Lightweight Approaches to the Verification of Functional Programs

Eddie Jones

A dissertation submitted to the University of Bristol in accordance with the requirements for award of the degree of PhD in Computer Science in the Faculty of Engineering.

September 2023

Word Count: 44911

Abstract

The constraints of pure functional programs are often applauded for the resulting safety and correctness guarantees. It is also claimed that these programs are easier to reason about and, therefore, verify. Despite being taken as fact within the community, the availability of effective verification tools tells a different story. This thesis focuses on two verification problems specific to functional programs — pattern-match safety and functional correctness. We develop two automated, lightweight verification tools with a focus on performance.

The first problem is to verify that a given functional program does not crash due to inexhaustive pattern-matching expressions in a function's definition. To this end, we present a refinement type system with a restricted form of structural subtyping and environment-level intersection. We describe a fully automated, sound and complete type inference procedure for this system which, under reasonable assumptions, is worst-case linear-time in the size of the program. Compositionality is essential to obtaining this complexity guarantee but is only enabled by the novel restriction we place on refinement types.

Other than expressive type systems, pure functional programs naturally lend themselves to equational specifications. These specifications are a desirable target for an automated verification tool because they are immediately accessible to the average programmer. Nevertheless, such a tool must tackle the thorny issue of proof by induction when verifying recursive programs over algebraic datatypes. We propose a new cyclic proof system that is well-adapted to equational reasoning over inductively defined datatypes. The key to our system is the way in which cyclic proofs and equational reasoning are mediated through the use of contextual substitution as a cut-like rule. We outline a performant proof search algorithm that relies on a number of supporting theoretical developments, including an alternative, incremental technique for checking the correctness of a candidate proof.

Acknowledgements

This thesis would not be possible without the dedicated support of my wonderful supervisors — Steven Ramsay and Luke Ong. I am grateful for their inexhaustive expertise and guidance through technical quagmires, but also to Steven, in particular, for introducing me to the world of academia and providing the perfect environment in which to grow. I would also like to acknowledge the Engineering and Physical Sciences Research Council and The National Cyber Security Centre via the UK Research Institute in Verified Trustworthy Software Systems for funding for my studentship, of which this thesis is the ultimate product.

My interested in this research area ultimately originates with the programming language seminar series at the University of Bristol, which I was encouraged to join by then-lecturer Nicolas Wu. I owe a great deal of thanks to every member of the group for creating a stimulating research environment. But I particularly want to convey my appreciation to Matthew Pickering and Minh Nguyen for their neighbourly attitudes, making me feel at home in our shared office, and Sam Frohlich for her unabated positivity.

Finally, it would be amiss if I were not to attempt to express my never-ending gratitude to the family and friends who have been subjected, by proxy, to the highs and lows of research. Despite the odd emotional maelstrom, I have been lucky enough to receive their unwavering support and kindness that anchored me and provided me with the wind I needed to complete my studentship.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

Eddie Jones
1st March 2024

Contents

1 Introduction **1**
- 1.1 Lightweight Verification . . . 1
- 1.2 Refinement Type Systems . . . 2
- 1.3 Inductive Equational Reasoning . . . 4
- 1.4 Thesis Outline and Contributions . . . 6

2 Preliminaries **9**
- 2.1 Syntax . . . 9
  - 2.1.1 Types and Type Schemes . . . 9
  - 2.1.2 Expressions . . . 10
  - 2.1.3 Typing Judgement . . . 12
- 2.2 Operational Semantics . . . 14
  - 2.2.1 Incremental Matching . . . 15
  - 2.2.2 Contextual Equivalence . . . 18
  - 2.2.3 An Aside on the Issue of Semantics . . . 20

3 Intensional Datatypes **21**
- 3.1 Introduction . . . 21
  - 3.1.1 Intensional Datatype Refinements . . . 22
  - 3.1.2 Compositionality and Complexity . . . 24
- 3.2 Declarative System . . . 26
  - 3.2.1 Intensional Datatype Environment . . . 26
  - 3.2.2 Subtyping Relation . . . 29
  - 3.2.3 Refinement Type System . . . 31
- 3.3 Algorithmic System . . . 34
  - 3.3.1 Constructor Set Constraints . . . 34
  - 3.3.2 Type Inference System . . . 36
- 3.4 Solving Constraints . . . 42
  - 3.4.1 Saturation and Restriction . . . 43
  - 3.4.2 Complexity Analysis . . . 47
- 3.5 Implementation . . . 50
  - 3.5.1 Performance . . . 50
  - 3.5.2 Limitations and Future Work . . . 52

4 CycleQ **55**
- 4.1 Introduction . . . 55
  - 4.1.1 Cyclic Proofs . . . 57
  - 4.1.2 Cyclic Equational Reasoning . . . 58

4.2 The CycleQ Proof System . . . 60
4.2.1 Cyclic Pre-proofs . . . 61
4.2.2 Global Soundness . . . 64
4.3 Rewriting Induction . . . 68
4.3.1 Inductionless and Implicit Induction . . . 68
4.3.2 Translation into CycleQ . . . 70
4.4 Efficient Proof Search . . . 75
4.4.1 Needed Sub-expressions . . . 75
4.4.2 Refining Cycle Formation . . . 77
4.4.3 Verifying Cycles . . . 79
4.5 Implementation . . . 85
4.5.1 Performance . . . 86
4.5.2 Limitations . . . 88

5 CycleQ$^{\Rightarrow}$ 90
5.1 Introduction . . . 90
5.2 Working with Hypotheses . . . 96
5.2.1 Conditional Equations . . . 97
5.2.2 Hypothetical Reduction . . . 98
5.2.3 Hypothesis Completion . . . 99
5.2.4 Solving Hypotheses . . . 106
5.3 Extended Proof System . . . 111
5.3.1 Cyclic Pre-proofs . . . 112
5.3.2 Local and Global Soundness . . . 115
5.4 Evaluation . . . 116

6 Conclusion 120

Appendix 134
A Proofs for Section 2.2 (Operational Semantics) . . . 134
B Proofs for Section 3.2 (Declarative System) . . . 139
C Proofs for Section 3.3 (Algorithmic System) . . . 141
D Proofs for Section 3.4 (Solving Constraints) . . . 148
E Proofs for Section 4.2 (The CycleQ Proof System) . . . 149
F Proofs for Section 4.3 (Rewriting Induction) . . . 151
G Proofs for Section 4.4 (Efficient Proof Search) . . . 155
H Proofs for Section 4.5 (Implementation) . . . 156
I Proofs for Section 5.2 (Working with Hypotheses) . . . 157
J Proofs for Section 5.3 (Extended Proof System) . . . 163

Chapter 1

Introduction

Functional programming is a paradigm with rigorous guiding principles. Its declarative style and a high-level of abstraction often results in code that is more concise, readable, and easier to maintain [1]. In particular, the prioritisation of immutable data and pure functions, i.e. those without side effects, eradicate most bugs related to memory management — some of the most prolific issues in industrial code bases [2]. The ensuing referential transparency also enables substitution and simplification principles that contribute to the comprehensibility of programs.

Nevertheless, these programs are not without faults. While testing is prevalent, it is fundamentally limited to detecting violations of, but not formally verifying, a limited set of properties. It is also worth noting that traditional approaches to testing, e.g. unit testing, can require a not-insignificant amount of additional work from the programmer, accounting for 50% of their time in some cases [3]. Although initially confined to academia, functional programming is becoming increasingly prevalent in a variety of safety-critical industries [4, 5, 6] as well as those processing highly-structured data [7]. Thus, developing practical tools that can automatically, formally verify the correctness of such programs is increasingly important. In this thesis, we develop two verification tools targeting functional programs and evaluate their effectiveness.

1.1 Lightweight Verification

In contrast to testing, formal verification aims to prove that a given program satisfies a formal specification of some desirable properties. Verification techniques differ depending on the language paradigm and target specification. However, they can be broadly categorised into model-checking algorithms and deductive verification [8]. Model-checking proceeds by exploring the program's state space, over which the specification is evaluated, whereas deductive verification derives proof obligations

that ensure the program meets its specification. As model checking is designed for automation, it usually requires a finite model of the program derived from an abstraction or explicit depth-bounds. Deductive verification, on the other hand, is traditionally performed in a semi-interactive environment using a combination of proof assistants such as Coq or Isabelle and SMT (satisfiability modulo theory) solvers and thus is more general [9].

The effort required for deductive verification can make it impractical for many projects. Likewise, model checking suffers from the "state explosion" problem whereby the verification task grows exponentially with the complexity of the system [10]. There has, therefore, been a shift in the formal methods' community towards more pragmatic, "lightweight" approaches to verification [11, 12]. Here lightweight refers to a partial analysis of a partial specification that may result in false positives or false negatives but can readily be integrated into the day-to-day software development cycles, thus providing immediate feedback to the programmer. Lightweight verification is usually an automated, or "push button", process. Nevertheless, it is important for the programmer to have some knowledge of the tool's limitations so that they can program defensively around these constraints or provide hints to the tool where necessary.

1.2 Refinement Type Systems

The first verification technique we develop is a refinement type system. Broadly speaking, refinement type systems are formal proof systems for asserting safety properties. That is, they demonstrate some undesirable behaviour, such as diving by zero, does not occur; hence the adage "well-typed programs don't go wrong" [13]. These systems are, by far, the most widespread form of program analysis due to their ease of use and scalability, and have a close connection to higher-order model checking [14].

Traditionally, there are two ways to interpret a type system: an *intrinsic* system distinguishes structurally different categories of expressions to ensure the program has a meaningful interpretation, whereas an *extrinsic* system predicates over some pre-existing semantics [15]. A refinement type system is, in essence, an extrinsic extension of an underlying intrinsic type system. In other words, these types "refine" underlying types by requiring their inhabitants to satisfy some additional properties [16, 17]. Refinement types are usually equipped with a partial order known as subtyping, which implies the inclusion of their respective inhabitants. Subtyping allows for the omission of irrelevant properties, enabling expressions to be given a precise type without restricting their use in more general contexts.

The verification problem for which we devise a refinement type system is that of pattern-match safety, i.e. the non-reachability of pattern-matching failures. Consider the simple function program in Figure 1.1. This program results in a pattern-matching

failure as the `head` function does not have a case corresponding to the `[]` constructor. As most standard type checkers do not distinguish between empty and non-empty lists, it cannot infer a sufficiently precise type for the `head` function. Rejecting such a primitive function would be overly restrictive, however. Instead, the notion of "going wrong" is weakened so that there is no guarantee of pattern-matching safety.

```
let head : List α → α
    head (x :: xs) = x
 in head []
```

Figure 1.1: An unsafe application of the `head` function.

The first refinement type system aiming to retain pattern-matching safety while permitting inexhaustive function definitions considered *regular refinements* of algebraic datatypes via intersection types, implicitly describing a regular tree grammar in the first-order case [16]. Under this system, refinement types encode a contract, guaranteeing that a given function does not result in a pattern-matching failure on the given set of inputs whilst also provide an over-approximation of the function's output in order to be compositional. For example, it could be correctly inferred that the `head` function is only well-defined on non-empty lists and that the subsequent call is unsafe. Note that the non-empty list refinement is still a subtype of lists making this approach less invasive than merely introducing a new, disjoint datatype that is isomorphic to the refinement such as Haskell's `Data.List.NonEmpty` datatype[1]. Nevertheless, this system still required the programmer to specify the relevant refinements and so is only partially automated.

Constraint-base Type Inference Many type systems, including the Hindley-Milner system, are implemented with *constraints*, either as part of type inference or persisting in the resulting type schemes [13, 18, 19]. In this context, constraints are a formal language for restricting a family of types, which can be instantiated differently depending on the demands of the context.

While a tractable unification algorithm enables the Hindley-Milner type system to scale well, it lacks support for subtyping and, thus, has limited use as a precise program analysis. Many type inference algorithms go further and emit subtyping constraints [20, 21]. However, this increase in expressivity naturally leads to an increase in complexity. Unlike the Hindley-Milner type system, the type variables appearing in the subtyping constraints cannot necessarily be assigned a most general solution due to their mixed variance, leading to an exponential number of type variables in the worst-case [22]. Even for non-polyvariant analyses, where there is no duplication, it

[1] https://hackage.haskell.org/package/base-4.18.0.0/docs/Data-List-NonEmpty.html

is typical to have quadratically-many constraints and a cubic-time upper-bound on testing satisfiability [23, 24].

Continuing this line of work, we propose a refinement type system for ensuring pattern-matching safety with a fully automated, constraint-based inference algorithm. As this system doesn't rely on user annotations, it must restrict the space of possible refinements. Our restriction is carefully designed so that constraints have limited complexity and, consequently, the inference algorithm is linear in the size of the program.

1.3 Inductive Equational Reasoning

Equational reasoning exploits the referential transparency of pure functional programs to represent specifications as equations. This highly expressive style of specification may include the equivalence of an optimised function and a more lucid implementation or the correctness of a typeclass instance which is required to uphold some algebraic theory, e.g. the associativity of monoids. A significant advantage of these specifications is that they can be immediately understood by any programmer familiar with the language's syntax, such as the functor laws for the list datatype displayed in Figure 1.2. Consequently, equational reasoning has seen widespread adoption in the functional programming community and often comprises a core element of introductory courses [25].

```
∀xs : List α. map id xs = xs

∀xs : List α, f : b → c, g : a → b.
    map (f ∘ g) xs = map f (map g xs)
```

Figure 1.2: The functor laws for the list datatype.

Unlike type systems, equational reasoning is not structural in that proof obligations are not derived from the structure of expressions nor are specifications associated with a single function. It also differs in that it does not necessarily overapproximate the semantics of a function. Although some properties concern abstractions such as the length of a list, e.g. `length ∘ map f = length`, other algebraic properties including associativity, commutativity, and idempotence do not fall into this category.

It is worth noting, however, that certain type systems can be simulated by equational reasoning. In particular, deterministic top-down tree automata can be directly embedded as functions in the programming language and used to assert type assignments as conditional equations. More generally still, by permitting a mixture of pro-

gram expressions and abstract symbols representing types, type checking and inference can be framed as a term rewriting problem [26].

In the latter part of this thesis, we will devise an automated equational reasoning tool. The objective of this system is to determine the universal validity of equations representing a program's specification, thus serving as a specialised, automated theorem prover for deductive verification. As such, it is necessary to take advantage of the domain's structure when creating the underlying proof system if it is to be practically automatable.

The Difficulty of Automating Induction Algebraic datatypes are a fundamental aspect of functional programming on which many standard algorithms rely [27]. As these datatypes and the functions manipulating them are typically recursive, an equational reasoning tool must have some support for proof by induction or analogous apparatus. Automating proof by induction is, however, notoriously difficult [28]. This complexity is formally described by its *non-analyticity* — the inability to, in general, construct a proof from sub-formulas of the goal alone. In other words, proofs by induction often require synthesising stronger hypotheses or auxiliary lemmas that might not be direct syntactic generalisations [29]. The main body of research towards the automation of proof by induction has thus focused on strategies for hypothesis strengthening and lemma generation [30, 31, 32, 33, 34]. Significant progress has been made in this area but often relies on heuristics that can impede performance even for simple properties.

In contrast to work described above, we put aside the issue of lemma generation and focus on developing an efficient proof system and proof search algorithm that can serve as the kernel of a verification system. As any lemma generation or hypothesis strengthening procedure is inherently inexhaustive, there is always a role for programmer guidance, which is likely to be more directed due to their understanding of the code base. Many people believe that deductive verification is overly challenging because it requires specific expertise from the user. However, this is not true of equational reasoning since the logic and target language overlap. What is more, the focus on heuristics in existing research has left core concerns, such as support for mutual induction, underdeveloped. When dealing with mutually inductive structures, the issue of non-analyticity becomes compounded as complementary induction hypotheses are required. Yet, mutual induction is not reducible to a combination of non-mutual induction and synthesis techniques; mutually inductive structures may contain an unbounded number of inductive sub-positions, e.g. rose trees, and thus cannot be represented by a standard inductive structure.

Cyclic Proofs Traditional proof systems consider finite derivation trees, for which soundness is justified by structural induction over the tree. Non-well-founded proof

theory generalises such systems to admit infinite derivations. The individual inferences constituting an infinite derivation must be well-formed, and thus the argument is locally sound, but this property is no longer sufficient to establish soundness as an argument can be constructed by assuming the conclusion ad infinitum. Instead, an additional "global soundness" condition ensures that only a finite, initial segment of the proof is required by any given instance of its conclusion.

A cyclic proof is a non-well-founded proof that only has finitely many distinct subderivations, i.e. an ω-regular derivation tree. This class is of particular importance for automated reasoning as they can be represented by finite graphs and typically have a decidable global soundness condition [35, 36, 37]. As the recursive structure of the problem is delegated to the cyclic structure of the proof graph, proof search is more flexible locally. In particular, there is no need to fix an induction scheme or even an induction hypothesis — all previously encountered judgements can be treated as candidate induction hypotheses and used cyclically to discharge related proof obligations, as long as these cycles satisfy the global soundness condition. This demand-driven approach to induction is distinctly powerful in the case of mutual induction that otherwise cannot proceed without synthesis.

Cyclic proofs have recently been proposed for a number of program analysis and synthesis tasks, as well as being used to formalise existing model-checking algorithms [38, 39, 40, 41]. However, prior to our work, no specialised system for equational reasoning about functional programs had been developed. Although Cyclist, a generic cyclic theorem prover, has been able to prove basic equational properties, its lack of first-class support for equational reasoning is limiting in practice [42]. Cyclic proofs are highly suited to reasoning about functional programs due to their first-class support for mutually and nested inductive datatypes and the capacity to mimic complex patterns of recursion by delaying the choice of induction schemes and hypotheses. Nevertheless, it is clear that equational reasoning requires distinguished proof mechanisms due to its intractable search space.

Thus, to suppose mutual induction and an efficient proof search algorithm, we base our proof system on cyclic proofs. Unlike existing cyclic proof systems, however, we prioritise equational reasoning by directly integrating it into cycle formation. This combination allows it to excel at automatically proving the equational properties of functional programs.

1.4 Thesis Outline and Contributions

As program verification is generally undecidable, there is a hard limit on the capacity of fully automated verification tools. In light of this, a common goal of our work is not to maximise the expressivity or coverage of our verification systems but rather their ease of use and efficiency. For example, we limit refinement types to eliminate

the need for user annotations, and we forgo hypothesis generalisation or lemma synthesis in pursuit of an effective core proof mechanism. As a result, the lightweight approaches we develop are sufficiently performant to be integrated into compilation pipelines and interactive development environments.

The structure and contributions of this thesis is as follows. Proofs that are not found inline are listed in the appendix.

- Chapter 2 introduces a simple functional programming language, including higher-order functions and algebraic datatypes, which our verification systems target. We provide an operational semantics for this language with a somewhat novel presentation that enables later theoretical developments.

  In actuality, our prototype tools are implemented as plugins for the Glasgow Haskell Compiler (GHC), targeting the GHC Core language [43]. This setup allows them to be used within the existing compilation pipeline and paves the way for more mature implementations in the future. As GHC Core is a more complex language than our calculus, we will comment on any significant and relevant differences throughout.

- The first verification system — *intensional refinement types* is presented in Chapter 3. Initially, we introduce a declarative refinement type system based on the novel notion of an *intensional refinement*. The purpose of this system is to provide the programmer with a formal understanding of which programs will be deemed safe by our type inference system, allowing them to program defensively within the system and decide whether to act on warnings produced by the type checker.

  Building on this, we present our constrained type inference system that is sound and complete with respect to the declarative system. We demonstrate that, under reasonable assumptions, our analysis is linear in the size of the program due to the novel restriction we place on the shape of refinements. To conclude this chapter, we discuss our prototype implementation and its performance and limitations.

  This work is derived from the paper 'Intensional datatype refinement: with application to scalable verification of pattern-match safety', which was presented at POPL '21 [44]. The present author's contribution was principally in the development of the type inference system and its implementation.

- Chapter 4 concerns CycleQ — a novel proof system for cyclic equational reasoning designed for automated proof search. After motivating the use of cyclic proofs for reasoning about functional programs at a high-level, we present our core proof system. The key contribution of the proof system is a novel cut-like rule that integrates equational reasoning directly into the formation of cycles,

thus largely mitigating the intractable search space associated with naïve equational reasoning. We go onto to show that this system, although remarkably simple, subsumes two alternatives to traditional proof by induction known as *proof by consistency* and *rewriting induction* when restricted to functional programs [45, 46].

The main technical difficulty associated with a cyclic proof system is in verifying the global soundness condition. To overcome this challenge, we introduce an alternative verification technique that can be performed incrementally, during proof construction. We then discuss the performance and limitations of our prototype implementation.

The work presented in this chapter is based on 'CycleQ: An Efficient Basis for Cyclic Equational Reasoning', which was published at PLDI '22 [47]. As the principal contributor to that paper, the core ideas, theoretical and empirical results are attributed to the present author.

- One of the principal limitations discovered by our evaluation of CycleQ is that many equational reasoning tasks require conditional reasoning, even if no conditions are explicitly present in the goal. In Chapter 5, we report on unpublished work that extends our proof system with support for conditional reasoning. As our focus is on efficiency, we replace the non-deterministic application of hypotheses, which would quickly lead to an intractable search space, with a novel algorithm for integrating hypotheses into the program's operational semantics as a confluent and terminating rewrite system. This mechanism is further accompanied by a narrowing-based procedure for determining viable instances of conditional lemmas. We conclude by showing that these additional mechanisms significantly expand the set of solvable problems without impeding the tool's performance.

  The work of this chapter is entirely attributed to the present author.

Chapter 2

Preliminaries

Throughout this thesis, we will work with MiniHask — a simple function programming language with algebraic datatypes and Hindley-Milner style polymorphism. As the name suggests, MiniHask is intended to represent a small subset of the Haskell programming language, for which we formalise our verification systems. In this chapter, we introduce its syntax and semantics and prove some basic properties of the language.

2.1 Syntax

2.1.1 Types and Type Schemes

Definition 2.1. We assume a finite collection of *datatype identifiers* $\mathbb{D}$, ranged over by d, each of which is equipped with a fixed arity, written $\text{arity}(d) \in \mathbb{N}$.

These datatype identifiers can be thought of as the names of (first-order) type constructors. The arity of a datatype identifier indicates the number of type parameters it takes, e.g. $\text{arity}(\texttt{Bool}) = 0$ and $\text{arity}(\texttt{List}) = 1$. For simplicity, our formalism does not account for higher-order type constructors; hence, type parameters are not differentiated by kind. However, it is straightforward to extend the system to include higher-order type constructors.

As is standard in Hindley-Milner style type systems, we distinguish *types* from *type schemes*. Proper type schemes apply to top-level definitions and constructors, encoding parametric polymorphism whereby their quantified variables can be instantiated with concrete types as required. Within the body of a function definition, on the other hand, expressions are only assigned monomorphic types, although these types may include type variables from their ambient context.

Definition 2.2. Types and type schemes are defined by the following grammar:

$$\begin{array}{ll} \mathsf{Ty} & \tau, \sigma ::= \alpha \mid d\ \tau_1\ \cdots\ \tau_{\mathsf{arity}(d)} \mid \tau \to \sigma \\ \mathsf{Sch} & \rho ::= \forall \alpha.\, \rho \mid \tau \end{array}$$

where α, β, γ etc. range over a countable set of type variables $\mathbb{A}$.

The function arrow associates to the right by default and type schemes are identified up to α-equivalence. Types of the form $d\ \tau_1\ \cdots\ \tau_{\mathsf{arity}(d)}$, such as `List Nat`, are referred to as *datatypes* and we write Dt to denote the set of such types. When we wish to restrict types to those containing only a subset of datatype identifers $D \subseteq \mathbb{D}$, we write $\mathsf{Ty}(D)$, $\mathsf{Sch}(D)$, and $\mathsf{Dt}(D)$.

Definition 2.3. The *free type variables* $\mathsf{FTV}(\tau) \subseteq \mathbb{A}$ of a type or type scheme τ is defined by recursion over the type as follows:

$$\mathsf{FTV}(\alpha) \coloneqq \{\alpha\} \qquad \mathsf{FTV}(\tau_1 \to \tau_2) \coloneqq \mathsf{FTV}(\tau_1) \cup \mathsf{FTV}(\tau_2)$$

$$\mathsf{FTV}(\forall \alpha.\, \rho) \coloneqq \mathsf{FTV}(\rho) \setminus \{\alpha\} \qquad \mathsf{FTV}(d\ \tau_1\ \cdots\ \tau_n) \coloneqq \bigcup_{i \le n} \mathsf{FTV}(\tau_i)$$

A type or type scheme is said to be *closed* is it contains no free type variables.

Definition 2.4. A *type-level substitution* $\Theta : \mathbb{A} \rightharpoonup \mathsf{Ty}$ is a partial map from type variables to types. The standard homomorphic extension of a substitution to a function on types, written using postfix notation $\tau\Theta$, is defined by recursion over the type as follows:

$$\alpha\Theta \coloneqq \begin{cases} \Theta(\alpha) & \text{if } \alpha \in \mathsf{dom}(\Theta) \\ \alpha & \text{otherwise} \end{cases} \qquad \begin{array}{c} (d\ \tau_1\ \cdots\ \tau_n)\Theta \coloneqq d\ (\tau_1\Theta)\ \cdots\ (\tau_n\Theta) \\ (\tau \to \sigma)\Theta \coloneqq \tau\Theta \to \sigma\Theta \end{array}$$

We will denote substitutions as sets of pairs $\alpha \mapsto \tau$ or, when directly applied to a type, using the notation $[\tau/\alpha]$.

2.1.2 Expressions

Definition 2.5. We assume a finite set of *constructors* $\mathbb{K}$, ranged over by k and its variants, that are equipped with a fixed arity, written $\mathsf{arity}(k) \in \mathbb{N}$.

Definition 2.6. The expressions of MiniHask are defined by the following grammar:

$$\begin{array}{ll} \mathsf{Exp} & e ::= x \mid k \mid e_1\ e_2 \mid \lambda x.\, e \\ & \quad \mid \texttt{case}\ e\ \texttt{of}\ \{k_1\ \overline{x}_1 \mapsto e_1;\ \ldots;\ k_n\ \overline{x}_n \mapsto e_n\} \end{array}$$

where x, y, z etc., range over a countable set of expression variables $\mathbb{V}$.

Application associates to the left by default and expressions are identified up to α-equivalence. The branch of a case expression associated with the constructor k is required to bind exactly arity(k) pattern variables. Furthermore, the branches of a case expression are assumed to be non-overlapping, i.e. there is no $i \neq j$ such that $k_i = k_j$. Thus, we also equate case expressions up to the permutation of their branches.

Definition 2.7. The *free variables* $\mathsf{FV}(e) \subseteq \mathbb{V}$ of an expression e is defined by recursion over the expression as follows.

$$\mathsf{FV}(x) \coloneqq \{x\} \qquad \mathsf{FV}(k) \coloneqq \emptyset$$

$$\mathsf{FV}(e_1\ e_2) \coloneqq \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \qquad \mathsf{FV}(\lambda x.\, e) \coloneqq \mathsf{FV}(e) \setminus \{x\}$$

$$\mathsf{FV}(\texttt{case}\ e\ \texttt{of}\ \{k_i\ \overline{x_i} \mapsto e_i \mid i \leq n\}) \coloneqq \mathsf{FV}(e) \cup \bigcup_{i \leq n} \{\mathsf{FV}(e_i) \setminus \{\overline{x_i}\}\}$$

Definition 2.8. An *(expression-level) substitution* $\theta : \mathbb{V} \rightharpoonup \mathsf{Exp}$ is a partial map from variables to expressions. As with type-level substitutions, substitutions written in a postfix position are interpreted by their homomorphic extension to expressions which is defined recursively as follows:

$$x\theta \coloneqq \begin{cases} \theta(x) & \text{if } x \in \mathrm{dom}(\theta) \\ x & \text{otherwise} \end{cases} \qquad k\theta \coloneqq k$$

$$(e_1\ e_2)\theta \coloneqq (e_1\theta)\ (e_2\theta) \qquad (\lambda x.\, e)\theta \coloneqq \lambda x.\, e\theta$$

$$(\texttt{case}\ e\ \texttt{of}\ \{k_i\ \overline{x_i} \mapsto e_i \mid i \leq n\})\theta \coloneqq \texttt{case}\ e\theta\ \texttt{of}\ \{k_i\ \overline{x_i} \mapsto e_i\theta \mid i \leq n\}$$

Following the Barendregt naming convention, we assume that any locally bound variables are distinct from those appearing in the ambient context (including those appearing in the substitution). In which case, the preceding definition is implicitly capture-avoiding.

Expression-level substitutions are denoted in the same manner as type-level substitutions: as a set of pairs $x \mapsto e$ or with the notation $[e/x]$ when directly applied to an expression. We write $\mathsf{FV}(\theta) \subseteq \mathbb{V}$ for the *free variables* of a substitution, which is defined as the set $\bigcup_{x \in \mathrm{dom}(\theta)} \mathsf{FV}(\theta(x))$, i.e. the free variables of the expressions in the substitutions codomain.

Definition 2.9. The composition of two substitution, written $\theta\theta'$, is defined by the map: $x \mapsto (x\theta)\theta'$ for all $x \in \mathrm{dom}(\theta) \cup \mathrm{dom}(\theta')$. Substitution forms a monoid action with respect to composition; that is, $e\emptyset = e$ and $e(\theta\theta') = (e\theta)\theta'$ for all expressions.

2.1.3 Typing Judgement

Definition 2.10. A *datatype environment* $\Delta : \mathbb{K} \rightharpoonup \mathsf{Sch}$ is a partial function from constructors to closed type schemes of the form:

$$\forall \alpha_1 \cdots \alpha_{\text{arity}(d)} . \tau_1 \to \cdots \to \tau_{\text{arity}(k)} \to d\ \alpha_1\ \cdots\ \alpha_{\text{arity}(d)}$$

where $d \in D$ is some datatype identifier. We write $k : \rho \in \Delta$ for some $k \in \mathbb{K}$ to indicate that $\Delta(k)$ is defined as ρ.

Note that constructors are required to be polymorphic in the parameters of the datatype that they ultimately return. In particular, we do not consider generalised algebraic datatypes or existential type parameters.

Definition 2.11. The *restricted datatype environment* $\Delta(d) \subseteq \Delta$ is defined as the subset of typings $k : \rho$ where the type scheme ρ ultimately returns an instance of the datatype identifier $d \in \mathbb{D}$. For some constructor $k : \rho \in \Delta(d)$, we identify $\Delta(d)(k)$ with the sequence of argument types $\tau_1, \ \ldots, \tau_{\text{arity}(k)}$ to the type scheme ρ as its return type is uniquely determined.

Typically, we will present datatype environments using Haskell-style data declarations such as the following example, which defines a datatype environment associated with the datatypes identifiers `Nat`, `List`, and `Tree`.

```
data Nat            data List α
  = Z                 = []
  | S Nat             | α :: List α

  data Tree α
    = Leaf
    | Node (Tree α) α (Tree α)
```

Figure 2.1: Example algebraic datatypes.

In order to aid readability, we will use the double colon to denote the "cons" constructor for lists when presenting MiniHask programs so that it is distinguished from the typing judgement. We will also use conventional list notation $[e_1, \ \ldots, \ e_n]$ (to be understood as $e_1 :: \cdots :: e_n :: []$) when convenient.

Definition 2.12. A *type environment* $\Gamma : \mathbb{V} \rightharpoonup \mathsf{Sch}$ is a partial map from variables to type schemes, with $x : \rho \in \Gamma$, as usual, indicating that $\Gamma(x)$ is defined as ρ. We will write $\mathsf{FTV}(\Gamma) \subseteq \mathbb{A}$ for the set $\bigcup_{x:\rho \in \Gamma} \mathsf{FTV}(\rho)$. As usual, we say that Γ is *closed* whenever $\mathsf{FTV}(\Gamma)$ is empty.

Type-level substitutions Θ are extended to act on type environments in the usual manner so that $\Gamma\Theta$ denotes the type environment $\{x : \tau\Theta \mid x : \tau \in \Gamma\}$.

Definition 2.13. For a given datatype environment Δ, the judgement $\Gamma \vdash e : \tau$, which indicates that the expression e can be given the type τ when its free variables are typed according to Γ, is defined inductively by the inference rules in Figure 2.2.

$$\frac{}{\Gamma \vdash x : \tau[\overline{\sigma/\alpha}]}\, x : \forall \overline{\alpha}.\, \tau \in \Gamma \qquad \frac{}{\Gamma \vdash k : \tau[\overline{\sigma/\alpha}]}\, k : \forall \overline{\alpha}.\, \tau \in \Delta$$

$$\frac{\Gamma \vdash e_1 : \tau \to \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \sigma} \qquad \frac{\Gamma \cup \{x : \tau\} \vdash e : \sigma}{\Gamma \vdash \lambda x.\, e : \tau \to \sigma}$$

$$\frac{\Gamma \vdash e : d\ \overline{\tau} \qquad (\forall i \leq n)\ \Gamma \cup \{\overline{x_i : \Delta(d)(k_i)[\overline{\tau/\alpha}]}\} \vdash e_i : \sigma}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{k_i\ \overline{x_i} \mapsto e_i \mid i \leq n\} : \sigma}$$

Figure 2.2: The typing rules for expressions.

Definition 2.14. The typing judgement $\Gamma \vdash \theta : \Gamma'$, for a substitution θ, indicates that $\Gamma \vdash \theta(x) : \tau$ for all $x : \forall \alpha_1, \ldots, \alpha_n.\, \tau \in \Gamma'$, where the type variables $\alpha_1, \ldots, \alpha_n$ do not appear freely in Γ. In which case, by induction, the homomorphic extension of θ preserves the well-typedness of expressions under the environment Γ', transforming the judgement $\Gamma' \vdash e : \tau$ into $\Gamma \vdash e\theta : \tau$.

Definition 2.15. A Σ-*program* P, for some closed type environment Σ, is a substitution such that $\Sigma \vdash P : \Sigma$. That is, a program maps each variable in the environment $f : \forall \alpha_1, \ldots, \alpha_n.\, \tau \in \Sigma$ to an expression $P(f)$ such that the typing judgement $\Sigma \vdash P(f) : \tau$ holds. Note that these definitions may themselves contain variables from the environment Σ, and thus allow for (mutually) recursive functions.

For a given program, we refer to the corresponding type environment Σ as the *program environment* and will typically reserve Σ for this role. Any variable bound by the program, i.e. an element of $\mathrm{dom}(\Sigma)$, will be referred to as *a program variable*, for which we reserve f, g, h. In general, we assume that any given type environment only map program variables to proper type schemes, and any other variables are assigned monomorphic types. That is to say all type environments should decompose into a program environment Σ and a local environment Γ such that only Σ contains proper type schemes. This restriction corresponds to the fact that we do not permit higher-ranked types and, therefore, any variables bound within an expression will be monomorphic.

Definition 2.16. A (well-typed) expression e is said to be *closed*, with respect to a given Σ-program, if its free variables are contained within Σ, i.e. $\Sigma \vdash e : \tau$ for some type τ.

As with datatype environments, we will typically present programs using Haskell-esque syntax as a set of equations. For example, the program in Figure 2.3 is to be understood as assigning the program variable `map` to an expression with two λ-abstractions that performs case analysis on its second argument in the body.

```
map : (α → β) → List α → List β
map f [] = []
map f (x :: xs) = f x :: map f xs
```

Figure 2.3: The definition of the `map` function.

Our type system is sufficient to ensure that programs don't go wrong in a very basic sense, e.g. expressions evaluating to a datatype are not treated as a function. However, programs can still get stuck due to inexhaustive pattern-matching expressions. The existence of such scenarios is the subject of the next chapter of this thesis. However, we will sometimes rely on a simple, conservative approximation of this property that is easily checked by compilers.

Definition 2.17. A program or expression is said to be *exhaustive* if it is well-typed under the following refinement of the typing rule for `case` expressions:

$$\frac{\Gamma \vdash e : d\,\overline{\tau} \qquad (\forall i \leq n) \quad \Gamma \cup \{\overline{x_i : \Delta(d)(k_i)[\overline{\tau/\alpha}]}\} \vdash e_i : \sigma}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{k_i\ \overline{x_i} \mapsto e_i \mid i \leq n\} : \sigma}$$

in which, additionally, $\mathrm{dom}(\Delta(d))$ is required to be a subset of $\{k_1, \ldots, k_n\}$, i.e. each possible constructor of the datatype d has a corresponding case. Otherwise, we say it is *inexhaustive*.

For example, the aforementioned `map` function is considered exhaustive as the sole case expression has a branch for each constructor associated with the `List` datatype identifier, i.e. `[]` and `::`. In contrast, the `head` function defined in the introduction is inexhaustive.

2.2 Operational Semantics

The semantics of MiniHask programs are given operationally as a reduction relation. In subsequent chapters, we will study equational properties over expressions where its operational semantics serves as a basis for symbolic reasoning. To minimise bookkeeping required when manipulating these expressions, we restrict our attention to those without binders, following the approach of Zeno that distinguishes applicative "terms" from "expressions" [32].

Definition 2.18. An *applicative expression* is an expression without binders, and an *applicative context* is just an applicative expression with a distinguished "hole" variable appearing exactly once. These two categories are formally defined by the following grammar:

$$a,\, b ::= x \mid k \mid a\ b$$
$$C[\cdot] ::= \cdot \mid C[\cdot]\ a \mid a\ C[\cdot]$$

As usual, we write $C[a]$ to denote the applicative expression that results from instantiating the hole with the applicative expression a.

It is not possible to simply unfold program definitions without introducing non-applicative expressions. Instead, we determine a series of applicative rewrite rules from a given program, effectively recovering the equation presentation of a program. Defining the reduction relation in this manner is also motivated by the desire to leverage well-established results concerning rewrite systems in subsequent chapters.

Definition 2.19. A *definitional expression* is an element of the following grammar:

$$d ::= \lambda x.\, d \mid \texttt{case}\ x\ \texttt{of}\ \{k_1\ \overline{x}_1 \mapsto d_1;\ \ldots;\ k_n\ \overline{x}_n \mapsto d_n\} \mid a$$

where a is an arbitrary applicative expression.

To construct the reduction relation, the program is first transformed so that each function definition is a definitional expression. This initial pre-processing is done by abstracting over the scrutinee of case expressions and then lifting any λ- or case-expressions into new top-level definitions when appearing in a non-definitional position, e.g. an argument. We do not formally justify the semantic preserving nature of such a transformation as λ-lifting is a standard technique and its extension to handle case expressions in this manner is merely an instance of β-expansion [48]. Thus, without loss of generality, we will assume programs only bind variables to definitional expressions.

2.2.1 Incremental Matching

Definitional expressions correspond closely to the "definitional trees" used to define strictly orthogonal rewrite systems, i.e. systems that distinguish between constructors and defined functions and for which there are no overlapping rules [49]. The advantage of presenting rewrite rules as definitional trees is that they provide additional control, analogous to fixing an evaluation strategy but for matching, making it easier to prove basic properties about the system such as confluence. In our case, the reduction relation is derived from the following matching operation.

Definition 2.20. *Incremental matching* $d\ a_1\ \cdots\ a_n \Downarrow_\theta a$ is a partial function from a definitional expression d, a series of applicative arguments a_1 through to a_n, and a

substitution θ, to an applicative expression a, which is defined in Figure 2.4 by recursion over the definitional expression d. For this function, we assume the substitution only contains applicative expressions.

$$\frac{}{a\ a_1\ \cdots\ a_n \Downarrow_\theta a\theta\ a_1\ \cdots\ a_n} \qquad \frac{d\ a_2\ \cdots\ a_n \Downarrow_{\theta \cup \{x \mapsto a_1\}} a}{(\lambda x.\, d)\ a_1\ \cdots\ a_n \Downarrow_\theta a}$$

$$\frac{d_i\ a_1\ \cdots\ a_m \Downarrow_{\theta \cup \{\overline{x_i \mapsto b}\}} a}{\texttt{case}\ x\ \texttt{of}\ \{k_i\ \overline{x}_i \mapsto d_i \mid i \leq n\}\ a_1\ \cdots\ a_m \Downarrow_\theta a}\ \theta(x) = k_i\ b_1\ \cdots\ b_\ell$$

Figure 2.4: The incremental matching function.

Intuitively, incremental matching permits the minimal amount of reduction necessary to eliminate a definitional expression, which may contain binders, resulting in an applicative expression that does not. For simplicity, the substitution is accumulated in the judgement instead of being directly applied to the definitional expression as this would result in a non-definitional expression.

Definition 2.21. For a given program P, we define the *one-step reduction relation* as the least binary relation on applicative expressions such that:

$$C[f\ a_1\ \cdots\ a_n] \to_P C[a]$$

whenever f is program variable for which $P(f)\ a_1\ \cdots\ a_n \Downarrow_\emptyset a$.

The *(many-step) reduction relation* $\to_P^*$ is then defined as the reflexive-transitive closure of the one-step rewrite relation.

The program's reduction relation alternates between exposing definitional expressions and eliminating them via incremental matching. Note that incremental matching must entirely eliminate the definitional expression before the reduction relation can move on to another sub-expression. In particular, incremental matching cannot reduce the expression that becomes the scurtinee of a case expression. However, as definitional expressions only pattern-match against λ-bound variables, their scurtinees are only instantiated with sub-expressions that can be reduced in a separate reduction step.

Lemma 2.1. Let P be a Σ-program. If $\Sigma \cup \Gamma \vdash f\ a_1\ \cdots\ a_n : \tau$ is an applicative expression and $P(f)\ a_1\ \cdots\ a_n \Downarrow_\emptyset b$ is defined, then $\Sigma \cup \Gamma \vdash b : \tau$.

Corollary 2.2. Let P be a Σ-program. If $\Sigma \cup \Gamma \vdash a : \tau$ is an applicative expression and $a \to_P^* b$, then $\Sigma \cup \Gamma \vdash b : \tau$.

The correspondence between incremental matching and a set of strictly-orthogonal rewrite rules is captured by the following two lemmas. The first show that incremental matching is closed under substitution and the application of additional arguments. Consequently, it is sufficient to consider a subset of incremental matches, i.e. instances of the incremental matching relation, as the underlying rewrite rules of the reduction relation from which all other incremental matches can be derived by instantiation.

Lemma 2.3. If $P(f)\ a_1\ \cdots\ a_n \Downarrow_\emptyset a$, then $P(f)\ a_1\theta\ \cdots\ a_n\theta \Downarrow_\emptyset a\theta$ for any substitution θ and $P(f)\ a_1\ \cdots\ a_n\ a_{n+1}\ \cdots\ a_m \Downarrow_\emptyset a\ a_{n+1}\ \cdots\ a_m$ for any applicative expressions $a_{n+1}, \ldots, a_m$.

The second lemma characterises such a subset of incremental matches as those linear, applicative expressions without program variables, referred to as *patterns*.

Definition 2.22. Formally, a pattern is an element of the following grammar:

$$p, q ::= x \mid k\ p_1\ \cdots\ p_{\text{arity}(k)}$$

where $x \notin \text{dom}(P)$ and each variables appears in at most one position, i.e. for any variable x there is at most one applicative context $C[\cdot]$ for which $p = C[x]$.

