lemma 24, we know it suffices to use $\mathfrak{L}_p$ in order for the generated constraints to have the minimum error source as the correct solution. We also know $\mathfrak{D} = \mathfrak{L}_p \setminus \mathfrak{L}$ consists of locations corresponding to usages of well-typed `let` definitions. By the definition of *Scope* and *PDefs*, this set can be ordered in a list so that each consecutive usage location is in the scope if we expand one by one from the beginning all usage locations appearing earlier in the list. For each such consecutive expansion inductively, Lemma 158 guarantees that the proper minimum error source before the expansion is still a proper minimum error source

after the expansion. After expanding all usages, our expansion location set $\mathfrak{L}$ is exactly $\mathfrak{L}_p$, guaranteeing the correct solution since in that case all error sources are in fact proper. □

Bibliography

[1] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *FPCA*, pages 31–41. ACM, 1993.

[2] A. W. Appel and D. A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001.

[3] V. Arceri, M. D. Preda, R. Giacobazzi, and I. Mastroeni. SEA: string executability analysis by abstract interpretation. *CoRR*, abs/1702.02406, 2017.

[4] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 203–213, 2001.

[5] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, pages 171–177. Springer-Verlag, 2011.

[6] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. *Satisfiability Modulo Theories*, chapter 26, pages 825–885. Volume 185 of Biere et al. [9], February 2009.

[7] C. Barrett, I. Shikanian, and C. Tinelli. An abstract decision procedure for a theory of inductive data types. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:21–46, 2007.

[8] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard – version 2.0. In *SMT*, 2010.

[9] A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009.

[10] N. Bjørner and A. Phan. νZ: Maximal Satisfaction with Z3. In *SCSS*, 2014.

[11] N. Bjørner, A. Phan, and L. Fleckenstein. νZ: An Optimizing SMT Solver. In *TACAS*, 2015.

[12] N. E. Boustani and J. Hage. Improving type error messages for generic java. *Higher-Order and Symbolic Computation*, 24(1-2):3–39, 2011.

[13] S. Chen and M. Erwig. Counter-factual typing for debugging type errors. In *POPL*, pages 583–594. ACM, 2014.

[14] O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *ICFP*, ICFP '01, pages 193–204. ACM, 2001.

[15] P. Cousot. Types as abstract interpretations. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 316–331. ACM, 1997.

[16] P. Cousot and R. Cousot. 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, pages 238–252. ACM, 1977.

[17] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282. ACM, 1979.

[18] P. Cousot and R. Cousot. Invited talk: Higher order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection, and PER analysis. In *Proceedings of the IEEE Computer Society 1994 International Conference on Computer Languages, May 16-19, 1994, Toulouse, France*, pages 95–112, 1994.

[19] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96, 1978.

[20] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212. ACM, 1982.

[21] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340. Springer-Verlag, 2008.

[22] D. Duggan and F. Bent. Explaining type inference. In *Science of Computer Programming*, pages 37–83, 1995.

[23] B. Dutertre and L. de Moura. The Yices SMT solver. Technical report, SRI International, 2006.

[24] EasyOCaml. http://easyocaml.forge.ocamlcore.org. [Online; accessed 10-March-2014].

[25] J. Eremondi, W. Swierstra, and J. Hage. A framework for improving error messages in dependently-typed languages. *Open Computer Science*, 9(1):1–32, 2019.

[26] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001.

[27] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, pages 191–202, 2002.

[28] T. S. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 268–277, 1991.

[29] R. Garcia, A. M. Clark, and É. Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 429–442, 2016.

[30] H. Gast. Explaining ML type errors by data flows. In *Implementation and Application of Functional Languages*, pages 72–89. Springer, 2005.

[31] R. Gori and G. Levi. An experiment in type inference and verification by abstract interpretation. In *Verification, Model Checking, and Abstract Interpretation, Third International Workshop, VMCAI 2002, Venice, Italy, January 21-22, 2002, Revised Papers*, pages 225–239, 2002.