Note that, as constructors must be fully applied in a pattern, only variables are matched against higher-order expressions.

Lemma 2.4. If $P(f)\ a_1\ \cdots\ a_n \Downarrow_\emptyset a$, then there exists some patterns $p_1, \ldots, p_n$ such that $\mathsf{FV}(p_i) \cap \mathsf{FV}(p_j) = \emptyset$ for all $i \neq j$ and a substitution θ such that $p_i\theta = a_i$ for all $i \leq n$ for which $P(f)\ p_1\ \cdots\ p_n \Downarrow_\emptyset$ is defined.

Proof. For this proof, we must first consider which generalisations of incremental matches are admissible. The following lemma implies that any sub-expressions that are applications with constructors in head position can be generalised.

Lemma 2.5. If $P(f)\ a_1\theta\ \cdots\ a_n\theta \Downarrow_\emptyset a$ but $P(f)\ a_1\ \cdots\ a_n \Downarrow_\emptyset$ is undefined, then $\theta(y)$ is of the form $k\ b_1\ \cdots\ b_{\text{arity}(k)}$ for at least one variable $x \in \bigcup_{i \leq n} \mathsf{FV}(a_i)$.

Now suppose $P(f)\ a_1\ \cdots\ a_n \Downarrow_\emptyset a$ is defined and consider the *minimal linear-generalisation* of the arguments to patterns $p_1, \ldots, p_n$ by a substitution θ such that $p_i\theta = a_i$ for all $i \leq n$. The patterns are linear in that $\mathsf{FV}(p_i) \cap \mathsf{FV}(p_j) = \emptyset$ for all $i \neq j$ as well as the requirement that each variable appears in at most one position within a given pattern. Here, minimality refers to the subsumption order where p is said to be at least as general as q just if $p\theta = q$ for some substitution θ. The existence of a minimal linear-generalisation is plain to see by structural induction over the expressions.

If $P(f)\ p_1\ \cdots\ p_n \Downarrow_\emptyset$ were not defined, then Lemma 2.5 would imply that there is some variable x for which $\theta(x)$ is of the form $k\ b_1\ \cdots\ b_m$, where m is the ar-

ity of k. However, if this were the case, then $p_1, \ldots, p_n$ and θ cannot be a minimal linear-generalisation as we could consider the less general sequence of patterns $(p_1, \ldots, p_n)[k\ x_1\ \cdots\ x_m/x]$ where $x_1, \ldots, x_m$ are fresh variables and the corresponding substitution θ' defined as:

$$\theta'(z) = \begin{cases} b_i & \text{if } z = x_i \\ \theta(z) & \text{otherwise} \end{cases}$$

Thus $P(f)\ p_1\ \cdots\ p_n \Downarrow_\emptyset$ is defined as required. □

Having shown that all incremental matches can be generated from patterns, it is easy to see that the program's reduction relation corresponds to a set of rewrite rules where the left-hand sides are of the form $f\ p_1\ \cdots\ p_n$ for some program variable $f \in \mathrm{dom}(P)$ and a linear series of patterns.

Corollary 2.6. The least binary relation such that $C[f\ p_1\theta\ \cdots\ p_n\theta] \to_P C[a\theta]$ whenever $P(f)\ p_1\ \cdots\ p_n \Downarrow_\emptyset a$ and $\mathsf{FV}(p_i) \cap \mathsf{FV}(p_j) = \emptyset$ for all $i \neq j$ is exactly the one-step reduction relation.

The rewrite system is clearly orthogonal as any overlapping matches are resolved by stability, see Lemma 2.3, and it is well-known that such systems are locally confluent. Finally, assuming the program is *terminating* in that there are no infinite reduction sequences $a_1 \to_P a_2 \to_P \cdots$, we can show that every expression has a unique *normal form*, written $a{\downarrow_P}$, i.e. there is a unique expression b such that $a \to_P^* b$ but for which there does not exist an expression c such that $b \to_P c$.

Theorem 2.7. If $a \to_P b_1$ and $a \to_P b_2$, then there exists an applicative expression c such that $b_1 \to_P^* c$ and $b_2 \to_P^* c$.

Corollary 2.8. Suppose P is a terminating program. Then, for any applicative expression a, the unique normal form $a{\downarrow_P}$ is well defined.

2.2.2 Contextual Equivalence

The closed normal forms of a datatype can be characterised by applications with constructors in head position; when all case expressions are exhaustive, any program variable becomes reducible once fully applied to closed arguments. This lemma is of particular importance as it allows us to justify case analysis on datatype expressions.

Lemma 2.9. If P is an exhaustive Σ-program, then the normal form $a{\downarrow_P}$ of a closed expression $\Sigma \vdash a : d\ \tau_1\ \cdots\ \tau_n$ is necessarily of the form $k\ a_1\ \cdots\ a_m$ for some $k \in \mathrm{dom}(\Delta(d))$.

Unfortunately, no such characterisation is possible for normal forms of higher-order type as they may only become reducible when provided with additional arguments. As a result, although the reduction relation is confluent, it is not sufficient to define the equivalence of closed expressions in purely syntactic terms via their normal forms. Instead, we employ contextual equivalence — the inability of the program to distinguish expressions by a certain class of contexts. In our case, we assume a nullary datatype identifier $\texttt{Bool} \in \mathbb{D}$ with two constructors $\texttt{False}, \texttt{True} : \texttt{Bool} \in \Delta(\texttt{Bool})$ that serve as a the basis for distinguishable values.

Definition 2.23. For a Σ-program P, we write $a \equiv_P b$ for two closed, applicative expressions of the same type $\Sigma \vdash a, b : \tau$ such that $(C[a])\!\downarrow_P \; = \; (C[b])\!\downarrow_P$ for all Boolean-valued typed contexts $\Sigma \cup \{x : \tau\} \vdash C[x] : \texttt{Bool}$. In which case, we say they are *P-equivalent.*

Lemma 2.10. For any well-typed program, equivalence is a congruence relation that contains the reduction relation. That is, it is closed under reflexivity, symmetry, transitivity, congruence, and reduction.

It is worth noting that, as we only consider applicative contexts, equivalence can be overly permissive. For example, in order to distinguish the two functions $\lambda f.\, f\ \texttt{False}\ \texttt{True}$ and $\lambda f.\, f\ \texttt{True}\ \texttt{False}$ it is necessary to provide a function such as $\lambda x\, y.\, x$ as an argument. However, there might not be a program variable assigned to this definitional expression, in which case we cannot construct an applicative context that distinguishes the two expressions. To resolve this problem, we assume the program already includes a complete set of combinators, such as `S`, `K`, and `I`, as well as pattern-matching primitives for each datatype identifier, e.g. a program variable $\texttt{case}_{\texttt{Nat}}$ of type $\texttt{Nat} \to \alpha \to (\texttt{Nat} \to \alpha) \to \alpha$ with the obvious definitions. These combinators allow us to construct applicative contexts that simulate any non-recursive expression, giving us a precise notion of contextual equivalence [50].

As expected, datatype expressions are contextually equivalent just if they result in matching constructors and all their arguments are also contextually equivalent. For expressions of a function type, equivalence asserts that they produce equivalent results on all equivalent arguments, thus satisfying function extensionality. Although the proof of function extensionality is outside the scope of this thesis, informally, it follows form the fact that definitional expressions can only branch according to the value of a datatype sub-expression and thus expressions of a function type can only be distinguished once fully applied.

Lemma 2.11. Let P be an exhaustive Σ-program. If $\Sigma \vdash a, b : d\ \tau_1\ \cdots\ \tau_n$ are two P-equivalent applicative expressions, then $a \to_P^* k\ a_1\ \cdots\ a_m$ and $b \to_P^* k\ b_1\ \cdots\ b_m$ for some $k \in \mathrm{dom}(\Delta(d))$ where $a_i \equiv_P b_i$ for all $i \leq m$. Furthermore, the converse holds by congruence.

Lemma 2.12. Let P be an exhaustive Σ-program. If $\Sigma \vdash a, b : \tau \to \sigma$ are two P-inequivalent applicative expressions, then there exists some $\Sigma \vdash c : \tau$ such that $a\ c \not\equiv_P b\ c$. Again, the converse holds by congruence.

2.2.3 An Aside on the Issue of Semantics

Haskell has become known as the de facto lazy programming language where, for example, it is a common idiom to take a prefix of an infinite list. The assumption that MiniHask programs terminate without regard to evaluation context, however, means that MiniHask programs can only simulate programs which behave equivalently under strict semantics; in particular, first-order datatypes must correspond to inductively defined finite trees. Despite this mismatch, it is by design that MiniHask is restricted in this manner, as variables should only be bound to finite values in order to be amenable to proof by induction. Furthermore, it is generally acknowledged that functional programmers are used to reasoning under the assumption that values are finite [51]. For example, if we consider infinite values, it would no longer be the case that the equation $\texttt{leq}\ x\ x = \texttt{True}$ holds as the left-hand side might diverge.

Although one could instead imagine an equation conditional on the finiteness of a value, e.g. $\texttt{isFinite}\ x \Rightarrow \texttt{leq}\ x\ x = \texttt{True}$, constructing a meaningful proof system for such properties remains challenging. Denotationally, infinite values are defined in domain theory as the least-upper bound of a chain of finite approximations [52]. In order to perform induction over domains, it is necessary to restrict our attention to continuous or "admissible" properties. However, conditional equations such as $x = \texttt{S}\ x \Rightarrow \texttt{True} = \texttt{False}$, which only holds of finite values, are not monotonic and, therefore, cannot be continuous.

Another common approach to reasoning about the equivalence of semantically infinite values is via *bisimulation*, which argues coinductively that no finite sequence of observations can distinguish two expressions [53]. Taking constructors as the basis for observation works well for closed expressions, but an effective proof system must also be able to construct a bisimulation for all instances of open expressions where free variables might prevent observation. Consider, the expression $\texttt{leq}\ x\ x$ as an example. In order to make an observation, we must perform case analysis on the variable x but, under the substitution $x \mapsto \texttt{S}\ x'$, no progress can been made as the expression merely reduces to $\texttt{leq}\ x'\ x'$ without producing a constructor. Thus one must appeal to some form of induction in order to consider all ground instances. Alternatively, a proof system based on bisimulation could incorporate reduction steps as observations and thus make progress even on open expressions. However, it is then necessary to choose between *strong bisimulation* where expressions are equivalent only if they reduce in parallel, so that $\texttt{leq}\ x\ x = \texttt{True}$ does not hold, or *weak bisimulation* which doesn't support standard equational reasoning within proofs [54, 55, 56].

Chapter 3

Intensional Datatypes

A scalable approach to the pattern-matching safety problem

3.1 Introduction

The pattern-match safety problem is to determine whether a given program can crash due to a non-exhaustive pattern-matching expression. One solution is to use a coverage checker that ensures all pattern matches are exhaustive for the entire datatype, as in Definition 2.17 [57, 58]. This approach disregards the possibility that some omitted cases are never actually reached and forces the programmer to code defensively, writing code they know will never be executed or refactoring their datatypes accordingly.

Consider, for example, the program in Figure 3.2 that converts arbitrary propositional formulas, defined by the following datatypes, into disjunctive normal form (represented as a list of lists of literals).

```
data Lit α
  = Atom α
  | NegAtom α
```

```
data Fm α
  = Lit (Lit α)
  | And (Fm α) (Fm α)
  | Or (Fm α) (Fm α)
  | Not (Fm α)
```

Figure 3.1: Algebraic datatypes representing literals and propositional formulas.

The formula is first transformed into negation normal form by the `nnf` function, whereby any `Not` constructors are recursively eliminated from the formula, before being passed to the `nnf2dnf` function. Although the latter function uses an inexhaustive pattern-matching expression, i.e. it doesn't have a clause corresponding to the `Not` constructor, if `nnf` has been implemented correctly no pattern-matching error will occur at runtime, and it would be redundant to provide such a clause. As it hap-

pens, there is a bug (highlighted in the program) that can cause a pattern-matching error to occur at runtime. Instead of correctly deriving the negation normal form of the first conjunct, it is merely wrapped in the `Not` constructor and will ultimately reach the inexhaustive pattern-matching expression in `nnf2dnf`.

```
dnf : Fm α → List (List α)
dnf = nnf2dnf ∘ nnf

nnf2dnf : Fm α → List (List α)
nnf2dnf (Lit a) = [[a]]
nnf2dnf (And p q) =
  [ append xs ys,
    | xs ← nnf2dnf p,
      ys ← nnf2dnf q
    ]
nnf2dnf (Or p q) =
  append (nnf2dnf p) (nnf2dnf q)

nnf : Fm α → Fm α
nnf (Lit l) = Lit l
nnf (And p q) =
  And (nnf p) (nnf q)
nnf (Not (And p q)) =
  Or (Not (nnf p)) (nnf (Not q))
                ⋮
```

Figure 3.2: A program snippet that converts arbitrary propositional formulas to disjunctive normal form.

In this chapter, we present a refinement type system that can be used to automatically detect possible pattern-matching errors or verify their absence. With our compositional and incremental approach, type inference can easily be incorporated into modern development environments. In particular, this means that open program expressions can be analysed, and only the modified parts of the code need to be reanalysed as changes are made. Our prototype implementation exhibits excellent performance, processing a range of production-scale packages from the Hackage database in the order of seconds.

3.1.1 Intensional Datatype Refinements

Sound and terminating program analyses are often conservative — there are programs without bugs that are rejected. Nevertheless, it is helpful to provide a large fragment of safe programs for which the analysis is guaranteed to be correct as this gives the programmer the opportunity to take action, such as by programming more defen-

sively, in order to manipulate their program into this fragment and thus be certain of verification success. One such fragment that we have already seen is restricted to exhaustive case expressions; our type system provides a tighter-bound with the following features:

- The set of datatypes is "completed" by adding all datatypes obtained by removing constructors from datatypes provided by the programmer. The key restriction is that the removal of constructors applies recursively, giving rise to *intensional refinements*. For example, the type of formulas in negation normal form is an intensional refinement that intuitively corresponds to the following data declaration:

```
data NNF α
  = Lit (Lit α)
  | And (NNF α) (NNF α)
  | Or (NNF α) (NNF α)
```

Figure 3.3: An intensional refinement that is included in the completed environment.

  Notice how not only is the `Not` constructor absent from this datatype, but every recursive occurrence of a formula is also required to be in negation normal form. In this way, intensional refinements can be seen as inductive invariants.

- Unlike the original set of datatypes, constructors no longer belong to a single datatype. Therefore, there is a natural notion of subtyping between intensional refinements incorporated into the type system through an unrestricted subsumption rule. Subtyping allows for increased compositionality, e.g. a conjunction of literals can always be passed to a function expecting a formula in negation normal form.

- The typing rule for case expressions enforces that the patterns are exhaustive *with respect to* the refinement type ascribed to the scrutinee. This constraint ensures that the analysis of matching is sound. In particular, programs can omit branches of a case expression just if typing of the scrutinee indicates that they are unreachable.

- Moreover, the typing rule for case expressions is *path-sensitive* so that the returned type only depends on those branches that are indeed reachable according to the type of scrutinee. Although path-sensitivity foregoes principal typing, it allows for increased precision in exchange. Path-sensitivity is essential for handling typical use cases where a single large datatype is defined, but parts of the program only work within a given fragment. Such scenarios are particularly

common in Elm-style web applications where certain pages may only be prepared to handle a subset of events defined by a global datatype.

The following function, for example, can be assigned two refinement types, neither of which can be substituted for the other: $(\alpha \to \beta) \to \texttt{Fm}\ \alpha \to \texttt{Fm}\ \alpha$ as well as $(\alpha \to \beta) \to \texttt{NNF}\ \alpha \to \texttt{NNF}\ \alpha$.

```
fmMap f (Lit a) =
  Lit (litMap f a)
fmMap f (And p q) =
  And (fmMap f p) (fmMap f q)
fmMap f (Or p q) =
  Or (fmMap f p) (fmMap f q)
```

Figure 3.4: A function with no principal type.

To account for the lack of principal typings, our type system permits environments with more than one refinement type binding for each free program variable, i.e. an environment-level intersection, making it a *polyvariant analysis* [59]

3.1.2 Compositionality and Complexity

Our analysis takes the form of a type inference procedure for the system described above. As is typical, inference proceeds by generating and solving typing constraints rather than directly considering all possible typings. The constraints represent the flow of data and can be conditional on the presence of certain constructors in datatypes in order to account for path-sensitivity.

A key goal of our work is to give some guarantee of the scalability of the analysis to large, real-world programs. We do this by ensuring that type inference, i.e. constraint generation and constraint solving, is linear in the size of the program. This complexity guarantee is achieved by exploiting compositionality in the type inference algorithm.

Compositional program analysis computes a summary of the behaviour of each component separately, such as a set of constraints in solved form that restrict the safe instances of a type scheme. For a component f_i, however, the size of its summary depends not only on its own complexity but also the size of summaries for $f_1, \ldots, f_{i-1}$ on which it may depend. In effect, summaries are duplicated under polyvariant analyses and can lead to a whole program analysis that is exponential in the number of components [22]. Even for non-polyvariant analyses, where there is no duplication, summaries can be quadratic in the number of components and have a cubic time lower bound on their solvability [23, 24]. Since this blow-up can often occur in practice, there is an extensive literature on simplification techniques by which large summaries may sometimes be replaced by more concise equivalents [22, 60, 61].

However, these approaches tend to be heuristical meaning that there will always be reasonable programs for which the heuristics are not well-tuned[1].

By contrast, the summaries of our system are designed so that their complexity doesn't increase with the number of dependencies but is instead bounded by the size of the component's "interface", i.e. the number of datatype identifiers in its underlying type scheme. We can thus guarantee a (parameterised) linear-time complexity for the overall analysis, as our compositional approach ensures that each set of constraints is unrelated to the size of the program. This remarkable property follows from Theorem 3.12:

> Suppose C is a set of constraints in solved form and I is an arbitrary subset of its refinement variables. Let $C\restriction_I$, referred to as the *restriction* of C to I, be those constraints in C that only concern the variables from I. Then every solution to $C\restriction_I$ can be extended to a solution to C.

In the restriction $C\restriction_I$, entire constraints are culled, including those involving a mixture of variables from both I and $V \setminus I$, where V is the total set of refinement variables. Intuitively, these mixed constraints impose compatibility requirements on the different components. What is significant about the above property is that the solved form guarantees not only that the part of the constraints concerned with $V \setminus I$ are internally consistent but, moreover, that the mixed constraints will be satisfiable no matter which solution to $C\restriction_I$ is chosen.

Theorem 3.12 relies on the existence of a most-general solution to non-interface variables, i.e. $V \setminus I$, with respect to interface variables. Any constraint system with the property of having most-general solutions can be used to perform a whole program analysis that is linear in the number of components. Such systems form "cylindric algebras" which are Boolean algebras with an internal notion of projection — a semantic analogue of existential quantification [19]. This framework includes the standard Hindley-Milner type systems via equality constraints (whereby the most-general unifier provides an appropriate solution to non-interface variables) and its extension to Haskell-style typeclass constraints. Nevertheless, this property should not be overlooked as we are unaware of any path-sensitive, polyvariant analysis satisfying it; in our case, it is only achieved through the novel, intensional restriction on datatype refinements.

Implementation

It is worth noting that our complexity analysis relies on the assumption that other parameters, such as the size of underlying types and size of the program, has a fixed upper-bound. The processes of computing a solved form that is sufficient to guarantee

[1] Although the "constraint abstraction" technique guarantees a quadratic complexity, it is limited to simple variable-variable constraints [62].

the above property is not straightforward. Our constraint solver, which is inspired by the resolution-based approach used in set constraint based program analysis [20, 63] and optimal solutions to the HORNSAT problem [64, 65], is worst-case exponential time in the number of constructors and refinement variables appearing in the interface.

Of course asymptotic complexity is only part of the story, and especially so when the constant factors depend upon several assumptions. Hence, we have implemented our system as a core plugin for GHC and tested it on a selection of packages from the Hackage database. The plugin takes a Haskell module to be compiled and performs our type inference algorithm to output a function summary for each top-level definition, as well as any type errors encountered. The average time taken to process each module is in the order of seconds and the results show very stark contrast between the total number of refinement variables generated during type inference (often greater than 10000) and the number of refinement variables in the interfaces (typically less than 20). As the number of variables with respect to which saturation is computed significantly contributes to the complexity of saturation, the scale of this reduction illustrates the importance of our restriction result.

3.2 Declarative System

3.2.1 Intensional Datatype Environment

In this chapter, we assume a fixed datatype environment $\underline{\Delta}$, which we will refer to as the *underlying datatype environment* and think of as being specified by the programmer via datatype definitions. We also define $\underline{D} \subseteq \mathbb{D}$ as the set of datatype identifiers appearing in the type schemes of this environment. Throughout, we will use the following datatypes — a locally nameless representation of a λ-calculus with arithmetic constants — as a running example of the underlying datatype environment.

```
data Lam                data Arith
  = Cst Arith             = Lit Int
  | BVr Int               | Add
  | FVr String            | Mul
  | Abs Lam
  | App Lam Lam
```

Figure 3.5: Algebraic datatypes representing locally nameless λ-terms with arithmetic constants.

Within this chapter, other than the underlying environment, we will use a more general definition of datatype environments that can be relations between construc-

tors and type schemes, rather than mere functions, and likewise for the type environments. That is to say constructors and program variables may be equipped with several type schemes. This flexibility is important as constructors may belong to different refinements of the same underlying datatype and, as discussed in the preceding section, there are no longer principal types for functions defined within the program.

Datatype environments are, however, restricted so that each datatype identifier assigns unique argument types of its inhabiting constructors. Thus, first-order datatypes resemble *deterministic* top-down tree automata. It is through this constraint that we only consider intensional refinements, whereby refinements must be applied consistently to the entire datatype environment; in particular, recursive occurrences of a datatype identifier inherit the same refinement.

Definition 3.1. A datatype environment $\Delta \subseteq \mathbb{K} \times \mathsf{Sch}$ is a binary relation between constructors and closed type schemes such that:

- As usual, for each $k : \rho \in \Delta$, the type scheme ρ is of the form:

$$\forall \alpha_1 \cdots \alpha_{\mathrm{arity}(d)} . \tau_1 \to \cdots \to \tau_{\mathrm{arity}(k)} \to d\ \alpha_1\ \cdots\ \alpha_{\mathrm{arity}(d)}$$

- Moreover, the restricted datatype environment $\Delta(d) \subseteq \Delta$, i.e. those typings $k : \rho$ where the type scheme ρ ultimately returns an instance of the datatype identified $d \in \mathbb{D}$, is functional.

Definition 3.2. As relations between constructors and their type schemes, datatype environments are naturally ordered by inclusion written $\Delta_1 \subseteq \Delta_2$. In which case, we say Δ_1 is a *(intensional) refinement* of Δ_2.

```
data CloApp                 data LArith
  = Cst LArith                = Lit Int
  | App CloApp CloApp         | Add
```

Figure 3.6: Datatypes representing an intensional refinement of λ-terms and arithmetic constants.

As datatype environments must assign unique type schemes to constructors for each given datatype identifier, the refinements of a given environment are determined just by the set of constructors associated with a given datatype identifier. More precisely, each function $\phi : \mathbb{D} \to \mathscr{P}(\mathrm{dom}(\Delta))$ maps a datatype identifier $d \in \mathbb{D}$ to a subset of constructors from $\mathrm{dom}(\Delta(d))$ determines an intensional refinement $\Delta_\phi \subseteq \Delta$ containing $k : \sigma \in \Delta_\phi$ just if $k : \sigma \in \Delta(d)$ and $k \in \phi(d)$. We write $\mathsf{Refine}(\Delta)$ for the set of such refinement functions. Consider, for example, the intensional refinement of our underlying environment $\underline{\Delta}$ presented in Figure 3.6 that describes closed,

applicative λ-terms with linear arithmetic. This datatype environment is determined precisely by a function ϕ_1 mapping the datatype identifier `Lam` to the set $\{\texttt{Cst}, \texttt{App}\}$ and, likewise, mapping `Arith` to the set $\{\texttt{Lit}, \texttt{Add}\}$.

For most typical programs, no one refinement environment will be universally applicable. Therefore, programs are assigned types under an expanded datatype environment consisting of all possible intensional refinements of the underlying environment.

Definition 3.3. For each underlying datatype identifier $d \in \underline{D}$ and refinement function $\phi \in \mathsf{Refine}(\Delta)$, we assume there is an *intensional datatype identifier*, written $\mathrm{inj}_\phi(d) \in \mathbb{D}$, that is not an element of $\underline{D}$. We write $D^* \subseteq \mathbb{D}$ for the set of intensional datatype identifiers and, likewise, we write Dt^* for the set of derived *intensional datatypes*. These datatype identifiers inherit their arity from the underlying datatype identifier, i.e. $\mathrm{arity}(\mathrm{inj}_\phi(d)) = \mathrm{arity}(d)$.

To help differentiate intensional datatype identifiers from their underlying counterparts, we will use $\underline{d}$ for elements of $\underline{D}$ and d for elements of D^*. Likewise, we will use $\underline{\tau}$ to range over those types with underlying datatype identifiers $\mathsf{Ty}(\underline{D})$, in contrast to τ, which ranges over types constructed from intensional datatype identifiers $\mathsf{Ty}(D^*)$. These types (or type schemes) are referred to as *refinement types*, and we refer to type environments with intensional datatype identifiers as *refinement environments*.

Definition 3.4. The *intensional refinement environment* Δ^*, is the coproduct of all intensional refinements of the underlying environment:

$$\Delta^* \coloneqq \bigsqcup_{\phi \in \mathsf{Refine}(\underline{\Delta})} \left\{ k : \mathrm{inj}_\phi(\rho) \mid k : \rho \in \underline{\Delta}_\phi \right\}$$

where $\mathrm{inj}_\phi(\rho)$ denotes the type scheme derived from ρ by replacing all underlying datatype identifier with their intensional refinement.

The intensional refinement environment contains a distinguished datatype identifier for every intensional refinement. For example, the aforementioned refinement of λ-terms to closed, applicative terms over linear arithmetic occurs within Δ^* as $\mathrm{inj}_{\phi_1}(\underline{\texttt{Lam}})$ alongside the type of closed, applicative λ-terms with arbitrary arithmetic which is represented by $\mathrm{inj}_{\phi_2}(\underline{\texttt{Lam}})$ under the definition $\phi_2(\underline{\texttt{Lam}}) = \{\texttt{Cst}, \texttt{App}\}$ and $\phi_2(\underline{\texttt{Arith}}) = \{\texttt{Lit}, \texttt{Add}, \texttt{Mul}\}$. Importantly, both these types can be used within the same program under our refinement type system even though they are derived from distinct intensional refinements. We shall write `CloAppLin` and `CloApp` to denote the datatypes $\mathrm{inj}_{\phi_1}(\underline{\texttt{Lam}})$ and $\mathrm{inj}_{\phi_2}(\underline{\texttt{Lam}})$ respectively. Note, that underlying datatypes are also reflected in the intensional datatype environment by the trivial refinement ϕ_*,

which maps each $\underline{d} \in \underline{D}$ to the complete set of constructors $\mathrm{dom}(\underline{\Delta}(\underline{d}))$. Thus, we will sometimes identify $\underline{d} \in \underline{D}$ with its trivial refinement $\mathrm{inj}_{\phi_*}(\underline{d}) \in D^*$.

Definition 3.5. For a refinement type or type scheme τ, the corresponding *underlying* type or type scheme, written $\mathcal{U}(\tau)$ is an underlying type defined recursively as follows:

$$\begin{aligned} \mathcal{U}(\forall\alpha.\,\rho) &\coloneqq \forall\alpha.\mathcal{U}(\rho) \\ \mathcal{U}(\tau_1 \to \tau_2) &\coloneqq \mathcal{U}(\tau_1) \to \mathcal{U}(\tau_2) \\ \mathcal{U}(\mathrm{inj}_\phi(\underline{d})\ \tau_1\ \cdots\ \tau_n) &\coloneqq \underline{d}\,\mathcal{U}(\tau_1)\ \cdots\ \mathcal{U}(\tau_n) \end{aligned}$$

In subsequent sections, we will assume that we are given a program equipped with a complete underlying typing; that is, every sub-expression is associated with a unique underlying type. Our task will be to find refinement types for expressions and a refinement environment for the program, whose underlying types coincide with the original.

3.2.2 Subtyping Relation

As we have already seen, datatype environments can be ordered by the set of constructors they support. In the intensional refinement environment, the combination of datatypes from different intensional refinements gives rise to a more substantial notion of refinement between individual types.

$$\textsf{(SShape)}\ \frac{}{\tau \not\sqsubseteq \sigma}\ \mathcal{U}(\tau) \neq \mathcal{U}(\sigma)$$

$$\textsf{(SMis)}\ \frac{}{d_1\ \overline{\tau} \not\sqsubseteq d_2\ \overline{\sigma}}\ \mathrm{dom}(\Delta^*(d_1)) \not\subseteq \mathrm{dom}(\Delta^*(d_2))$$

$$\textsf{(SSim)}\ \frac{\Delta^*(d_1)(k)_i[\overline{\tau/\alpha}] \not\sqsubseteq \Delta^*(d_2)(k)_i[\overline{\sigma/\alpha}]}{d_1\ \overline{\tau} \not\sqsubseteq d_2\ \overline{\sigma}}\ \begin{array}{l} k \in \mathrm{dom}(\Delta^*(d_1)) \\ i \leq \mathrm{arity}(k) \end{array}$$

$$\textsf{(SArrL)}\ \frac{\tau_2 \not\sqsubseteq \tau_1}{\tau_1 \to \sigma_1 \not\sqsubseteq \tau_2 \to \sigma_2} \qquad \textsf{(SArrR)}\ \frac{\sigma_1 \not\sqsubseteq \sigma_2}{\tau_1 \to \sigma_1 \not\sqsubseteq \tau_2 \to \sigma_2}$$

Figure 3.7: The complement of the subtyping relation.

Definition 3.6 (Subtyping)**.** The subtyping relation $\tau \sqsubseteq \sigma$ is defined coinductively via its complement, which is characterised inductively by the inference rules in Figure 3.7. For type schemes, we define $\forall\overline{\alpha}.\,\tau \sqsubseteq \forall\overline{\alpha}.\,\sigma$ simply as $\tau \sqsubseteq \sigma$.

Furthermore, we write $\Gamma \sqsubseteq \Gamma'$ when, for all $x : \rho \in \Gamma'$, there exists $x : \rho \in \Gamma$ such that $\rho \sqsubseteq \rho'$ (recall that refinement type environments may contain several types associated with a single variable).

Lemma 3.1. Subtyping is a preorder, i.e. it is reflexive and transitive.

We give the definition of subtyping coinductively because, as usual, there is a notion of simulation that arises naturally from our coalgebraic view of datatype environments. Consequently, it is most straightforward to think of the inference rules as constructing finite refutations of subtype inequalities $\tau_1 \not\sqsubseteq \tau_2$, which ultimately fail to hold either because the types τ_1 and τ_2 have a different underlying shape, or because τ_1 provides some constructor that τ_2 does not. Returning to our running example, the fact that $\texttt{CloAppLin} \to \texttt{String}$ is not a subtype of $\texttt{CloApp} \to \texttt{String}$ can be witnessed by the following derivation that relies on $\texttt{Mul} \in \mathsf{inj}_{\phi_1}(\underline{\texttt{Arith}}) \setminus \mathsf{inj}_{\phi_2}(\underline{\texttt{Arith}})$.

$$\dfrac{\dfrac{\dfrac{}{\phi_1(\underline{\texttt{Arith}}) \not\sqsubseteq \phi_2(\underline{\texttt{Arith}})}\,(\mathsf{SMis})}{\mathsf{inj}_{\phi_1}(\underline{\texttt{Arith}}) \not\sqsubseteq \mathsf{inj}_{\phi_2}(\underline{\texttt{Arith}})}\,(\mathsf{SSim})}{\texttt{CloApp} \not\sqsubseteq \texttt{CloAppLin}}\,(\mathsf{SArrL})$$
$$\texttt{CloAppLin} \to \texttt{String} \not\sqsubseteq \texttt{CloApp} \to \texttt{String}$$

Conversely, we can use the coinduction principle to show that $\tau_1 \sqsubseteq \tau_2$ by constructing a *model* of the subtyping relation that contains the pair of types in question, i.e. a set of pairs whose complement satisfies all defining rules from Figure 3.7. For example, the subtyping $\texttt{CloApp} \to \underline{\texttt{String}} \sqsubseteq \texttt{CloAppLin} \to \underline{\texttt{String}}$ is witnessed by the following model:

$$\left\{ \begin{array}{l} (\texttt{CloApp} \to \underline{\texttt{String}},\ \texttt{CloAppLin} \to \underline{\texttt{String}}), \\ (\texttt{CloAppLin},\ \texttt{CloApp}), (\underline{\texttt{String}},\ \underline{\texttt{String}}), \\ (\mathsf{inj}_{\phi_2}(\underline{\texttt{Arith}}),\ \mathsf{inj}_{\phi_1}(\underline{\texttt{Arith}})),\ (\underline{\texttt{Int}},\ \underline{\texttt{Int}}) \end{array} \right\}$$

Such models quickly become unwieldy when the types increase in complexity as they contain redundant information. For example, the first pair of types ($\texttt{CloApp} \to \underline{\texttt{String}}$, $\texttt{CloAppLin} \to \underline{\texttt{String}}$) is already implied by the two subsequent pairs ($\texttt{CloAppLin}$, $\texttt{CloApp}$) and ($\underline{\texttt{String}}$, $\underline{\texttt{String}}$) as a result of the rules for subtyping function arrows. We can improve this situation by observing that the definition of a model is approximated by a coinductive part, concerning just datatypes, and an inductive part, by which a subtyping relationship between datatypes is then lifted to all types. Consequently, we need only find a model of the coinductive part. In Lemma 3.2, we formalise this argument by lifting of a well-behaved relation between datatypes to a model of subtyping. As a result, we can derive the aforementioned subtyping by exhibiting the simpler relation: $\{(\mathsf{inj}_{\phi_2}(\underline{\texttt{Arith}}),\ \mathsf{inj}_{\phi_1}(\underline{\texttt{Arith}})),\ (\texttt{CloAppLin},\ \texttt{CloApp})\}$.

$$\frac{}{\mathsf{Ty}(R)(\alpha,\, \alpha)} \qquad \frac{}{\mathsf{Ty}(R)(d_1\ \overline{\tau},\, d_2\ \overline{\sigma})}\,(d_1\ \overline{\tau},\, d_2\ \overline{\sigma}) \in R$$

$$\frac{\mathsf{Ty}(R)(\tau',\, \tau) \quad \mathsf{Ty}(R)(\sigma,\, \sigma')}{\mathsf{Ty}(R)(\tau \to \sigma,\, \tau' \to \sigma')}$$

Figure 3.8: Inductively defined rules for simulation.

Definition 3.7. Let $R \subseteq \mathsf{Dt}^* \times \mathsf{Dt}^*$ be a binary relation on intensional datatypes. Then its *lifting* to types, written $\mathsf{Ty}(R)$, is defined inductively by the inference rules in Figure 3.8.

Lemma 3.2 (Simulation)**.** Suppose $R \subseteq \mathsf{Dt}^* \times \mathsf{Dt}^*$ is a binary relation on intensional datatypes such that, for any pair of intensional datatypes $(d_1\ \overline{\tau},\, d_2\ \overline{\sigma}) \in R$ and constructor $k \in \mathrm{dom}(\Delta^*(d_1))$, the following properties hold:

1. $k \in \mathrm{dom}(\Delta^*(d_2))$
2. $\mathcal{U}(d_1\ \overline{\tau}) = \mathcal{U}(d_2\ \overline{\sigma})$.
3. $\mathsf{Ty}(R)(\Delta(d_1)(k)[\overline{\tau/\alpha}]_i,\, \Delta(d_2)(k)[\overline{\sigma/\alpha}]_i)$ for all $i \leq \mathrm{arity}(k)$.

Then it follows that $\mathsf{Ty}(R)$ is included in the subtyping relation.

3.2.3 Refinement Type System

In this section, we present a refinement type system whose purpose is to exclude the possibility of pattern-matching errors. To achieve this, the typing rule for pattern-matching expressions requires that cases are exhaustive according to the type of the scrutinised expression. However, the use of intensional refinements means that the scrutinised expression can be given a more precise type than is possible in the underlying type system. Its refinement type allows the system to safely conclude that it never evaluates to certain constructors, i.e. those not included according to its refinement function. Thus, the case expression can be deemed exhaustive despite constructors from the underlying type not being handled, distinguishing this system from exhaustivity as defined in Definition 2.17.

As our generalisation of type environments allow for multiple types, top-level definitions can be given several type schemes, which is equivalent to allowing environment-level intersection types. However, all types assigned to a given variable will refine the same underlying type.

Definition 3.8. The type assignment system $\Gamma \Vdash e : \tau$ is defined inductively by the inference rules in Figure 3.9. Note that these derivations ensure that $\mathcal{U}(\Gamma) \vdash e : \mathcal{U}(\tau)$, where $\mathcal{U}(\cdot)$ distributes over environments in the obvious way.

$$\text{(TVar)}\ \frac{}{\Gamma \Vdash x : \tau[\overline{\sigma/\alpha}]}\, x : \forall\overline{\alpha}.\,\tau \in \Gamma \qquad \text{(TSub)}\ \frac{\Gamma \Vdash e : \tau}{\Gamma \Vdash e : \sigma}\, \tau \sqsubseteq \sigma$$

$$\text{(TCon)}\ \frac{}{\Gamma \Vdash k : \tau[\overline{\sigma/\alpha}]}\, k : \forall\overline{\alpha}.\,\tau \in \Delta^*(d)$$

$$\text{(TAbs)}\ \frac{\Gamma \cup \{x : \tau\} \Vdash e : \sigma}{\Gamma \Vdash \lambda x.\, e : \tau \to \sigma} \qquad \text{(TApp)}\ \frac{\Gamma \Vdash e_1 : \tau \to \sigma \quad \Gamma \Vdash e_2 : \tau}{\Gamma \Vdash e_1\ e_2 : \sigma}$$

$$\text{(TCase)}\ \frac{\Gamma \Vdash e : d\,\overline{\tau} \quad (\forall k_i \in \mathrm{dom}(\Delta^*(d)))\ \Gamma \cup \Gamma_i \Vdash e_i : \sigma}{\Gamma \Vdash \texttt{case}\ e\ \texttt{of}\ \{k_i\ \overline{x_i} \mapsto e_i \mid i \leq n\} : \sigma}$$
$$\text{where } \mathrm{dom}(\Delta^*(d)) \subseteq \{k_1, \ldots, k_n\}$$
$$\text{and } \Gamma_i = \{x_i : \Delta^*(d)(k_i)[\overline{\tau/\alpha}]\}$$

Figure 3.9: Refinement typing rules for expressions.

The system is conceptually similar to the underlying Hindley-Milner style system presented in Chapter 2, but note the following deviations:

- Any suitable intensional refinement of a datatype identifier, in accordance with the intensional environment, can be used in order to type a constructor or the scrutinee of a case statement.

- The notion of subtyping from the previous section is incorporated through a subsumption rule.

- Pattern-matching expressions must be exhaustive in order to guarantee that programs cannot lead to a pattern-matching error at runtime. Nevertheless, this restriction is with respect to the refinement type of the scrutinee and thus is distinct from our earlier definition of exhaustivity.

- The rule (TCase) not only requires that the branches are exhaustive but only considers those that are reachable, i.e. when the constructor of the corresponding pattern is in the datatype of the scrutinee. That is to say, unreachable branches can safely be ignored and do not contribute to the type of the case expression. This relaxation incorporating path-sensitivity only makes sense for a refinement-type system because reachability is encoded by choosing an appropriate refinement for the scrutinee.

Consider, as an example, the function in Figure 3.10 that applies a substitution to a λ-term and has underlying type $\underline{\texttt{List}}\ (\underline{\texttt{String}} \times \underline{\texttt{Lam}}) \to \underline{\texttt{Lam}} \to \underline{\texttt{Lam}}$, assuming that `lookup` has underlying type $\forall\alpha.\, \underline{\texttt{List}}\ (\underline{\texttt{String}} \times \alpha) \to \underline{\texttt{String}} \to \alpha$[2].

```
subst theta t =
  case t of
    FVr x → lookup theta x
    Cst c → Cst c
    App u v →
      App (subst theta u) (subst theta v)
```

Figure 3.10: Function performing substitution on closed λ-terms.

Recall that `ClosApp` denotes closed, applicative terms, i.e. the subtype of `Lam` without the `FVr`, `Abs`, and `BVr` constructors. Let us also define `AppLam` as the subtype of `Lam` datatype under the intensional refinement that removes the `Abs` and `BVr` constructors but retains `Cst`, `FVar`, and `App`. Then the refinement type $\underline{\texttt{List}}\ (\underline{\texttt{String}} \times \texttt{ClosApp}) \to \texttt{AppLam} \to \texttt{ClosApp}$ can be assigned to this function, encoding the fact that the resulting expression is also closed when a substitution replaces all free variables with a closed expression. This refinement typing is possible due to a combination of the features of the system. First, the abstraction rule can assume the bound variable `theta` has the refinement type $\underline{\texttt{List}}\ (\underline{\texttt{String}} \times \texttt{ClosApp})$. It then follows that the sub-expression `lookup theta` x can be given the refinement type `ClosApp`. Second, the rule (TCase) is applicable only because we have chosen a refinement of the scrutinee without the `Abs` and `BVr` constructors and thus the pattern-matching expression is exhaustive. Then, in the body of the case expression, u and v are also assigned the type `ClosApp`, as intensional refinements apply recursive throughout a datatype, so that the sub-expressions `subst theta` u and `subst theta` v can be assigned the type `ClosApp` as required.

Definition 3.9. The central problem in which we are interested is the typeability under the aforementioned system of a program P, which is assumed to be well-typed with respect to some underlying program environment $\underline{\Sigma}$ but is *not* necessarily exhaustive under Definition 2.17. This involves finding a closed refinement environment Σ such that, for each $f : \forall\overline{\alpha}.\, \tau \in \Sigma$, we can derive $\Sigma \Vdash P(f) : \tau$. When such an environment exists, we say that $(\underline{\Delta}, \underline{\Sigma}, P)$ constitutes a positive instance of the *refinement typeability problem*.

The following theorem shows that a positive instance of the refinement typeability is sufficient to ensure that datatype expressions do not fail to evaluate and, therefore,

[2]In reality, no exhaustive function of the aforementioned type exists and such a function would more typically wrap its return type in a failure or "option" monad. We omit this detail from our example for the sake of simplicity.

there are no reachable pattern-matching failures. Conceptually, this proof is similar to Lemma 2.9 but relies on our refinement type system instead of the coarser notion of exhaustivity.

Theorem 3.3. Suppose $(\underline{\Delta}, \underline{\Sigma}, P)$ is a positive instance of the refinement typeability problem witnessed by the refinement environment Σ. Then the normal form $a{\downarrow_P}$ of an applicative expression $\Sigma \Vdash a : d\ \tau_1\ \cdots\ \tau_n$ is necessarily of the form $k\ a_1\ \cdots\ a_{\mathrm{arity}(k)}$ for some $k \in \mathrm{dom}(\Delta^*(d))$.

As discussed in the introduction, allowing several types for each expression ensures they can be used in different contexts. This approach is more lightweight than a full intersection type system as intersections are not permitted under function arrows, and arguably easier for programmers to reason about if types are to be considered as specifications. When it comes to algorithmic inference, however, the nondeterministic aspects of this system are problematic. Instead of enumerating possible types, we rely on refinement polymorphism to summarise every possible typing of a program variable compactly by a single constrained type scheme. Polymorphism of this kind is no different from that of the Hindley-Milner system, which could equally be viewed as an infinite intersection type system, or indeed allowing several typings of the same variable in an environment. The rest of this chapter concerns the algorithmic solution to aforementioned typeability problem.

3.3 Algorithmic System

3.3.1 Constructor Set Constraints

We assume a countable set of *refinement variables* $\mathbb{X}$, ranged over by X, Y, Z etc. The purpose of these refinement variables is to represent an unknown refinement function $\phi \in \mathrm{Refine}(\underline{\Delta})$, in the same way that a type variable represents an unknown type. Notice that these refinements functions are in 1–1 correspondence with intensional refinements of $\underline{\Delta}$. By abuse of notation, we will treat $X \in \mathbb{X}$ as both a refinement function and the corresponding datatype environment.

Definition 3.10. A *constructor set expression*, typically S, is either a finite set of constructors $\{k_1, \ldots, k_n\}$ or a pair $X(\underline{d})$ consisting of a refinement variable $X \in \mathbb{X}$ and an underlying datatype identifier $\underline{d} \in \underline{D}$. The underlying datatype identifier of a set expression is defined as follows:

$$\mathcal{U}(X(\underline{d})) \coloneqq \underline{d} \qquad\qquad \mathcal{U}(\{k_1, \ldots, k_n\}) \coloneqq \underline{d} \quad \text{where } \forall i \leq n.\, k_i \in \mathrm{dom}(\underline{\Delta}(d))$$

We only consider those constructor set expression for which the underlying datatype identifier is well-defined, i.e. we do not consider sets of constructors belonging to different underlying datatypes.

Definition 3.11. An *inclusion constraint* is a pair of constructor set expressions with the same underlying datatype identifier, written $S_1 \subseteq S_2$. When S_1 is a singleton $\{k\}$, we will write the constraint as $k \in S_2$ instead.

A *conditional constraint*, hereafter just *constraint*, is a pair $\phi \mathbin{?} S_1 \subseteq S_2$ consisting of a set of inclusion constraints ϕ, referred to as the *guard*, and an inclusion constraint $S_1 \subseteq S_2$, referred to as the *body*. We will only consider conditional constraints in which each element of the guard is of the form $k \in X(\underline{d})$. When the guard is trivial, i.e. the empty set, we shall omit it and just write the body. Furthermore, we shall sometimes guard a set of constraints C, writing $\phi \mathbin{?} C$ to denote the set of constraints of the form $\phi \cup \psi \mathbin{?} S_1 \subseteq S_2$ where $\psi \mathbin{?} S_1 \subseteq S_2 \in C$.

Definition 3.12. The *free refinement variables* of a set expression, constraint, or set of constraints $\mathsf{FRV}(C) \subseteq \mathbb{X}$ is defined recursively as follows:

$$\mathsf{FRV}(X(\underline{d})) \coloneqq \{X\} \qquad \mathsf{FRV}(\{k_1, \ldots, k_n\}) \coloneqq \emptyset$$

$$\mathsf{FRV}(S_1 \subseteq S_2) \coloneqq \mathsf{FRV}(S_1) \cup \mathsf{FRV}(S_2)$$

$$\mathsf{FRV}(\phi \mathbin{?} S_1 \subseteq S_2) \coloneqq \mathsf{FRV}(\phi) \cup \mathsf{FRV}(S_1) \cup \mathsf{FRV}(S_2)$$

$$\mathsf{FRV}(C) \coloneqq \bigcup_{\phi ? S_1 \subseteq S_2 \in C} \mathsf{FRV}(\phi \mathbin{?} S_1 \subseteq S_2)$$

Constraints restrict the possible datatype environments that a refinement variable can represent. Intuitively, an inclusion $S_1 \subseteq S_2$ implies that the constructors denoted by S_1 are included in the constructors denoted by S_2, and this is lifted to conditional constraints in the obvious manner. In particular, the constraint $k \in X(\underline{d})$ implies that whatever refinement function is assigned to X must include the constructor k, i.e. an expression of the type $\mathrm{inj}_X(\underline{d})$ may evaluate to the constructor k, and the constraint $X(d) \subseteq \{k_1, \ldots, k_n\}$ limits the refinement function so that X can only include these constructors, i.e. an expression of the type $\mathrm{inj}_X(\underline{d})$ may *not* evaluate to some constructor $k \notin \{k_1, \ldots, k_n\}$.

Definition 3.13. A *constructor set assignment*, hereafter just *assignment*, is a total map $\theta : \mathbb{X} \to \mathsf{Refine}(\underline{\Delta})$ from refinement variables to refinement functions. Constructor set expressions are interpreted by such an assignment as follows:

$$\theta[\![X(\underline{d})]\!] \coloneqq \theta(X)(\underline{d}) \qquad \theta[\![\{k_1, \ldots, k_n\}]\!] \coloneqq \{k_1, \ldots, k_n\}$$

We write $\theta_1 \equiv_U \theta_2$, and say that θ_1 and θ_2 are equivalent on the set $U \subseteq \mathbb{X}$, just if $\theta_1(X) = \theta_2(X)$ for each $X \in U$.

Definition 3.14. An inclusion constraint $S_1 \subseteq S_2$ is said to be *satisfied* by an assignment θ, written $\theta \vDash S_1 \subseteq S_2$, just if $\theta[\![S_1]\!]$ is included in $\theta[\![S_2]\!]$. Furthermore, a conditional constraint $\phi \,?\, S_1 \subseteq S_2$ is said to be satisfied by an assignment, written $\theta \vDash \phi \,?\, S_1 \subseteq S_2$, just if $\theta \vDash S_1 \subseteq S_2$ or there exists some inclusion constraint $k \in X(d) \in \phi$ in the guard such that $\theta \nvDash k \in X(d)$.

Definition 3.15. A *solution* to a set of constraints C is an assignment θ that satisfies every constraint in C. In which case we write $\theta \vDash C$. A set of constraints is said to be *solvable* or *satisfiable* whenever it has a solution.

The full set constraint language is known to correspond to the monadic class of first-order propositions [66]. By applying the translation of that paper, it can be shown that guarded constraints of the form laid out above are (monadic) Horn clauses where constructors are interpreted as constants. This observation will become relevant later when we discuss the process by which the solvability of a set of constraints is determined.

3.3.2 Type Inference System

Since our system is effectively syntax directed, the subsumption rule can be factored into the other syntax-directed rules and type inference follows a standard pattern of constraint generation and satisfiability checking (see e.g. [19]). The constraints are derived from subtype inequalities between refinement types, which, as a result of Lemma 3.2, can be reduced to conditional inclusion constraints between refinement variables and sets of datatype constructors. To enable this approach, we extend the language of types so as to allow datatypes parameterised by refinement variables.

Definition 3.16. The *extended types* are types where datatype identifiers are replaced by intensional refinements parameterised by a refinement variable:

$$\tau, \sigma ::= \cdots \mid \mathrm{inj}_X(\underline{d})\ \overline{\tau}$$

with $X \in \mathbb{X}$. Note that the arguments to an intensional datatype identifier are also extended types. Therefore, types such as $\mathrm{inj}_X(\texttt{List})\ \mathrm{inj}_Y(\texttt{String})$ are well-formed.

Assignments extend to extended types homomorphically, where $\tau\theta$ is the unextended type that results from replacing $\mathrm{inj}_X(\underline{d})$ by $\mathrm{inj}_{\theta(X)}(\underline{d})$. Likewise, we define $\mathsf{FRV}(\tau) \subseteq \mathbb{X}$ as the set of refinement variables occurring in the type τ in the obvious manner.

Recall that refinement datatype identifiers are of the form $\mathrm{inj}_\phi(\underline{d})$ and should be thought of as specifying the refinement of $\underline{d} \in \underline{D}$ whose datatype definition is given by the refinement function ϕ. The task of inference is to constrain these refinement

functions so that the expression in question is typeable under any satisfying assignment. If the constraints are solvable, there exists appropriate refinement functions with respect to which the program can be deemed well-typed.

Definition 3.17. A *constrained type scheme* ρ has the form $\forall\overline{\alpha}.\,\forall\overline{X}.\,C \supset \tau$ where C is a set of constraints such that $\mathsf{FRV}(C) \subseteq \overline{X}$ and τ is an extended type such that $\mathsf{FRV}(\tau) \subseteq \overline{X}$; that is, constrained type schemes don't contain any free refinement variables. When C is the empty set, we will just write $\forall\overline{\alpha}.\,\forall\overline{X}.\,\tau$.

Typically, there is not a most general instance of a constrained type scheme. By which we mean, there is not a single assignment to the constrained type scheme's refinement variables that satisfies its constraints and for which all other satisfying instances are supertypes. For example, consider the following recursive function that acts as the identity on the `Lam` datatype:

```
lamId (Cst a) = Cst a
lamId (BVr x) = BVr x
lamId (FVr x) = FVr x
lamId (Abs u) = Abs (lamId u)
lamId (App u v) =
  App (lamId u) (lamId v)
```

Figure 3.11: The identify function on λ-terms.

Let us suppose this function is assigned a constrained type scheme of the form $\forall XY.\,C \supset \mathsf{inj}_X(\texttt{Lam}) \to \mathsf{inj}_Y(\texttt{Lam})$. The constraints appearing in C must ensure that, if the input type contains the `Cst` constructor, then the output type also contains the `Cst` constructor. Thus, the constraint $\texttt{Cst} \in X(\texttt{Lam})\ ?\ \texttt{Cst} \in Y(\texttt{Lam})$ will appear in C and likewise for each other constructor of the `Lam` datatype. Equivalently, we may assert that $X(\texttt{Lam}) \subseteq Y(\texttt{Lam})$.

Informally speaking, the most general instances of a constrained type scheme must balance the assignment to refinement variables appearing in covariant and contravariant positions with respect to subtyping. Types are more general when their contravariant refinement variables (e.g. an input to a function type) are assigned more constructors, but they are less general when their covariant refinement variables (e.g. the output of a function type) are assigned more constructors. In the above example, the refinement variable X appears in a contravariant position and Y appears in a covariant position and, therefore, the most general type must maximise the assignment to X whilst minimising the assignment to Y. However, this requirement is at odds with the inclusion constraint $X(\texttt{Lam}) \subseteq Y(\texttt{Lam})$ — a more general input type must have a less general output type in order to satisfy this constraint, and likewise a more general output type requires a less general input type. For example, both the types

$\texttt{ClosApp} \to \texttt{ClosApp}$ and $\texttt{AppLam} \to \texttt{AppLam}$ are valid instances of the constrained type scheme but, whilst the latter is more general with regards to the arguments it can be passed, its output type contains more constructors and thus can be passed to fewer functions. As a result, there is no universally most general type.

Definition 3.18. A *constrained type environment* Γ is a function from variables to constrained type schemes, for which we write $f : \rho \in \Gamma$ as usual where ρ is a constrained type scheme.

For such an environment, $\mathsf{FRV}(\Gamma) \subseteq \mathbb{X}$ is defined as the set of refinement variables appearing in any type in the constrained type environment. Note that this set will be empty for the program environment but may contain refinement variables associated with locally bound variables. Additionally, we define $\Gamma\theta$ as the refinement type environment whose types result from instantiating any free refinement variables appearing in Γ according to the assignment θ.

Note that constrained type environments are functional, i.e. there is only one constrained type scheme assigned to each variable. Although there is no best solution to a set of constraints, constrained type schemes give us an internal representation of the set of *all* possible type schemes; hence it is sufficient to have a single constrained type scheme for each program variable. Constrained program environments can be understood as compact descriptions of an intersection of ordinary program environments, which is made precise by the following definition.

Definition 3.19. For a constrained type environment Γ, we define $(\!|\Gamma|\!)$ as the refinement environment obtained by instantiating every constrained type scheme with all of its solution:

$$\left\{x : \forall\overline{\alpha}.\, \tau\theta \mid \forall\overline{\alpha}.\, \forall\overline{X}.\, C \supset \tau \in \Gamma,\, \theta \vDash C\right\}$$

For the purpose of defining type inference algorithmically, we will assume programs are given as a finite sequence of function definitions with their underlying type $f_1 : \underline{\rho_1} = e_1,\ \ldots,\ f_n : \underline{\rho_n} = e_n$ such that each e_i can only use a program variable f_j if $i \leq j$, i.e. function definitions are topologically sorted by their dependencies. A constrained type environment for the program can thus be inferred incrementally, with recursive calls being handled standardly by forbidding polymorphic recursion. For simplicity, we do not permit mutually recursive functions as these can be simulated by direct recursion. In the implementation, however, several function definitions can belong to the same recursive group.

Typical presentations of type inference by constraint generation involve choosing fresh type variables, which are then constrained. Since we work with refinement types, it is more convenient to choose fresh *refinement type templates*, which are refinement types that are parameterised everywhere by fresh refinement variables. This construction is possible in the setting of a refinement type system as, at the point at

which inference would choose a fresh type, the underlying shape of the type is already known. In other words, a type is said to be fresh if each occurrence of a free refinement variables is fresh in the sense that they are not already in scope or elsewhere in the type.

Definition 3.20. We write $\mathsf{Fresh}(X)$ to assert that X is a refinement variable that is not already in scope, e.g. in the free refinement variables of the local typing environment. This predicate is extended to types by $\mathsf{Fresh}_{\underline{\tau}}(\tau)$ which is defined recursively as follows:

- For all $\alpha \in \mathbb{A}$, $\mathsf{Fresh}_{\alpha}(\alpha)$.
- For any $d \in \underline{D}$, $\mathsf{Fresh}_{d}(\mathsf{inj}_X(d)\ \overline{\tau})$ just if $\mathsf{Fresh}(X)$ and, for each $\mathsf{Fresh}_{\underline{\tau_i}}(\tau_i)$ for each type-level argument.
- Finally, $\mathsf{Fresh}_{\underline{\tau}\to\underline{\sigma}}(\tau \to \sigma)$ just if $\mathsf{Fresh}_{\underline{\tau}}(\tau)$ and $\mathsf{Fresh}_{\underline{\sigma}}(\sigma)$.

We will also write $\mathsf{Fresh}_{\overline{\underline{\tau}}}(\overline{\tau})$ to assert that a sequence of types are fresh.

Definition 3.21. Constrained type inference is split into three components:

Subtyping (Figure 3.12) For a given pair of extended types τ and σ, the judgement $H \Vdash \tau \sqsubseteq \sigma \Longrightarrow C$ is used to infer a set of constraints C that imply $\tau\theta \sqsubseteq \sigma\theta$ for any satisfying assignment θ. Here H, referred to as the *history*, is a set of pairs of the form $(\mathsf{inj}_X(\underline{d})\ \overline{\tau},\ \mathsf{inj}_Y(\underline{d})\ \overline{\sigma})$ that is used to track which subtyping relations have already been covered by the inferred constraints, capturing the coinductive nature of subtyping. Algorithmically, these parameters are used to prevent loops occurring when inferring constraints between recursive datatypes as we will discuss below. We simply write $\Vdash \tau \sqsubseteq \sigma \Longrightarrow C$ when the history is empty.

Expressions (Figure 3.13) In the context of a constrained environment Γ, and an underlying type $\underline{\tau}$, the judgement $\Gamma \Vdash e : \underline{\tau} \Longrightarrow \tau,\ C$ algorithmically infers an extended type τ and constraints C that describes the assignments under which the expression is typeable.

The underlying type is used to generate fresh, extended types in the case of (IAbs) and (ICase) and to determine the type parameters of program variables and constructors. We will sometimes omit the underlying type from the judgement when it is superfluous, simply writing $\Gamma \Vdash e \Longrightarrow \tau,\ C$.

Programs (Figure 3.14) Finally, the judgement for programs $\Vdash P \Longrightarrow \Sigma$ produces a constrained type environment Σ containing a constrained type scheme for each variable defined by P.

Constrained type generation via these inference systems follows a well established pattern for expressions and programs (see e.g. [19] for a general treatment of the

$$\text{(ISTyVar)}\ \frac{}{H \Vdash \alpha \sqsubseteq \alpha \Longrightarrow \emptyset}$$

$$\text{(ISArr)}\ \frac{H \Vdash \tau' \sqsubseteq \tau \Longrightarrow C_1 \quad H \Vdash \sigma \sqsubseteq \sigma' \Longrightarrow C_2}{H \Vdash \tau \to \sigma \sqsubseteq \tau' \to \sigma' \Longrightarrow C_1 \cup C_2}$$

$$\text{(ISData)}\ \frac{(\forall ki)\ H' \Vdash \mathsf{inj}_X(\Delta(\underline{d})(k)_i)[\overline{\tau/\alpha}] \sqsubseteq \mathsf{inj}_Y(\Delta(\underline{d})(k)_i)[\overline{\sigma/\alpha}] \Longrightarrow C_{ki}}{H \Vdash \mathsf{inj}_X(\underline{d})\ \overline{\tau} \sqsubseteq \mathsf{inj}_Y(\underline{d})\ \overline{\sigma} \Longrightarrow C}$$
$$\text{where } C = \textstyle\bigcup_{k,\,i} k \in X(\underline{d})\ ?\ C_{ki} \cup \{X(\underline{d}) \subseteq Y(\underline{d})\}$$
$$\text{and } H' = H \cup \{(\mathsf{inj}_X(\underline{d})\ \overline{\tau},\ \mathsf{inj}_Y(\underline{d})\ \overline{\sigma})\}$$

$$\text{(ISStop)}\ \frac{}{H \Vdash \mathsf{inj}_X(\underline{d})\ \overline{\tau} \sqsubseteq \mathsf{inj}_Y(\underline{d})\ \overline{\sigma} \Longrightarrow \emptyset}$$
$$\text{where } (\mathsf{inj}_X(\underline{d})\ \overline{\tau},\ \mathsf{inj}_Y(\underline{d})\ \overline{\sigma}) \in H$$

Figure 3.12: Constraint inference for subtyping.

non-refinement case), so we concentrate on the inference rules for subtyping. Like the more standard inference rules for expressions, the inference rules for subtyping generate a derivation tree and a system of constraints whose solution guarantees the correctness of the corresponding instance of the derivation tree. However, in the case of a subtyping inference, the derivation tree is not a coinductive proof under the system presented in Figure 3.7 but rather a proof that the solution constitutes a simulation in the sense of Lemma 3.2. For example, the conclusion of (ISData) yields the constraints $\{X(\underline{d}) \subseteq Y(\underline{d})\} \cup \bigcup_{k \in \mathbb{K}} k \in X(\underline{d})\ ?\ C_{k_i}$. The first constraint encodes the requirement that all constructors in $X(\underline{d})$ appear in $Y(\underline{d})$ whereas the second constraint arises from the second requirement placed upon a simulation: if X is assigned a refinement function ϕ for which $k \in \phi(\underline{d})$, the corresponding argument types of this constructor should again be related. This latter set of constraints ensures that no refutation could be derived via the (SSim) rule.

When constructing a simulation argument for recursive algebraic datatypes, it is not sufficient to inductively derive constraints that ensure the subtyping relation holds between the type of constructors as such a process will evidently enter into a loop so that no (non-trivial) subtyping relations could be inferred! It is for this reason that the algorithm subtyping judgement $H \Vdash \tau \sqsubseteq \sigma \Longrightarrow C$ is additionally parameterised by a history H consisting of pairs of the form $(\mathsf{inj}_X(\underline{d})\ \overline{\tau},\ \mathsf{inj}_Y(\underline{d})\ \overline{\sigma})$. Informally speaking, this set denotes those subtyping relations that are already implied by the set of constraints being accumulated. When a new subtyping constraint between datatypes is encountered, the rule (ISData) is applied and the history is extended ac-

cordingly. Upon encountering a subtyping constraint that arises from recursive occurrences of a datatype, on the other hand, we do not derive any new constraints but rather terminate the inference via the (ISStop) rule. Intuitively, the history can be thought of as a coinductive hypothesis that prevents subtyping inference from entering lead into a loop when accumulating constraints.

To show that the subtyping inference judgement is sound, in the sense that any solution to the constraints will ensure the subtyping relation holds, we borrow the idea of *stratification* that is often useful when dealing with coinductive structures [67, 68]. Stratification introduces an infinite sequence of over-approximations to a coinductive relation that ultimately converge upon it. As with the subtyping relation itself, each approximation is defined via its complement, which is restricted to only support derivations of lengths up to the given stratification level.

Definition 3.22. We write $\tau \not\sqsubseteq_j \sigma$ if there exists a derivation using the inference rules of the system presented in Figure 3.7 with at most $j \geq 0$ inferences, otherwise we write $\tau \sqsubseteq_j \sigma$ when no such derivation exists. Additionally, we will write $\theta \vDash_j H$ if $\tau \sqsubseteq_j \sigma$ holds for each pair $(\tau, \sigma) \in H$.

Note that, although $\tau \sqsubseteq_j \sigma$ does not necessarily imply $\tau \sqsubseteq \sigma$ for any given $j \geq 0$, the true subtyping relation is recovered as the intersection of this family.

Lemma 3.4. Suppose $H \Vdash \tau \sqsubseteq \sigma \Longrightarrow C$ is a subtyping inference and θ is an assignment such that $\theta \vDash_j H$ and $\theta \vDash C$. Then we have that $\tau\theta \sqsubseteq_j \sigma\theta$.

Lemma 3.5. Suppose $H \Vdash \tau \sqsubseteq \sigma \Longrightarrow C$ is a subtyping inference and θ is an assignment such that $\theta \vDash_j H$ for all $j \geq 0$ and $\tau\theta \sqsubseteq \sigma\theta$. Then we have that $\theta \vDash C$.

Corollary 3.6. Suppose $\vDash \tau \sqsubseteq \sigma \Rightarrow C$ is a subtyping inference and θ is an assignment. Then $\tau\theta \sqsubseteq \sigma\theta$ if, and only if, $\theta \vDash C$.

Having shown the subtyping inference is sound and complete, we show that the type inferred for an expression guarantees that it is typeable under any assignment that satisfies associated constraints. To show the completeness of inference for expressions is somewhat more complex as the declarative system permits the unrestricted use of (SSub), i.e. the type of an expression can be arbitrarily weakened at any point in the derivation. Therefore, we show instead that, for any type derivable under the declarative system, there is an assignment to the constraints generated by type inference that provides a stronger type.

Lemma 3.7. Suppose $\Gamma \Vdash e \Longrightarrow \tau$, C is an instance of type inference and θ is an assignment such that $\theta \vDash C$. Then $(\!|\Gamma\theta|\!) \Vdash e : \tau\theta$.

Lemma 3.8. Suppose $\Gamma' \Vdash e : \tau'$ and $\Gamma \Vdash e \Longrightarrow \tau$, C is an instance of type inference for which there exists some assignment θ such that $(\!|\Gamma\theta|\!) \sqsubseteq \Gamma'$. Then there exists an

$$\text{(IVar)}\ \frac{}{\Gamma \Vdash x : \underline{\tau}[\overline{\underline{\sigma}/\alpha}] \Longrightarrow \tau[\overline{Y/X}][\overline{\sigma/\alpha}],\ C[\overline{Y/X}]}\ \mathsf{Fresh}(\overline{Y}),\ \mathsf{Fresh}_{\overline{\underline{\sigma}}}(\overline{\sigma})$$
$$\text{where } f : \forall\overline{\alpha}.\,\forall\overline{X}.\,C \supset \tau \in \Gamma$$

$$\text{(ICon)}\ \frac{}{\Gamma \Vdash k : \underline{\tau}[\overline{\underline{\sigma}/\alpha}] \Longrightarrow \mathsf{inj}_X(\underline{\tau})[\overline{\sigma/\alpha}],\ \{k \in X(\underline{d})\}}\ \mathsf{Fresh}(X),\ \mathsf{Fresh}_{\overline{\underline{\sigma}}}(\overline{\sigma})$$
$$\text{where } k : \forall\overline{\alpha}.\,\underline{\tau} \in \underline{\Delta}(d)$$

$$\text{(IAbs)}\ \frac{\Gamma,\, x : \tau \Vdash e \Longrightarrow \sigma,\, C}{\Gamma \Vdash \lambda x.\, e : \underline{\tau} \to \underline{\sigma} \Longrightarrow \tau \to \sigma,\, C}\ \mathsf{Fresh}_{\underline{\tau}}(\tau)$$

$$\text{(IApp)}\ \frac{\Gamma \Vdash e_1 \Longrightarrow \tau \to \sigma,\, C_1 \quad \Gamma \Vdash e_2 \Longrightarrow \tau',\, C_2 \quad \Vdash \tau' \sqsubseteq \tau,\, C_3}{\Gamma \Vdash e_1\ e_2 \Longrightarrow \sigma,\, C_1 \cup C_2 \cup C_3}$$

$$\text{(ICase)}\ \frac{\Gamma \Vdash e \Longrightarrow \mathsf{inj}_X(\underline{d})\ \overline{\tau},\, C_0 \qquad \begin{array}{c}(\forall i \le m)\ \Gamma \cup \Gamma_i \Vdash e_i \Longrightarrow \sigma_i,\, C_i \\ (\forall i \le m)\ \Vdash \sigma_i \sqsubseteq \sigma \Longrightarrow C_i'\end{array}}{\Gamma \Vdash \mathtt{case}\ e\ \mathtt{of}\ \{k_i\ \overline{x_i}\ \mapsto e_i \mid i \le n\} : \underline{\sigma} \Longrightarrow \sigma,\, C_0 \cup C}\ \mathsf{Fresh}_{\underline{\sigma}}(\sigma)$$
$$\text{where } \Gamma_i = \overline{\{x_i : \mathsf{inj}_X(\Delta(\underline{d})(k_i))[\overline{\tau/\alpha}]\}}$$
$$\text{and } C = \{X(\underline{d}) \subseteq \{k_1, \ldots, k_m\}\} \cup \bigcup_{i \le m} k_i \in X(\underline{d})\ ?\ C_i \cup C_i'$$

Figure 3.13: Type inference for expressions.

assignment θ' that (1) satisfies C, (2) agrees with θ on the refinement variables of Γ, i.e. $\theta' \equiv_{\mathsf{FRV}(\Gamma)} \theta$, and such that (3) $\tau\theta' \sqsubseteq \tau'$.

Finally, we can prove the soundness and completeness for program inference and thereby provide an algorithmic solution to the typeability problem.

Theorem 3.9. Let P be a program such that $\vdash P \Longrightarrow \Sigma$. Then, $(\underline{\Delta}, \underline{\Sigma}, P)$ is a positive instance of the refinement typeability problem if, and only if, $(\!|\Sigma|\!)$ provides at least one type to every program variable.

3.4 Solving Constraints

Determining the solvability of constraints can be achieved through a process of saturation, which involves deriving all the consequences under a simple set of inference rules. This process is a generalisation of the transitive closure of simple inclusion constraint graphs and, more generally, an instance of Horn clause resolution. We will show that, once saturated, the solvability of a set of constraints depends solely on the

$$\frac{}{\Vdash \varepsilon \Longrightarrow \emptyset}$$

$$\frac{\Vdash P \Longrightarrow \Sigma \quad \Sigma \cup \{f : \tau\} \Vdash e \Longrightarrow \tau',\, C_1 \quad \Vdash \tau' \sqsubseteq \tau \Longrightarrow C_2}{\Vdash P;\, f : \forall \overline{\alpha}.\, \underline{\tau} = e \Longrightarrow \Sigma \cup \{f : \rho\}}\, \mathsf{Fresh}_{\underline{\tau}}(\tau)$$
$$\text{where } \rho = \forall \overline{\alpha}.\, \forall \overline{X}.\, C_1 \supset C_2 \subset \tau$$

Figure 3.14: Type inference for programs.

presence of constraints without refinement variables i.e. $k \in \{k_1, \ldots, k_m\}$. These constraints are either trivially valid or unsatisfiable.

Furthermore, saturated sets of constraints have a remarkable property — they can be restricted to a subset of refinement variables whilst remaining faithful to the set of satisfying assignments. That is to say, it is possible to safely omit constraints from the saturated set except those that only relate to a given subset of refinement variables. Any solution to this restricted set can be extended to a complete solution without any loss of generality. The subset of refinement variables in question are those that appear in the function's type, e.g. its inputs and outputs, which we will refer to as its *interface*. To allow for full generality with regards to the context in which a function is used, the interface variables of a type should not be instantiated. On the other hand, it is sufficient to merely know that non-interface variables have a solution with respect to the interface variables, as their assignment only imposes compatibility constraints and does not directly affect a function's type nor the contexts in which it can be used. Thus, non-interface variables can be viewed as existentially quantified for which saturation implicitly determines a solution in terms of the interface variables. It is from this construction that we derive our linear-time (parameterised) complexity. As the size of the restricted constraints no longer depends on the number of preceding function definitions or their complexity, the exponential blow-up is avoided.

3.4.1 Saturation and Restriction

Definition 3.23. A constraint is said to be *atomic* just if its body has one of the following four shapes:

$$X(\underline{d}) \subseteq Y(\underline{d}) \quad X(\underline{d}) \subseteq \{k_1, \ldots, k_m\} \quad k \in X(\underline{d}) \quad k \in \emptyset$$

i.e. it is not of the form $\{k_1, \ldots, k_m\} \subseteq S$ or $k \in \{k_1, \ldots, k_m\}$ where $m > 0$.

An atomic constraint is said to be *trivially unsatisfiable* if it is unguarded and its body is of the form $k \in \emptyset$. Any constraint set with a trivially unsatisfiable constraint is itself trivially unsatisfiable.

By applying standard identities, every constraint is equivalent to a set of atomic constraints. In particular, a constraint of the form $k \in \{k_1, \ldots, k_m\}$ is equivalent to the empty set of atomic constraints (i.e. can be eliminated) whenever k is equal to some k_i, and is equivalent to $k \in \emptyset$ otherwise.

Definition 3.24. An atomic constraint set, i.e. one that only contains atomic constraints, is said to be saturated just if it is closed under the saturation rules in Figure 3.15. We write $\mathsf{Sat}(C)$ for the smallest saturated atomic constraint set containing C, which is clearly finite for any finite C as there are finitely many constructors and finitely many refinement variables appearing in the original set.

Recall that the full set constraint language is exactly the monadic class of first-order propositions [66] and that our guarded fragment corresponds to (monadic) Horn clauses. The aforementioned saturation rules, therefore, amount to special cases of resolution for Horn clauses.

$$\textsf{(Trans)}\ \frac{\phi \mathbin{?} S_1 \subseteq S_2 \quad \psi \mathbin{?} S_2 \subseteq S_3}{\phi \cup \psi \mathbin{?} S_1 \subseteq S_3}$$

$$\textsf{(Sat)}\ \frac{\phi \mathbin{?} k \in X(\underline{d}) \quad \psi \cup \{k \in X(\underline{d})\} \mathbin{?} S_1 \subseteq S_2}{\phi \cup \psi \mathbin{?} S_1 \subseteq S_2}$$

$$\textsf{(Weak)}\ \frac{\phi \mathbin{?} X(\underline{d}) \subseteq Y(\underline{d}) \quad \psi \cup \{k \in Y(\underline{d})\} \mathbin{?} S_1 \subseteq S_2}{\phi \cup \psi \cup \{k \in X(\underline{d})\} \mathbin{?} S_1 \subseteq S_2}$$

Figure 3.15: Saturation rules for atomic constraints.

Theorem 3.10. For any assignment θ and atomic set of constraints, we have that $\theta \vDash C$ if, and only, if $\theta \vDash \mathsf{Sat}(C)$.

When there are no trivially unsatisfiable constraints in $\mathsf{Sat}(C)$, we can construct a solution to the constraints as follows. For each variable X occurring in C, let the refinement function ϕ_X be defined by:

$$\phi_X(\underline{d}) \coloneqq \{k \mid k \in X(\underline{d}) \in \mathsf{Sat}(C)\}$$

and let the assignment θ map each refinement variable X to the refinement function ϕ_X. This constitutes a solution as any guards which the solution satisfies are eliminated in the saturated set through the (Sat) rule, likewise indirect inclusions are accounted for by (Trans) and (Weak). Note that this solution is, in particular, the minimal solution.

Theorem 3.11. A set of atomic constraints C is satisfiable if, and only if, $\mathsf{Sat}(C)$ does not have any trivially unsatisfiable constraints.

Having established that a constraint set is solvable, we are interested in constructing a general solution to a certain subset of the refinement variables, i.e. the non-interface variables that correspond to the sub-expression of a function's body, in terms of those appearing in its interface. However, the interface variables should be left uninstantiated as there is not generally a most general instance that will be applicable in all contexts. More formally, for constraints C describing a set of types $\{\tau\theta \mid \theta \vDash C\}$, we should consider two solutions θ_1 and θ_2 to be equivalent whenever they agree on the free refinement variables of τ, i.e. the interface. That is to say, typeability is agnostic to the refinement function assigned to internal variables.

Definition 3.25. Let C be a saturated set of constraints and $I \subseteq \mathbb{X}$ be some set of refinement variables, called the *interface variables*. Then we define the *restriction* $C{\restriction_I}$ as the set $\{\phi \mathbin{?} S_1 \subseteq S_2 \in C \mid \mathsf{FRV}(\phi \mathbin{?} S_1 \subseteq S_2) \subseteq I\}$.

The restriction of a set of constraints is quite severe, since it simply discards any constraints that are not *solely* comprised of interface variables. However, a remarkably strong consequence of saturation is that, whenever C is solvable, every solution of $\mathsf{Sat}(C){\restriction_I}$ may be extended to a solution of C, independent of the choice of I. Since every solution to $\mathsf{Sat}(C)$ trivially restricts to a solution of $\mathsf{Sat}(C){\restriction_I}$, it follows that the solutions of $\mathsf{Sat}(C){\restriction_I}$ are exactly the restriction of the solution of C, i.e. no solutions are lost through restriction.

If we were not to saturate, however, then solutions to the restriction cannot necessarily be extended. Consider, for example, the following constraint set C:

$$\begin{aligned} &\texttt{Cst} \in X(\texttt{Lam}) && X(\texttt{Lam}) \subseteq Y(\texttt{Lam}) \\ &\texttt{FVr} \in Y(\texttt{Lam}) \mathbin{?} Y(\texttt{Lam}) \subseteq Z(\texttt{Lam}) && Z(\texttt{Lam}) \subseteq \{\texttt{FVr}, \texttt{Cst}\} \end{aligned}$$

This set is not saturated as we can derive $\texttt{FVr} \in X(\texttt{Lam}) \mathbin{?} X(\texttt{Lam}) \subseteq \{\texttt{FVr}, \texttt{Cst}\}$ through the rules presented in Figure 3.15. Therefore, although the solution:

$$\theta(Z)(\underline{d}) = \emptyset \qquad \theta(X)(\underline{d}) = \begin{cases} \{\texttt{Cst}, \texttt{FVr}, \texttt{App}\} & \text{if } \underline{d} = \texttt{Lam} \\ \emptyset & \text{otherwise} \end{cases}$$

satisfies the restriction to $\{X, Z\}$, it is not a solution to the constraint set; in particular, it violates the aforementioned derivable constraint.

Theorem 3.12 (Restriction/Extension)**.** Suppose C is a set of saturated constraints and $I \subseteq \mathbb{X}$ is a subset of refinement variables. Let θ be an assignment such that $\theta \vDash C{\restriction_I}$, i.e. a solution to the restricted set. Then there exists an assignment θ' such

that $\theta' \equiv_I \theta$ and $\theta' \vDash C$, i.e. a solution to the original set that agrees on the refinement function assigned to interface variables.

Proof. To construct our extended solution, we generalise the argument presented in Section 3.4.1. That construction assigned each refinement variable its minimal refinement function by considering atomic constraints of the form $k \in Y(\underline{d})$. In this more general setting, we must also consider constraints of the form $X(\underline{d}) \subseteq Y(\underline{d})$ where X is an interface variable and those guarded by interface variables as we cannot assume the interface assignment is minimal. Nevertheless, saturation ensures that the constructed assignment is indeed a solution.

Let θ be our assignment to interface variables that is a solution to the restricted set. We consider an extended assignment θ' that is defined as $\theta(X)$ for any interface variable $X \in I$ and, for each non-interface variable $Y \in \mathbb{X} \setminus I$, it defines as the refinement function that maps each underlying datatype $\underline{d} \in \underline{D}$ to the set:

$$\bigcup_{\phi ? S \subseteq Y(\underline{d}) \in C} \{S\theta \mid \theta \vDash \phi,\ \mathsf{FRV}(\phi) \cup \mathsf{FRV}(S) \subseteq I\}$$

First, we shall show that, whenever the guards of a constraint are satisfied by θ', the body of the constraint must have already been satisfied by θ. In other words, the extended solution does not make any arbitrary choices.

Lemma 3.13. If there is an atomic constraint $\phi \,?\, S_1 \subseteq S_2 \in C$ in a saturated set of constraints such that $\theta' \vDash \phi$, then there is another constraint $\psi \,?\, S_1 \subseteq S_2 \in C$ such that $\theta \vDash \psi$ and $\mathsf{FRV}(\psi) \subseteq I$.

Proof. Our proof is by induction on the cardinality of $\phi\restriction_{\mathbb{X} \setminus I}$, i.e. the number of constraints $k \in Y(\underline{d}) \in \phi$ in the guard where Y is a non-interface variable. Note that, for this proof, we treat guards as multisets of constraints.

- The base case is trivial as $\theta \vDash \emptyset$.
- Otherwise, suppose we have that $\phi = \{k \in Y(\underline{d})\} \uplus \phi'$ where $Y \in \mathbb{X} \setminus I$ and $k \in \theta'(Y(\underline{d}))$. There must exist, therefore, some constraint $\psi \,?\, S \subseteq Y(\underline{d})$ such that $\theta \vDash \psi$ and $k \in S\theta$ where $\mathsf{FRV}(S) \cup \mathsf{FRV}(\psi) \subseteq I$. There are two possible forms S may have:
  - If it is the $\{k\}$, then there is an additional constraint $\psi \cup \phi' \,?\, S_1 \subseteq S_2$ as the set is closed under the (Sat) rule. Note that the guard of this constraint contains one fewer non-interface constraints and is still satisfied by θ'. Thus, by induction, there is a constraint $\psi' \,?\, S_1 \subseteq S_2$ whose guard contains no non-interface variables as required.
  - If, on the other hand, S is of the form $X(\underline{d})$ for some $X \in I$, then we have that $\{k \in X(\underline{d})\} \uplus \phi' \,?\, S_1 \subseteq S_2$ is in the saturated set as a result of

the (Weak) rule. Again, there is one fewer non-interface constraint in the guard of this constraint and is still satisfied by θ'. Thus, by induction, there is a constraint $\psi' \mathbin{?} S_1 \subseteq S_2$ whose guard contains no non-interface variables as required.

□

Now suppose we have a constraint $\phi \mathbin{?} S_1 \subseteq S_2 \in C$ whose guard is satisfied by θ'. As a consequence of the above lemma, there is also a constraint $\psi \mathbin{?} S_1 \subseteq S_2 \in C$ whose guard, which only concerns interface variables, is satisfied by θ. If $\mathsf{FRV}(S_1) \cup \mathsf{FRV}(S_2) \subseteq I$, then we immediately know that the constraint is satisfied as it appears in the restricted set, for which θ is a solution. Now consider the possible forms of S_1 and S_2 containing non-interface variables:

- An inclusion $S_1 \subseteq Y(\underline{d})$ where $Y \in \mathbb{X} \setminus I$ and $\mathsf{FRV}(S_1) \subseteq I$ is clearly satisfied by construction.
- Suppose, on the other hand, we have a constraint of the form $Y(\underline{d}) \subseteq S_2$ where $Y \in \mathbb{X} \setminus I$ and $\mathsf{FRV}(S_2) \subseteq I$. For each $k \in \theta'(Y)(\underline{d})$, there must be a constraint $\phi' \mathbin{?} S \subseteq Y(\underline{d})$ such that $\theta \vDash \phi'$ and $k \in S\theta$ where $\mathsf{FRV}(S) \cup \mathsf{FRV}(\phi') \subseteq I$. By (Trans), therefore, we can derive the additional constraint $\psi \cup \phi' \mathbin{?} S \subseteq S_2$. As this constraint appears in the restriction, it must be satisfied by θ. Therefore, $k \in S_2\theta'$ as required.
- Finally, suppose both S_1 and S_2 contain non-interface variables, i.e. the constraint is of the form $Y(\underline{d}) \subseteq Z(\underline{d})$ where $Y, Z \in \mathbb{X} \setminus I$. As before, for each $k \in \theta'(Y)(\underline{d})$, there must be a constraint $\phi' \mathbin{?} S \subseteq Y(\underline{d})$ such that $\theta \vDash \phi'$ and $k \in S\theta$ where $\mathsf{FRV}(S) \cup \mathsf{FRV}(\phi') \subseteq I$. And, therefore, by (Trans), there is an additional constraint $\psi \cup \phi' \mathbin{?} S \subseteq Z(\underline{d})$. In which case, $k \in \theta'(Z)(\underline{d})$ by construction.

□

3.4.2 Complexity Analysis

Although our inference procedure is compositional, i.e. it breaks programs down into top-level definitions, and expressions into sub-expressions that can be analysed in isolation, this is no guarantee of its efficiency. As we have described it in Section 3.3, the number of constraints inferred by type inference depends on the size of the expression — constraints are generated at most syntax nodes and propagated to the root. In fact, as is well known for constrained type inference, the situation is much worse as a whole set of constraints is imported from the program signature when inferring the type of a program variable. Again, the number of constraints associated with

that program variable will depend on not only its definition, but the number of constraints associated with program variables on which it depends and so on. As a result of this duplication, the number of constraints generated across the entire program can quickly become exponential in the number of function definitions.

Consider an inference $\Gamma \Vdash e \Longrightarrow \tau, C$. As it stands, the number of refinement variables occurring in C will depend upon the size of e and the size of every definition on which e depends. To avoid the constraint explosion problem caused by duplicating previously inferred constraint sets, we utilize the restriction operator. At each step during inference, we compute $\mathsf{Sat}(C)\restriction_I$ where the interface I is taken to be the free refinement variables of the context and the inferred type, i.e. $\mathsf{FRV}(\Gamma) \cup \mathsf{FRV}(\tau)$. Clearly no context in which this expression appears can directly interact with any other refinement variables and, as previously discussed, any solution to the restricted, saturated set can be extended to a solution to the original set.

By applying our restriction strategy, the number of refinement variables appearing in the constrained type scheme of a top-level function depends only on the number of refinement variables appearing in its type. In other words, the number of constraints appearing in the type scheme ascribed to a program variable is independent on the number of preceding definitions.