[32] R. Gori and G. Levi. Properties of a type abstract interpreter. In *Verification, Model Checking, and Abstract Interpretation, 4th International Conference, VMCAI 2003, New York, NY, USA, January 9-11, 2002, Proceedings*, pages 132–145, 2003.

[33] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, pages 189–224, 2004.

[34] J. Hage. *Language Implementation Patterns: Create your own Domain-Specific and General Programming Languages*, by terence parr, pragmatic bookshelf, http://www.pragprog.com, ISBN 9781934356456. *J. Funct. Program.*, 21(2):215–217, 2011.

[35] J. Hage and B. Heeren. Heuristics for type error discovery and recovery. In *Implementation and Application of Functional Languages, 18th International Symp osium, IFL 2006, Budapest, Hungary, September 4-6, 2006, Revised Selected Papers*, pages 199–216, 2006.

[36] J. Hage and B. Heeren. Strategies for solving constraints in type and effect systems. *Electr. Notes Theor. Comput. Sci.*, 236:163–183, 2009.

[37] R. Harper. Constructing type systems over an operational semantics. *J. Symb. Comput.*, 14(1):71–84, 1992.

[38] The Haskell Programming Language. http://www.haskell.org/. [Online; accessed 15-March-2014].

[39] M. Hassan, C. Urban, M. Eilers, and P. Müller. Maxsmt-based type inference for python 3. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II*, pages 12–19, 2018.

[40] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:2960, 1969.

[41] S. Jagannathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 393–407. ACM, 1995.

[42] S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for javascript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings*, pages 238–255, 2009.

[43] R. Jhala, R. Majumdar, and A. Rybalchenko. HMC: verifying functional programs using abstract interpreters. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 470–485, 2011.

[44] N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. *Theor. Comput. Sci.*, 375(1-3):120–136, Apr. 2007.

[45] N. D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*, pages 296–306, 1986.

[46] N. D. Jones and M. Rosendahl. Higher-order minimal function graphs. *Journal of Functional and Logic Programming*, 1997(2), 1997.

[47] M. Jose and R. Majumdar. Bug-Assist: Assisting Fault Localization in ANSI-C Programs. In *CAV*, pages 504–509. Springer-Verlag, 2011.

[48] M. Kazerounian, N. Vazou, A. Bourgerie, J. S. Foster, and E. Torlak. Refinement types for ruby. In *Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9, 2018, Proceedings*, pages 269–290, 2018.

[49] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. ML Typability is DEXTIME-Complete. In *CAAP*, pages 206–220, 1990.

[50] S. Kim and K. Choe. String analysis as an abstract interpretation. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pages 294–308, 2011.

[51] K. Knowles and C. Flanagan. Type reconstruction for general refinement types. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 505–519, 2007.

[52] S. K. Lahiri and S. Qadeer. Complexity and algorithms for monomial and clausal predicate abstraction. In *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 214–229. Springer, 2009.

[53] N. Lehmann and É. Tanter. Gradual refinement types. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 775–788, 2017.

[54] B. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *PLDI*. ACM Press, 2007.

[55] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. OCaml Manual: Module Pervasives. `http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html`. [Online; accessed 14-March-2014].

[56] C. M. Li and F. Manyà. *MaxSAT, Hard and Soft Constraints*, chapter 19, pages 613–631. Volume 185 of Biere et al. [9], February 2009.

[57] H. G. Mairson. Deciding ML Typability is Complete for Deterministic Exponential Time. In *POPL*, pages 382–401, 1990.

[58] J. Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3):10:1–10:33, 2012.

[59] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[60] B. Monsuez. Polymorphic typing by abstract interpretation. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 217–228. Springer, 1992.

[61] B. Monsuez. Polymorphic types and widening operators. In *Static Analysis*, pages 267–281. Springer, 1993.

[62] B. Monsuez. Polymorphic typing for call-by-name semantics. In *Formal Methods in Programming and Their Applications*, pages 156–169. Springer, 1993.

[63] B. Monsuez. System f and abstract interpretation. In *International Static Analysis Symposium*, pages 279–295. Springer, 1995.