Theorem 3.14. Under the assumption that the size of types and the size of each function definition is bounded, the complexity of type inference is $\mathcal{O}(N)$ where N is the number of function symbols.

There is, however, an extensive cost associated with performing saturation before we can take the restriction. Let us take K to be the number of constructors and D to be the number of datatypes. A simple analysis of the form of atomic constraints over V refinement variables exploiting the fact that constructors uniquely determine an underlying datatype yields the bound:

$$\begin{aligned}&\mathcal{O}(2^{K\cdot V} \cdot (K \cdot V + V^2 \cdot D + V \cdot 2^K)) \\ &\quad \subseteq \mathcal{O}(2^{K\cdot V + K} \cdot K \cdot V^2 \cdot D)\end{aligned}$$

as a subset of simple inclusions $k \in X(\underline{d})$ appears in the guard and the body may either be of the form $k \in X(\underline{d})$, $X(\underline{d}) \subseteq Y(\underline{d})$, or $X(\underline{d}) \subseteq \{k_1, \ldots, k_n\}$ for a subset of constructors.

In light of the large number of possible constraints, it is crucial that the algorithm for computing the saturated set of constraints is efficient. To this end, we observe that the construction in Theorem 3.12 can also be framed as computing a minimal solution, merely over a different partial order to the one used in Theorem 3.11. Recall

the assignment to a non-interface variable Y:

$$\bigcup_{\phi ? S \subseteq Y(\underline{d}) \in C} \{S\theta \mid \theta \vDash \phi,\ \mathsf{FRV}(\phi) \cup \mathsf{FRV}(S) \subseteq I\}$$

By abstracting over the assignment θ, it is plain to see that the set of constructors can be recovered from the set of pairs $(\phi,\ S)$ for which there exists some $\phi\ ?\ S \subseteq Y(\underline{d}) \in C$ such that $\mathsf{FRV}(\phi) \cup \mathsf{FRV}(S) \subseteq I$. We refer to such pairs as *interface values*; for any assignment to the interface variables θ, an interface value $(\phi,\ S)$ is interpreted as the set $\{k \mid k \in S\theta,\ \theta \vDash \phi\}$. In other words, non-interface variables can first be interpreted by a set of interface values, which is then used to determine their lower bound for any given assignment.

Using this insight, we employ a standard algorithm for computing the least-fixed point over the set of interface values instead of the naïve saturation algorithm that iteratively applies the rules in Figure 3.15. This algorithm generalises the optimal solution to the HORNSAT problem to partial orders of finite height, where there are no infinite chains [64, 65]. At a high-level, the algorithm not only keeps track of the current, accumulated valuation mapping refinement variables to sets of interface values, but also creates a map that associates constraints with the refinement variables on which they depend. This structure ensures that only the affected constraints are revisited when a new interface value for a refinement variable is encountered.

The asymptotic complexity of this algorithm is linear in the number of entries in the valuation $V \cdot D$ and the height of the partial order, i.e. the longest chain [69]. In our case, the partial order is defined as inclusion on sets of interface values, of which there are at most $\mathcal{O}(2^{K \cdot I} \cdot (K + I))$. Therefore, the worst-case complexity of saturation can be bounded by:

$$\mathcal{O}(2^{K \cdot I} \cdot (K + I) \cdot V \cdot D)$$

or $\mathcal{O}(2^I \cdot I \cdot V)$, once we fix the number of constructors and datatype identifiers.

Although this result is still exponential in the number of interface variables, this is a drastic improvement on the naïve saturation algorithm that can only be bounded by the total number of possible constraints, which is exponential in non-interface variables as well. Our experimental evaluation found that, while interface variables rarely exceed a hundred, there may be several thousand refinement variables in total associated with large recursive groups. Therefore, the distinction between these two complexity results is significant. In the following section, we will show that the exponential cost of saturation does not prevent our performant program-level algorithm.

3.5 Implementation

We have implemented a prototype of the inference algorithm presented in the preceding chapters as a core plugin for GHC 9.2.8 [43]. Through an additional compiler flag, the user can run our type checker as another stage of compilation whereby any unsatisfiable set of constraints raises a warning that specifies the source of the constructor and the incomplete pattern matching expression that it may reach. In addition to running the type checker on individual modules, a binary interface file is generated, enabling other modules within the same package to use the inferred constraints.

The resulting analysis provides a certificate of safety for the package modulo the safe use of its dependencies. Since we do not analyse the dependencies of packages unless an interface file is present, datatypes that are defined outside the current package are treated as base types and not refined. Not permitting refinements of standard datatypes, such as lists, was particularly beneficial for efficiency. In practice, considering their refinements leads to unnecessary constraints without actually improving expressivity. For the sake of performance, we also do not consider refinements of datatypes with less than two constructors (e.g. boxed integers and records). This relatively small departure from the theory is a substantial improvement to the efficiency of the tool due to the number of records and newtypes that are found in typical Haskell programs. Likewise, when a case expression is exhaustive, we do not infer a constraint of the form $X(\underline{d}) \subseteq \{k_1, \ldots, k_n\}$ as this is universally valid.

As previously mentioned, GHC's core language is significantly more powerful than the small language presented in Chapter 2. Specifically, our prototype implementation does not have a proper treatment of higher-rank types, type classes (which are represented as higher-ranked functions in GHC core), or casts and coercions. These features are simply over-approximated by an "ambiguous" refinement type. One relevant difference from our formalism, however, is that case expressions in GHC core are necessarily complete, with missing branches being implemented by an explicit exception. Our tool uses the results of internal analyses in GHC to identify expressions that necessarily throw an exception and treat such branches as missing [70].

3.5.1 Performance

To test the performance of our prototype, we applied it to a number of packages taken from the Hackage database:

- `aeson` 2.2.0 a performant JSON serialisation library.
- `algebraic-graphs` 0.8 a library for algebraic graph construction and transformation.

- `containers` 0.6.7 a standard selection of classic functional data structures such as sets and finite maps.
- `haskeline` 0.8.2 a library for writing command-line interfaces.
- `pretty` 1.1.3 a collection of pretty printing combinators.
- `sbv` 10.2 an SMT based theorem prover.
- `time` 1.12.2 a library for manipulating time, clocks, and calendars.
- `unordered-containers` 0.2.19 a set of hashing-based containers for datatypes without a natural/efficient ordering.

These packages were selected to test our tool in a range of contexts and scales. In the case of the `containers` package, we have removed the `Data.Sequence` module and any functions that depend on it for reasons which we discuss below. We recorded the time taken for our inference algorithm to process each module within these packages across an average 10 runs on the Windows Subsystem for Linux, running on an 11th Gen Intel® Core™ i7–1185G7 @ 3.00GHz/1.80GHz processor with 16.0GB of RAM. A summary of the results[3] are presented in Table 3.1.

In addition to the elapsed time, we recorded the total number of top-level definitions (N), the total number of refinement variables generated during inference (V), and the largest interface encountered (I). The contrast between these latter two figures gives some indication of how intractable the analysis may become be without the restriction operator.

The two columns labelled (D) and (K) are a metric of the number of underlying datatypes identifiers and constructors respectively. Rather than being the total number, however, they denote the largest number of underlying datatypes identifiers and constructors in any given *slice*. A slice of a given underlying datatype identifier $\underline{d}$ is the set of datatypes $D_{\underline{d}} \subseteq \underline{D}$ that are transitively reachable through the type of constructor, e.g. $\underline{\texttt{Arith}}$ is in the slice of $\underline{\texttt{Lam}}$ due to the `Cst` constructor. The slice of a datatype is a more accurate predictor of the number of constraints generated by inference as subtyping, which tends to contribute the most constraints, reconstructs the slices of datatype identifiers through the (ISData) rule. Consider, for example, the subtyping constraint $\text{inj}_X(\underline{\texttt{Lam}}) \sqsubseteq \text{inj}_Y(\underline{\texttt{Lam}})$ that induces set inclusion constraints concerning $X(\underline{\texttt{Arith}})$ and $Y(\underline{\texttt{Arith}})$ as well as $X(\underline{\texttt{Lam}})$ and $Y(\underline{\texttt{Lam}})$, but it will not give rise to any constraints of datatype identifiers outside its slice.

The final two columns indicate the number of warnings found in the given package and the average time taken. Typically, many of the warnings can be traced to a single pair of constructor and inexhaustive case expression but as they factor through

[3]The results here differ from those presented in the original paper due to a bug causing the elapsed times to be incorrectly reported as well as updates to GHC and the benchmark packages.

Package	N	D	K	V	I	Warnings	Time (s)
`aeson`	499	6	24	1408	21	0	0.66
`algebraic-graphs`	768	4	16	1494	8	0	0.15
`containers`	785	2	6	8705	23	38	0.89
`haskeline`	482	4	15	2964	10	0	0.21
`pretty`	163	4	16	1037	16	8	1.31
`sbv`	1550	12	173	13584	36	98	11.59
`time`	352	2	10	464	5	20	0.08
`unordered-containers`	230	3	17	2203	15	0	0.29

Table 3.1: Intensional benchmark summaries for packages taken from the Hackage database.

different functions and appear in different top-level definitions they are reported separately. We did not find any true positives, but this is not surprising since these are mature packages. Recall that our inference algorithm is sound and complete *with respect to* the declarative system presented in Section 3.2. This means that, if the algorithm produces a warning for a given program, then that program cannot be typed by an intensional program environment and, if no warnings are produced, then the program cannot produce a pattern-matching error. However, it is not the case that our declarative system (and, therefore, our inference algorithm) is complete with respect to the pattern-match safety problem. It is in this sense that we observe false positives: there exists programs that are pattern-match safe but are not typeable by our system.

3.5.2 Limitations and Future Work

As previously mentioned, we did not include the `Data.Sequence` module from the `containers` package. The reason for this omission is that the tool was unable to process the large number of constraints produced by this module. More specifically, the top-level recursive group consisting of the functions `addDigits2`, `addDigits3`, `addDigits4`, `appendTree2`, `appendTree3`, and `appendTree4` has an abnormally large interface consisting of 90 refinement variables, due to the large number of inputs and the nesting of datatypes. As our inference algorithm is exponential in the number of interface refinement variables, it is no surprise that it was unable to fully saturate the associated constraints. This scenario clearly outlines the drawback of our approach: when the number of interface variables is small, the tool is extremely performant and exhibits an asymptotically linear-time complexity; on the other hand,

when the number of interface variable grows with the complexity of the project, the algorithm becomes exponentially expensive.

Fully automated program analyses necessarily compromise on expressivity. In our case, we emphasised performance and imposed the intensional restriction of refinements in order to achieve a linear-time asymptotic complexity under the assumption that functions have a bounded interface, leading to good performance in practice. As a consequence of this restriction, our system is unable to express certain common patterns of refinement, the most notable of which is the lack of a non-empty list refinement. Recall that the intensional refinements of a datatype identifier are defined by the removal of constructors. For the list datatype, therefore, there are three proper refinements: one with no constructors, one with just the `[]` constructor, and one with just the `::` constructor.

```
data List₁ α      data List₂ α      data List₃ α
                    = []                = α :: List₃ α
```

Figure 3.16: The possible refinements of the `List` datatype.

The datatype identifier does not characterise non-empty lists but rather infinite lists, as the refinement must apply recursively throughout the datatype environment. For most programs, these refinements will not be useful. Following the coalgebraic view of datatype environments as a generalisation of tree automata, intensional refinements give us the option to remove transitions, i.e. constructors, from a given state, i.e. datatype identifier. Creating a non-empty list refinement, on the other hand, requires the introduction of a new state of the automaton.

As it is often useful to distinguish a "non-empty" or "singleton" case from a variety of data structures, it may be prudent to consider unrolling datatype definitions by explicitly introducing a new datatype identifier, defined in the underlying environment as the following.

```
data List α              data List′ α
  = []                     = []
  | α :: List′ α           | α :: List′ α
```

Figure 3.17: An unrolling of the `List` datatype.

From these datatypes, the type of non-empty and singleton lists can be constructed as intensional refinements of the `List` α datatype. The introduction of overlapping datatype identifiers into the underlying environment would complicate type inference as it is no longer clear which datatype the constructor `[]` should refine. However, it

is worth noting that any intensional refinement of `List'` α can be simulated as an intensional refinement of `List` α.

A similar system could also be used to allow arbitrary, user specified refinements by ensuring there is enough datatype identifiers to be simulated by intensional refinements. As part of future work, we plan to investigate the integration of these features into our inference algorithm.

Chapter 4

CycleQ

An Efficient Basis for Cyclic Equational Reasoning

4.1 Introduction

Equational properties are a powerful yet terse way of expressing the behaviour of pure functional programs. A key advantage of this style of specification, which has led to its widespread use, is that the programmer needn't familiarise themself with an external logic, e.g. separation logic. Such properties are immediately recognisable to any functional programmer, e.g:

$$\forall xs \texttt{ : List } \alpha\texttt{. map id xs = xs}$$

Figure 4.1: The first functor law for lists.

When reasoning about recursive programs that manipulate algebraic datatypes, proof by induction is often essential. Unfortunately, inductive theorem proving is notoriously difficult to automate. The incompleteness and non-analyticity of sufficiently expressive proof systems excludes the possibility of a universal decision procedure [71]. Here, non-analyticity refers to the inability to perform cut elimination on inductive proofs; that is to say, a proof cannot necessarily be constructed from sub-formulas of the goal alone. Thus, in many cases, auxiliary lemmas are required, and induction hypotheses must be strengthened.

Even if we restrict our attention to what we might loosely imagine as those functional programs that occur in practice, the situation is still extremely complex. Functional programmers readily employ a variety of inductive and mutually inductive datatypes and rarely restrict themselves to functions defined by structured recursion schemes. Hence, if we are to rely on explicit induction schemes, not only do we need a scheme for each datatype, but we should expect that these schemes can be nested,

combined for mutual induction, or generalised to account for complex patterns of recursion that are not directly structural [72].

Despite the apparent difficulties, there are already several tools that have been successfully able to automatically prove equational properties of functional programs. However, to the best of our knowledge, none have a smooth treatment of the more complicated induction schemes that are frequently required in practice. For example, proofs that require mutual induction are not supported by default in HipSpec [73], IsaPlanner [74], or Zeno [32], and reasoning about mutually recursive functions is described as being "a bit awkward" in the ACL2 manual [75]. This limitation is unsurprising as mutual induction not only depends on strengthening an induction hypothesis but entirely synthesising a complementary hypothesis for other datatypes within the recursive group.

Consider, for example, the following mutually inductive datatypes:

```
data Tm α                    data Exp α
  = Var α                      = MkExp (Tm α) Int
  | Cst Int
  | App (Exp α) (Exp α)

mapE : (α → β) → Exp α → Exp β
mapE f (MkExp t n) = MkExp (mapT t) n

mapT : (α → β) → Tm α → Tm β
mapT f (Var x) = Var (f x)
mapT f (Cst n) = Cst n
mapT f (App e₁ e₂) =
  App (mapE f e₁) (mapE f e₂)
```

Figure 4.2: Datatypes for `Int` annotated expressions and their mapping functions.

These datatypes are intended to model a simple expression language where each sub-expression is annotated with an integer, perhaps a line number marking its provenance. The function `mapE` witnesses the functoriality of the `Exp` datatype. However, it is not possible to show that Figure 4.1 holds without simultaneously showing that it holds for `mapT`, which is not a syntactic generalisation. Even in this case, where it should be quite straightforward to infer the required property, the process of strengthening the hypothesis cannot be performed as part of proof search itself because the mutual induction scheme requires both induction hypotheses to be known upfront.

Instead of generating specialised induction schemes, most recent advances in the area of automated proof induction have been focused on discovering suitable lemmas to compensate. Lemma discovery and hypothesis strengthening techniques can be powerful but have a number of weaknesses including the generation of irrelevant

lemmas, a tendency to over generalise, and lack of scalability for complex formulas [29]. Ultimately, such techniques are crucial in all but the simplest proofs, and so any improvement to the underlying induction mechanism, reducing the burden on lemma generation heuristics, is worthwhile. It is also worth noting that no lemma discovery strategy can be complete, and so there will always be a role for the user in supplementing proof search with their own insight.

Thus, leaving aside the issues of lemma discovery and hypothesis strengthening, we explore an alternative system that doesn't rely on explicit induction schemes — cyclic proofs. In this context, we introduce a novel technique for equational reasoning and a simple proof system based on our mechanism that seamlessly supports complex forms of inductive arguments, such as nested or mutual induction.

4.1.1 Cyclic Proofs

Non-well-founded proof theory is distinguished from classic proof theory by the use of possibly infinitely deep derivation trees, which are well suited to expressing arguments by infinite descent. Cyclic proofs occur within this field as those proofs whose derivations are regular, i.e. have finitely many distinct sub-derivations, and are thus representable as a finite graph [76, 77]. Of course, not all circular proofs represent sound arguments; an additional correctness criterion is required to compensate, usually taking the form of an ω-regular condition on the infinite paths through a proof.

Cyclic proofs are naturally of theoretical interest in domains with some notion of fixed-point and have enabled a number of elegant proof theoretic results [78, 79, 80, 81]. In the case of first-order logic extended with inductive definitions, for example, they are known to be stronger than traditional proof systems in terms of logical complexity, i.e. degree of quantifier alternation [82]. One of the major forces driving research into cyclic proofs, however, has been applications in the field of programming languages, whereby their first-class support for recursion has the potential to improve on the state-of-the-art. For example, cyclic proofs have already been employed in program synthesis and to verify termination [38, 40].

Automated cyclic proof systems are also better suited to the exploratory nature of goal-directed proof search in settings involving recursion or inductive domains as they can avoid committing to a fixed induction scheme in advance, instead discovering a bespoke circular argument justified post-hoc [42]. Although the generation of induction schemes specific to the recursive functions in question, i.e. "recursion analysis", and the selection of induction hypotheses based on deductive proof search, i.e. "lazy induction", have previously been considered, these approaches have only been shown applicable to problems of limited complexity [71, 83]. Cyclic proofs can imitate such approaches when appropriate but is not limited to them.

The cyclic proof in Figure 4.3 demonstrates the "lazy" style of induction whereby the recursive structure of the argument is postponed until an instance of a previous equation is required. Here, and elsewhere, the cycle is represented by labelling a node in the proof graph, e.g. labelling the root 1, and referencing this label elsewhere as a premise, e.g. (1), without further justification. The proof can be read in a goal-oriented manner with the root node, which ultimately acts as an induction hypothesis, reoccurring naturally through a combination of case analysis, reduction, and simple equational reasoning without being pre-determined as an induction hypothesis or requiring the corresponding property of `Tm` to be known up-front.

$$\dfrac{\dfrac{\dfrac{(2) \qquad \dfrac{}{\vdash n = n}\,(\text{Refl})}{\vdash \texttt{MkExp}\ (\texttt{mapT}\ \texttt{id}\ t)\ n = \texttt{MkExp}\ t\ n}\,(\text{Cong})}{\vdash \texttt{mapE}\ \texttt{id}\ (\texttt{MkExp}\ t\ n) = \texttt{MkExp}\ t\ n}\,(\text{Reduce})}{1{:}\ \vdash \texttt{mapE}\ \texttt{id}\ e = e}\,(\text{Case})$$

$$\dfrac{\dfrac{\dfrac{}{\vdash \texttt{Var}\ x = \texttt{Var}\ x}\,(\text{Refl})}{\vdash \texttt{mapT}\ \texttt{id}\ (\texttt{Var}\ x) = \texttt{Var}\ x}\,(\text{Reduce}) \qquad \vdots \quad \vdash \texttt{mapT}\ \texttt{id}\ (\texttt{Cst}\ n) = \texttt{Cst}\ n \quad (3)}{2{:}\ \vdash \texttt{mapT}\ \texttt{id}\ t = t}\,(\text{Case})$$

$$\dfrac{\dfrac{\dfrac{(1)}{\vdash \texttt{mapE}\ \texttt{id}\ e_1 = e_1}\,(\text{Inst}) \qquad \dfrac{(1)}{\vdash \texttt{mapE}\ \texttt{id}\ e_2 = e_2}\,(\text{Inst})}{\vdash \texttt{App}\ (\texttt{mapE}\ \texttt{id}\ e_1)\ (\texttt{mapE}\ \texttt{id}\ e_2) = \texttt{App}\ e_1\ e_2}\,(\text{Cong})}{3{:}\ \vdash \texttt{mapT}\ \texttt{id}\ (\texttt{App}\ e_1\ e_2) = \texttt{App}\ e_1\ e_2}\,(\text{Reduce})$$

Figure 4.3: A cyclic proof of $\vdash \texttt{mapE}\ \texttt{id}\ e = e$.

4.1.2 Cyclic Equational Reasoning

The cycles that feature in Figure 4.3 instantiate an ancestral node to directly justify their conclusion without requiring any other premises. In Brotherston, Gorogiannis and Petersen's state-of-the-art Cyclist theorem prover, cycles are formed precisely in this manner — a node that is logically stronger than the current proof obligation is identified and used to discharge it [42]. Being generic across logics, the exact justification used to form cycles in their system will vary but typically is limited to some combination of weakening, instantiation, or, in the case of separation logic, the frame rule. When it comes to equational reasoning, however, this mechanism is insufficient.

The standard axiomatisation of equational reasoning quickly leads to an intractable space of proofs and cannot be handled effectively by a goal-oriented theorem prover without a specialised mechanism that takes advantage of structural information. For example, while Cyclist has been shown to handle mutual induction problems with ease, its lack of native support for equational reasoning means that it has difficulty with heavily-equational properties, such as $\vdash \mathtt{add}\ x\ y = \mathtt{add}\ y\ x$ which encodes the commutativity of addition. The authors conjecture that a proof could be obtained if the auxiliary lemma $\vdash \mathtt{add}\ x\ (\mathtt{S}\ y) = \mathtt{S}\ (\mathtt{add}\ x\ y)$ were supplied as a hint.

However, a variant of this required lemma can be derived from information already present in the proof without relying on generalisation or some other ad-hoc means of synthesis. The partial proof in Figure 4.4 shows the S case of the commutativity property, where the original goal is labelled (1). This proof is not discovered by Cyclist because the cycle is obscured behind several equational reasoning steps, which crucially omits additional obligations in the right-hand branch.

$$\dfrac{\dfrac{\dfrac{\dfrac{(1)}{\vdash \mathtt{add}\ x'\ y = \mathtt{add}\ y\ x'}\,(\mathsf{Inst})}{\vdash \mathtt{S}\ (\mathtt{add}\ x'\ y) = \mathtt{S}\ (\mathtt{add}\ y\ x')}\,(\mathsf{Cong}) \qquad \vdots \quad \vdash \mathtt{S}\ (\mathtt{add}\ y\ x') = \mathtt{add}\ y\ (\mathtt{S}\ x')}{\vdash \mathtt{S}\ (\mathtt{add}\ x'\ y) = \mathtt{add}\ y\ (\mathtt{S}\ x')}\,(\mathsf{Trans})}{\vdash \mathtt{add}\ (\mathtt{S}\ x')\ y = \mathtt{add}\ y\ (\mathtt{S}\ x')}\,(\mathsf{Reduce})$$

Figure 4.4: A partial proof of commutativity of addition.

It is disingenuous to claim that nothing has been synthesised when reading this proof in a strictly bottom-up, i.e. goal-oriented, manner as transitivity seemingly introduces a new expression (in pink) for the transitivity step. Our key observation, however, is that endeavouring to form cycles can itself be used to guide equational reasoning. The aforementioned expression can be derived, without synthesis, through a combination of bottom-up and top-down reasoning, exploiting the fact that the equation labelled (1) is already known. Standardly, proof rules that are suitable for goal-oriented reasoning should have only a few variants of the premises that justify a given instance of the conclusion, so that proof search has a low branching factor, which is not the case for the standard presentation of transitivity. In a cyclic proof system, on the other hand, ancestral nodes can also be used as premises allowing more flexible proof rules without drastically increasing the branching factor.

Since, in general, we cannot expect there to be an exact syntactical relationship between the target node and its ancestor, the formation of cycles is closely related to the use of cuts in the proof. Indeed, Tsukada and Unno [41], have demonstrated that many efficient model checking techniques can be viewed as the introduction of cut-like rules into cyclic proofs that allow proof obligations to be discharged earlier. In line with this body of work, we propose to incorporate basic equational reasoning into

the formation of cycles through a cut-like rule (Subst), which uses an ancestral and non-ancestral nodes alike to rewrite a sub-expression of the current proof obligation instead of completely discharging it.

$$(\text{Subst})\,\frac{\vdash a = b \quad \vdash C[b\theta] = c}{\vdash C[a\theta] = c}$$

We refer to the left-hand premise of this rule as the *lemma* and the right-hand premise as the *continuation*. As these names might suggest, the rule says that a given lemma $a = b$ can be used to simplify the current conclusion $C[a\theta] = c$ and emit a new proof obligation $C[b\theta] = c$. This rule doesn't involve any generalisation or synthesis, however, as the choice of lemmas is restricted to pre-existing nodes or externally supplied lemmas. As an example of its application, the proof segment in Figure 4.4 can be compressed to a single instance of (Subst) that our proof search algorithm will successfully discover:

$$(\text{Subst})\,\frac{(1) \quad \vdash \mathtt{S}\ (\mathtt{add}\ y\ x') = \mathtt{add}\ y\ (\mathtt{S}\ x')}{\vdash \mathtt{S}\ (\mathtt{add}\ x'\ y) = \mathtt{add}\ y\ (\mathtt{S}\ x')}$$

Because this acts as a single rule, there is no need to heuristically generate proof segments with the hope of forming a cycle but instead apply it directly to simplify a proof obligation. Note also that the (Inst) rule used in Figure 4.3 is actually shorthand for the combination of (Subst) with a continuation that is trivially discharged by (Refl). That is to say, our contextual substitution rule subsumes instantiation.

Although, in principle, the lemma needn't be derived from the proof's root node, e.g. it may be supplied by a human or conjectured by a theory exploration tool, we found that our proof search algorithm is often able to produce proofs whilst only selecting lemmas that already occur as nodes within the same proof graph without the need to invoke any potentially costly lemma discovery strategies.

4.2 The CycleQ Proof System

In the rest of this chapter, we shall assume an exhaustive and terminating MiniHask program P with program environment Σ and datatype environment Δ.

Definition 4.1. An *equation* is of the form $\Gamma \vdash a = b$, where Γ is a type environment and where a and b are applicative expressions such that $\Gamma \vdash a, b : \tau$. We write Eq for the set of well-formed equations. For convenience, we will also use $\doteq$ to indicate an unordered equation rather than duplicating proof rules or including a specific rule for symmetry.

The type environment of an equation contains any universally quantified variables in addition to the program variables. We will assume that Γ always contains the

program environment Σ and, as usual, that all other variables are monomorphic. Furthermore, we will sometimes omit the equation's type environment as it can typically be inferred from the context.

Standardly, equations are interpreted as the set of valuations, i.e. closing assignments to their universally quantified variables, for which both sides are contextually equivalent according to Definition 2.23.

Definition 4.2. A *valuation* of a type environment Γ is a pair (Θ, θ) where Θ is a type-level substitution such that $\Theta(\alpha)$ is closed for each type variable $\alpha \in \mathsf{FTV}(\Gamma)$ and θ is an expression-level substitution such that $\Sigma \vdash \theta : \Gamma\Theta \setminus \Sigma$. Through the abuse of notation, we will typically identify a valuation just by its expression-level substitution as a suitable type-level substitution can be inferred.

An equation $\Gamma \vdash a = b$ is said to be *satisfied* by a valuation of Γ just if both sides are equivalent under θ, i.e. $a\theta \equiv_P b\theta$, in which case we write $\theta \vDash a = b$. When an equation is satisfied by all valuations, then it is said to be *valid*.

4.2.1 Cyclic Pre-proofs

$$\textsf{(Refl)}\ \frac{}{\Gamma \vdash a \doteq a} \qquad \textsf{(Reduce)}\ \frac{\Gamma \vdash a' \doteq b' \quad a \to_P^* a' \quad b \to_P^* b'}{\Gamma \vdash a \doteq b}$$

$$\textsf{(Cong)}\ \frac{(\forall i \leq n) \quad \Gamma \vdash a_i \doteq b_i}{\Gamma \vdash k\ a_1\ \cdots\ a_n \doteq k\ b_1\ \cdots\ b_n}$$

$$\textsf{(FunExt)}\ \frac{\Gamma \cup \{x : \tau\} \vdash a\ x \doteq b\ x}{\Gamma \vdash a \doteq b} \quad \text{where } \Gamma \vdash a,\ b : \tau \to \sigma$$

$$\textsf{(Subst)}\ \frac{\Gamma_2 \vdash a \doteq b \quad \Gamma_1 \vdash C[b\theta] \doteq c}{\Gamma_1 \vdash C[a\theta] \doteq c}\ \Gamma_1 \vdash \theta : \Gamma_2\Theta \setminus \Sigma$$

$$\textsf{(Case)}\ \frac{(\forall k \in K)\ \Gamma \cup \Gamma_k \vdash a[k\ x_1\ \cdots\ x_n/x] \doteq b[k\ x_1\ \cdots\ x_n/x]}{\Gamma \cup \{x : d\ \overline{\tau}\} \vdash a \doteq b}$$
$$\text{where } K = \overline{\mathrm{dom}(\Delta(d))}$$
$$\text{and } \Gamma_k = \{x : \Delta(d)(k)[\overline{\tau/\alpha}]\}$$

Figure 4.5: The inference rules for CycleQ pre-proofs.

As previously mentioned, the infinitary nature of cyclic proofs means that they require an additional condition in order to correspond to sound arguments. Thus, the standard approach to defining cyclic proofs first introduces *pre-proofs* before considering the subclass of proofs proper [77, 84]. In this section, we define pre-proofs and the inference rules that can be used to construct them.

In addition to the (Subst) rule, the proof system consist of: reflexivity for discharging trivial equations, a rule for simplifying an equation according to the reduction relation, congruence for constructors, function extensionality, and a rule for performing case analysis on variables. The proof rules are named according to their goal-oriented interpretation, hence the reductions in the (Reduce) rule go from conclusion to premise and, in the case of the (Subst) rule, the titles *lemma* and *continuation* are used for left- and right-hand premises respectively.

Although the congruence rule can be simulated by (Subst), we distinguish it because it is not intended as a mechanism for creating cycles. This rule is restricted to only apply to constructors so that it can be applied eagerly during proof search without lost of generality. Note also that transitivity can be simulated by (Subst) and symmetry is included by the use of unordered equations.

Definition 4.3. A *cyclic pre-proof* is a tuple $(V,\ E,\ \lambda,\ \rho)$ consisting of:

- A finite set of *nodes* V and an *edge function* $E : V \rightharpoonup V^*$ that determines the underlying structure of the proof graph, mapping conclusions to their premises.

  For $E(v) = v_1 \cdots v_n$, we say that each v_i is a *child* of v and, when a node has no children we say it is a *leaf*. Notices that the edge function distinguishes nodes with no children from those for which the edge function is undefined. When $E(v)$ is undefined, we say that v is an *axiom*, and we denote the set of such nodes as Ax. Unlike other nodes, axioms are not justified within the proof and thus allow for lemmas supplied externally, e.g. as guided by the user or from a lemma discovery tool.

- Each node is labelled with an equation by $\lambda : V \to \mathsf{Eq}$ and each non-axiomatic node is labelled with an instance of a rule from Figure 4.5 by $\rho : V \setminus \mathrm{Ax} \to \mathsf{Rule}$, which are subject to the requirement that, for any node $v \in V \setminus \mathrm{Ax}$ with children $E(v) = v_1,\ \ldots,\ v_n$, the following inference is well-formed.

$$\rho(v)\ \frac{\lambda(v_1) \quad \cdots \quad \lambda(v_n)}{\lambda(v)}$$

  We informally assume that ρ fully determines the value of meta-variables in the inference rules. For example, if $\rho(v)$ is (Subst), then the substitution and context are also known.

A trivial example of a pre-proof can be constructed using substitution to rewrite any equation according to itself, thus assuming the exact equation which is to be proven, as in Figure 4.6. Recall that, when depicting cyclic proofs, the premise (1) is to be understood as a reference to the labelled node 1: $\vdash \texttt{True} = \texttt{False}$, which forms the cycle in this case.

$$\textsf{(Subst)}\ \frac{\textsf{(1)}\quad \textsf{(Refl)}\ \dfrac{}{\vdash \texttt{False} = \texttt{False}}}{1{:}\ \vdash \texttt{True} = \texttt{False}}$$

Figure 4.6: A trivially unsound pre-proof.

Although this example clearly illustrates that pre-proofs do not necessarily represent sound arguments, they are, however, *locally sound* in the sense that the premises of each inference rule justify its conclusion. In the context of a pre-proof, this property is witnessed by directly relating each valuation of an equation to valuations of its premises that are necessary for concluding that it is satisfied.

Definition 4.4. Let $(V,\ E,\ \lambda,\ \rho)$ be a pre-proof. For a pair of nodes $v_1,\ v_2 \in V$ such that v_2 is a child of v_1, i.e. a premise, with valuations θ_1 and θ_2 of their respective equations $\lambda(v_1)$ and $\lambda(v_2)$, the *necessary precursor* relation $(v_1,\ \theta_1) \rightarrowtriangle (v_2,\ \theta_2)$ is defined as follows depending on the rule $\rho(v_1)$:

- (Refl) As there are no premises to reflexivity, there are no necessary precursors.
- (Reduce), (Cong) Across these rules, the necessary precursor of a valuation is just the valuation itself. That is, $(v_1,\ \theta_1) \rightarrowtriangle (v_2,\ \theta_1)$ where v_2 is any child of the node v_1.
- (FunExt) When the type environment Γ is extended with a fresh variable $x : \tau_1$ in accordance with function extensionality, a valuation of the sole premise v_2 is a necessary precursor if it agrees on all variables other than the fresh variable. That is, $(v_1,\ \theta_1) \rightarrowtriangle (v_2,\ \theta_2)$ whenever $\theta_1(y) = \theta_2(y)$ for all $y \in \mathrm{dom}(\Gamma)$.
- (Subst) Let θ be the substitution instance of the lemma.
  - The necessary precursor of the first premise, i.e. the lemma, is related to the valuation of the conclusion by the instance of the lemma in question. That is, $(v_1,\ \theta_1) \rightarrowtriangle (v_2,\ \theta\theta_1)$ when v_2 is the lemma.
  - For the continuation, on the other hand, the necessary precursor is the same valuation: $(v_1,\ \theta_1) \rightarrowtriangle (v_2,\ \theta_1)$ when v_2 is the continuation.
- (Case) Suppose $x : d\,\overline{\tau}$ is the variable upon which case analysis is performed and $x_1,\ \ldots,\ x_n$ are the fresh variables. We know that the expression $\theta(x)\!\downarrow_P$ is of

the form $k\ a_1\ \cdots\ a_n$ for some $k \in \Delta(d)$. Thus, $(v_1,\ \theta_1) \rightarrowtriangle (v_2,\ \theta_1 \cup \{\overline{x \mapsto a}\})$ whenever v_2 is the premise associated with this constructor. Clearly, no other premise is necessary as, by Lemma 2.11, there is no other way to satisfy the equation implicit in case analysis.

Consider the partial proof taken from Figure 4.3:

$$\dfrac{\dfrac{\dfrac{\vdots}{\vdash \mathtt{mapE}\ \mathtt{id}\ y_1 = y_1}\ (\text{Subst})\ \dfrac{(1) \qquad \dfrac{}{\vdash y_1 = y_1}\ (\text{Refl})}{\vdash \mathtt{mapE}\ \mathtt{id}\ y_1 = y_1}}{\vdash \mathtt{App}\ (\mathtt{mapE}\ \mathtt{id}\ y_1)\ (\mathtt{mapE}\ \mathtt{id}\ y_2) = \mathtt{App}\ y_1\ y_2}\ (\text{Cong})}{\vdash \mathtt{mapE}\ \mathtt{id}\ (\mathtt{App}\ y_1\ y_2) = \mathtt{App}\ y_1\ y_2}\ (\text{Reduce})}{\vdash \mathtt{mapT}\ \mathtt{id}\ x = x}\ (\text{Case})$$

A valuation θ of the conclusion may be of the form $\{x \mapsto \mathtt{App}\ a_1\ a_2\}$ where a_1 and a_2 are closed expressions. In which case, its satisfaction only depends on the App case. Hence, there the valuation $\theta \cup \{y_1 \mapsto a_1,\ y_2 \mapsto a_2\}$ associated with the right-hand premise is a necessary precursor and there are no necessary precursors for the left-hand premise. Now considering the node justified by (Subst) with the lemma 1: $\vdash \mathtt{mapE}\ \mathtt{id}\ x = x$, there are two necessary precursors to the valuation $\theta \cup \{y_1 \mapsto a_1,\ y_2 \mapsto a_2\}$: one associated with the continuation, which is the same valuation, and one associated with the lemma, corresponding to the specific instance in use, namely $\{x \mapsto a_1\}$.

The following states the essential property that precursors must witness — the contrapositive of local soundness, i.e. from an invalid conclusion, we can derive an invalid premise.

Theorem 4.1 (Local Soundness)**.** Let $v_1 \in V \setminus \mathrm{Ax}$ be a non-axiomatic node within the pre-proof $(V,\ E,\ \lambda,\ \rho)$ such that $\theta_1 \not\models \lambda(v_1)$ for some valuation θ_1. Then, there exist a child node $v_2 \in E(v_1)$ with a necessary precursor θ_2, i.e. where $(v_1,\ \theta_1) \rightarrowtriangle (v_2,\ \theta_2)$, for which $\theta_2 \not\models \lambda(v_2)$.

4.2.2 Global Soundness

In a proof system with finite derivation trees, local soundness is sufficient for concluding the validity of any given node by induction — an invalid node must have an invalid child and so on until we reach a leaf. However, in a cyclic proof system, we may never reach a leaf and instead follow an infinite sequence of invalid nodes. In the pre-proof from Figure 4.6, for example, there is an infinite sequence of necessary precursors $(1,\ \emptyset) \rightarrowtriangle (1,\ \emptyset) \rightarrowtriangle \cdots$ that follows from the fact that the root node is justified in terms of itself.

The soundness of a cyclic pre-proof depends instead on the precursor relation being well-founded, i.e. there are no such infinite sequences. In essence, for any given

valuation, we should be able to restrict and unfold a sound pre-proof into a finite derivation tree that is still a sufficient witness for that instance. Clearly, this would exclude our trivially false pre-proof as the sequence of precursors remains constant.

Definition 4.5. Within a given pre-proof $(V,\ E,\ \lambda,\ \rho)$, sequences of necessary precursors follow *paths* — a (possibly) infinite sequence of nodes $(v_i)_{i\in\mathbb{N}}$ where $v_{i+1} \in E(v_i)$ is a child of $v_i \in V$ for each $i \in \mathbb{N}$. A *path segment* is a finite sequence of nodes that meets the same criteria.

In a proof without cycles, there are only finitely many paths and, consequently, the necessary precursor relation is well-founded. To ensure the necessary precursor relation is well-founded in general, however, each path is equipped with a corresponding sequence of expressions that, once instantiated with the precursors, is infinitely decreasing. If the order on expressions is itself well-founded, then the necessary precursor relation must be as well. We refer to this property as the *global soundness condition*.

Definition 4.6. A partial order on applicative expressions $\leq$ is said to be *stable* if the inequality $a \leq b$ implies $a\theta \leq b\theta$ for any substitution θ.

Definition 4.7. Let $\leq$ be a well-founded, stable partial order. A $\leq$-*trace* along the path $(v_i)_{i\in\mathbb{N}}$ is an infinite sequence of applicative expressions $(t_i)_{i\in\mathbb{N}}$ subject to the following constraints:

- If $\lambda(v_i) = \Gamma_i \vdash a_i = b_i$, then $\mathsf{FV}(t_i) \subseteq \mathrm{dom}(\Gamma_i \setminus \Sigma)$, i.e. the trace expression may only depend on the free variables of the equation associated with the given node.
- If $\rho(v_i)$ is (Case) for some $i \in \mathbb{N}$ where $x : d\,\overline{\tau}$ is the variable upon which case analysis is performed and v_{i+1} is the premise associated with the constructor $k \in \Delta(d)$ using fresh variables $x_1, \ldots, x_n$, then $t_{i+1} \leq t_i[k\ x_1\ \cdots\ x_n/x]$.
- If $\rho(v_i)$ is (Subst) with substitution θ and v_{i+1} is the lemma, then $t_{i+1}\theta \leq t_i$ and, if v_i is the continuation, then $t_{i+1} \leq t_i$.
- Finally, in all other cases, it is required that $t_{i+1} \leq t_i$.

In each of the above cases, when the inequality is strict $t_{i+1} < t_i$, we say that the index i is a *progress point* of that trace. As with paths, a finite sequence of applicative expressions meeting the above criteria is referred to as a *trace segment*.

Naturally, the content of a trace depends on the cyclic proof system in question. In Brotherston's work on first-order logic with inductive definitions, for example, traces consists of inductively defined atoms appearing in the antecedent and progress points occur through unfolding their definition [77]. As we discuss in a future section, part

of our intention with this work is to relate cyclic proofs to rewriting induction [46]. To accommodate this goal, we do not consider a fixed ordering on expressions, and, as a result, the proof rules responsible for creating progress points will vary. For example, our implementation of proof search is based on the substructural order where progress points are introduced by the (Case) rule, but they would arise from the (Reduce) rule under a reduction order. It is for this reason that progress points are not tied to a specific proof rule as is the case in Brotherston's generic cyclic proof system [85].

The following lemma shows that the requirement placed on traces is sufficient to witness a decrease once instantiated with a sequence of necessary precursors along its path. In particular, if a path has a trace with infinitely many progress points and the trace's ordering is well-founded, there can be no corresponding infinite sequences of precursors. And, if every path has such a trace, the precursor relation is well-founded.

Lemma 4.2. Let $(V,\ E,\ \lambda,\ \rho)$ be a cyclic pre-proof with some path $(v)_{i\in\mathbb{N}}$ and suppose $(t)_{i\in\mathbb{N}}$ is a $\leq$-trace along this path. If θ_i is a valuation of some node $v_i \in V$ and $(v_i,\ \theta_i) \twoheadrightarrow (v_{i+1},\ \theta_{i+1})$, then $t_{i+1}\theta_{i+1} \leq t_i\theta_i$ and $t_{i+1}\theta_{i+1} < t_i\theta_i$ when i is a progress-point.

A cyclic pre-proof is said to satisfy the *global soundness condition* if every path has a suffix that can be equipped with a trace that has infinitely many progress points. In which case, we will refer to it as a *cyclic proof*. Allowing for traces that only cover a suffix of their associated paths makes the global soundness conditional more general as traces might not extend to the root node. For example, a trace may follow an expression that is introduced by an inference rule such as (Subst). Moreover, if a path doesn't have an infinitely decreasing trace, then there exists an ultimately periodic path that doesn't have an infinitely decreasing trace, although it is not the case that every path is ultimately periodic. This property follows from the fact that non-empty ω-regular languages (e.g. paths without an infinitely decreasing trace) necessarily contain a regular word (e.g. an ultimately period path without an infinitely decreasing trace). Therefore, it suffices to find a trace for every cycle, rather than every infinite path, where the traces of this cycle need not extend to the root. It is not, however, always sufficient to assign a single trace expression to each node as distinct cycles may overlap.

Lemma 4.3. Let $(V,\ E,\ \lambda,\ \rho)$ be a cyclic pre-proof and $\leq$ a stable, well-founded partial order. If, for every path $(v_i)_{i\in\mathbb{N}}$, there is some index $j \in \mathbb{N}$ and a $\leq$-trace $(t_i)_{i\in\mathbb{N}}$ along the corresponding suffix $(v_{i+j})_{i\in\mathbb{N}}$ with infinitely many progress points. Then, the precursor $\twoheadrightarrow$ relation is well-founded.

It follows from local soundness that a valuation not satisfying the equation of a given node induces an infinite sequence of necessary precursors, which contradicts the preceding lemma if the global soundness condition is satisfied. Therefore, every node in a cyclic proof is labelled by a valid equation.

Theorem 4.4 (Global soundness)**.** Let (V, E, λ, ρ) be a cyclic proof such that, for every axiom $v \in \mathrm{Ax}$, the associated equation $\lambda(v)$ is valid. Then, for every other node $v \in V \setminus \mathrm{Ax}$, the associated equation $\lambda(v)$ is also valid.

Commutativity of Addition Revisited

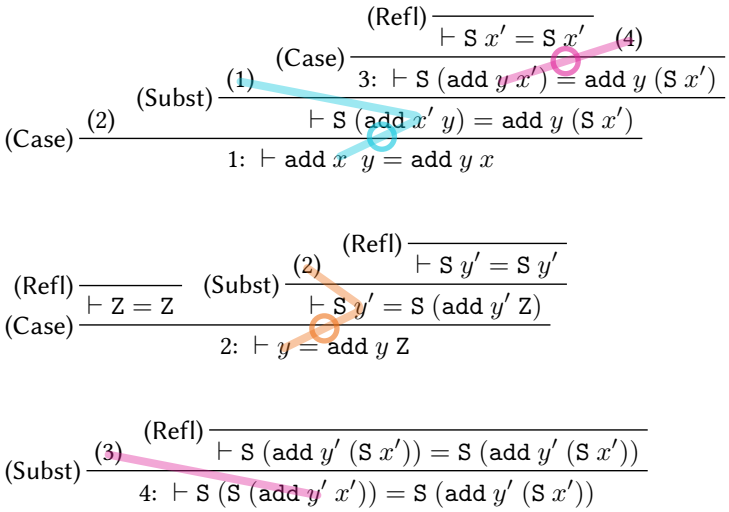

Figure 4.7: A cyclic proof of the commutativity of addition.

As an example of a global soundness condition, consider the pre-proof in Figure 4.7 for the commutativity of addition where (Reduce) nodes have been elided for compactness. We will show that this is indeed a cyclic proof using the substructural ordering.

Lemma 4.5. The substructural order $\trianglelefteq$ on applicative expressions, defined as follows, is well-founded and stable.

$$a \trianglelefteq b \iff \exists C[\cdot].\, C[a] = b$$

To this end, we must consider each of the following cycles and show they each have a trace with infinitely many progress points. These traces are informally depicted as coloured lines in the pre-proof with progress points marked by circles, following their presentation in [81].

Blue There is a cycle starting at node (1) that follows the successor case and the lemma of the (Subst) rule, returning to the root node. For its trace, consider the sequence of expressions x, x', x, ... This sequence is a valid trace as it meets the two criteria $x' \triangleleft x[\mathtt{S}\ x'/x]$ and $x[x'/x] \trianglelefteq x'$ for the (Case) and (Subst)

rules respectively. What is more, in the first edge along this path, the trace strictly decreases via the transition from the variable x to the variable x', which corresponds to a proper sub-expression in any sequence of necessary precursors along this path. Therefore, this trace segment contains a progress point. As we are considering a cycle, this trace will pass through infinitely many progress points as required.

Orange, **Pink** There are two similar cycles starting at nodes (2) and (3) respectively that again follow the successor case and the lemma of the (Subst) rule, before returning to their respectively start nodes. A trace for each of these cycles is constructed much like the previous trace but following the expressions y, y', y, ... and equally has infinitely many progress points.

4.3 Rewriting Induction

In the introduction to this chapter, we contrasted cyclic proofs with explicit induction, by which we mean proof systems based on induction schemes. However, cyclic proof systems are not the only paradigmatic alternatives to be proposed in response to the shortcomings of automated explicit induction. In this section, we will explore the relationship between cyclic proofs and two such alternatives: "inductionless induction" and "implicit induction".

4.3.1 Inductionless and Implicit Induction

Inductionless Induction

Originally proposed by Musser in the 80s, "inductionless induction" or "proof by consistency" relies on a rare proof theoretic property— *strong completeness* [86] A proof system is strongly complete if every consistent formula is a theorem. In which case, one can prove a formula is a theorem by assuming it holds and then demonstrating that there is no contradiction [45]. The strongly consistent proof system in question is inductive equational logic where an equation holds whenever all closed instances are derivable from the standard inference rules of equational reasoning, i.e. reflexivity, symmetry, transitivity, congruence, and instantiation. Assuming a sufficiently expressive theory, consistency is encoded by an assertion which is evidently true under the initial model, e.g. `False` $\neq$ `True`.

The difficulty associated with proving the consistency of a theory limits the application of this strategy to saturation-based systems, such as the Knuth-Bendix procedure, that dynamically derive consequences [87]. The Knuth-Bendix procedure takes as input a set of universally quantified equations and, when successful, produces an equivalent, confluent and terminating rewrite system. If the equation in question, in

conjunction with the original theory, can be converted into such a rewrite system, then it is straightforward to test that `False` $\neq$ `True` by comparing their unique normal forms.

A more recent line of work has investigated the integration of induction into standard saturation-based theorem proving [88, 89]. However, as the theorem to be shown is incorporated as a goal clause, i.e. in its negated form, and combined with an induction axiom, these approaches do not fall into the category of inductionless induction.

Implicit Induction

At a high-level, the Knuth-Bendix completion proceeds by simplifying and deleting equations before orienting them so as to be interpreted as rewrite rules. The orientation is determined by a well-founded order on expressions that is compatible with context and substitution so that no infinite reduction sequences can occur, thus guaranteeing termination. In order to produce a confluent rewrite system, the procedure must also consider *critical pairs* — expressions that can be rewritten in two different ways by overlapping rules. Critical pairs are resolved by the addition of a new hypothesis to be incorporated.

A large body of work considers a specialisation of this procedure to the task of performing inductionless induction [90]. In particular, the "inductive completion" procedure directly incorporates rules governing constructors, eschewing the need for an additional consistency check [91]. A further refinement of the inductive completion procedure noted that only those critical pairs arising between the original theory and the hypotheses need to be considered, not the critical pairs deduced from hypotheses themselves [92]. Although such frameworks are still categorisable as inductionless induction, their soundness is no longer derived from confluence. Later, Reddy distilled the simple essence of this mechanism into 'Term rewriting induction', now often referred to simply as rewriting induction, which relies on the well-founded nature of the ordering directly [46]. Rewriting induction is not a form of proof by consistency but, nevertheless, avoids selecting an explicit induction scheme beforehand; hence, "implicit induction". As part of this work, Reddy also demonstrated how this approach subsumes proof by consistency under the Knuth-Bendix completion procedure.

Beware the Old Jade and the Zombie!

> *Do not to fumble around with the zombie of "inductionless induction"! The experts in implicit induction have spent a lot of time with it, mostly to bury it — Clause-Peter Wirth [93]*

The inductionless and implicit approaches to inductive theorem proving are pleasingly, yet deceptively, simple. Ultimately, they must still solve the same problem as

faced by the explicit approach — namely, finding a finite cyclic representation for an infinite argument.

Saturation may be guaranteed, eventually, to expose any inconsistencies, but it will encounter many superfluous inferences in the process, limiting its efficiency [94]. The refinements of the completion procedure attempt to ameliorate this limitation but do not address the fact that equations must be orientable with respect to the fixed ordering. As a result, these systems are not only highly sensitive to the choice of order but are thwarted by theorems such as the commutativity of addition, the symmetry of which is inherently unorientable.

Although there are extensions to the completion procedure and rewriting induction that allow for unorientable equations, the increase in complexity detracts from the principal advantage of these approaches, namely their simplicity [95, 96]. Instead of further refining the completion strategy, the most actively developed automated theorem prover based on implicit induction — SPIKE — has incorporated some elements of explicit induction [97, 98]. And, as far as the authors are aware, there is no active automated theorem prover based on proof by consistency alone.

4.3.2 Translation into CycleQ

Despite the aforementioned problems with inductionless and implicit induction, these approaches share many of the advantages of cyclic proofs. For example, they naturally support mutual induction and do not require a fixed induction scheme in advance. In the remainder of this section, we will show that inductionless induction is subsumed by our cyclic proof calculus, presented in Section 4.2, using rewriting induction as a stepping stone. The key observation is that the unconstrained use of hypotheses in Reddy's system naturally gives rise to the structure of cyclic pre-proofs, and that global soundness is guaranteed by construction as the lemmas are oriented. From the subsumption of rewriting induction, it follows transitively that proof by consistency is also subsumed:

$$\text{Inductive Completion} \overset{\text{Section 5 [46]}}{\leq} \text{Rewriting Induction} \overset{\text{Theorem 4.10}}{\lesssim} \text{CycleQ}$$

One of the principal developments of the inductive completion procedure and rewriting induction was its independence from confluence, although the rewrite system is still required to be terminating. Because we target functional programs however, the equational theory is inherently confluent and thus CycleQ only subsumes rewriting induction in this case. Nevertheless, it is worth noting that the soundness of our proof system is not derived from confluence.

Expansion and Case Analysis

As previously mentioned, rewriting induction implicitly considers critical pairs, i.e. expressions that can be rewritten in multiple ways, that arise between the program and the equations that are being proven (but not between these equations and any existing hypotheses).

As our rewrite system is defined by a set of strictly orthogonal rewrite rules, it is sufficient to consider *basic* expressions to determine the critical pairs. We then define the critical pairs associated with an equation via the *expansion* operator that unifies a given basic sub-expression with a reduction rule from the program.

Definition 4.8. A basic expression is an applicative expression of the form $f\ p_1\ \cdots\ p_n$ where $f \in \mathrm{dom}(\Sigma)$ is a program variable and p_i are (possibly non-linear) patterns.

Definition 4.9. For a given context $C[\cdot]$, the expansion operator is defined as follows:

$$\begin{aligned}&\mathsf{Expand}(C[f\ p_1\ \cdots\ p_n] = b) = \\ &\quad \{C[c]\theta = b\theta \mid P(f)\ p'_1\ \cdots\ p'_n \Downarrow_\emptyset c,\ \theta = \mathrm{mgu}(\overline{p}, \overline{p}')\}\end{aligned}$$

where $f\ p_1 \cdots p_n$ is a basic expression, $p'_1, \ldots, p'_n$ are patterns, and the substitution $\theta = \mathrm{mgu}(\overline{p}, \overline{p'})$, when it is defined, denotes the most-general unifier such that $p_i\theta = p'_i\theta$ for all $i \leq n$.

The expansion of a basic expression generates a "cover set" of substitutions — a set of instances to which every closed expression reduces. As our rewrite rules are determined by patterns, it effectively performs case analysis instantiating variables with constructors. The first component of our translation from rewriting induction to CycleQ takes advantage of this fact to simulate expansion via a case analysis tree. However, another critical part of the definition of expansion is that a reduction step has occurred, and the left-hand side is, therefore, strictly smaller. In other words, it marks a progress point under the reduction order. Each branch of the constructed tree thus must also simulate this reduction step.

Lemma 4.6. For any equation $\Gamma \vdash C[a] = b$ where $\Gamma \vdash a : d\ \overline{\tau}$ is a basic expression, there is a finite derivation tree using only instances of the (Case) rule and exactly one instance of the (Reduce) rule per branch where the leaves are labelled by equations from the set $\mathsf{Expand}(C[a] = b)$ and the root is labelled $\Gamma \vdash C[a] = b$.

From Derivations to Pre-proof

Definition 4.10. A well-founded partial order on applicative expressions $\leq$ is *stable* if $a \leq b$ implies $a\theta \leq b\theta$ for any substitution θ. If such an order is also closed under context, i.e. $a \leq b$ implies $C[a] \leq C[b]$ for any applicative context $C[\cdot]$, then we say it is a *reduction order*.

An example of a reduction order is $\rightarrow_P^*$ for a terminating program P. For this section, we shall assume that $\leq$ is a fixed reduction order that is compatible with the program's reduction relation. That is, if $a \rightarrow_P b$, then $a > b$. One could consider the reduction relation itself as the choice of ordering. However, this would be overly limiting as lemmas and induction hypotheses must also be orientable, and are unlikely to be mere instances of reduction.

Definition 4.11. The inference rules of rewriting induction manipulate pairs $(H,\ G)$ where H consists of oriented equations $\Gamma \vdash a = b$ such that $a > b$ and G is a set of unoriented equations, by which we mean the set is closed under symmetry. We refer to these sets as the *hypotheses* and *goals* respectively. The judgement $\vdash (H,\ G)$ defined inductively in Figure 4.8 expresses that all the goals are valid when all the hypotheses are also valid.

$$\textsf{(End)}\ \frac{}{\vdash (H,\ \emptyset)} \qquad \textsf{(Delete)}\ \frac{\vdash (H,\ G)}{\vdash (H,\ G \cup \{\Gamma \vdash a \doteq a\})}$$

$$\textsf{(Simplify)}\ \frac{\vdash (H,\ G \cup \{\Gamma \vdash a' \doteq b\})}{\vdash (H,\ G \cup \{\Gamma \vdash a \doteq b\})}\ a \rightarrow_P^* a'$$

$$\textsf{(Hypothesis)}\ \frac{\vdash (H,\ G \cup \{\Gamma \vdash C[a'\theta] \doteq b\})}{\vdash (H,\ G \cup \{\Gamma \vdash C[a\theta] \doteq b\})}\ \begin{array}{l}\Delta \vdash a = a' \in H \\ \Gamma \vdash \theta : \Delta\Theta\end{array}$$

$$\textsf{(Expand)}\ \frac{\vdash (H \cup \{\Gamma \vdash C[a] = b\},\ G \cup \textsf{Expand}(C[a] = b))}{\vdash (H,\ G \cup \{\Gamma \vdash C[a] \doteq b\})}$$
$$\text{where } C[a] < b$$
$$\text{and } \Gamma \vdash a : d\ \overline{\tau} \text{ is basic}$$

Figure 4.8: The proof rules of rewriting induction.

As the program P is not presented as a set of rewrite rules, we have two separate rules for the use of hypotheses and the program's reduction relation, whereas rewriting induction was originally presented with a single simplification rule that combined the two. Although the hypotheses are treated as rewrite rules that supplement the program's reduction relation, they needn't be strictly orthogonal. For example, $\vdash \texttt{add}\ (\texttt{add}\ x\ y)\ z = \texttt{add}\ x\ (\texttt{add}\ y\ z)$ is a valid hypothesis despite overlapping with the existing rules governing the reduction of `add` and matching against a non-pattern expression.

The translation into a cyclic pre-proof proceeds by structural induction on the rewriting induction derivation, following the close correspondence between the in-

ference rules of the two systems:

$$\begin{array}{lcl} \text{(Delete)} & \rightsquigarrow & \text{(Refl)} \\ \text{(Simplify)} & \rightsquigarrow & \text{(Reduce)} \\ \text{(Hypothesis)} & \rightsquigarrow & \text{(Subst)} \\ \text{(Expand)} & \rightsquigarrow & \text{(Case), (Reduce)} \end{array}$$

As cyclic pre-proofs discharge their hypotheses globally rather than locally, intermediate stages in the translation use axioms to simulate hypotheses that are justified later in the construction of the pre-proof.

Lemma 4.7. If $\vdash (H, G)$ is a rewriting induction derivation, then there exists a cyclic pre-proof (V, E, λ, ρ) where $\lambda(\mathrm{Ax}) \subseteq H$ and, for each goal $\Gamma \vdash a \doteq b$ in the set G, there is a node $v \in V \setminus \mathrm{Ax}$ such that $\lambda(v) = \Gamma \vdash a \doteq b$ modulo orientation.

Global Soundness by Construction

The pre-proof constructed in Lemma 4.7 can be equipped with the structure of a cyclic proof, i.e. a trace with infinitely many progress points for a suffix of each path, using a variation of the reduction order. The trace follows the left-hand side of each equation in the pre-proof, where progress points occur as a result of the reduction step in Lemma 4.6. For this trace to be preserved following the (Subst) rule into the lemma, we need an ordering that also includes the sub-expression relation. We can construct a suitable order with this property from any given reduction order.

Definition 4.12. The reflexive-transitive closure of the union of the partial order $\leq$ and the substructural order, i.e. $(\leq \cup \trianglelefteq)^*$, is referred to as the *substructural extension* of $\leq$ and is denoted $\leq_{\mathrm{sub}}$.

Lemma 4.8. If $\leq$ is a reduction order, then $\leq_{\mathrm{sub}}$ is stable and well-founded.

We will now show that the global soundness condition is indeed satisfied using the substructural extension of the given ordering. It is plain to see that all cycles arising in the constructed pre-proof use a hypothesis that was introduced via (Expand). As this rule is represented as a case analysis tree that ultimately enables a reduction step, there is a progress point in each branch.

Lemma 4.9. For any rewriting induction derivation $\vdash (H, G)$, the pre-proof constructed in Lemma 4.7 satisfies the following invariants:

- Along a path $(v_i)_{i \in \mathbb{N}}$, let $\lambda(v_i) = \Gamma_i \vdash a_i = b_i$ be the corresponding equation. The sequence of left-hand sides $(a_i)_{i \in \mathbb{N}}$ is monotonically decreasing with respect to the substructural extension of the reduction order $\leq$.

- Within every cycle $v_1, \ldots, v_n$, there is at least one $i \leq n$ for which $\rho(v_i)$ is an instance of (Reduce) and where the trace expression has a progress point.

Theorem 4.10. For any rewriting induction derivation $\vdash (\emptyset, G)$, there exists a cyclic proof (V, E, λ, ρ) where, for all goals $\Gamma \vdash a \doteq b \in G$, there is a node $v \in T \setminus \mathrm{Ax}$ such that $\lambda(v) = \Gamma \vdash a \doteq b$ modulo orientation.

As a simple example of our translation, Figure 4.9 shows a rewriting induction derivation of the equation $\vdash \mathtt{append}\ xs\ [] = xs$ where H denotes the hypothesis set $\{\vdash \mathtt{append}\ xs\ [] = xs\}$. The corresponding cyclic proof is shown in Figure 4.10 where the trace, i.e. the left-hand expression of each equation, is indicated in blue. Note the progress point occurs at the reduction step.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash (H, \emptyset)}\,\text{(End)}}{\vdash (H, \{\vdash y :: ys = y :: ys\})}\,\text{(Delete)}}{\vdash (H, \{\vdash y :: \mathtt{append}\ ys\ [] = y :: ys\})}\,\text{(Hypothesis)}}{\vdash (H, \{\vdash [] = [], \vdash y :: (\mathtt{append}\ ys\ []) = y :: ys\})}\,\text{(Delete)}}{\vdash (\emptyset, \{\vdash \mathtt{append}\ xs\ [] = xs\})}\,\text{(Expand)}$$

Figure 4.9: A rewriting induction derivation of $\vdash \mathtt{append}\ xs\ [] = xs$.

$$\dfrac{\dfrac{\dfrac{}{\vdash y :: ys = y :: ys}\,\text{(Refl)}}{\vdash y :: \mathtt{append}\ ys\ [] = y : ys}\,\text{(1)}\,\text{(Subst)}}{\vdash \mathtt{append}\ (y :: ys)\ [] = y :: ys}\,\text{(Reduce)}}{1\colon \vdash \mathtt{append}\ xs\ [] = xs}\,\text{(2)}\,\text{(Case)}$$

$$\dfrac{\dfrac{}{\vdash [] = []}\,\text{(Refl)}}{2\colon \vdash \mathtt{append}\ []\ [] = []}\,\text{(Reduce)}$$

Figure 4.10: The cyclic proof corresponding to the rewriting induction derivation in Figure 4.9.

Since our publication of this work, it has separately been shown that rewriting induction proofs can be transformed into a restricted form of cyclic proof and back again [99, 100]. The cyclic proof systems in these results, however, are not specialised for equational reasoning and the relationship is instead characterised by treating inference rules as rewriting rules over sequents. Furthermore, their translation only applies to cut-free proofs and thus is not able to simulate our calculus. The combination of implicit and explicit induction employed in the SPIKE prover, which closely

resembles a cyclic proof system, is also known to subsume rewriting induction. However, the conjectures are associated with a history to be individually checked in lieu of a more general global soundness condition [97].

4.4 Efficient Proof Search

Thus far, we have seen a cyclic proof system specialised to equational reasoning. Its subsumption of rewriting induction, a significantly more constrained system, already provides one possible proof search algorithm. In the implementation of our CycleQ tool, however, we follow a different approach that is less hampered by the use of a reduction ordering to orient lemmas so that it can serve as a basis for a practical inductive theorem prover.

It is not hard to see a tension between being more general and focusing on efficiency. Not requiring lemmas to be oriented immediately leads to an intractable number of lemmas with many redundancies. Furthermore, as the cyclic proof will no longer satisfy the global soundness condition by construction, we must also consider its verification. In this section, we will address these concerns by restricting both the creation and application of lemmas and adapting an existing technique concerning program termination to verify the global soundness condition.

4.4.1 Needed Sub-expressions

As with the Zeno inductive theorem prover [32], a guiding principle of our proof search strategy is the aim to reduce expressions. Intuitively, each reduction step provides more information about the behaviour of an expression and thus provides an opportunity to witness the equivalence or inequivalence of two expressions. This directive manifests itself in the prioritisation of (Reduce) over any other rule and the strategy by which variables chosen for case analysis are selected. The possible variables selected for case analysis are those that are "needed" in order to reduce the target equation. More specifically, a variable is needed within some expression if it must ultimately be instantiated by a constructor-lead application for the parent expression to reach a normal form, i.e. some part of the parent expression's definition branches on the value of the variable [101]. Although the program's reduction relation isn't tied to a particular evaluation strategy, whether a variable is needed or not is determined by a lazy evaluation strategy.

Definition 4.13. The set of *needed sub-expressions* $\mathsf{Need}(a)$ for a given applicative expression a is defined as follows:

$$\begin{aligned}
&\mathsf{Need}(x\ a_1\ \cdots\ a_n) \coloneqq \emptyset \\
&\mathsf{Need}(k\ a_1\ \cdots\ a_n) \coloneqq \emptyset \\
&\mathsf{Need}(f\ a_1\ \cdots\ a_n) \coloneqq \mathsf{Need}^{P(f)}_{\emptyset}(a_1, \ldots, a_n)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{Need}^{a}_{\theta}(a_1, \ldots, a_n) \coloneqq \mathsf{Need}(a\theta\ a_1\ \cdots\ a_n) \\
&\mathsf{Need}^{\lambda x.\, d}_{\theta}(a_1, \ldots, a_n) \coloneqq \mathsf{Need}^{d}_{\theta \cup \{x \mapsto a_1\}}(a_2, \ldots, a_n) \\
&\mathsf{Need}^{\mathtt{case}\ x\ \mathtt{of}\ \{k_i\ \overline{x}_i \mapsto d_i \mid i \leq n\}}_{\theta}(a_1, \ldots, a_n) \\
&\qquad \coloneqq \begin{cases}
\mathsf{Need}^{d_i}_{\theta \cup \{\overline{x_i \mapsto b_i}\}}(a_1, \ldots, a_n) & \text{if } \theta(x) = k_i\ b_1\ \cdots\ b_\ell \\
\theta(x) \cup \mathsf{Need}(\theta(x)) & \text{otherwise}
\end{cases}
\end{aligned}$$

A *needed variable* is merely a variable sub-expression that is needed. We do not consider the set of needed variables directly as, in the subsequent chapter, we will discuss an extension of CycleQ with support conditional equations whereby case analysis can be performed on whole sub-expressions.

The set of needed sub-expressions is naturally in close correspondence with the program reduction relation, alternating between applicative and definitional expressions. It is well-defined for a terminating program and, in the implementation, is computed simultaneously alongside the expression's normal form. The following two lemmas show the correctness of the definition, i.e. a needed sub-expression is indeed a proper sub-expression and that it must be instantiated for the parent expression to reduce to a normal form.

Lemma 4.11. Suppose $\Gamma \vdash a : \tau$ and $b \in \mathsf{Need}(a)$, then b is a proper datatype sub-expression, i.e. $b \lhd a$ and $\Gamma \vdash b : d\ \overline{\tau}$, and is not an application with a constructor in head position.

Lemma 4.12. If $a \to_P^* k\ a_1\ \cdots\ a_n$, then $\mathsf{Need}(a)$ is empty.

Note the final case in the definition of a needed sub-expression: if a definitional expression attempts to pattern match on an argument that is not an application with a constructor in head position, then both the scrutinee itself and, transitively, any needed sub-expressions of the scrutinee are considered needed. For example, in the following expression both $\mathtt{leq}\ x\ y$ and x are considered needed.

$$\mathtt{case}\ \mathtt{leq}\ x\ y\ \mathtt{of}\ \{\cdots\}$$

While performing case analysis on x will provide us with more precise information; it is not necessarily more advantageous than performing case analysis on $\texttt{leq}\ x\ y$ due to the non-analytic nature of induction.

4.4.2 Refining Cycle Formation

While the substitution of equals for equals is an appropriate technique for detecting cycles, its unconstrained use also creates many redundancies during proof search, incurring a severe performance penalty. Instead of requiring lemmas or their instances to be oriented, we restrict lemmas based on the inference rule used to justify them. Specifically, only those nodes that are axioms or justified by (Case) or (FunEx) can be used as lemmas in the (Subst) rule.

Despite its simplicity, this rule-based restriction is surprisingly effective. For example, while the cyclic proof of the commutativity of addition in Figure 4.7 has 12 nodes, only 2 are applicable lemmas. Nevertheless, we conjecture that any proof that respects the aforementioned strategy where nodes are normalised before proceeding can be simulated under this restriction. We informally justify this claim by considering the following set of proof transformations, which translates part of the lemma's proof into the continuations proof.

- (Refl) Clearly, any lemma justified by reflexivity is redundant.
- (Reduce)
$$\textsf{(Subst)}\dfrac{\textsf{(Reduce)}\dfrac{\vdash a' \doteq b'}{\vdash a \doteq b} \quad \textsf{(Reduce)}\dfrac{\vdash b'' \doteq c}{\vdash C[b\theta] \doteq c}}{\vdash C[a\theta] \doteq c}$$

  For a proof fragment of this form, we already know the conclusion $C[a\theta] \doteq c$ is in normal form, else the (Reduce) rule would be applied instead. Therefore, it must also be the case that a is in normal form and the reduction step only applies to the complementary side of the lemma, i.e. $a = a'$ and $b \to_P^* b'$. By stability under substitution and context, the continuation $\vdash C[b\theta] \doteq c$ is evidently reducible and thus must be justified by the reduction $C[b\theta] \to_P^* b''$ as, again, equations are eagerly normalised by the (Reduce) rule.

  To translate such a proof into a proof where the lemma is not justified by reduction is straightforward as we can use the reduced lemma in its place. It follows from confluence that the new continuation $C[b'\theta]$ also reduces to b'', and thus we may reuse the continuation from the original proof as its justification:

$$\textsf{(Subst)}\dfrac{\vdash a \doteq b' \quad \textsf{(Reduce)}\dfrac{\vdash b'' \doteq c}{\vdash C[b'\theta] \doteq c}}{\vdash C[a\theta] \doteq c}$$

Note that the possible traces annotating paths through the proof remain unchanged as no new cycles have been introduced, merely cut short, and trace expressions are unaffected by the (Reduce) rule.

- (Cong)

$$\text{(Subst)}\ \dfrac{\text{(Cong)}\ \dfrac{(\forall i \leq n) \quad \vdash a_i \doteq b_i}{\vdash k\ a_1\ \cdots\ a_n \doteq k\ b_1\ \cdots\ b_n} \quad \vdash C[(k\ b_1\ \cdots\ b_n)\theta] \doteq c}{\vdash C[(k\ a_1\ \cdots\ a_n)\theta] \doteq c}$$

The application of a lemma justified by congruence can be simulated by applying each of its premises in sequence, as in the following proof fragment. Again, the space of possible traces is unchanged as (Cong) does not affect trace expressions.

$$\text{(Subst)}\ \dfrac{\begin{array}{cc} & \vdash a_n \doteq b_n \quad \vdash C[(k\ b_1\ b_2 \cdots\ b_n)\theta] \doteq c \\ \iddots & \vdots \\ \vdash a_1 \doteq b_1 & \vdash C[(k\ b_1\ a_2 \cdots\ a_n)\theta] \doteq c \end{array}}{\vdash C[(k\ a_1\ \cdots\ a_n)\theta] \doteq c}$$

- (Subst)

$$\text{(Subst)}\ \dfrac{\text{(Subst)}\ \dfrac{\vdash a \doteq b_1 \quad \vdash C_1[b_1\theta_1] \doteq b_2}{\vdash C_1[a\theta_1] \doteq b_2} \quad \vdash C_2[b_2\theta_2] \doteq c}{\vdash C_2[(C_1[a\theta_1])\theta_2] \doteq c}$$

Finally, we consider lemmas that are themselves justified by the use of another lemma. In this case, we can perform a re-association of the nested (Subst) using the composition of contexts and substitutions to directly apply the latent lemma to the conclusion.

$$\text{(Subst)}\ \dfrac{\vdash a \doteq b_1 \quad \text{(Subst)}\ \dfrac{\vdash C_1[b_1\theta_1] \doteq b_2 \quad \vdash C_2[b_2\theta_2] \doteq c}{\vdash C_2[(C_1[b_1\theta_1])\theta_2] \doteq c}}{\vdash C_2[(C_1[a\theta_1])\theta_2] \doteq c}$$

The traces following the continuations are unchanged, as in the previous cases. Any path passing through both lemmas in the original proof is equipped with trace expression t_1, t_2, t_3 such that $t_1 \geq t_2\theta_2$ and $t_2 \geq t_3\theta_1$. By stability, we can use the trace $t_1 \geq t_3\theta_1\theta_2$, corresponding to the composed substitutions, to annotate the new path segment.

For traditional proof systems with finite derivation trees, the above local transformation could be extended by induction to eliminate all instances of the (Subst) rule from a complete proof tree that do not comply with our search strategy. The same can not easily be said of a cyclic proof system where the creation, deletion, and alteration of proof nodes can impact a completely different part of the proof. Instead, we argue that this transformation can be applied at the frontier of proof search. In particular, as our proof search strategy is goal-oriented and uses a lemma to derive a new proof obligation, i.e. the continuation, it is fair to assume that the nodes of the continuation are not used elsewhere in the proof and can be modified freely.

A natural question to ask is whether this restriction is without loss of generality: if there exists a cyclic proof with a given conclusion, is it possible to re-create it so that only axiomatic lemmas and those justified by (Case) and (FunEx) are used? It is important to set aside those proofs that cannot be discovered by our proof search algorithm when answering this question, hence the assumption that nodes are normalised with (Reduce) in the above transformations. Unfortunately, it still is not obvious that our transformation can simulate any such proof as there are two important caveats to the final case:

- First, the equation $C_1[b_1\theta_1] \doteq b_2$ was emitted as obligation in the original proof but, appearing as a lemma in the resulting proof, is implicitly assumed to exist elsewhere.

- Second, the new expression $C_2[(C_1[b_1\theta_1])\theta_2]$ appearing in the continuation might be reducible; in which case, the aforementioned lemma would not be applied even if it did exist as the (Reduce) rule takes priority.

Although the resulting proof is well-formed, it is not necessarily within the target fragment, i.e. "discoverable", and we cannot claim that the reduction is complete without further investigation. However, we are yet to find a proof that can be discovered in a goal-oriented manner that falls outside this fragment.

4.4.3 Verifying Cycles

As our definition of a trace and its progress points is highly generic, it is undecidable whether a sufficient set of traces exists or not, even once given an ordering. This is a significant divergence from proofs by structural induction, whose validity is effectively a syntactic well-formedness condition, and existing cyclic proof systems with a finite space of possible traces [77]. As the declarative proof system is designed to support a number of proof search algorithms, we opted to retain generality and do not, for example, restrict trace expressions to be a sub-expression of its associated node as is done with cyclic arithmetic [102].

Nevertheless, having given up the correct-by-construction approach to cyclic proof search that relies on overly restrictive expression orderings, we must limit our attention to a decidable fragment of traces. In particular, we only consider traces composed of variables and restrict our attention to those that are well-formed under the substructural ordering. In this case, the global soundness condition becomes decidable.

A comparable result was first shown by reduction to Büchi automata in the context of a cyclic proof system for the modal μ-calculus with explicit approximants [35]. In essence, two ω-regular languages are extracted from a cyclic pre-proof, capturing the language of paths and traces respectively. It can then be decided whether the path language is included in the trace language, thus ensuring every path has an infinitely progressing trace. Since then, this approach has been adapted to a number of cyclic proof systems including Brotherston's work on first-order logic with inductive definitions [77].

Unfortunately, deciding the inclusion of Büchi automata involves the construction of the complement automaton which is, in the worst case, exponential in the number of states and is known to produce large automata in practice [103]. Here the number of states in the automata is proportional to the number of nodes in the proof graph. Therefore, the procedure becomes too onerous if several candidate proofs, the majority of which may be unsound, need to be checked throughout the proof search. In the Cyclist theorem prover, for example, verifying the global soundness condition could take a significant proportion of the proof time [42]. This approach to verifying cyclic proofs fails to take advantage of the incremental nature of the goal-oriented proof search where possible pre-proofs share a common prefix, and thus much of the work can be re-used. Furthermore, as soon as a cycle that does not satisfy the global condition is detected, there is no benefit in attempting to complete the pre-proof. Thus, maintaining the global soundness condition as an invariant of proof search, rather than verifying it upon completion, is advantageous.

To this end, we adapt previous work on termination analysis to develop a more incremental solution to the problem of verifying global soundness. The edges of a cyclic pre-proof are annotated with an abstract domain that encodes the ω-regular language of traces — size-change graphs [103]. Size-change graphs are composed along path segments to encode the space of possible traces. Their compositionality allows the workload to be performed as each node is uncovered, through a generalised transitive closure of the pre-proof. As a result, the global soundness condition is represented explicitly. Furthermore, our size-change based approach is only cubic in the number of states and exponential in the number of variables within proof node, which is likely to be significantly smaller.

It is no coincidence that the termination problem and global soundness condition are intimately linked. The Curry-Howard correspondence tells us that traditional well-founded proofs can be seen as well-typed programs. The generalisation of this

correspondence that applies to cyclic pre-proofs produces programs with general recursion. For the logical interpretation to be sound, however, the programs must terminate, or else some types will be incorrectly inhabited by divergence. Interactive theorem provers such as Agda do not require the explicit use of structural induction (e.g. combinators capturing recursion schemes) but instead admit recursive programs that are subsequently checked for termination [104]. The underlying logic of such systems is thus most aptly described by a cyclic proof system.

Although the size-change based approach to verifying the global soundness condition has previously been alluded to, we are unaware of any cyclic proof system that implements it [85]. After the publication of the work on which this chapter is based, a closely related "Ramsey-based" trace condition was formalised for an abstract cyclic proof calculus [84].

Size-Change Graphs

Definition 4.14. Let (V, E, λ, ρ) be a cyclic pre-proof with two nodes $v_1, v_2 \in V$ labelled by the equations $\Gamma_1 \vdash a_1 = b_1$ and $\Gamma_2 \vdash a_2 = b_2$ respectively. A *size-change graph* $G : v_1 \to v_2$ between these two nodes is a labelled bipartite graph whose nodes are drawn from the non-program variables of Γ_1 and Γ_2. That is, G is a subset of triples (x_1, ℓ, x_2) consisting of $x \in \mathrm{dom}(\Gamma_1 \setminus \Sigma)$, $y \in \mathrm{dom}(\Gamma_2 \setminus \Sigma)$, and some label $\ell \in \{=, \triangleright\}$. We refer to edges labelled by $=$ as *equality edges* and those labelled by $\triangleright$ as *progress edges*.

As we shall see later, size-change graphs describe the set of possible traces along a given path segment. More specifically, an edge (x, ℓ, y) indicates that a valid trace segment can be formed starting from the variable x and ending with the variable y and, when labelled $\triangleright$, that the trace contains a progress point. Crucially, the size-change graphs for any given path segment do not need to be constructed independently but are the result of composition.

Definition 4.15. If $G_1 : v_1 \to v_2$ and $G_2 : v_2 \to v_3$ are two size-change graphs, then the composition $G_1 \circ G_2 : v_1 \to v_3$ is defined as follows:

$$G_1 \circ G_2 \coloneqq \{(x_1, \ell_1 \circ \ell_2, x_3) \mid (x_1, \ell_1, x_2) \in G_1, (x_2, \ell_2, x_3) \in G_2\}$$

where $\ell_1 \circ \ell_2$ is defined as $\triangleright$ just if either label is $\triangleright$ and $=$ otherwise.

The composition of size-change graphs includes an edge $(x_1, =, x_3)$ whenever there exists an intermediate variable x_2 and two edges $(x_1, =, x_2)$ and $(x_2, =, x_3)$, and likewise there is a progressing edge if either edge is a progressing. Intuitively, this definition corresponds to the fact that a trace segment $x_1, \ldots, x_2$ can be combined with a trace segment $x_2, \ldots, x_3$ between their respective nodes and that the resulting trace segment will contain a progress point if either sub-trace has a progress point.

Definition 4.16. Initially, a size-change graph $G_{v_1,\, v_2}$ is associated with each pair of nodes $v_1,\, v_2 \in V$ in a cyclic pre-proof $(V,\, E,\, \lambda,\, \rho)$ where $v_2 \in E(v_1)$ is a child of $v_1 \in V$. As size-change graphs are intended to capture the space of possible traces along their path segments, the definition of this family is dependent on the justification of v_1 mimicking Definition 4.7:

- When $\rho(v_1)$ is an instance of (Reduce), (Cong), or (FunEx), the size-change graph $G_{v_1,\, v_2}$ is the size-change graph $\{(x,\, =,\, x) \mid x \in \mathrm{dom}(\Gamma)\}$, where Γ is the type environment consisting of non-program variables that are common to both equations.

- When $\rho(v_1)$ is (Case) and v_2 is the premise associated with the constructor k and substitution $\{x \mapsto k\; x_1\; \cdots\; x_n\}$, the corresponding size-change graph has a progressing edge $(x,\, \triangleright,\, x_i)$ for all $i \leq n$ as x_i is a proper sub-expression of the x under the necessary precursor of a valuation. For all other non-program variables $x \in \Gamma \setminus \Sigma$, there is an equality edge $(x,\, =,\, x)$.

- Finally, when $\rho(v_1)$ is an instance of (Subst) using the substitution θ and v_2 is the lemma, there is an equality edge $(\theta(y),\, =,\, y) \in G_{v_1,\, v_2}$ whenever $\theta(y)$ is variable for some $y \in \mathrm{dom}(\Delta)$ in the lemma's type environment. This size-change graph simply captures the corresponding trace condition $t_1\theta \leq t_2$ but restricted to variable trace expressions. Otherwise, if v_2 is the continuation, there is an equality edge $(x,\, =,\, x)$ for any non-program variable $x \in \Gamma \setminus \Sigma$.

The following two results formalise the intuition that size-change graphs represent possible trace segment along a given path segment.

Lemma 4.13. Let $(V,\, E,\, \lambda,\, \rho)$ be a cyclic pre-proof with nodes $v_1,\, v_2 \in V$ where $v_2 \in E(v_1)$. Then, whenever there is an edge $(x_1,\, \ell,\, x_2) \in G_{v_1,\, v_2}$, there is also a trace $x_1,\, x_2$ along this path segment. Furthermore, if labelled $\triangleright$, this trace segment has a progress point.

Corollary 4.14. Let $(V,\, E,\, \lambda,\, \rho)$ be a cyclic pre-proof with a path segment $v_1,\, \ldots,\, v_n$ where $n > 1$. Then, whenever there is an edge $(x_1,\, \ell,\, x_n)$ in the composition of size-change graphs along this path segment $G_{v_1,\, v_2} \circ \cdots \circ G_{v_{n-1},\, v_n}$, there is also a trace $x_1,\, \ldots,\, x_n$ consisting solely of variables along this path segment. Furthermore, if labelled $\triangleright$, this trace segment has a progress point.

Having defined the size-change graphs for each edge in a pre-proof and shown that they do indeed give rise to trace segments, we can derive trace segments for a given path segment in the pre-proof by composing the size-change graphs along it. This process is applied to all path segments in the pre-proof by the construction of the *closure* of the proof graph.

Definition 4.17. For a given cyclic pre-proof $C = (V, E, \lambda, \rho)$, we define $\mathsf{Cl}(C)$ as the least family of size-change graphs such that:

- For each $v_1, v_2 \in V$ such that G_{v_1, v_2} is defined, $G_{v_1, v_2} \in \mathsf{Cl}(C)$.
- If $G_1 : v_1 \to v_2 \in \mathsf{Cl}(C)$ and $G_2 : v_2 \to v_3 \in \mathsf{Cl}(C)$, then $G_1 \circ G_2 \in \mathsf{Cl}(C)$.

It is plain to see that the closure of a cyclic pre-proof is finite as there are finitely many nodes, each of which has finitely many variables, and thus there can only be finitely many size-change graphs between two nodes. Nevertheless, this property should not be disregarded as it is essential for deriving decidability. The following theorem, derived from [103], shows that the global soundness condition can be decided just by verifying that all idempotent size-change graphs in the closure have a variable that decreases to itself, the infinite iteration of which clearly constitutes an infinitely progressing trace.

Theorem 4.15 (Ramsey's theorem)**.** Let $\mathbb{N}_2$ denote the 2-element sets $\{i, j\} \subseteq \mathbb{N}$ and suppose $f : \mathbb{N}_2 \to A$ is a function from these sets to a finite set A. Then there exists some $a \in A$ and an infinite set $I \subseteq \mathbb{N}$ such that, for any 2-element set $\{i, j\} \subseteq I$, we have that $f(\{i, j\}) = a$.

Theorem 4.16. Suppose $C = (V, E, \lambda, \rho)$ is a cyclic pre-proof for which every size-change graph $G : v \to v \in \mathsf{Cl}(C)$ in the closure that is *idempotent*, in that $G \circ G = G$, has a progressing edge $(x, \triangleright, x) \in G$. Then C satisfies the global soundness condition and is a cyclic proof.

Proof. Suppose, for the sake of contradiction, that $C = (V, E, \lambda, \rho)$ is a cyclic pre-proof that does not satisfy the global soundness condition. Therefore, there must exist a path $(v_i)_{i \in \mathbb{N}}$ with no infinitely progressing trace.