[64] B. Monsuez. Using abstract interpretation to define a strictness type inference system. In *Proceedings of the 1995 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 122–133. ACM, 1995.

[65] C. Mossin. Higher-order value flow graphs. *Nord. J. Comput.*, 5(3):214–234, 1998.

[66] N. Narodytska and F. Bacchus. Maximum satisfiability using core-guided maxsat resolution. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[67] M. Neubauer and P. Thiemann. Discriminative sum types locate the source of type errors. In *ICFP*, pages 15–26. ACM Press, 2003.

[68] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

[69] H. R. Nielson and F. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 332–345, 1997.

[70] OCaml. http://ocaml.org. [Online; accessed 2-February-2014].

[71] M. Odersky, M. Sulzmann, and M. Wehr. Type inference with constrained types. *TAPOS*, 5(1):35–55, 1999.

[72] Z. Pavlinovic, T. King, and T. Wies. Finding minimum type error sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 525–542, 2014.

[73] Z. Pavlinovic, T. King, and T. Wies. Practical smt-based type error localization. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 412–423, 2015.

[74] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[75] R. Piskac, T. Wies, and D. Zufferey. Grasshopper. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 124–139. Springer, 2014.

[76] J. Plevyak and A. A. Chien. Iterative flow analysis, 1995.

[77] V. Rahli, J. B. Wells, J. Pirie, and F. Kamareddine. Skalpel: A type error slicer for standard ML. *Electr. Notes Theor. Comput. Sci.*, 312:197–213, 2015.

[78] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169, 2008.

[79] P. M. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 131–144, 2010.

[80] D. S. Scott. Data types as lattices. *SIAM J. Comput.*, 5(3):522–587, 1976.

[81] E. L. Seidel, H. Sibghat, K. Chaudhuri, W. Weimer, and R. Jhala. Learning to blame: localizing novice type errors with data-driven diagnosis. *PACMPL*, 1(OOPSLA):60:1–60:27, 2017.

[82] SHErrLoc. http://www.cs.cornell.edu/projects/sherrloc/. [Online; accessed 22-April-2015].

[83] G. Singh, M. Püschel, and M. T. Vechev. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 46–59, 2017.

[84] P. J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in haskell. In *Haskell*, pages 72–83. ACM, 2003.

[85] P. J. Stuckey, M. Sulzmann, and J. Wazny. Improving type error diagnosis. In *ACM SIGPLAN Workshop on Haskell*, pages 80–91. ACM, 2004.

[86] M. Sulzmann. An overview of the chameleon system. In *The Third Asian Workshop on Programming Languages and Systems, APLAS'02, Shanghai Jiao Tong University, Shanghai, China, November 29 - December 1, 2002, Proceedings*, pages 16–30, 2002.

[87] M. Sulzmann, M. Müller, and C. Zenger. Hindley/Milner style type systems in constraint form. *Res. Rep. ACRC-99-009, University of South Australia, School of Computer and Information Science. July*, 1999.

[88] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5, 06 1955.

[89] F. Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Trans. Softw. Eng. Methodol.*, pages 5–55, 2001.

[90] N. Vazou, A. Bakst, and R. Jhala. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 48–61, 2015.

[91] N. Vazou, P. M. Rondon, and R. Jhala. Abstract refinement types. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 209–228, 2013.

[92] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. P. Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 269–282, 2014.

[93] P. Vekris, B. Cosman, and R. Jhala. Refinement types for typescript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 310–325, 2016.

[94] M. Wand. Finding the source of type errors. In *POPL*, pages 38–43. ACM, 1986.

[95] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 214–227, 1999.

[96] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227. ACM, 1999.

[97] D. Zhang and A. C. Myers. Toward general diagnosis of static errors. In *POPL*, pages 569–581. ACM, 2014.

[98] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A hardware design language for timing-sensitive information-flow security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Lan-*

guages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015, pages 503–516, 2015.

[99] H. Zhu and S. Jagannathan. Compositional and lightweight dependent type inference for ML. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, pages 295–314, 2013.