Consider the function that associates a pair of indices $i < j \in \mathbb{N}$ to the size-change graph $G_{v_i, v_{i+1}} \circ \cdots \circ G_{v_{j-1}, v_j} \in \mathsf{Cl}(C)$. Ramsay's theorem implies the existence of some infinite set of indices $I \subseteq \mathbb{N}$ for which any pair $i < j \in I$ corresponds to the same size-change graph, which we shall denote $G^\star$. That is, the subset of size-change graphs from the closure $\{G_{v_i, v_{i+1}} \circ \cdots \circ G_{v_{j-1}, v_j} \mid i < j \in I\} \subseteq \mathsf{Cl}(C)$ is uniquely inhabited by $G^\star$.

By the pigeonhole principle, some node $v^\star \in V$ must be visited infinitely often in the sub-sequence generated by indices I. That is to say, $G^\star \in \mathsf{Cl}(C)$ is associated with a path from $v^\star$ to $v^\star$. Furthermore, from any three indices $i < j < k \in I$, we can observe that $G^\star$ is idempotent:

$$\begin{aligned} G^\star &= (G_{v_i, v_{i+1}} \circ \cdots \circ G_{v_{j-1}, v_j}) \circ (G_{v_j, v_{j+1}} \circ \cdots \circ G_{v_{k-1}, v_k}) \\ &= G^\star \circ G^\star \end{aligned}$$

It thus follows from our assumption that there exists some variable x with a progressing edge $(x, \triangleright, x)$ in $G^\star$. And, therefore, by Corollary 4.14 there is a trace seg-

ment $x, \ldots, x$ with a progress point for any path segment $v_i, \ldots, v_j$ with $i < j \in I$. Finally, as there are infinitely many indices from I in the path, we can construct an infinitely progressing trace from each of these trace segments. □

Although we do not formally reproduce the proof here, size-change based termination is also complete for variable traces, see [103]. That is, if a cyclic pre-proof satisfies the global soundness condition using only variable traces, then every size-change $G : v \to v \in \mathsf{CI}(C)$ such that $G \circ G = G$ has a progressing edge $(x, \triangleright, x) \in G$. Intuitively, if this were not the case and there were such an idempotent size-change graph G without a progressing edge, we could construct an infinite path by iterating this path segment whose traces are described by G (as it is idempotent) but for which there is no corresponding progressing trace.

Complexity Analysis

Although the size-change based termination problem is also known to be worst-case exponential-time, in practice it is often more efficient than checking the inclusion of Büchi automata due to the subsumption between size-change graphs [103]. The exponential worst-case complexity comes from the number of possible size-change graphs between two nodes $\mathcal{O}(3^{m \cdot m})$ where m is the maximum number of variables appearing in any equation of a pre-proof graph. In practice, however, m is likely to be small and only a few of the possible size-change graphs will be encountered in a given pre-proof. Assuming a fixed upper-bound on the number of variables, the construction of the closure is cubic $\mathcal{O}(n^3)$ in the number of proof nodes as it can be derived from the generalisation of the Floyd-Warshall algorithm [105].

Our principal motivation for switching to size-change based global soundness, however, is the possibility of constructing the closure of the pre-proof in an incremental manner during proof search. The addition of a fixed number of nodes only incurs a cost of $\mathcal{O}(n^2)$ in the worst-case [106]. Thus, if we assume proof search proceeds up to depth d and each stage introduces a constant number of nodes, then the worst-case complexity associated with computing the closure incrementally for the ℓ^{th} stage of proof search is $\mathcal{O}(\ell^2 \cdot 2^\ell)$ and the sum over all stages:

$$\begin{aligned} &\mathcal{O}\left(\sum_{\ell=1}^{d} \ell^2 \cdot 2^\ell\right) \\ &\quad = \mathcal{O}\left(d^2 \cdot \sum_{\ell=1}^{d} 2^\ell\right) \\ &\quad = \mathcal{O}\left(d^2 \cdot 2^d\right) \end{aligned}$$

Verifying the leaves of proof search in a non-incremental manner, on the other hand, is asymptotically worse with a complexity of $\mathcal{O}(d^3 \cdot 2^d)$ as there are $\mathcal{O}(2^d)$ leaves each of which has $\mathcal{O}(d)$ nodes.

Alternative Global Soundness Conditions

As the global soundness condition is inherently non-local and cycles may overlap, several different traces segment may be required for any given cycle. A *trace manifold*, as proposed by Brotherston, is an alternative trace-based condition where only basic cycles are equipped with trace segments but restricted in such a way that they can be "glued together" to form consistent traces for derived cycles [85]. Although the trace manifold condition is less complex, it requires pre-proofs to be in *cycle normal form* where the companion of each bud is one of its ancestors. The conversion of an arbitrary pre-proof into cycle normal form can result in an exponentially larger proof and thus brings into question whether any efficiency is gained in practice.

An alternative line of work validates the global soundness condition locally for each basic cycle using ordering constraints [107, 108]. Although their algorithm is worst-case polynomial-time, it is unclear exactly what class of cyclic proofs are verifiable under this approach. On the other hand, a restriction of size-change based termination has been developed that is also worst-case polynomial-time with the intuition that trace terms cannot "move between" variables [109]. We do not adopt this restriction as again it is not clear what impact this has on the space of provable properties or the size of their corresponding proofs.

4.5 Implementation

We implemented a prototype of our CycleQ as a plugin for GHC 9.2.8. As a core plugin, it receives the small subset of Haskell used internally by the GHC compiler and converts this code into a MiniHask program. The user adds properties of interest to their program as top-level functions, using the following syntax, and the plugin will attempt to prove them at compile-time.

```
{-# ANN mapId CycleQ.assertion #-}
mapId :: List α → Formula
mapId xs = map id xs ≡ xs
```

Figure 4.11: Haskell syntax for specifying equations to be proven.

The pragma specifies to the CycleQ plugin that the associated definition is to be interpreted as an assertion that it will then attempt to prove. Each universally quantified variable is encoded as an argument to a function whose body ultimately

dictates the form of the equation. The pragma in Figure 4.11, in particular, specifies that CycleQ should attempt to construct a proof of the equation `map id` $xs = xs$.

The annotation additionally allows the user to specify a number of parameters for proof search: the "fuel" which controls the depth of proof search, any axiomatic lemmas that can be used, and optionally an output destination to which a visualisation of the proof is written if proof search is successful.

The authors of the Cyclist theorem prover originally conjectured that breadth-first search would be most efficient when attempting to construct a cyclic proof due to the possibility of using non-ancestral links which significantly reduce the size of proofs [42]. However, in practice, it was observed that the high branching factor makes depth-first search more practical. These analyses, in fact, belong to two orthogonal dimensions: the choice of node to be justified and the higher-level strategy used by proof search to select from candidate proofs. In our implementation, we select nodes within a partial proof in a first-in-first-out order as to retain the possibility of discovering non-ancestral links, e.g. using cousin nodes as lemmas. At the proof search level, on the other hand, we employ iterative deepening depth-first proof search where the depth bound is determined by the fuel parameter. Although iterative deepening ultimately pursues proofs in a breadth-first search manner, our proof search algorithm has a lower branching factor due to the optimisations discussed in the preceding sections and thus performs well in practice. Furthermore, iterative deepening favours smaller proofs where the cost associated with verifying the global soundness condition is generally lower whilst avoiding the large memory requirements of breadth-first search.

When several of the inference rules from Section 4.2 are applicable, they are prioritised in the following order: (Refl), (Cong), (FunEx), (Reduce) with no prioritisation given to (Case) or (Subst). Proof search applies the former rules eagerly without backtracking, whereas, if none of them are applicable, the latter two rules produce a branch point in the search space. In the implementation, this behaviour is encoded by only reducing the fuel, i.e. increasing the depth, when an instance of the (Case) or (Subst) rules are applied. While we have not formally justified the relative completeness of this strategy, it greatly reduces the branching factor of proof search as previously discussed and, in practice, doesn't appear to limit the set of provable properties.

4.5.1 Performance

There are very few implementations of cyclic proof systems, and their performance on equational problems is not well understood. The Cyclist system, which is certainly the most developed, is known to have difficulty with equational reasoning [42]. The primary objective of this evaluation is to demonstrate that our system, although simple, is reasonably efficient and the incremental use of size-change based termination

provides a significant improvement over the offline approach, which itself is known to be more efficient in practice than approaches based on Büchi automata [103]. As mentioned in the introduction, our objective was not to develop a tool that solves as many problems as possible without human intervention, but rather one that could adequately serve as a robust basis for a more interactive model of proof search.

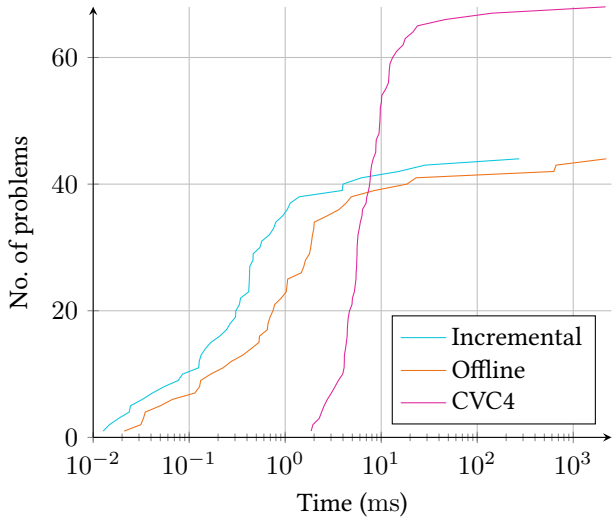

Figure 4.12: Number of benchmark problems solved within the given time bound.

We tested the tool against a standard benchmark suite of 85 induction problems concerning natural numbers, lists, and trees, originally used to test the IsaPlanner tool [74]. The results were obtained as an average over 10 runs with the fuel parameter set to 10 on the windows sub-system for Linux, running on an 11th Gen Intel® Core™ i7–1185G7 @ 3.00GHz/1.80GHz processor with 16.0GB of RAM. The number of benchmark problems solved in a given time is shown in Figure 4.12, including both the incremental and offline approach to checking the global soundness condition. Our tool was able to solve 44 of the problems (14 were not in scope as they concerned conditional equations), with most being solved in under 1ms and all but one in under 30ms. Furthermore, these results confirm that verifying the global soundness condition in an incremental manner is superior to the offline approach in terms of performance. On average, it was approximately 5 times faster and, in the worst-case, it was nearly 60 times faster.

Since the original benchmark set was not designed to test mutual induction, and indeed many tools simply do not support mutual induction, we also designed a small number of problems around the representation of annotated, mutually recursive syntax trees, as shown in the introduction to this chapter. The mutual induction bench-

marks were solved, on average, in approximately 1.5ms. The specification of these problems can be found in Figure 2 of the appendix.

We have compared the efficiency of our prototype implementation with CVC4 version 1.8 using the `--quant-ind` flag and a time out of 30s [110]. Unlike most other automated inductive theorem provers, it is possible to disable sub-goal generation in CVC4, i.e. limit strengthening and lemma synthesis. This restriction means that we are able to draw a fair comparison as the heuristics used in sub-goal generation and more complex techniques such as theory exploration naturally increase the cost of proof search. As the figure demonstrates, CycleQ was able to solve a number of benchmark problems significantly faster. It is hard to draw a comprehensive comparison from this data, however, as CVC4 ultimately solves more problems for reasons we discuss below.

4.5.2 Limitations

Although the tool performs efficiently on those 44 benchmark problems that it is able to solve, this number is relatively small. By comparison: HipSpec solved 80, Zeno 82, CVC4 80 (68 without sub-goal generation), ACL2 74, Inductive Horn Clause Solving 68, IsaPlanner 47, and Dafny 44 (as reported by [73, 111]). Our analysis of the problems that CycleQ was unable to solve indicates that its limitations can be attributed to two missing features: *conditional equations* and *lemma discovery*. Both limitations are essentially orthogonal to cyclic reasoning; they are not a consequence of our cycle formation method nor our overall proof search strategy. In the subsequent chapter, we will outline an extension to CycleQ with conditional equations that will enable a more meaningful comparison with CVC4.

Many of the standard benchmark problems are conditional equations and are thus out of scope for the proof system presented in this chapter. What is more, the tool's inability to express conditional equations, and perform hypothetical reasoning, limits its ability to prove unconditional equations. Many properties concern functions that are defined by pattern matching on non-variable expressions, such as `count`, `filter`, and `sort`. As we will discuss in the following chapter, the proofs of such properties often depend on imitating the program definitions by performing case analysis on the scrutinees. In general, case analysis of a non-variable expression cannot be reduced to a finite case analysis of the variables it concerns, and thus it is necessary to introduce a conditional equation to express these hypothetical cases.

Most comparable tools incorporate some form of lemma discovery technique, which is very powerful but orthogonal to this work. Of the remaining 7 benchmark problems, all were designed to test a tool's capacity to synthesise lemmas and indeed required lemmas before being proven by the state of the art HipSpec theorem prover [112]. Although CycleQ does not attempt to synthesis the necessary lemmas, it is able to solve most of these problems when supplied with very simple lemmas.

Specifically, property 47 was provable if the commutativity of `max` was assumed, likewise properties 54, 65, 69, and 81 were provable if the commutativity of `add` was assumed.

It is worth noting, however, that CycleQ solved a number of the benchmark problems designed to test hypothesis strengthening and lemma discovery, despite not having a specialised tactic for either. For example, problem 50 was solved in approximately 0.5ms with no user supplied lemmas. By comparison, HipSpec fails to prove the same result after approximately 40s, an attempt that involved 15 synthesised lemmas, 4 of which failed [112]. This particular problem illustrates the need for a robust basis for inductive theorem proving regardless of heuristics for generalisation or synthesis — one which is not sensitive to problem specific parameters such as the choice of induction variable and scheme.

Chapter 5

CycleQ$^{\Rightarrow}$

5.1 Introduction

In the previous chapter, we introduced a cyclic proof system for equational reasoning about functional programs — CycleQ. The main theoretical innovation of this work was the way in which cycle formation and equational reasoning are mediated through the use of a cut-like rule that performs contextual substitution. Although the system demonstrated promising results in terms of efficiency, it's lack of support for conditional equations seriously limited the number of verification problems for which it could automatically derive a proof. In particular, 14 out of the 85 standard benchmark problems were themselves conditional equations. What is more, as this chapter will demonstrate, the inability of the system to prove an additional 20 unconditional problems is due to the need to perform conditional reasoning internally within proofs.

```
count : Nat → List Nat → Nat
count x [] = Z
count x (y :: ys) =
  case x == y of
    True → S (count x y)
    False → count x y

∀x xs. count x (x :: xs) ≡ S (count x xs)
```

Figure 5.1: An example function that depends on the case analysis of a non-variable expression.

To understand why conditional reasoning is necessary, even when trying to proof an unconditional property, consider the program and equational property in Figure 5.1. Upon reducing the left-hand side of this equation under the program defi-

nitions, we see that reduction is blocked by the needed sub-expression $x == x$. One way to proceed with proving the stated equation is to perform case analysis on this sub-expression. In the case where we assume that $x == x = \texttt{True}$, the resulting proof obligation may be trivially discharged by using this hypothesis to simplify the target equation, and the complementary case can be proven to be absurd by induction on x. The proof in Figure 5.2 gives a fragment of this proof where (Reduce) is used to indicate the application of a hypothesis.

$$\dfrac{\dfrac{\dfrac{}{\vdash x == x = \texttt{True} \Rightarrow \texttt{S}\ (\texttt{count}\ x\ xs) = \texttt{S}\ (\texttt{count}\ x\ xs)}\ (\text{Refl})}{\vdash x == x = \texttt{True} \Rightarrow \texttt{count}\ x\ (x :: xs) = \texttt{S}\ (\texttt{count}\ x\ xs)}\ (\text{Reduce})}{\vdash \texttt{count}\ x\ (x :: xs) = \texttt{S}\ (\texttt{count}\ x\ xs)}\ (\text{Case}) \quad \vdots$$

Figure 5.2: A partial proof of $\texttt{count}\ x\ (x :: xs) = \texttt{S}\ (\texttt{count}\ x\ xs)$.

In order to perform case analyses as described in the previous example, it is necessary to express conditional equations and perform hypothetical reasoning. In particular, the assumed equations cannot be characterised by finite case analysis on variables alone. It is worth noting that, in this particular example, it would also be possible to conjecture, and then prove, the desired equation as the alternative case is absurd. However, this strategy does not extend to properties such as $\texttt{all}\ (\texttt{filter}\ p\ xs) = \texttt{True}$ where the needed sub-expressions of the form $p\ x$ do not take on a single, universal value.

In this chapter, we present a new proof system CycleQ$^{\Rightarrow}$ which extends our cyclic proof system with support for conditional reasoning. To avoid drastically increasing the size of the search space, whilst accommodating this increase in expressivity, we develop two specialised mechanisms for managing hypotheses: one for determining how and when hypotheses should be used to simplify the consequent of conditional equations and one for determining which instances of a conditional lemma are applicable in a given context.

Normalising Conditional Equations

The first challenge when integrating conditional reasoning is to determine how and when hypotheses should be used to simplify the consequent of a conditional equation, corresponding to the application of the (Reduce) rule in the previous proof fragment. Ideally, the proof search algorithm should be able to use as much of the information present in the hypotheses as possible to simplify the consequent. However, naïvely considering the arbitrary application of hypotheses would lead to an intractable search space.

One initial solution you might imagine is to represent all equivalent representation of the consequent modulo hypotheses as a single node within a proof, i.e. the equivalence classes generated by the congruence closure of the hypotheses. Although it is possible to compactly represent these equivalence classes using a data structure such as an e-graph, and thereby reduce the redundancy of naïve equational reasoning, there are a number of complications to this approach [113]. First, it would no longer be possible to normalise proof obligations under the program's reduction relation. As each proof obligation would depend on a regular language of expressions under this scheme, normalisation involves constructing the language of normal forms for each element in the original language. However, the set of normal forms is not necessarily regular, and the known algorithms for constructing this set are only guaranteed to terminate in overly restrict cases [114]. As a result, only an approximation to the (Reduce) rule could be used in practice. Second, compact representations of equivalence classes such as e-graphs do not support "destructive" rewriting. That is to say that redexes persist after reduction, nullifying any advantage gained by our prioritisation of reduction discussed in the previous chapter, as well as leading to extremely large memory usage. Finally, detecting the relevant instances of a lemma (i.e. those that relate to a sub-expression of the current proof obligation) would require matching one regular language against another. Here, matching regular language involves instantiating variables, which are treated as elements of the language's signature, so that the two languages have a non-empty intersection. The problem with matching regular languages in practice is that it is known to be EXPTIME-hard [115]. Consequently, computing relevant lemmas could become exponentially costly for complex expressions even before determining whether the lemma's hypotheses are satisfied.

Clearly, neither selecting an arbitrary representation of the consequent through ad-hoc use of hypotheses nor attempting to simultaneously manage each equivalent representation are practical from the perspective of proof search. Instead, we adapt the inductive completion procedure to convert the hypotheses of a conditional equation, in the conjunction with the program definitions, into a confluent and terminating rewrite system. Subsequently, we can derive a normal form for the consequent of a conditional equation that subsumes normalisation under the program's reduction relation whilst integrating much of the information present in its hypotheses. Providing a canonical representative for the consequent in this manner enables the efficiency of our proof search algorithm to extend to the conditional setting as it avoids drastically increasing the search space due to the multitude of ways in which hypotheses could be applied, as well as maintaining a simple method for matching proof obligations against lemmas.

Although the inductive completion procedure results in a terminating rewrite system, the procedure itself may diverge. The essential issue that the inductive completion procedure encounters is the same as that encountered during the normalisation

of a regular language — the inability to finitely represent the equational classes generated by a rewrite system and a set of closed equations. In contrast, our procedure is guaranteed to terminate as we carefully restrict those hypotheses that can be oriented. Instead of rejecting cases with unorientable hypotheses, however, the resulting rewrite system under-approximates the equational theory of the antecedent by marking such hypotheses as unusable. These unusable hypotheses are set aside as they may later become orientable.

Refuting Hypotheses

The second mechanism we introduce for managing hypotheses determines which instances of a conditional lemma are applicable in a given context. In order to understand the full generality of this problem, it will be useful to first explore an additional cycle mechanism that is required when handling conditional equations.

The introduction of hypotheses creates the possibility that a conditional equation is vacuously true, i.e. there are no valuations satisfying the hypotheses and nothing is said about the consequent. It is essential that a proof system for conditional reasoning can complete such branches by refuting the satisfiability of hypotheses. Demonstrating that a set of hypotheses is unsatisfiable, however, may itself require inductive reasoning and thus the formation of cycles.

In the original version of CycleQ, the only mechanism for forming cycles was via the (Subst) rule that uses a lemma to rewrite the goal equation with the objective of making progress towards reflexivity, whereby the branch can ultimately be discharged. When trying to refute the satisfiability of a set of hypotheses, cycles may take on a different form: showing that the hypotheses are unsatisfiable by relating them to the hypotheses of a lemma that is cyclically known to be unsatisfiable. In purely logical terms, there is no need to distinguish these two modes as any equation follows vacuously from unsatisfiable hypotheses and thus can be used to immediately discharge the goal. In practice, however, there is no guarantee of a syntactic correspondence between the vacuously true instance of the lemma and the required equation.

Consider the example problem given in Figure 5.1 where a proof may be constructed by performing case analysis on the needed sub-expression $x == x$. One resulting proof obligation can be discharged by rewriting the consequent according to the hypothesis $x == x = \texttt{True}$ under the normalisation procedure before applying reflexivity. We previously claimed that the complementary branch, with the assumed hypothesis $x == x = \texttt{False}$, can be proven absurd by induction on x. Let us examine the structure of this argument more closely.

The induced proof obligation $x == x = \texttt{False} \Rightarrow \texttt{count}\ x\ (x :: xs) = \texttt{S}\ (\texttt{count}\ x\ xs)$ is first normalised using the given hypothesis, resulting in the sim-

plified conditional equation $x == x = \texttt{False} \Rightarrow \texttt{count}\ x\ xs = \texttt{S}\ (\texttt{count}\ x\ xs)$. To this end, we perform case analysis on the variable x, leaving use with two cases to consider. In the first case, where x is taken to be $\texttt{Z}$, we have the hypothesis $\texttt{Z} == \texttt{Z} = \texttt{False}$ that is clearly unsatisfiable, and so the proof obligation can be immediately discharged. On the other hand, when x is taken to be $\texttt{S}\ x'$, we are left with the following proof obligation after simplification:

$$x' == x' = \texttt{False} \Rightarrow \texttt{count}\ (\texttt{S}\ x')\ xs = \texttt{S}\ (\texttt{count}\ (\texttt{S}\ x')\ xs)$$

If we are to complete this inductive proof using the (Subst) rule to form a cycle, we must instantiate a previously encountered lemma to simplify the current proof obligation. However, the only lemmas that matches the consequent require instantiating x with $\texttt{S}\ x'$, which would not admit an infinitely progressing trace. As a result, we are unable to complete this proof using the existing cycle formation mechanism. The problem here arises from the fact that we should not be focused on the consequent at all; rather, it is the hypotheses that we're trying to show are absurd and the form of consequent is incidental. In particular, we can see that matching the hypothesis $x' == x' = \texttt{False}$ against its ancestor $x == x = \texttt{False}$ would admit an infinitely progressing trace, namely x, x', x etc.

Instead of relying on the syntactic form of the lemma's consequent in these cases, we introduce a new mode of proof search — *refutation*, which is characterised by formulas with no positive literals, i.e. with hypotheses but no consequent. Formulas in refutation mode can no longer be discharged through reflexivity and instead must be shown to have unsatisfiable hypotheses. As lemmas, however, these formulas allow for a more flexible cycle formation mechanism so that branches of a pre-proof can be completely discharged without relying on a syntactic correspondence and rewriting some sub-expression. This mechanism is encoded in the proof system through an additional cut-like rule:

$$(\textsf{Subst})_{\bot}\ \frac{\Gamma_2 \vdash H_2 \Rightarrow \bot \quad \Gamma_1 \vdash \theta : (\Gamma_2 \setminus \Sigma)\Theta}{\Gamma_1 \vdash H_1 \Rightarrow \phi \ \vDash H_1 \Rightarrow H_2\theta}$$

This rule takes a proof obligation that can be either a normal formula or one in refutation mode and discharges it via a lemma that is (cyclically) known to have unsatisfiable hypotheses. As with the original (Subst) rule, the premise serves as a lemma and may be another node within the proof graph or an externally supplied lemma. The key difference with this rule is that it does not require a sub-expression of the conclusion to be matched with the lemma, merely requiring that an instance of the lemma's hypotheses are implied by the conclusion's hypotheses, encoded through the side-condition $\vDash H_1 \Rightarrow H_2\theta$. For example, we may complete the previous example by showing that the obligations hypotheses $x' == x' = \texttt{False}$ imply an instance of

the lemma's hypotheses $(x == x = \texttt{False})[x'/x]$. Figure 5.3 displays this branch of the proof. Note that this proof segment additionally uses the (Refute) rule to move into the proof obligation into refutation mode and uses (Absurd) to discharge trivially unsatisfiable hypotheses. Instances of the (Reduce) rule have been omitted for compactness.

$$\dfrac{\dfrac{\dfrac{}{\vdash \texttt{True} = \texttt{False} \Rightarrow \bot}\,(\text{Absurd}) \quad \dfrac{(1)}{\vdash x' == x' = \texttt{False} \Rightarrow \bot}\,(\text{Subst})_{\bot}}{1{:}\ \vdash x == x = \texttt{False} \Rightarrow \bot}\,(\text{Case})}{\vdash x == x = \texttt{False} \Rightarrow \texttt{count}\ x\ xs = \texttt{S}\ (\texttt{count}\ x\ xs)}\,(\text{Refute})$$

Figure 5.3: The complementary branch of the proof in Figure 5.2.

Solving Hypotheses

An important aspect of proof search is computing a set of inferences that could be applied to a given proof obligation. The newly introduced $(\text{Subst})_{\bot}$ rule is only applicable in cases where the conclusion's hypotheses (i.e. the assumptions of the current proof obligation) imply the desired instance of the lemma's hypotheses, as enforced by its side-condition. Therefore, in order to make use of this rule with a given lemma, proof search must first compute a set of instances of the lemma's hypotheses that are implied by the assumptions of the current proof obligation. In the case of our previous example, we arrived at the proof obligation $x' == x' = \texttt{False} \Rightarrow \bot$ that we wish to discharge via the $(\text{Subst})_{\bot}$ rule, using the lemma $x == x = \texttt{False} \Rightarrow \bot$. In order to do so, we wish to find a substitution for which the implication $x' == x' = \texttt{False} \Rightarrow (x == x = \texttt{False})\theta$ is universally valid. The second procedure introduced in this chapter builds on our mechanism for normalising conditional equation to compute an applicable set of substitution.

In the original proof system presented in Section 4.2, the (Subst) rule is applied when an instance of one side of the lemma's equation matches against a sub-expression of the current proof obligation. Matching expressions in this way partially determines the instance of the lemma, but it might not be fully determined if there are variables appearing in only one side of the lemma's equation, or indeed in its hypotheses. In the unconditional case, any uninstantiated variables may simply persist into the continuation, which is sound under the assumption that all types are inhabited. However, the same strategy cannot be applied in the presence of conditional equations: once a possible match has been detected, we can partially instantiate the lemma's hypotheses but, unlike the unconditional case, there is no guarantee of a solution. Thus, the application of the (Subst) rule also requires us to find instance of the lemma's hypotheses that valid under the hypotheses of the current proof obligation, albeit where

the instance is partially determined by matching the consequent of the lemma with a sub-expression of the proof obligation.

Logically, this task can be framed as a unification problem modulo theory — to find instances of a set of equations that are valid in a given equational theory. In our case, the equational theory consists of the program's reduction relation and the hypotheses of the current proof obligation. Here the variables appearing in the lemma and the proof obligation take on distinct roles. The former we will refer to as *existential* as proof search is concerned with finding an appropriate substitution instance, whereas the latter are deemed *universal* as the implication must hold for all valuations of these variables, and thus they cannot be instantiated.

Having organised hypotheses into a confluent and terminating rewrite system, we provide a procedure for solving a lemma's hypotheses using *narrowing* [116]. Narrowing is a powerful technique that extends a rewrite relation so that syntactic unification is used in lieu of matching. In other words, under the narrowing relation, existential variables (i.e. those appearing in the lemma's hypotheses) are instantiated precisely when this enables a reduction step. For example, the expression $x == x$ is in normal form, but it may be narrowed to `False` under the hypothesis $x' == x' = \texttt{False}$ by unifying the existential variable x with the universal variable x' (i.e. under the substitution $\{x \mapsto x'\}$).

For a confluent and terminating rewrite system, narrowing is a semi-decision procedure for solving unification problems modulo equational theory. The fact that it is only a semi-decision procedure is to be expected from the infinitary set of most-general unifiers and the undecidable nature of such problems. As non-termination is clearly not viable when computing applicable inference rules, we limit narrowing steps to only unify with the hypotheses of the proof obligation, rather than the program's reduction relation. Unlike the program's reduction relation, the variables appearing in the proof obligation's hypotheses (i.e. the universal variables) cannot be instantiated in order to make a narrowing step. Therefore, non-decreasing narrowing steps reduce the number of existential variables by instantiating them with expressions built from universal variable, and thus can only be applied a finite number of times. It is worth noting, however, that the program's reduction relation is still eagerly used to simplify expressions that arise during unification.

5.2 Working with Hypotheses

In Section 5.3, we will introduce the proof rules for handling conditional equations. First, however, we will define their syntax and semantics and the novel mechanisms for normalising and solving hypotheses outlined in the introduction to this chapter.

5.2.1 Conditional Equations

Definition 5.1. The syntax for equations is extended to *clauses* generated by the following grammar:

$$\begin{array}{ll} \text{Clause} & C ::= H \Rightarrow a = b \mid H \Rightarrow \bot \\ \text{Hypotheses} & H ::= \top \mid H_1 \wedge H_2 \mid a = b \end{array}$$

The left- and right-hand side of the implication are referred to as the *hypotheses* and the *consequent* respectively. We will use ϕ, ψ, etc., to range over the consequents. If the consequent of a clause is $\bot$, then it is said to be in *refutation mode*, and it is said to be in *normal mode* otherwise. Conjunction is implicitly assumed to satisfy the usual axioms of commutativity, associativity, and idempotence, thus we will treat hypotheses as a set of atomic equations where $\top$ denotes the empty set. As in the previous chapter, we will also use $\doteq$ to denote an unordered equation in the consequent of a clause, but hypotheses are always oriented, as we will discuss in the subsequent sections.

$$\frac{\Gamma \vdash H\ \mathsf{wf} \quad \Gamma \vdash a : \tau \quad \Gamma \vdash b : \tau}{\Gamma \vdash H \Rightarrow a = b\ \mathsf{wf}} \qquad \frac{\Gamma \vdash H\ \mathsf{wf}}{\Gamma \vdash H \Rightarrow \bot\ \mathsf{wf}}$$

$$\frac{}{\Gamma \vdash \top\ \mathsf{wf}} \qquad \frac{\Gamma \vdash a : \tau \quad \Gamma \vdash b : \tau}{\Gamma \vdash a = b\ \mathsf{wf}} \qquad \frac{\Gamma \vdash H_1\ \mathsf{wf} \quad \Gamma \vdash H_2\ \mathsf{wf}}{\Gamma \vdash H_1 \wedge H_2\ \mathsf{wf}}$$

Figure 5.4: Well-formedness rules for clauses.

Definition 5.2. A clause is said to be *well-formed* for a given type environment if the judgement $\Gamma \vdash C$ wf can be derived from the inference rules in Figure 5.4.

Unless stated otherwise, all clauses and hypotheses are equipped with a type environment for which they are well-formed and that, as usual, contains the program environment. We will sometimes annotate equations in a set of hypotheses by their type, writing $a = b : \tau \in H$ to indicate that $\Gamma \vdash a, b : \tau$ where Γ is the type environment for which the hypotheses and consequent are well-formed.

Definition 5.3. A well-formed clause, set of hypotheses, or equation $\Gamma \vdash C$ wf is said to *be satisfied* by a valuation of its type environment θ (see Definition 4.2) following the standard logical interpretation of the connectives:

$$\theta \vDash H \Rightarrow \bot \iff \theta \nvDash H \qquad \theta \vDash \top \iff \top$$

$$\theta \vDash H_1 \wedge H_2 \iff \theta \vDash H_1 \wedge \theta \vDash H_2 \qquad \theta \vDash a = b \iff a\theta \equiv_P b\theta$$

$$\theta \vDash H \Rightarrow a = b \iff \theta \nvDash H \vee \theta \vDash a = b$$

Likewise, a clause is said to be *valid* if it is satisfied by all valuations of its type environment, in which case we write $\vDash C$. Note that a clause in refutation mode is valid just if its hypotheses are unsatisfiable.

5.2.2 Hypothetical Reduction

Hypotheses can be employed to simplify the consequent of a clause, providing additional information about sub-expressions under local assumptions. Rather than define a separate proof rule to encode this mechanism, we extend the program's reduction relation with the hypotheses — *hypothetical reduction*. Combining the program's reduction relation with hypotheses in this manner will give us a setting in which a normal form for conditional equations can be established.

Definition 5.4. For a set of well-formed hypotheses H, the hypothetical reduction relation $H \vdash a \rightsquigarrow b$ is defined by the inference rules in Figure 5.5. Recall that hypotheses are oriented so that $a = b \in H$ does not necessarily imply that $b = a \in H$. As with the program reduction relation, we write $H \vdash a_1 \rightsquigarrow^* a_2$ for reflexive-transitive closure of the hypothetical reduction relation.

$$\frac{}{H \vdash C[a] \rightsquigarrow C[b]}\, a = b \in H$$

$$\frac{}{H \vdash C[(f\ p_1\ \cdots\ p_n)\theta] \rightsquigarrow C[b\theta]}\, \begin{array}{l} P(f)\ p_1\ \cdots\ p_n \Downarrow_\emptyset b \\ p_1, \ldots, p_n \text{ patterns} \end{array}$$

Figure 5.5: Rules of hypothetical reduction.

Note that hypothetical reduction does not instantiate the variables of a hypothesis when it is applied. This is because the hypotheses constrain their free variables, rather than being quantified equations that describe a general property of the program. Therefore, from the point of view of hypothetical reduction, the free variables of the hypotheses are effectively constants. We will refer to these variables are universal variables.

Hypothetical reduction clearly subsumes the program's reduction relation as a consequence of Lemma 2.4. Furthermore, as the following lemmas indicate, hypothetical reduction preserves the type of expressions and their equivalence under any valuation satisfying the hypotheses. We also show a limited form of stability, where the substitution instance is required to be disjoint from the hypotheses.

Lemma 5.1. Suppose $\Gamma \vdash H$ wf and $\Gamma \vdash a : \tau$. If $H \vdash a \rightsquigarrow b$, then $\Gamma \vdash b : \tau$.

Lemma 5.2. If $H \vdash a \rightsquigarrow^* b$, then $H \Rightarrow a = b$ is valid.

Lemma 5.3. Suppose $\Gamma \vdash H$ wf and $H \vdash a \rightsquigarrow^* b$. Then $H \vdash C[a\theta] \rightsquigarrow^* C[b\theta]$ for any context $C[\cdot]$ and any substitution θ such that $\mathrm{dom}(\theta) \cap \mathrm{dom}(\Gamma) = \emptyset$.

5.2.3 Hypothesis Completion

For certain sets of hypotheses, hypothetical reduction is neither terminating nor confluent. That is to say, some expressions may diverge or not reduce to a common normal forms. As the motivation for introducing hypothetical reduction was to replace the non-deterministic application of proof rules governing the use of hypotheses with a deterministic procedure for normalising goals, both termination and confluence are essential. Unless hypothetical reduction is terminating, even depth-bound proof search may diverge. Furthermore, if there were multiple normal forms for conditional equations proof search would not be able to comply with the various restrictions to proof search discussed in Section 4.4, from which the system derives its efficiency.

Hypotheses that lead to multiple normal forms can be immediately constructed from visibly contradictory conjunction such as $p = \texttt{True} \wedge p = \texttt{False}$. Such unsatisfiable hypotheses cannot be discounted since the role of the antecedent is to limit the cases in which its consequent must be satisfied, else they are redundant. However, expressions can also have multiple normal forms under satisfiable hypotheses. Consider, for example, the hypotheses $xs = y :: ys$ and $\texttt{reverse}\ xs = z :: zs$. Under these hypotheses, the expression $\texttt{reverse}\ xs$ can be reduced to both $z :: zs$ and $\texttt{append}\ (\texttt{reverse}\ ys)\ [y]$. Finally, a hypothesis such as $xs = x :: xs$ evidently induces divergence for the expression xs under hypothetical reduction.

To prevent each of the aforementioned scenarios, we adapt the Knuth-Bendix completion procedure for transforming a set of equations into a confluent and terminating rewrite system [87]. At a high-level, the Knuth-Bendix completion proceeds by simplifying and deleting equations before orienting them as to be interpreted as rewrite rules. The orientation is determined by a well-founded order on applicative expressions that is compatible with reduction (i.e. closed under context and substitution) so that no infinite reduction sequences can occur, thus guaranteeing normalisation. In order to produce a confluent rewrite system, however, we must also consider *critical pairs*. Critical pairs are expressions that can be rewritten in two different ways by overlapping rules, such as $\texttt{reverse}\ xs$ in the previous example. This conflict is resolved by adding a new equation, e.g. $z :: zs = \texttt{append}\ (\texttt{reverse}\ ys)\ [y]$, that must also be oriented and incorporated into the rewrite system. The tension between these two processes makes the Knuth-Bendix completion procedure, in general, only a semi-decision procedure.

Constructing an ordering that extends the program's reduction relation in a non-trivial way is a challenging task as the union of well-founded relations is rarely well-founded. Additionally, although the program is assumed to terminate, there are many

different termination schemes encountered in practice, e.g. those based on size-change graphs vs lexicographical path orderings, that cannot easily be unified [103, 117]. For this reason, we merely assume a reduction ordering $\leq$, in the sense of Definition 4.10, that is compatible with the program's reduction relation, i.e. if $a \to_P b$, then $a > b$. In Section 5.4, we will discuss the ordering used in our prototype implementation.

As previously mentioned, in addition to orienting hypothesis, the completion procedure must also consider critical pairs. The relevance of critical pairs is that they can be used to characterise confluence, as the Critical Pairs Lemma, stated below, demonstrates. In particular, if all critical pairs are joinable, then hypothetical narrowing is locally confluent. It is for this reason that the completion procedure derives additional equations to resolve critical pairs. As with the orthogonality of the program's reduction relation, the Critical Pairs Lemma implies global confluence if the rewrite system does not contain any infinite reduction sequences.

Definition 5.5. Formally, a hypothesis $a = b \in H$ has *critical overlap* with:

1. A hypothesis of the form $C[a] = c \in H$, for which the corresponding critical pair is the equation $C[b] = c$.

2. A reduction $a \to_P c$ for some applicative expression c, producing the critical pair $b = c$.

3. Or, if there is some non-trivial context $C[\cdot]$ and a non-variable applicative expression a' such that $a = a'\theta$ for some substitution θ, where $C[a']$ is of the form $f\ p_1\ \cdots\ p_n$ and $P(f)\ p_1\ \cdots\ p_n \Downarrow_\emptyset c$ is defined for some c. Then the hypothesis has a critical overlap with the reduction $C[a']\theta \to_P c\theta$ and the corresponding critical pair is the equation $C[b] = c\theta$.

   Note that the use of linear patterns in the program's reduction relation, see Lemma 2.4, implies that $C[a']\theta$ is simply equivalent to $C[a]$.

In the latter case, the context must be non-trivial else it merely degenerates to an instance of a type 2 overlap. If, on the other hand, a' is a variable x, then the quasi-critical pair is already joinable as $C[b] \to_P c[b/x]$ (see Lemma 2.3) and the hypothetical reduction $H \vdash c[a/x] \rightsquigarrow^* c[b/x]$ is derivable by repeated application of the hypothesis $a = b \in H$ in each sub-position. Type 1 and type 2 overlaps are less involved than the usual definition of critical pairs as free variables appearing in a hypothesis are effectively constants; being quantified at the clause level and not within the hypotheses themselves they cannot be instantiated by hypothetical reduction.

Lemma 5.4 (The Critical Pairs Lemma [118])**.** Suppose every critical pair $a = b$ of a hypothesis set H is joinable, i.e. there exists some applicative expression c such that $H \vdash a \rightsquigarrow^* c$ and $H \vdash b \rightsquigarrow^* c$. Then the applicative reduction relation $H \vdash \cdot \rightsquigarrow^* \cdot$

is locally confluent. That is, if $H \vdash a \rightsquigarrow b_1$ and $H \vdash a \rightsquigarrow b_2$, then $H \vdash b_1 \rightsquigarrow^* c$ and $H \vdash b_2 \rightsquigarrow^* c$ for some applicative expression c.

Corollary 5.5. Suppose H is a hypothesis set satisfying the pre-condition of the Critical Pairs Lemma and such that hypothetical reduction terminates, i.e. there does not exist an infinite chain of applicative expressions $H \vdash a_1 \rightsquigarrow a_2 \rightsquigarrow \cdots$, then hypothetical reduction is confluent.

Despite starting with a confluent and terminating program, we could very quickly run into a diverging instance of the completion procedure when trying to incorporate hypotheses. For example, the equation $xs = x :: xs$ must be oriented as $x :: xs = xs$ if the ordering under by completion is both well-founded and closed under context. But this orientation will induce type 3 critical pairs with expressions such as $\texttt{map}\ f\ (x :: xs)$, which could be reduced to both $f\ x :: \texttt{map}\ f\ xs$ and $\texttt{map}\ f\ xs$ under the aforementioned hypothesis. The completion procedure attempts to resolve critical pairs by introducing a new equation that restores confluence; in this case, $f\ x :: \texttt{map}\ f\ xs = \texttt{map}\ f\ xs$. However, the new equation induces a further critical overlap arising from the expression $\texttt{map}\ g\ (x :: \texttt{map}\ f\ xs)$, which can be also be reduced in two separate ways, and so on ad infinitum.

The first two types of critical overlap always yield smaller critical pairs as they merely involve simplify an existing hypothesis with another oriented hypothesis or a reduction step. Diverging instances of the completion procedure, therefore, result from type 3 overlaps, which introduces a non-trivial context and instantiates pattern variables, and thus doesn't necessarily result in a smaller critical pair. Fortunately, due to the strictly orthogonal nature of programs, type 3 overlaps can be isolated to the cases where there are higher-order equations or equations oriented so that the left-hand side is an application headed by a constructor. We can thus characterise the expressions that can yield such divergence instances of the completion procedure by the following definition.

Definition 5.6. For a given type environment Γ, a *stable* expression is an applicative expression that is either of the form:

- $x\ a_1\ \cdots\ a_n$ for some non-program variable x,
- Or, $f\ a_1\ \cdots\ a_n$ for some program variable f such that $\Gamma \vdash f\ a_1\ \cdots\ a_n : d\ \overline{\tau}$ or $\Gamma \vdash f\ a_1\ \cdots\ a_n : \alpha$

Otherwise, it is said to be *unstable*.

Lemma 5.6. Let $\Gamma \vdash H\ \mathsf{wf}$ be a set of hypotheses and suppose there is some hypothesis $a = b \in H$ that has a type 3 critical overlap. Then a is unstable.

With the preceding lemma in mind, we restrict the orientation of a hypothesis so that the left-hand side must be stable. When the greater of the two expressions is unstable, the hypothesis cannot be oriented. The ordering used to orient equations should, therefore, be designed so that unstable expressions are smaller than stable ones. In particular, constructor should be dominated by variables and applications where possible. Although unstable left-hand sides cannot always be avoided, when both sides of an equation are headed by a constructor, we can either emit smaller equations concerning their arguments whilst preserving the set of satisfied valuation or deduce there are no satisfying valuations if the constructors in question do not match.

We are now ready to define the hypothesis completion procedure. Recall that the objective of this procedure is to transform a set of hypotheses so that hypothetical reduction is confluent and terminating. The resulting set of hypotheses is constructed by simplifying and orienting equations, in addition to the introduction of critical pairs of type 1 and type 2. Type 3 critical pairs are not considered, as unlike the other two they may lead to divergence. Consequently, the procedure may terminate with some additional hypotheses that failed to be integrated.

Definition 5.7. The *hypothesis completion procedure* is presented as an inference system over *configurations* that are either pairs $\langle E,\ R\rangle$, where E are hypotheses yet to be oriented and R are hypotheses that have already been oriented, or the contradictory configuration $\lightning$. We write $\langle E,\ R\rangle \vdash \langle E',\ R'\rangle$ (or $\langle E,\ R\rangle \vdash \lightning$ resp.) to indicate that $\langle E',\ R'\rangle$ is obtained from $\langle E',\ R'\rangle$ by the inference rules in Figure 5.6, in which $\uplus$ denotes the disjoint union.

A configuration is said to be *terminal* if there are no further derivable configurations and *well-oriented* if $a > b$ and a is stable for any equation $a = b \in R$. Note that contradictory configurations are always terminal.

First, we will show that hypothesis completion is sound in that any derived equation or rewrite rule is implied by its premise. It therefore preserves the set of satisfying valuations or correctly deduces unsatisfiability.

Lemma 5.7 (Soundness)**.** Let $\langle E,\ R\rangle \vdash \langle E',\ R'\rangle$ be an inference of the hypothesis completion procedure. Then, the set of satisfying instances is preserved, i.e. the clause $E \wedge R \Rightarrow a = b$ is valid for any hypothesis $a = b \in E' \cup R'$.

Lemma 5.8 (Soundness)**.** If $\langle E,\ R\rangle \vdash \lightning$ is an inference of the hypothesis completion procedure, then there is no valuation θ such that $\theta \vDash E \wedge R$.

Next, we show that once there are no applicable rules for hypothesis completion the hypotheses satisfy the pre-condition of the critical pairs lemma and thus lead to a locally confluent system. Moreover, as they are necessarily oriented, hypothesis

$$\text{(Delete)}\ \frac{\langle E \uplus \{a \doteq a\},\ R\rangle}{\langle E,\ R\rangle} \qquad \text{(Orient)}\ \frac{\langle E \uplus \{a \doteq b\},\ R\rangle}{\langle E,\ R \cup \{a = b\}\rangle}\ \begin{array}{l} a > b \\ a \text{ is stable} \end{array}$$

$$\text{(Simplify)}\ \frac{\langle E \uplus \{a \doteq b\},\ R\rangle}{\langle E \cup \{a \doteq b'\},\ R\rangle}\ R \vdash b \rightsquigarrow b'$$

$$\text{(Compose)}\ \frac{\langle E,\ R \uplus \{a = b\}\rangle}{\langle E,\ R \cup \{a = b'\}\rangle}\ R \vdash b \rightsquigarrow b'$$

$$\text{(Collapse)}\ \frac{\langle E,\ R \uplus \{a = b\}\rangle}{\langle E \cup \{a' \doteq b\},\ R\rangle}\ R \vdash a \rightsquigarrow a'$$

$$\text{(Match)}\ \frac{\langle E \uplus \{k\ a_1\ \cdots\ a_n \doteq k\ b_1\ \cdots\ b_n\},\ R\rangle}{\langle E \cup \{a_i \doteq b_i \mid i \leq n\},\ R\rangle}\ k \in \mathbb{K}$$

$$\text{(Fail)}\ \frac{\langle E \uplus \{k\ a_1\ \cdots\ a_n \doteq k'\ b_1\ \cdots\ b_m\},\ R\rangle}{\lightning}\ k \neq k' \in \mathbb{K}$$

Figure 5.6: The inference rules of the hypothesis completion procedure.

reduction is globally confluent. Note that, unlike the Knuth-Bendix completion procedure, terminal configurations may have hypotheses that cannot be incorporated into the rewrite system without possibly leading to a divergent instance of the completion procedure, i.e. may induce type-3 overlaps, and thus are left as unoriented equations. It is for this reason that the hypothetical reduction under the resulting set of oriented hypotheses cannot be said to be complete.

Lemma 5.9 (Correctness)**.** Suppose $\langle E,\ R\rangle$ is a terminal configuration that is well-oriented. Then R has no critical overlaps, i.e. hypothetical reduction under R is confluent and terminating.

We will conclude this section by showing that, for any starting set of hypotheses, a terminal configuration is reached whose oriented hypotheses satisfy the pre-condition of the preceding lemma and thus, for which, hypothetical reduction is confluent and normalising. This result amounts to showing that the completion procedure cannot diverge. First, note that the side-condition on the (Orient) rule means hypothesis completion preserves well-orientation.

Lemma 5.10. Suppose $\langle E,\ R\rangle \vdash \langle E',\ R'\rangle$ is an inference of the hypothesis completion procedure where $\langle E,\ R\rangle$ is well-oriented. Then $\langle E',\ R'\rangle$ is also well-oriented.

To see that hypothetical completion has no infinite runs, we adapt the Dershowitz-Manna multiset ordering [119]. Configurations are treated as multisets of labelled expressions $(a,\, \mathfrak{o})$ or $(a,\, \mathfrak{u})$ where the second component indicates whether it appears in an oriented or unoriented equation respectively.

Definition 5.8. For a given configuration $\langle E,\, R\rangle$, let $\wr E,\, R\int$ denote the following function on labelled expressions:

$$\wr E,\, R\int(a,\, \mathfrak{o}) \coloneqq \#\{a_1 = a_2 \in R \mid a \in \{a_1,\, a_2\}\}$$
$$\wr E,\, R\int(a,\, \mathfrak{u}) \coloneqq \#\{a_1 \doteq a_2 \in E \mid a \in \{a_1,\, a_2\}\}$$

which counts the occurrences of the given expression in the oriented or unoriented equations of the configuration respectively.

Intuitively, the multiset extension of a given ordering permits a decrease if an element has been removed and replaced by any number of smaller elements. This ordering is well-founded whenever the underlying ordering is well-founded. In our case, these steps primarily consist of replacing an equation with a smaller equation, i.e. where either side is smaller under the compatible ordering. However, it must also support the (Orient) rule, which does not decrease the expressions of an equation but rather changes the label associated with the equation. Hence, we consider oriented equations as smaller than unoriented equations.

Definition 5.9. The ordering $p >_{\mathfrak{u}/\mathfrak{o}} q$ on labelled expressions is defined by the following cases:

- $(a,\, \ell) >_{\mathfrak{u}/\mathfrak{o}} (a',\, \ell')$ whenever $a >_{\text{sub}} a'$ regardless of the labellings ℓ and ℓ'.
- Or, $(a,\, \mathfrak{u}) >_{\mathfrak{u}/\mathfrak{o}} (a',\, \mathfrak{o})$ whenever $a \geq_{\text{sub}} a'$.

where $\geq_{\text{sub}}$ refers to the substructural extension of the ordering used by completion, see Definition 4.12.

Definition 5.10. The ordering on labelled equation is extended to non-contradictory configurations, written $\langle E,\, R\rangle \gg \langle E',\, R'\rangle$, whenever the following conditions are met:

- $E \neq E'$ or $R \neq R'$
- And, if $\wr E,\, R\int(p) < \wr E',\, R'\int(p)$ for any labelled expression p, there exists some other labelled expression q such that $p <_{\mathfrak{u}/\mathfrak{o}} q$ and $\wr E,\, R\int(q) > \wr E',\, R'\int(q)$.

Lemma 5.11. $>_{\mathfrak{u}/\mathfrak{o}}$ is well-founded.

Corollary 5.12. $\gg$ is well-founded [119].

Having established a well-founded ordering on configurations, we can conclude that there can be no infinite runs of the hypothesis completion if each inference results in a decrease with respect to this order. The following lemma shows exactly that. Most rules result in a decrease as the first component of a labelled expression, i.e. the applicative expression itself, decreases through hypothetical reduction under a well-oriented set of rules. The (Match) rule is similar, although it relies on the inclusion of the sub-expression relation in the substructural extension of our compatible ordering. Finally, the (Orient) rule does not change any expressions but relabels them as oriented, thus also resulting in a decrease.

Lemma 5.13. Suppose $\langle E,\ R\rangle$ is a well-oriented configuration and $\langle E,\ R\rangle \vdash \langle E',\ R'\rangle$ is an inference of the hypothesis completion. Then, $\langle E,\ R\rangle \gg \langle E',\ R'\rangle$.

Corollary 5.14. There is no infinite run of the hypothesis completion starting from a well-oriented configuration $\langle E,\ R\rangle$.

For any starting set of hypotheses, the hypothesis completion procedure can be used to either demonstrate that a set of hypotheses is unsatisfiable or extract an implied set of well-oriented hypotheses for which hypothetical reduction is confluent and terminating. The former case allows us to completely discharge a proof obligation as the clause is trivially valid, and the latter case allows us to simplify the consequent of a clause in a deterministic manner.

Definition 5.11. We write $\langle E,\ R\rangle \downarrow R'$ to denote that there exists a sequence of hypothesis completion inferences starting from $\langle E,\ R\rangle$ and ending in a terminal configuration $\langle E',\ R'\rangle$. Similarly, we write $\langle E,\ R\rangle \downarrow \lightning$ if a contradictory configuration is reached.

As Corollary 5.14 shows, it must either be the case that $\langle E,\ R\rangle \downarrow \lightning$ or there exists a set of hypotheses R' such that $\langle E,\ R\rangle \downarrow R'$. Furthermore, in the latter case, R' is well-oriented as a result of Lemma 5.10. Note, however, that there is no guarantee that the completion procedure is functional as the terminal configuration may depend on the order in which rules are applied. Nevertheless, any resulting set of hypotheses will enable proof search to normalise the consequent.

Consider the hypothesis set $xs = y :: ys \wedge \texttt{reverse}\ xs = z :: zs$ from our earlier example of a non-confluent hypothetical reduction. One possible run of the hypothesis completion is as follows:

$$
\begin{aligned}
&\langle \{xs \doteq y :: ys,\ \texttt{reverse}\ xs = z :: zs\},\ \emptyset\rangle \\
&\quad \vdash \langle \{xs \doteq y :: ys\},\ \{\texttt{reverse}\ xs = z :: zs\}\rangle \\
&\quad \vdash \langle \emptyset,\ \{xs = y :: ys,\ \texttt{reverse}\ xs = z :: zs\}\rangle \\
&\quad \vdash \langle \{\texttt{reverse}\ (y :: ys) \doteq z :: zs\},\ \{xs = y :: ys\}\rangle \\
&\quad \vdash \langle \{\texttt{append}\ (\texttt{reverse}\ ys)\ [y] \doteq z :: zs\},\ \{xs = y :: ys\}\rangle \\
&\quad \vdash \langle \emptyset,\ \{xs = y :: ys,\ \texttt{append}\ (\texttt{reverse}\ ys)\ [y] = z :: zs\}\rangle
\end{aligned}
$$

First, both equations are oriented by the (Orient) rule assuming $xs > y$, ys. As we have already seen, this hypothesis set isn't confluent, and thus we cannot have reached a terminal configuration. Indeed, the hypothesis concerning `reverse` xs can be rewritten under the (Collapse) rule, producing a new unoriented equation. This expression is simplified further by the (Simplify) before being re-oriented by the (Orient) rule. As there are no further applicable rules, hypothetical reduction under the hypotheses $xs = y :: ys$ and `append` (`reverse` ys) $[y] = z :: zs$, which result from the completion procedure, is confluent and terminating. In particular, the expression `reverse` xs now has a unique normal form, namely $z :: zs$.

To demonstrate how the hypothesis completion procedure can be used in an incremental manner, now consider adding the hypothesis $xs \doteq x :: xs$. This new equation would be simplified under the existing hypotheses and deconstructed by the (Match) rule. Only one of the resulting equations is orientable, however, and the terminal configuration still has unoriented equations as the larger side is unstable.

$$\begin{aligned}
&\langle \{xs \doteq x :: xs\},\ R\rangle \\
&\quad \vdash \langle \{y :: ys \doteq x :: xs\},\ R\rangle \\
&\quad \vdash \langle \{y :: ys \doteq x :: y :: ys\},\ R\rangle \\
&\quad \vdash \langle \{y \doteq x,\ ys \doteq y : ys\},\ R\rangle \\
&\quad \vdash \langle \{ys \doteq y :: ys\},\ \{x = y\} \cup \{R\}\rangle
\end{aligned}$$

where R denotes the set $\{xs = y :: ys,$ `append` (`reverse` ys) $[y] = z :: zs\}$ that results from the previous instance of hypothesis completion.

As the correctness of the hypothesis completion procedure is derived solely from the well-orientation of configurations, which is an invariant, we do not need to recompute the completion in its entirety, i.e. starting with an empty set of oriented equation, when a new equation is introduced. We exploit this fact in the implementation of our proof search algorithm by directly representing the hypotheses of a clause as a configuration of the hypothesis completion procedure.

5.2.4 Solving Hypotheses

As mentioned in the introduction to this chapter, the proof rules for applying lemmas induce a unification problem modulo equational theory when lemmas are conditional. That is, we must find a substitution instance of a set of equations that follow from the given equational theory. This is because, even after matching the lemma with the target equation, there may be uninstantiated variables that cannot take on an arbitrary value as they are constrained according to the lemma's hypotheses.

$$\text{(Subst)}\ \frac{\Gamma_2 \vdash H_2 \Rightarrow a \doteq c \quad \Gamma_1 \vdash H_1 \Rightarrow C[c\theta] \doteq b \quad \Gamma_1 \vdash \theta : (\Gamma_2 \setminus \Sigma)\Theta}{\Gamma_1 \vdash H_1 \Rightarrow C[a\theta] \doteq b \qquad \models H_1 \Rightarrow H_2\theta}$$

The proof rule above shows the general substitution principle for conditional equations. The side-condition $\models H_1 \Rightarrow H_2\theta$ constrains the substitution θ to be a solution to the lemma's hypotheses H_2 under the hypotheses H_1, i.e. any valuation that satisfies H_1 must also satisfy each of the hypotheses in $H_2\theta$, so that the lemma is applicable in this context. As we need only show that the lemma's hypotheses are valid when assuming the hypotheses of the conclusion, this latter set, in conjunction with the program's rewrite system, constitute the equational theory with respect to which we must perform unification.

It is not hard to see that unification modulo equational theory is, in general, undecidable due to the expressive power of arbitrary equational theories. For confluent and terminating rewrite systems, however, there does exist a complete semi-decision procedure based on *narrowing* [116]. Narrowing generalises a reduction relation so that sub-expressions are unified with the left-hand side of rewrite rules instead of being matched against them. For example, the expression $\texttt{append}\ xs\ (\texttt{append}\ ys\ xs)$ is in normal form, but we may narrow it to the expression $[z, z]$ by first unifying xs with the pattern $z :: zs$ and then subsequently unifying ys and zs with $[]$. Thus, narrowing is able to simulate the reduction of instances of a given expression rather than the expression itself, such as the instance $\{xs \mapsto [z], ys \mapsto []\}$ in the previous example. When combined with syntactic unification, narrowing is sufficient for solving unification problems modulo equational theory.

Although we can use hypothesis completion to extract a confluent and terminating instance of hypothetical reduction from the current proof obligation's hypotheses, it must also be the case that our unification algorithm terminates with the resulting rewrite system. Without a guarantee of termination, proof search may get stuck when applying a lemma, preventing the application of more productive lemmas. The source of non-termination in the narrowing procedure is the possibly endless introduction of existential variables. Recall that we use "existential" to refer to the variables that the unification procedure may instantiate, which is initially those variables appearing in the lemma's hypotheses. In contrast, "universal" variables are those of the current proof obligation, which cannot be instantiated as part of the unification process. Suppose, for example, the expression $\texttt{append}\ xs\ []$ appears in the lemma's hypotheses. It can be unified with the rewrite rule $\texttt{append}\ (y :: ys)\ zs \to_P y :: \texttt{append}\ ys\ zs$, under the substitution $\{xs \mapsto y :: ys, zs \mapsto []\}$, and then subsequently reduced to the expression $y :: \texttt{append}\ ys\ []$. In doing so, new existential variables are introduced which capture the free variables of the rewrite rule that were not instantiated; namely,

the variables y and ys. Moreover, the reduct has a sub-expression that is a variant of the original and, for which, the same narrowing step can be simulated ad infinitum.

To ensure termination, therefore, we forgo completeness by restricting the possible narrowing steps. Our restriction comes from the observation that unifying sub-expressions with a hypothesis of the current proof obligation only instantiates existential variables with expressions built from the universal variables to which the hypothesis applies. Therefore, when narrowing against a hypothesis, the number of existential variables in the unification problem decreases as the universal variables cannot be instantiated as part of the unification process. From this observation, we find our terminating restriction where a sub-expression may match against any rewrite rule (i.e. either a hypothesis or an instance of incremental matching) but may only be unified with a hypothesis whereby the number of existential variables is reduced. Recall that, as the hypotheses have to be oriented normal reduction steps decreases the size of the expressions with respect to the ordering used by the completion procedure. In this section, we define a *hypothetical narrowing* procedure that exactly captures this process.

Definition 5.12. The *hypothetical narrowing* relation $\Gamma,\ H \vdash a \overset{?}{\rightsquigarrow}_\theta b$ is defined by the inference rules in Figure 5.7. In addition to the assumed hypotheses, this procedure is parameterised by a type environment that dictates the variables that can be instantiated, i.e. the existential variables, and an accumulated substitution for these variables. We assume that $\Gamma \cap \Sigma = \emptyset$, i.e. no program variables are treated as existential.

$$\textsf{(Reduce)}\ \frac{H \vdash a \rightsquigarrow b}{\Gamma,\ H \vdash a \overset{?}{\rightsquigarrow}_\emptyset b}$$

$$\textsf{(Unify)}\ \frac{a\theta = b \in H \quad \mathrm{dom}(\theta) \subseteq \mathrm{dom}(\Gamma) \cap \mathsf{FV}(a)}{\Gamma,\ H \vdash C[a] \overset{?}{\rightsquigarrow}_\theta C\theta[b]}$$

Figure 5.7: The hypothetical narrowing relation.

Definition 5.13. The *many-step hypothetical narrowing* relation $\Gamma,\ H \vdash a \overset{?}{\rightsquigarrow}^*_\theta b$ is defined in Figure 5.8 and is analogous to the reflexive-transitive closure of the hypothetical narrowing relation but where the substitution is accumulated.

The correctness of hypothetical narrowing is given by the following lemma, showing that it produces an instance of the initial expression that reduces to the latter under the given hypotheses. For the sake of simplicity, we do not enforce that the substitution is well-typed as part of the narrowing judgement. This oversight is mitigated by an additional side-condition of the (Subst) and (Subst)$_\bot$ rules that checks the substi-

$$\frac{}{\Gamma,\ H \vdash a \overset{?}{\rightsquigarrow}^{*}_{\emptyset} a} \qquad \frac{\Gamma,\ H \vdash a \overset{?}{\rightsquigarrow}_{\theta} b \quad \Gamma,\ H \vdash b \overset{?}{\rightsquigarrow}^{*}_{\theta'} c}{\Gamma,\ H \vdash a \overset{?}{\rightsquigarrow}^{*}_{\theta \cup \theta'} c} \text{ where } \mathrm{dom}(\theta) \cap \mathrm{dom}(\theta') = \emptyset$$

Figure 5.8: The many-step hypothetical narrowing relation.

tution is well-typed, although in practice we verify that the accumulated substitutions are well-typed during narrowing.

Lemma 5.15. Suppose $\Gamma_1 \vdash H$ wf is a well-formed set of hypotheses and Γ_2 is a type environment that is disjoint from Γ_1, i.e. $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$. If $\Gamma_2,\ H \vdash a \overset{?}{\rightsquigarrow}^{*}_{\theta} b$ is an instance of hypothetical narrowing, then $\mathrm{dom}(\theta) \subseteq \Gamma_2$ and $H \vdash a\theta \rightsquigarrow^{*} b$ is an instance of hypothetical reduction.

As previously mentioned, narrowing cannot be guaranteed to terminate even when the underlying rewrite system is terminating. This divergence arises from the introduction of instantiation of variables with large terms as part of narrowing steps. In our case, however, proper narrowing steps are restricted to only apply to hypotheses, which only contain universal variables that cannot be instantiated. Therefore, narrowing either decreases the number of existential variables or results in smaller expressions. Therefore, as there are only finitely many existential variables appearing in the hypotheses being solved, hypothetical narrowing terminates under well-oriented hypotheses.

Lemma 5.16. Let $>$ be a reduction order that is compatible with the program's reduction relation. Suppose $\Gamma_1 \vdash H$ wf are well-oriented hypotheses, i.e. $a > b$ for all equations $a = b \in H$, and Γ_2 is a type environment that is disjoint from Γ_1. Then, for any hypothetical narrowing $\Gamma_2,\ H \vdash a \overset{?}{\rightsquigarrow}_{\theta} b$, either $\mathrm{dom}(\theta)$ is non-empty or $a > b$.

Theorem 5.17. Suppose $\Gamma_1 \vdash H$ wf are well-oriented hypotheses, Γ_2 is a type environment that is disjoint from Γ_1, and a is an applicative expression. Then there is no infinite sequence of expressions $(a_i)_{i \in \mathbb{N}}$ and substitutions $(\theta_i)_{i \in \mathbb{N}}$ such that $\Gamma_2,\ H \vdash a_i \overset{?}{\rightsquigarrow}_{\theta_i} a_{i+1}$ for all $i \in \mathbb{N}$.

Corollary 5.18. Suppose $\Gamma_1 \vdash H$ wf are well-oriented hypotheses, Γ_2 is a type environment that is disjoint from Γ_1, and a is an applicative expression. Then there are finitely many substitutions θ and expressions b such that $\Gamma_2,\ H \vdash a \overset{?}{\rightsquigarrow}^{*}_{\theta} b$.

As there can be no infinite derivations of hypothetical narrowing, we can enumerate instances of an expression modulo hypothetical reduction. Therefore, an equation can be solved by narrowing each side until syntactically unifiable expressions are reached. This process can easily be extended to a set of hypotheses by solving each equation and, when possible, combining solutions.

Theorem 5.19. Suppose $\Gamma_2,\ H \vdash a \overset{?}{\rightsquigarrow}^*_\theta a'$ and $\Gamma_2,\ H \vdash b\theta \overset{?}{\rightsquigarrow}^*_{\theta'} b'$ are two instances of hypothetical narrowing such that $a'\theta'\theta'' = b'\theta''$ for some substitution θ'' where $\mathrm{dom}(\theta'') \subseteq \mathrm{dom}(\Gamma_2)$. Then the composite substitution $\theta_{\text{sol}} = \theta\theta'\theta''$ is a solution to the equation $a = b$ under the hypotheses H, i.e. the clause $H \Rightarrow a\theta_{\text{sol}} = b\theta_{\text{sol}}$ is valid.

Now let us examine an example of the hypothetical narrowing process. Consider the equation $\texttt{not}\ (\texttt{leq}\ x\ y) \doteq p$ that we would like to solve with hypothetical narrowing under the well-oriented hypotheses $\texttt{leq}\ a\ a = \texttt{True}$ and $\texttt{leq}\ a\ b = \texttt{False}$. One possible solution is derived from the hypothetical narrowing steps:

$$\texttt{not}\ (\texttt{leq}\ x\ y) \overset{?}{\rightsquigarrow}_{\{x \mapsto a,\, y \mapsto a\}} \texttt{not}\ \texttt{True} \overset{?}{\rightsquigarrow}_{\emptyset} \texttt{False}$$

and the unifier $\{p \mapsto \texttt{False}\}$, which gives rise to the solution $\{x \mapsto a,\ y \mapsto a,\ p \mapsto \texttt{False}\}$. Alternatively, unification with the second hypothesis results in the solution $\{x \mapsto a,\ y \mapsto b,\ p \mapsto \texttt{True}\}$. As a result of our restriction, completeness is sacrificed for the sake of termination by only unifying sub-expressions with hypotheses. For example, the solution $\{x \mapsto \texttt{Z},\ y \mapsto \texttt{Z},\ p \mapsto \texttt{False}\}$ would not be discovered, despite being valid, as it only arises through unification with the reduction $\texttt{leq Z Z} \to_P \texttt{True}$ rather than with a hypothesis.

Narrowing Strategies

Since its conception, narrowing has received much attention with many refinements being developed. We will briefly comment on why we don't specialise the above procedure to one of the common strategy. The first such strategy, "basic narrowing", only permits the application of narrowing steps to sub-expressions that were not introduced as a result of unification, exploiting the confluence of the rewrite system [120]. In our case, basic narrowing doesn't result in any improvement as narrowing steps cannot introduce existential variables that could be narrowed further. There are also many strategies that are specialised to strictly orthogonal rewrite systems [101, 121]. Although the program's reduction relation is strictly orthogonal, the same cannot be said for its combination with hypotheses. Hence, these strategies are not applicable.

Another approach to solving unification problems with a confluent and terminating rewrite system is "one-sided paramodulation" or "lazy narrowing" [122, 123]. One-sided paramodulation interleaves the syntactic unification procedure with narrowing using an outermost restructuring rule to implicitly match an expression with a hypothesis:

$$\begin{aligned} f\ s_1\ \cdots\ s_n = t \Rightarrow \\ \quad s_1 = l_1 \wedge \cdots \wedge s_n = l_n \wedge r = t \end{aligned}$$

where $f\ l_1\ \cdots\ l_n = r$ is a hypothesis. One-sided paramodulation can often be more efficient than narrowing as its derivations avoid over specialising solution, typically

subsuming many narrowing derivations. In our case, however, it is harder to reason about the termination properties of one-sided paramodulation as the restructuring rule introduces new expressions that are not necessarily smaller than original under the completion ordering.

5.3 Extended Proof System

In the previous section, we discussed the various mechanical aspects of working with hypotheses. Namely, we introduced a saturation-based procedure to normalise clauses and a technique for solving hypotheses using narrowing. These two procedures play a central role in the implementation of the extension of the proof system to support conditional equations, but they are not sufficient for showing that hypotheses are unsatisfiable — a crucial route to discharging goals in a conditional setting.

When the hypothesis completion ends in a contradictory configuration, the hypotheses are known to be unsatisfiable, but this does not cover all such cases. Consider, for example, the hypothesis $x == x = \texttt{False}$ that is clearly unsatisfiable despite being easily organised into a confluent and terminating rewrite system. It can only be shown to be unsatisfiable by inductively considering the value of x, such as through a cyclic proof of the following form:

$$\dfrac{\dfrac{}{\texttt{Z} == \texttt{Z} = \texttt{False} \Rightarrow \bot} \quad \dfrac{\dfrac{(1)}{y == y = \texttt{False} \Rightarrow \bot}}{\texttt{S}\ y == \texttt{S}\ y = \texttt{False} \Rightarrow \bot}}{1\text{: } x == x = \texttt{False} \Rightarrow \bot}$$

In the left-hand branch, the hypothesis completion procedure will demonstrate that the hypotheses are unsatisfiable as $\texttt{Z} == \texttt{Z} \to_P^* \texttt{True} \neq \texttt{False}$. On the other hand, the hypothesis $\texttt{S}\ y == \texttt{S}\ y = \texttt{False}$ can be simplified but is still not visibly unsatisfiable. In order to complete this branch, the proof must create a cycle using the root node as a lemma with the instance $\{x \mapsto y\}$. However, CycleQ's original cycle formation rule does not apply here as the lemma has no positive equation with which the goal could be rewritten. It is clear that an additional rule for forming cycles is necessary to complete such proofs.

In this section, we introduce a cyclic proof system for conditional equational reasoning with two distinct modes of operation: the normal mode as in our original proof system and a "refutation mode" where the formulas in question have no positive literals. As an example of the necessity of the refutation mode consider the function in Figure 5.9 and the equation $\texttt{silly}\ x = \texttt{Z}$, which we will try to prove within the normal mode.

In order to make a reduction step and expose the behaviour of this program, it is necessary to perform case analysis on $x == x$. The case when $x == x$ is $\texttt{True}$ can

```
silly : Nat → Nat
silly x =
  case x == x of
    False -> x
    True -> Z
```

Figure 5.9: A trivial, yet conditional, function.

be trivially discharged, so consider the remaining proof obligation:

$$1\text{: } x == x = \texttt{False} \Rightarrow x = \texttt{Z}$$

To proceed with the proof, we perform case analysis on x and, after simplification, we are left with the following clause: $x' == x' = \texttt{False} \Rightarrow \texttt{S}\ x' = \texttt{Z}$. By using the consequent of clause (1) with the instance $\{x \mapsto \texttt{S}\ x'\}$, we could rewrite this clause to reflexivity and discharge the proof obligation. The lemma's hypotheses are indeed satisfied by this instance, but there is no progressing trace as the only relevant variable has increased. The lemma that would actually enable us to complete this proof would be of the form $x == x = \texttt{False} \Rightarrow \texttt{S}\ x = \texttt{Z}$, which is a seemingly arbitrary variant.

The intuition to be derived from the above example is that, when trying to complete a proof with unsatisfiable hypotheses, equational consequents provide no relevant information — they are not known to be satisfied by any instances. Rewriting a proof obligation towards reflexivity is, therefore, misguided as it relies on the syntactic form of irrelevant equations. Instead, once in refutation mode, we avoid such meaningless transformations and the correct use of the clause is enforced. For example, if the aforementioned lemma was in refutation mode, then we could complete the preceding proof without having to artificially syntheses a lemma that aligns with the syntactic form of the proof obligation.

Refutation mode is characterised by clauses without positive literals, represented with $\bot$ as their consequent. This alternative mode restricts the space of proofs as lemmas in refutation mode can only be used to discharge goals directly via an additional proof rule for cycle formation.

5.3.1 Cyclic Pre-proofs

Recall that a cyclic pre-proof is a tuple $(V,\ E,\ \lambda,\ \rho)$ where V is a finite set of nodes and $E : V \rightharpoonup V^*$ determines the underlying structure of the proof graph. The latter two components λ and ρ previously assigned nodes equations and inference rules from Figure 4.5 respectively. In the extension, however, they equip nodes with a clause and an inference rule from the extended set in Figure 5.10 respectively. Other than the rules for reflexivity, congruence, and function extensionality, which are largely the

same as in the original system with no special interaction with the hypotheses, the interpretation of proof rules is as follows:

- (Refute) This rule is applied non-deterministically to move a clause into refutation mode. As it is not always appropriate, it is not prioritised over any other rule. However, when the consequent is visibly unsatisfiable this rule is applied eagerly.

  One could imagine a system that randomly tests the hypotheses and only permits the application of this rule if no satisfying instances are found. This approach would reduce the number of times it is inaccurately applied. Nevertheless, the results in Section 5.4 show that an implementation without such checks can still be sufficiently performant.

- (Absurd), (Reduce) When the hypothesis completion procedure produces a contradictory configuration, we can discharge the proof obligation with (Absurd) The soundness of hypothesis completion implies that such hypotheses are unsatisfiable and the clause is vacuously true.

  Otherwise, we perform hypothetical reduction with the resulting saturated hypotheses. As with the original CycleQ proof search algorithm, reduction is prioritised as it exposes more of the behaviour of expressions. In this case, reduction also includes relevant consequences of hypotheses.

  As the hypothesis completion procedure can be expensive, but must be computed when deciding which of these inference rules are applicable, the implementation stores the terminal configuration as part of the clause. In Section 5.2, we showed that the hypothesis completion procedure can build on the existing configuration as new hypotheses are added, largely mitigating the cost of repeated saturation.

- (Subst), $(\text{Subst})_\bot$ As previously discussed the new calculus has two mechanisms for forming cycles: the normal mode which uses a pre-existing lemma to rewrite the goal producing a new obligation, i.e. the "continuation", and the refutation mode which discharges the goal entirely for lemmas that indicate the hypotheses are unsatisfiable.

  In either case, we must find a substitution θ for which $\vDash H_1 \Rightarrow H_2\theta$ where H_1 and H_2 are the goal's and lemma's hypotheses respectively, although the substitution will be partially instantiated in normal mode as a result of matching sub-expressions. Instead of using H_1 directly to solve the lemma's hypotheses, we again use the set of hypotheses produced by the hypothesis completion procedure for which hypothetical reduction is confluent and terminating. Under these conditions, the narrowing based unification algorithm discussed in Section 5.2 computes a finite set of solutions.

- (Case) Finally, the rule for case analysis differs from the unconditional system as it allows for case analysis on arbitrary expressions rather than being restricted to variables, introducing a new hypothesis corresponding to the relevant case. Although there are many more sub-expressions than variables, we still restrict our attention to those that are "needed" according to the definition in Section 4.4 in order to guide proof search with the aim of making a reduction step.

  The ordering used during the hypothesis completion procedure is designed so that the hypotheses resulting from case analysis are always orientable. That is to say, reduction can indeed make use of the new hypothesis to replace any occurrence of the subject of case analysis with the corresponding case.

$$\text{(Refl)}\ \frac{}{\Gamma \vdash H \Rightarrow a \doteq a} \qquad \text{(Refute)}\ \frac{\Gamma \vdash H \Rightarrow \bot}{\Gamma \vdash H \Rightarrow a \doteq b}$$

$$\text{(Absurd)}\ \frac{}{\Gamma \vdash H \Rightarrow \phi}\ \langle H,\, \emptyset \rangle \vdash^* \lightning$$

$$\text{(Cong)}\ \frac{(\forall i \leq n) \quad \Gamma \vdash H \Rightarrow a_i \doteq b_i}{\Gamma \vdash H \Rightarrow k\ a_1\ \cdots\ a_n \doteq k\ b_1\ \cdots\ b_n}$$

$$\text{(Reduce)}\ \frac{\Gamma \vdash H' \cup R \Rightarrow a' \doteq b'}{\Gamma \vdash H \Rightarrow a \doteq b}\ \begin{array}{l} \langle H,\, \emptyset \rangle \vdash^* \langle H',\, R \rangle \\ R \vdash a \rightsquigarrow^* a' \\ R \vdash b \rightsquigarrow^* b' \end{array}$$

$$\text{(FunExt)}\ \frac{\Gamma \cup \{x : \tau_1\} \vdash H \Rightarrow a\ x \doteq b\ x}{\Gamma \vdash H \Rightarrow a \doteq b}\ \Gamma \vdash a,\, b : \tau_1 \to \tau_2$$

$$\text{(Subst)}_{\bot}\ \frac{\Gamma_2 \vdash H_2 \Rightarrow \bot}{\Gamma_1 \vdash H_1 \Rightarrow \phi}\ \begin{array}{l} \Gamma_1 \vdash \theta : (\Gamma_2 \setminus \Sigma)\Theta \\ \vDash H_1 \Rightarrow H_2\theta \end{array}$$

$$\text{(Subst)}\ \frac{\Gamma_2 \vdash H_2 \Rightarrow a \doteq b \quad \Gamma_1 \vdash H_1 \Rightarrow C[b\theta] \doteq c}{\Gamma_1 \vdash H_1 \Rightarrow C[a\theta] \doteq c}\ \begin{array}{l} \Gamma_1 \vdash \theta : (\Gamma_2 \setminus \Sigma)\Theta \\ \vDash H_1 \Rightarrow H_2\theta \end{array}$$

$$\text{(Case)}\ \frac{(\forall k \in \mathrm{dom}(\Delta(d)))\ \Gamma \cup \Gamma_k \vdash H \wedge a = k\ x_1\ \cdots\ x_n \Rightarrow \phi}{\Gamma \vdash H \Rightarrow \phi}$$

$$\text{where } \Gamma \vdash a : d\ \overline{\tau}$$
$$\text{and } \Gamma_k = \{\overline{x : \Delta(d)(k)[\overline{\tau/\alpha}]}\}$$

Figure 5.10: The inference rules for extended pre-proofs.

5.3.2 Local and Global Soundness

As with the unconditional proof system, the soundness property of cyclic proofs is decomposed into the local soundness of the proof rules, which applies to all pre-proofs, and the global soundness condition constraining the shape of cycles and thus distinguishing a class of proper proofs.

Recall the necessary precursor relation $\twoheadrightarrow$ on pairs (v, θ) where $v \in V$ is a node in a pre-proof and θ a valuation of the clause $\lambda(v)$ that determines the dependencies between valuations of nodes. The idea is that if a node v_1 is not satisfied by a valuation θ_1, then there is a necessary precursor, i.e. a node v_2 and a valuation θ_2 of its clause, such that $(v_1, \theta_1) \twoheadrightarrow (v_2, \theta_2)$ and v_2 is also not satisfied by θ_2. The definition of the necessary precursor relation is the same as in Section 4.2 with the following exceptions to account for the new inference rules. In each case, v_2 is a child of v_1, i.e. a premise, and the precursory valuations are determined by the inference rule $\rho(v_1)$.

- (Refute) As v_2 is satisfied by a valuation just if its hypotheses are unsatisfied, in which case v_1 is also satisfied, we have that $(v_1, \theta) \twoheadrightarrow (v_2, \theta)$ for any appropriate valuation θ.
- (Absurd) As with reflexivity, this rule does not give rise to any necessary precursors as it is always valid.
- $(\text{Subst})_\perp$ Let θ be the substitution instance of the lemma. As with the normal (Subst) rule, the relevant instances of the lemma are related to the valuation of the conclusion by θ. Therefore, $(v_1, \theta_1) \twoheadrightarrow (v_2, \theta\theta_1)$.
- (Case) where $a : d\,\overline{\tau}$ is the expression upon which case analysis is performed and $x_1, \ldots, x_n$ are the fresh variables. We know that $(a\theta)\!\downarrow_P$ is of the form $k\,a_1 \cdots a_n$ for some $k \in \Delta(d)$. Thus, $(v_1, \theta_1) \twoheadrightarrow (v_2, \theta_1 \cup \{\overline{x \mapsto a}\})$ whenever v_2 is the premise associated with this constructor.

Theorem 5.20 (Local Soundness)**.** Let $v_1 \in V \setminus \text{Ax}$ be a non-axiom node within the pre-proof (V, E, λ, ρ) such that $\theta_1 \not\Vdash \lambda(v_1)$ for some valuation θ_1. Then there exists a necessary precursor (v_2, θ_2) such that $\theta_2 \not\Vdash \lambda(v_2)$.

An invalid node thus gives rise to an invalid axiom or infinite sequence of invalid nodes. To extend local soundness to global soundness we must show that the necessary precursor relation is well-founded for the given pre-proof, in which case there can be no such infinite sequences. This is done by assigning a trace to all paths within a pre-proof, see Definition 4.5, which have infinitely many progress points. Given a stable, well-founded partial-order on applicative expressions, a $\leq$-trace along a path $(v_i)_{i\in\mathbb{N}}$ is an infinite sequence of applicative expressions $(t_i)_{i\in\mathbb{N}}$ subject to the following constraints:

- If $\lambda(v_i) = \Gamma_i \vdash H \Rightarrow a_i = b_i$, then $\mathsf{FV}(t_i) \subseteq \mathrm{dom}(\Gamma_i \setminus \Sigma)$, i.e. the trace term may only depend on the free variables of the equation associated with the given node.

- If $\rho(v_i)$ is (Case) for some $i \in \mathbb{N}$ where $x : d\ \overline{\tau}$ is the variable upon which case analysis is performed and v_{i+1} is the premise associated with the constructor $k \in \Delta(d)$ using fresh variables $x_1, \ldots, x_n$, then $t_{i+1} \leq t_i[k\ x_1\ \cdots\ x_n/x]$. If, on the other hand, case analysis is applied to a non-variable expression, it must be the case that $t_{i+1} \leq t_i$.

- If $\rho(v_i)$ is (Subst) with substitution θ and v_{i+1} is the lemma, then $t_{i+1}\theta \leq t_i$ and, if v_i is the continuation, then $t_{i+1} \leq t_i$.

- If $\rho(v_i)$ is $(\mathrm{Subst})_\bot$ with substitution θ and v_{i+1} is the lemma, then it must be the case that $t_{i+1}\theta \leq t_i$.

- And, in all other cases, $t_{i+1} \leq t_i$.

Lemma 5.21. Let (V, E, λ, ρ) be a cyclic pre-proof with a path $(v)_{i\in\mathbb{N}}$ and suppose $(t)_{i\in\mathbb{N}}$ is a $\leq$-trace along this path. If θ_i is a valuation of some node $v_i \in T$ and $(v_i, \theta_i) \rightarrowtriangle (v_{i+1}, \theta_{i+1})$, then $t_{i+1}\theta_{i+1} \leq t_i\theta_i$ and $t_{i+1}\theta_{i+1} < t_i\theta_i$ when i is a progress-point.

As with the unconditional proof system, the existence of an infinitely progressing trace for each path implies that the necessary precursor relation is well-founded. Once again, when combined with local soundness, we can conclude that every equation in such a cyclic pre-proof is valid if its axioms are.

Theorem 5.22 (Global Soundness)**.** Let (V, E, λ, ρ) be a cyclic proof such that, for every axiom $v \in \mathrm{Ax}$, the associated equation $\lambda(v)$ is valid. Then, for every other node $v \in V \setminus \mathrm{Ax}$, the associated equation $\lambda(v)$ is also valid.

5.4 Evaluation

The original implementation of CycleQ was limited by two key factors: the lack of conditional equations and the lack of theory exploration. In this chapter, we introduced an extended proof system CycleQ$^{\Rightarrow}$ that supports conditional equations. This proof system is supported by a number of theoretical developments that are designed to enable efficient proof search despite the increase in expressivity.

As with the first iteration, we have implemented the extended system, including the hypothesis completion procedure and a narrowing-based approach to solving hypotheses, as well as the additional refutation mode discussed in the preceding section, as a plugin for GHC 9.2.8. The global soundness condition is again verified using

size-change graphs that we have previously shown to enable efficient incremental computation. The purpose of this evaluation is, therefore, two-fold: to demonstrate the mechanisms for managing hypotheses do not impede on the performance of the tool and that they are sufficient for solving those benchmark problems that were previously identified as requiring conditional equations, but not generalisation or the generation of auxiliary lemmas as this is an orthogonal concern.

For the two principle contributions of this chapter, a reduction ordering is required that is compatible with the program's reduction relation. This ordering is used to construct a confluent and terminating rewrite system via the hypothesis completion procedure and then used to argue that our narrowing-based procedure for solving hypotheses terminates. The construction of such an ordering for an arbitrary program is difficult as not much of the structure of the program's reduction relation can be gathered just given that it is terminating, and the combination of arbitrary well-founded ordering is rarely well-founded itself. In our prototype implementation, we orient hypotheses according to the lexicographical path ordering where constructors have the least precedence so that more hypotheses are orientable (see Definition 5.6) [117]. This ordering works well in practice, as the following evaluation demonstrates, but it is not compatible with every possible program. As no instances of non-termination were encountered, we suspect that a similar ordering can be constructed for a common subset of programs that includes all benchmark problems. We intend to investigate this observation in more detail as future work.

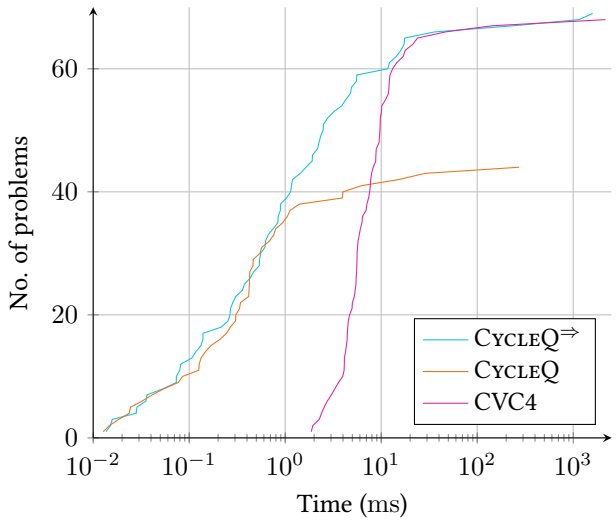

Figure 5.11: Number of benchmark problems solved within the given time bound.

As with the original implementation, we tested the tool against a standard benchmark suite of 85 induction problems concerning natural numbers, lists, and trees,

originally used to test the IsaPlanner tool [74]. The results were obtained as an average over 10 runs with the fuel parameter set to 10 on the windows sub-system for Linux, running on am 11th Gen Intel® Core™ i7–1185G7 @ 3.00GHz/1.80GHz processor with 16.0GB of RAM. The number of benchmark problems solved in a given time is shown in Figure 5.11 in addition to the results for the non-conditional implementation.

Our prototype implementation of CycleQ$^{\Rightarrow}$ was able to find proofs for an additional 24 benchmark problems; solving 69 (∼80%) in total. Furthermore, the new features for handling conditional equation appeared to have very little impact on efficiency. In particular, the new iteration of our equational reasoning tool was still able to solve all but one of the original 44 problems solved by CycleQ in under 30ms. These results indicate that the hypothesis completion procedure and narrowing-based solving mechanism are viable approaches to working with conditional equations when performing proof search. The effectiveness of the hypothesis completion procedure is further witnessed by the fact that no instance encountered more than one unorientable equation. Perhaps more surprisingly, the unconstrained use of (Refute) does not appear to lead to an intractably large search space as long as it is given a lower priority than the other rules that can be applied without loss of generality.

Recall that from the previous chapter, we compared the efficiency of CycleQ with the inductive extension of CVC4, whose benchmark results are also presented in Figure 5.11. Although CycleQ was more efficient in many cases, it was hard to draw a true comparison as CVC4 ultimately solved more benchmarks. With the introduction of conditional equations, however, it is plain to see that our prototype implementation of CycleQ$^{\Rightarrow}$ is significantly more efficient on simple problems and comparably efficient on more difficult problems.

To verify that the techniques developed in this chapter are sufficient for conditional reasoning, we need to identify those benchmark problems that require generalisation or the generation of auxiliary lemmas as this is an orthogonal concern that we have not attempted to address. Out of the remaining 18 unsolved problems that were solved by the HipSpec system, all required the generation of auxiliary lemmas [112]. Furthermore, our proof search algorithm was able to solve one more benchmark problem than CVC4, which only solves 68 problems without mapping inductive types to existing theories such as the integers [110]. This tool was selected for comparison due to the lack of sub-goal generation.

The unsolved problems either require simple properties such as the commutativity of addition to be known or depend on more complex functions such as `sort` and `reverse`. One distinguishing feature of these functions is that they are not "treeless", unlike `add`, `take`, and `max` where the tool performs well, in that they build-up intermediate structures [124]. Appropriate induction hypotheses are often discovered naturally for treeless functions merely by unfolding their definitions as there is no in-

direct recursion. Recursive calls in these more complex functions, on the other hand, make use of an accumulation parameter, which typically require strengthening, or are guarded by another function, e.g. `insert` or `append`, that typically require an additional lemma to circumnavigate. As part of future work, we intend to establish the exact connection between the class of functions and those proofs that can be discovered without auxiliary lemmas.

Chapter 6

Conclusion

Functional programs have many appealing properties when it comes to reasoning about their correctness. They are often pure, and process immutable structures, so there is no question with regards to memory safety. Despite these advantages there is a limited set of tools available that can automatically and formally prove properties of such programs. In this thesis, we presented two lightweight approaches to the verification of a small, functional programming language with algebraic datatypes and Hindley-Milner style polymorphism. These systems are lightweight in that they sacrifice completeness for efficiency in order to be usable in day-to-day software development cycles, rather than being limited to safety critical applications where it makes sense to dedicate time to more comprehensive styles of verification, e.g. deductive verification in an interactive theorem prover.

The first of these verification systems is a refinement type system that can be used to guarantee a program is a positive instance of the pattern-match safety problem; that is, it cannot give rise to a pattern-matching error at runtime. Intuitively, the declarative system can be seen as automatically "completing" a datatype environment by adding instances of datatypes where constructors have been removed. In practice, our constrained-type inference procedure provides an algorithmic solution to the problem that is shown to be, in the worst case, linear in the size of the program. This complexity result was achieved by restricting the constraints associated with each function so that they are bounded by the size of its underlying type, under the assumption that the size of these underlying types are indeed bounded. As a result, the algorithm was able to analyse large scale packages with over a thousand definitions in the order of seconds. Nevertheless, we found one recursive group where our assumption breaks down, and the tool was unable to handle the large number of induced constraints.

In order to soundly restrict constraints in this manner, it was necessary to limit the possible refinements of datatypes to those that apply recursively, i.e. recursive

occurrence of a datatype under a constructor is refined in the same manner; namely intensional refinements. Consequently, some common datatypes do not have useful refinements, such as the list datatype that can only be (non-trivially) refined to empty lists or infinite lists, neither of which are particularly useful in practice. Although it can be argued that this is acceptable from a fully-automated and scalable analysis, we nevertheless intend to extend the system to better handle such cases. There appear to be two directions for doing so: allow the user to specify non-intensional refinements of interest (e.g. non-empty lists), or automatically extend the intensional environment with common patterns such as a one-level unfolding, from which the non-empty list refinement can be derived. Both cases would require a fundamental change to the type inference system as the constraints rely on the assumption that each constructor is associated with a single underlying datatype identifier. It is as yet unclear how much this extension would compromise performance.

The second verification system we considered was an equational reasoning system for algebraic datatypes based on cyclic proofs rather than the traditional approach to proof by induction with explicit induction schemes. Equational specifications are an appealing target for a lightweight verification system as they do not require the user to be familiar with a complex program logic, meaning they can be more easily integrated into the development cycle. Furthermore, they are significantly more expressive than type systems and are not limited to safety properties. Naturally this expressivity comes with complexity which, in this case, arises from the interaction between equational reasoning and the limitations of automated inductive reasoning. We developed a novel cyclic proof system to mitigate this complexity — CycleQ. Our prototype implementation demonstrates that the corresponding proof search algorithm is efficient when combined with a size-change based method for checking the global soundness condition. It was able to automatically prove many properties about simple functions in under a millisecond.

The first iteration of our proof system, although efficient, was not able to prove some of the more complex properties in the benchmark suite due to the lack of support for conditional equations. This meant that we were not able to provide a fair comparison to any other state-of-the-art inductive theorem provers. In the final chapter of this thesis, we introduced CycleQ $^{\Rightarrow}$ that additionally has support for conditional equations. To maintain the promising results under this increase in expressivity, we developed a number of specialised mechanisms for handling conditional reasoning, including the integration of hypotheses into the program's reduction relation in a deterministic manner and a terminating algorithm for finding solutions to a lemma's hypotheses. Our prototype implementation demonstrated that these new features enable a large number of problems to be solved with no significant decrease in efficiency, meaning that our tool surpasses CVC4 (without sub-goal generation) in both the number of problems solved and efficiency.

The reason we draw a comparison with CVC4 is precisely because its sub-goal generation heuristics can be disabled. This restriction is necessary for a fair comparison as the heuristics necessarily inhibit the tool's performance in exchange for being able to solve a greater number of problems. However, it is naturally of interest to see if our cyclic proof systems can be extended in this manner with some support of lemma generation or heuristical strengthening while retaining its good performance characteristics. Based on our analysis of the yet unproven benchmark problems, there appears to be a correlation between functions that are treeless (i.e. do not built-up intermediate structures) and properties that can be proven without auxiliary lemmas or strengthening. This observation is perhaps unsurprising as treeless programs have a simple recursive structure and construct their outputs directly, meaning that more of their behaviour is exposed by case analysis and reduction alone. If a formal connection can be established, the types of generalisations and the space of auxiliary lemmas could then be restricted in order to retain efficiency whilst supporting a more powerful algorithm.

Preceding this future work, a clear understanding of the "discoverable" cyclic proofs is required. In the setting of explicit induction, defining such a fragment is straightforward by removing cut-like rules and restricting generalisation. However, the non-local nature of cyclic proofs requires a more detailed analysis. What is more, such a model of discoverable proof may also enable a formal justification of the restriction we place upon proof search. It is important to understand the boundaries of our proof search algorithm not just for the sake of further theoretical developments but to provide the user with a clear guide as to when the tool will work out of the box, as is the case with our declarative type system for intensional refinement types.

To summarise, we developed two lightweight verification systems for a functional programming language. Our focus was on efficiency and predictability rather than precision or the ability to solve all target problems. These systems are ideal for widespread use due to their easy-to-understand foundations, derived from the idea of removing constructors or through equational reasoning. However, it is ultimately necessary for verification tools to handle more complex scenarios, even if the tools are no longer fully automated. Overall, the systems presented in this paper provide an efficient foundation for their intended purposes, but further work is required in order to develop them into mature tools.

Bibliography

[1] John Hughes. 'Why functional programming matters'. In: *The Computer Journal*, vol. 32 (1989), pp. 98–107. DOI: 10.1093/comjnl/32.2.98.

[2] Matt Miller. 'Pursuing durable safety for systems software'. In: *Symposium on Information and Communications Technology Security*. 2020.

[3] Koen Claessen and John Hughes. 'Quickcheck: a lightweight tool for random testing of Haskell programs'. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. Association for Computing Machinery, 2000, pp. 268–279. DOI: 10.1145/351240.351266.

[4] Simon Marlow et al. 'There is no fork: an abstraction for efficient, concurrent, and concise data access'. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. Association for Computing Machinery, 2014, pp. 325–337. DOI: 10.1145/2628136.2628144.

[5] Victor Allombert, Mathias Bourgoin and Julien Tesson. 'Introduction to the tezos blockchain'. In: *International Conference on High Performance Computing & Simulation*. IEEE, 2019, pp. 1–10. DOI: 10.1109/HPCS48598.2019.9188227.

[6] Ákos Hajdu et al. 'Inferl: scalable and extensible erlang static analysis'. In: *Proceedings of the 21st ACM SIGPLAN International Workshop on Erlang*. Association for Computing Machinery, 2022, pp. 33–39. DOI: 10.1145/3546186.3549929.

[7] Patrick Thomson et al. 'Fusing industry and academia at github (experience report)'. In: *Proc. ACM Program. Lang.*, vol. 6 (2022). DOI: 10.1145/3547639.

[8] J. N. Reed, J. E. Sinclair and F. Guigand. 'Deductive reasoning versus model checking: two formal approaches for system development'. In: *IFM '99*. Springer London, 1999, pp. 375–394. DOI: 10.1007/978-1-4471-0851-1_20.

[9] Jean-Christophe Filliâtre. 'Deductive software verification'. In: *International Journal on Software Tools for Technology Transfer*, vol. 13 (2011), pp. 397–403. DOI: 10.1007/s10009-011-0211-0.

[10] Edmund M. Clarke et al. 'Model checking and the state explosion problem'. In: *Tools for Practical Software Verification*. Springer Berlin Heidelberg, 2012, pp. 1–30. DOI: 10.1007/978-3-642-35746-6_1.

[11] Sten Agerholm and Peter Gorm Larsen. 'A lightweight approach to formal methods'. In: *Applied Formal Methods*. Springer Berlin Heidelberg, 1999, pp. 168–183. DOI: 10.1007/3-540-48257-1_10.

[12] Anna Zamansky et al. 'Towards classification of lightweight formal methods'. In: *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering*. SciTePress, 2018, pp. 305–313. DOI: 10.5220/0006770803050313.

[13] Robin Milner. 'A theory of type polymorphism in programming'. In: *Journal of Computer and System Sciences*, vol. 17 (1978), pp. 348–375. DOI: 10.1016/0022-0000(78)90014-4.

[14] Luke Ong. 'Higher-order model checking: an overview'. In: *30th Annual ACM/IEEE Symposium on Logic in Computer Science*. 2015, pp. 1–15. DOI: 10.1109/LICS.2015.9.

[15] John C. Reynolds. 'What do types mean? — from intrinsic to extrinsic semantics'. In: *Programming Methodology*. Springer New York, 2003, pp. 309–327. DOI: 10.1007/978-0-387-21798-7_15.

[16] Tim Freeman and Frank Pfenning. 'Refinement types for ML'. In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. Association for Computing Machinery, 1991, pp. 268–277. DOI: 10.1145/113445.113468.

[17] Hongwei Xi and Frank Pfenning. 'Dependent types in practical programming'. In: *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, 1999, pp. 214–227. DOI: 10.1145/292540.292560.

[18] Cordelia V. Hall et al. 'Type classes in haskell'. In: *ACM Transactions on Programming Languages and Systems*, vol. 18 (1996), pp. 109–138. DOI: 10.1145/227699.227700.

[19] Martin Odersky, Martin Sulzmann and Martin Wehr. 'Type inference with constrained types'. In: *Foundations of Object-Oriented Languages*, vol. 5 (1999), pp. 35–55. DOI: 10.1002/(SICI)1096-9942(199901/03)5:1<35::AID-TAPO4>3.0.CO;2-4.

[20] Alexander Aiken and Edward L. Wimmers. 'Type inclusion constraints and type inference'. In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. Association for Computing Machinery, 1993, pp. 31–41. DOI: 10.1145/165180.165188.

[21] Lionel Parreaux. 'The simple essence of algebraic subtyping: principal type inference with subtyping made easy (functional pearl)'. In: *Proc. ACM Program. Lang.*, vol. 4 (2020). DOI: 10.1145/3409006.

[22] Manuel Fahndrich and Alexander Aiken. 'Making set-constraint based program analyses scale'. In: *First Workshop on Set Constraints*. University of California at Berkeley, 1996.

[23] David Melski and Thomas Reps. 'Interconvertbility of set constraints and context-free language reachability'. In: *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. Association for Computing Machinery, 1997, pp. 74–89. DOI: 10.1145/258993.259006.

[24] Nevin Heintze and David McAllester. 'On the cubic bottleneck in subtyping and flow analysis'. In: *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 1997, p. 342. DOI: 10.1109/LICS.1997.614960.

[25] Richard Bird. *Thinking functionally with haskell*. Cambridge University Press, 2014. DOI: 10.1017/cbo9781316092415.

[26] Timothée Haudebourg, Thomas Genet and Thomas Jensen. 'Regular language type inference with term rewriting'. In: *Proc. ACM Program. Lang.*, vol. 4 (2020). DOI: 10.1145/3408994.

[27] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1998. DOI: 10.1017/cbo9780511530104.

[28] Alan Bundy. 'The automation of proof by mathematical induction'. In: *Handbook of Automated Reasoning*. North-Holland, 2001. Chap. 13, pp. 845–911. DOI: 10.1016/b978-044450813-3/50015-1.

[29] Moa Johansson. 'Lemma discovery for induction'. In: *International Conference on Intelligent Computer Mathematics*. Springer International Publishing, 2019, pp. 125–139. DOI: 10.1007/978-3-030-23250-4_9.

[30] Rovert Boyer and J. Srother Moore. 'Generalization'. In: *A Computational Logic*. Academic Press, 1979. Chap. XII, pp. 151–158. DOI: 10.1016/B978-0-12-122950-4.50016-4.

[31] Andrew Ireland and Alan Bundy. 'Productive use of failure in inductive proof'. In: *Automated Mathematical Induction*. Springer Netherlands, 1996, pp. 79–111. DOI: 10.1007/978-94-009-1675-3_3.

[32] William Sonnex, Sophia Drossopoulou and Susan Eisenbach. 'Zeno: an automated prover for properties of recursive data structures'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2012, pp. 407–421. DOI: 10.1007/978-3-642-28756-5_28.

[33] Koen Claessen, Nicholas Smallbone and John Hughes. 'Quickspec: guessing formal specifications using testing'. In: *Tests and Proofs*. Springer Berlin Heidelberg, 2010, pp. 6–21. DOI: 10.1007/978-3-642-13977-2_3.

[34] Koen Claessen et al. 'Automating inductive proofs using theory exploration'. In: *Automated Deduction*. Springer Berlin Heidelberg, 2013, pp. 392–406. DOI: 10.1007/978-3-642-38574-2_27.

[35] Christoph Sprenger and Mads Dam. 'On global induction mechanisms in a μ-calculus with explicit approximations'. In: *RAIRO - Theoretical Informatics and Applications*, vol. 37 (2003), pp. 365–391. DOI: 10.1051/ita:2003024.

[36] James Brotherston and Alex Simpson. 'Sequent calculi for induction and infinite descent'. In: *Journal of Logic and Computation*, vol. 21 (2010), pp. 1177–1216. DOI: 10.1093/logcom/exq052.

[37] Bahareh Afshari and Dominik Wehr. 'Abstract cyclic proofs'. In: *Logic, Language, Information, and Computation*. Springer International Publishing, 2022, pp. 309–325. DOI: 10.1007/978-3-031-15298-6_20.

[38] James Brotherston, Richard Bornat and Cristiano Calcagno. 'Cyclic proofs of program termination in separation logic'. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, 2008, pp. 101–112. DOI: 10.1145/1328438.1328453.

[39] James Brotherston, Dino Distefano and Rasmus Lerchedahl Petersen. 'Automated cyclic entailment proofs in separation logic'. In: *Automated Deduction*. Springer Berlin Heidelberg, 2011, pp. 131–146. DOI: 10.1007/978-3-642-22438-6_12.

[40] Shachar Itzhaky et al. 'Cyclic program synthesis'. In: *International Conference on Programming Language Design and Implementation*. Association for Computing Machinery, 2021, pp. 944–959. DOI: 10.1145/3453483.3454087.

[41] Takeshi Tsukada and Hiroshi Unno. 'Software model-checking as cyclic-proof search'. In: *Proc. ACM Program. Lang.*, vol. 6 (2022). DOI: 10.1145/3498725.

[42] James Brotherston, Nikos Gorogiannis and Rasmus L. Petersen. 'A generic cyclic theorem prover'. In: *Programming Languages and Systems*. Springer Berlin Heidelberg, 2012, pp. 350–367. DOI: 10.1007/978-3-642-35182-2_25.

[43] *GHC user's guide*. 9.2.8. 2023. URL: https://downloads.haskell.org/ghc/9.2.8/docs/users_guide.pdf.

[44] Eddie Jones and Steven Ramsay. 'Intensional datatype refinement: with application to scalable verification of pattern-match safety'. In: *Proceedings of the ACM on Programming Languages*, vol. 5 (2021), pp. 1–29. DOI: 10.1145/3434336.

[45] Deepak Kapur and David R. Musser. 'Proof by consistency'. In: *Artificial Intelligence*, vol. 31 (1987), pp. 125–157. DOI: 10.1016/0004-3702(87)90017-8.

[46] Uday S. Reddy. 'Term rewriting induction'. In: *International Conference on Automated Deduction*. Springer, 1990, pp. 162–177. DOI: 10.1007/3-540-52885-7_86.

[47] Eddie Jones, C.-H. Luke Ong and Steven Ramsay. 'Cycleq: an efficient basis for cyclic equational reasoning'. In: *International Conference on Programming Language Design and Implementation*. Association for Computing Machinery, 2022, pp. 395–409. DOI: 10.1145/3519939.3523731.

[48] Adam Fischbach and John Hannan. 'Specification and correctness of lambda lifting'. In: *Journal of Functional Programming*, vol. 13 (2003), pp. 509–543. DOI: 10.1017/S0956796802004604.

[49] Sergio Antoy. 'Definitional trees'. In: *International Conference on Algebraic and Logic Programming*. Vol. 3. Springer, Berlin, Heidelberg, 1992, pp. 143–157. DOI: 10.1007/BFb0013825.

[50] Silvia Gilezan. 'A note on typed combinators and typed lambda terms'. In: *Review of Research Faculty of Science, Mathematics Series* (1993), pp. 319–329.

[51] Nils Anders Danielsson et al. 'Fast and loose reasoning is morally correct'. In: *Symposium on Principles of Programming Languages*. Association for Computing Machinery, 2006, pp. 206–217. DOI: 10.1145/1111037.1111056.

[52] Samson Abramsky and Achim Jung. 'Domain theory'. In: *Handbook of Logic in Computer Science*. Vol. 3. Oxford University Press, 1995, pp. 1–168. DOI: 10.5555/218742.218744.

[53] Andrew Gordon. 'Bisimilarity as a theory of functional programming'. In: *Electronic Notes in Theoretical Computer Science*, vol. 1 (1995), pp. 232–252. DOI: 10.1016/S1571-0661(04)80013-6.

[54] Davide Sangiorgi and Robin Milner. 'The problem of "weak bisimulation up to"'. In: *International Conference on Concurrency Theory*. Springer, 1992, pp. 32–46. DOI: 10.1007/BFb0084781.

[55] Damien Pous. 'Weak bisimulation up to elaboration'. In: *International Conference on Concurrency Theory*. Springer, 2006, pp. 390–405. DOI: 10.1007/11817949_26.

[56] John C. Kolesar, Ruzica Piskac and William T. Hallahan. 'Checking equivalence in a non-strict language'. In: *Proc. ACM Program. Lang.*, vol. 6 (2022). DOI: 10.1145/3563340.

[57] Luc Maranget. 'Warnings for pattern matching'. In: *Journal of Functional Programming*, vol. 17 (2007), pp. 387–421. DOI: 10.1017/S0956796807006223.

[58] Sebastian Graf, Simon Peyton Jones and Ryan G Scott. 'Lower your guards: a compositional pattern-match coverage checker'. In: *Proc. ACM Program. Lang.*, vol. 4. DOI: 10.1145/3408989.

[59] Jens Palsberg and Christina Pavlopoulou. 'From polyvariant flow information to intersection and union types'. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, 1998, pp. 197–208. DOI: 10.1145/268946.268963.

[60] Manuel Fähndrich et al. 'Partial online cycle elimination in inclusion constraint graphs'. In: (1998), pp. 85–96. DOI: 10.1145/277650.277667.

[61] Zhendong Su, Manuel Fähndrich and Alexander Aiken. 'Projection merging: reducing redundancies in inclusion constraint graphs'. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Association for Computing Machinery, 2000, pp. 81–95. DOI: 10.1145/325694.325706.

[62] Jörgen Gustavsson and Josef Josef Svenningsson. 'Constraint abstractions'. In: *Symposium on Program as Data Objects*. Springer, 2001, pp. 63–83. DOI: 10.1007/3-540-44978-7_5.

[63] Alexander Aiken and Edward L. Wimmers. 'Solving systems of set constraints'. In: *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 1992, pp. 329–340. DOI: 10.1109/LICS.1992.185545.

[64] William F. Dowling and Jean H. Gallier. 'Linear-time algorithms for testing the satisfiability of propositional horn formulae'. In: *The Journal of Logic Programming*, vol. 1 (1984), pp. 267–284. DOI: https://doi.org/10.1016/0743-1066(84)90014-1.

[65] Xinxin Liu and Scott A. Smolka. 'Simple linear-time algorithms for minimal fixed points'. In: *25th International Colloquium on Colloquium on Automata, Languages, and Programming*. Springer. 1998, pp. 53–66. DOI: 10.1007/BFb0055040.

[66] Leo Bachmair, Harald Ganzinger and Uwe Waldmann. 'Set constraints are the monadic class'. In: *Symposium on Logic in Computer Science*. IEEE, 1993, pp. 75–83. DOI: 10.1109/LICS.1993.287598.

[67] Michael Brandt and Fritz Henglein. 'Coinductive axiomatization of recursive type equality and subtyping'. In: *Typed Lambda Calculi and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 63–81. DOI: 10.1007/3-540-62688-3_29.

[68] Davide Sangiorgi. 'On the origins of bisimulation and coinduction'. In: *ACM Trans. Program. Lang. Syst.*, vol. 31 (2009). DOI: 10.1145/1516507.1516510.

[69] B. Vergauwen, J. Wauman and J. Lewi. 'Efficient fixpoint computation'. In: *Static Analysis*. Springer Berlin Heidelberg, 1994, pp. 314–328. DOI: 10.1007/3-540-58485-4_49.

[70] Ilya Sergey, Simon Peyton Jones and Dimitrios Vytiniotis. 'Theory and practice of demand analysis in Haskell'. Unpublished. 2014.

[71] Alan Bundy. 'The automation of proof by mathematical induction'. In: *Handbook of Automated Reasoning*. Vol. 1. North Holland, 2001, pp. 845–911. DOI: 10.1016/B978-044450813-3/50015-1.

[72] Christoph Walther. 'Computing induction axioms'. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 1992, pp. 381–392. DOI: 10.1007/BFb0013076.

[73] Koen Claessen et al. 'Automating inductive proofs using theory exploration'. In: *International Conference on Automated Deduction*. Springer, 2013, pp. 392–406. DOI: 10.1007/978-3-642-38574-2_27.

[74] Lucas Dixon and Jacques Fleuriot. 'IsaPlanner: a prototype proof planner in Isabelle'. In: *International Conference on Automated Deduction*. Springer, 2003, pp. 279–283. DOI: 10.1007/978-3-540-45085-6_22.

[75] Matt Kaufmann and J Strother Moore. *ACL2 user's manual*. 8.5. 2023. URL: https://www.cs.utexas.edu/users/moore/acl2/.

[76] Christoph Sprenger and Mads Dam. 'On the structure of inductive reasoning: circular and tree-shaped proofs in the μ-calculus'. In: *International Conference on Foundations of Software Science and Computation Structures*. Springer, 2003, pp. 425–440. DOI: 10.1007/3-540-36576-1_27.

[77] James Brotherston. 'Cyclic proofs for first-order logic with inductive definitions'. In: *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer, 2005, pp. 78–92. DOI: 10.1007/11554554_8.

[78] Anupam Das and Damien Pous. 'A cut-free cyclic proof system for Kleene algebra'. In: *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer, 2017, pp. 261–277. DOI: 10.1007/978-3-319-66902-1_16.

[79] Bahareh Afshari and Graham E Leigh. 'Cut-free completeness for Modal μ-calculus'. In: *Symposium on Logic in Computer Science*. IEEE, 2017, pp. 1–12. DOI: 10.1109/LICS.2017.8005088.

[80] Anupam Das, Amina Doumane and Damien Pous. 'Left-handed completeness for Kleene algebra via Cyclic proofs'. In: *International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. EasyChair Publications, 2018, pp. 271–289. DOI: 10.29007/hzq3.

[81] Denis Kuperberg, Laureline Pinault and Damien Pous. 'Cyclic proofs, System T, and the power of contraction'. In: *Proceedings of the ACM on Programming Languages*, vol. 5 (2021), pp. 1–28. DOI: 10.1145/3434282.

[82] Anupam Das. 'On the logical complexity of cyclic arithmetic'. In: *Logical Methods in Computer Science*, vol. 16 (2020). DOI: 10.23638/LMCS-16(1:1)2020.

[83] Martin Protzen. 'Lazy generation of induction hypotheses'. In: *International Conference on Automated Deduction*. Springer, pp. 42–56. DOI: 10.1007/3-540-58156-1_4.

[84] Bahareh Afshari and Dominik Wehr. 'Abstract cyclic proofs'. In: *International Workshop on Logic, Language, Information, and Computation*. Springer, 2022, pp. 309–325. DOI: 10.1007/978-3-031-15298-6_20.

[85] James Brotherston and Alex Simpson. 'Sequent calculi for induction and infinite descent'. In: *Journal of Logic and Computation*, vol. 21 (2011), pp. 1177–1216. DOI: 10.1093/logcom/exq052.

[86] David R. Musser. 'On proving inductive properties of abstract data types'. In: *Symposium on Principles of Programming Languages*. Association for Computing Machinery, 1980, pp. 154–162. DOI: 10.1145/567446.567461.

[87] Donald E. Knuth and Peter B. Bendix. 'Simple word problems in universal algebras'. In: *Automation of Reasoning 2: Classical Papers on Computational Logic 1967–1970*. Springer, 1983, pp. 342–376. DOI: 10.1007/978-3-642-81955-1_23.

[88] Simon Cruanes. 'Superposition with structural induction'. In: *International Symposium on Frontiers of Combining Systems*. Springer, 2017, pp. 172–188. DOI: 10.1007/978-3-319-66167-4_10.

[89] Giles Reger and Andrei Voronkov. 'Induction in saturation-based proof search'. In: *International Conference on Automated Deduction*. Springer, 2019, pp. 477–494. DOI: 10.1007/978-3-030-29436-6_28.

[90] Hubert Comon. 'Inductionless induction'. In: *Handbook of Automated Reasoning*. North-Holland, 2001, pp. 913–962. DOI: 10.1016/B978-044450813-3/50016-3.

[91] Gerard Huet and Jean-Marie Hullot. 'Proofs by induction in equational theories with constructors'. In: *Journal of Computer and System Sciences*, vol. 25 (1982), pp. 239–266. DOI: 10.1016/0022-0000(82)90006-X.

[92] Laurent Fribourg. 'A strong restriction of the inductive completion procedure'. In: *International Colloquium on Automata, Languages, and Programming*. Springer, 1986, pp. 105–115. DOI: 10.1007/3-540-16761-7_60.

[93] Claus-Peter Wirth. 'History and future of implicit and inductionless induction: beware the old jade and the zombie!' In: *Mechanizing Mathematical Reasoning* (2005), pp. 192–203. DOI: 10.1007/978-3-540-32254-2_12.

[94] Stephen J. Garland and John V. Guttag. 'Inductive methods for reasoning about abstract data types'. In: *Symposium on Principles of Programming Languages*. Association for Computing Machinery, pp. 219–228. DOI: 10.1145/73560.73579.

[95] Leo Bachmair, Nachum Dershowitz and David A. Plaisted. 'Completion without failure'. In: *Rewriting Techniques*. Elsevier, 1989, pp. 1–30. DOI: 10.1016/B978-0-12-046371-8.50007-9.

[96] Takahito Aoto. 'Dealing with non-orientable equations in rewriting induction'. In: *International Conference on Rewriting Techniques and Applications*. Springer, 2006, pp. 242–256. DOI: 10.1007/11805618_18.

[97] Sorin Stratulat. 'A unified view of induction reasoning for first-order logic'. In: *Turing-100, The Alan Turing Centenary Conference*. EasyChair, 2012, pp. 326–352. DOI: 10.29007/nsx4.

[98] Sorin Stratulat. 'SPIKE, an automatic theorem prover—revisited'. In: *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, 2020, pp. 93–96. DOI: 10.1109/SYNASC51798.2020.00025.

[99] Shujun Zhang and Naoki Nishida. 'On transforming cut-and quantifier-free cyclic proofs into rewriting-induction proofs'. In: *International Symposium on Functional and Logic Programming*. Springer, 2022, pp. 262–281. DOI: 10.1007/978-3-030-99461-7_15.

[100] Shujun Zhang and Naoki Nishida. 'On transforming rewriting-induction proofs for logical-connective-free sequents into cyclic proofs'. In: *9th International Workshop on Rewriting Techniques for Program Transformations and Evaluation*. EasyChair, 2022.

[101] Sergio Antoy, Rachid Echahed and Michael Hanus. 'A needed narrowing strategy'. In: *Journal of the ACM*, vol. 47 (2000), pp. 776–822. DOI: 10.1145/347476.347484.

[102] Alex Simpson. 'Cyclic arithmetic is equivalent to Peano arithmetic'. In: *International Conference on Foundations of Software Science and Computation Structures*. Springer, 2017, pp. 283–300. DOI: 10.1007/978-3-662-54458-7_17.

[103] Chin Soon Lee, Neil D. Jones and Amir M. Ben-Amram. 'The size-change principle for program termination'. In: *Symposium on Principles of programming languages*. 2001, pp. 81–92. DOI: 10.1145/360204.360210.

[104] Andreas Abel and Thorsten Altenkirch. 'A predicative analysis of structural recursion'. In: *Journal of Functional Programming*, vol. 12 (2002). DOI: 10.1017/S0956796801004191.

[105] Peter Höfner and Bernhard Möller. 'Dijkstra, floyd and warshall meet kleene'. In: *Formal Aspects of Computing*, vol. 24 (2012), pp. 459–476. DOI: 10.1007/s00165-012-0245-4.

[106] Sushant S Khopkar et al. 'Efficient algorithms for incremental all-pairs shortest-paths, closeness and betweenness in social network analysis'. In: *Social Network Analysis and Mining*, vol. 4 (2014), pp. 1–20. DOI: 10.1007/s13278-014-0220-6.

[107] Sorin Stratulat. 'Cyclic proofs with ordering constraints'. In: *Automated Reasoning with Analytic Tableaux and Related Methods*. Springer, 2017, pp. 311–327. DOI: 10.1007/978-3-319-66902-1_19.

[108] Sorin Stratulat. 'Validating back-links of FOLID cyclic pre-proofs'. In: *International Workshop on Classical Logic and Computation*. Open Publishing Association, 2018, pp. 39–53. DOI: 10.4204/EPTCS.281.4.

[109] Amir M Ben-Amram and Chin Soon Lee. 'Program termination analysis in polynomial time'. In: *Transactions on Programming Languages and Systems*, vol. 29 (2007), pp. 1–37. DOI: 10.1145/1180475.1180480.

[110] Andrew Reynolds and Viktor Kuncak. 'Induction for SMT solvers'. In: *Verification, Model Checking, and Abstract Interpretation*. Springer Berlin Heidelberg, 2015, pp. 80–98. DOI: 10.1007/978-3-662-46081-8_5.

[111] Hiroshi Unno, Sho Torii and Hiroki Sakamoto. 'Automating induction for solving Horn clauses'. In: *International Conference Computer Aided Verification*. Springer, 2017, pp. 571–591. DOI: 10.1007/978-3-319-63390-9_30.

[112] Dan Rosén. *HipSpec results*. URL: https://danr.github.io/hipspec/.

[113] Charles Gregory Nelson. *Techniques for program verification*. Stanford University, 1980.

[114] Thomas Genet. 'Termination criteria for tree automata completion'. In: *Journal of Logical and Algebraic Methods in Programming*, vol. 85 (2016), pp. 3–33. DOI: 10.1016/j.jlamp.2015.05.003.

[115] Margus Veanes. *On computational complexity of basic decision problems of finite tree automata*. Tech. rep. 1997.

[116] Michael Fay. 'First-order unification in an equational theory'. In: *Proceedings of the 4th Workshop on Automated Deduction, Austin, Texas, 1979*. 1979.

[117] Nachum Dershowitz. 'Orderings for term-rewriting systems'. In: *Theoretical computer science*, vol. 17 (1982), pp. 279–301. DOI: 10.1016/0304-3975(82)90026-3.

[118] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998. DOI: 10.1017/CBO9781139172752.

[119] Nachum Dershowitz and Zohar Manna. 'Proving termination with multiset orderings'. In: *Commun. ACM*, vol. 22 (1979), pp. 465–476. DOI: 10.1145/359138.359142.

[120] Jean-Marie Hullot. 'Canonical forms and unification'. In: *International Conference on Automated Deduction*. Springer Berlin Heidelberg. 1980, pp. 318–334. DOI: 10.1007/3-540-10009-1_25.

[121] Rachid Echahed. 'On completeness of narrowing strategies'. In: *Colloquium on Trees in Algebra and Programming*. Springer Berlin Heidelberg. 1988, pp. 89–101. DOI: 10.1007/BFb0026098.

[122] Nachum Dershowitz and G. Sivakumar. 'Solving goals in equational languages'. In: *Conditional Term Rewriting Systems*. Springer Berlin Heidelberg. 1988, pp. 45–55. DOI: 10.1007/3-540-19242-5_4.

[123] Aart Middeldorp, Satoshi Okui and Tetsuo Ida. 'Lazy narrowing: strong completeness and eager variable elimination'. In: *Theoretical Computer Science*, vol. 167 (1996), pp. 95–130. DOI: 10.1016/0304-3975(96)00071-0.

[124] Philip Wadler. 'Deforestation: transforming programs to eliminate trees'. In: *Theoretical Computer Science*, vol. 73 (1990), pp. 231–248. DOI: 10.1016/0304-3975(90)90147-A.

Appendix

A Proofs for Section 2.2 (Operational Semantics)

Lemma 2.1. Let P be a Σ-program. If $\Sigma \cup \Gamma \vdash f\ a_1\ \cdots\ a_n : \tau$ is an applicative expression and $P(f)\ a_1\ \cdots\ a_n \Downarrow_\emptyset b$ is defined, then $\Sigma \cup \Gamma \vdash b : \tau$.

Proof. We will show that, if $\Sigma \cup \Gamma \vdash d\theta\ a_1\ \cdots\ a_n : \tau$ and $d\ a_1\ \cdots\ a_n \Downarrow_\theta a$, then $\Sigma \cup \Gamma \vdash a : \tau$ by induction on the incremental matching relation.

- In the base case, where $a\ a_1\ \cdots\ a_n \Downarrow_\theta a\theta\ a_1\ \cdots\ a_n$, we have by assumption that $\Sigma \cup \Gamma \vdash a\theta\ a_1\ \cdots\ a_n : \tau$ as required.
- Suppose $(\lambda x.\, d)\ a_1\ \cdots\ a_n \Downarrow_\theta b$ and that $\Sigma \cup \Gamma \vdash (\lambda x.\, d\theta)\ a_1\ \cdots\ a_n : \tau$. By inversion, we have that $d\ a_2\ \cdots\ a_n \Downarrow_{\theta \cup \{x \mapsto a_1\}} b$ and $\Sigma \cup \Gamma \cup \{x : \tau_1\} \vdash d\theta : \tau_2 \to \cdots \to \tau_n \to \tau$ where $\Sigma \cup \Gamma \vdash a_i : \tau_i$ for all $i \leq n$. Thus $\Sigma \cup \Gamma \vdash d(\theta \cup \{x \mapsto a_1\}) : \tau_2 \to \cdots \to \tau_n \to \tau$ and, by induction, $\Sigma \cup \Gamma \vdash b : \tau$ as required.
- Finally, suppose $\Sigma \cup \Gamma \vdash \texttt{case}\ \theta(x)\ \texttt{of}\ \{k_i\ \overline{x}_i \mapsto d_i\theta \mid i \leq n\}\ a_1\ \cdots\ a_n : \tau$ and $\texttt{case}\ x\ \texttt{of}\ \{k_i\ \overline{x}_i \mapsto d_i \mid i \leq n\}\ a_1\ \cdots\ a_n \Downarrow_\theta a$ is due to the incremental matching $d_i\ a_1\ \cdots\ a_n \Downarrow_{\theta \cup \{\overline{x_i \mapsto b}\}} a$. By inversion, we have that $\Sigma \cup \Gamma \cup \{\overline{x_i : \Delta(d)(k_i)[\overline{\sigma/\alpha}]}\} \vdash d_i\theta : \tau_1 \to \cdots \to \tau_n \to \tau$ for each $i \leq n$ and $\Sigma \cup \Gamma \vdash \theta(x) : d\,\overline{\sigma}$. Therefore, $\Sigma \cup \Gamma \vdash d_i(\theta \cup \{\overline{x_i \mapsto b_i}\}) : \tau_1 \to \cdots \to \tau_n \to \tau$. It follow from the induction hypothesis that $\Sigma \cup \Gamma \vdash b : \tau$ as required.

□

Corollary 2.2. Let P be a Σ-program. If $\Sigma \cup \Gamma \vdash a : \tau$ is an applicative expression and $a \to_P^* b$, then $\Sigma \cup \Gamma \vdash b : \tau$.

Proof. Suppose $C[f\ a_1\ \cdots\ a_n] \to_P C[a]$ is due to $P(f)\ a_1\ \cdots\ a_n \Downarrow_\emptyset a$. It is straightforward to see by induction that on the context $C[\cdot]$ that, whenever $\Sigma \cup \Gamma \vdash C[f\ a_1\ \cdots\ a_n] : \tau$, there must exists a type σ such that $\Sigma \cup \Gamma \vdash a_1\ \cdots\ a_n : \sigma$ and $\Sigma \cup \Gamma \cup \{x : \sigma\} \vdash_{\Delta,\Sigma} C[x] : \tau$. By applying Lemma 2.1, we have that $\Sigma \cup \Gamma \vdash b : \sigma$ and thus $\Sigma \cup \Gamma \vdash C[b] : \tau$ as required.

It then follows by induction that this property extends to the many-step reduction relation. □

Lemma 2.3. If $P(f)\ a_1\ \cdots\ a_n \Downarrow_\emptyset a$, then $P(f)\ a_1\theta\ \cdots\ a_n\theta \Downarrow_\emptyset a\theta$ for any substitution θ and $P(f)\ a_1\ \cdots\ a_n\ a_{n+1}\ \cdots\ a_m \Downarrow_\emptyset a\ a_{n+1}\ \cdots\ a_m$ for any applicative expressions $a_{n+1}, \ldots, a_m$.

Proof. We will show, by structural induction on the definitional expression d, that $d\ a_1\ \cdots\ a_n \Downarrow_{\theta'} a$ implies $d\ a_1\theta\ \cdots\ a_n\theta\ a_{n+1}\ \cdots\ a_m \Downarrow_{\theta'\theta} a\theta\ a_{n+1}\ \cdots\ a_m$ for any substitution θ and applicative expressions $a_{n+1}, \ldots, a_m$.

- In the base case of an applicative expression a, we have that $a\ a_1\ \cdots\ a_n \Downarrow_{\theta'} a\theta'\ a_1\ \cdots\ a_n$. Furthermore, by definition, $a\ a_1\theta\ \cdots\ a_n\theta\ a_{n+1}\ \cdots\ a_m \Downarrow_{\theta'\theta} a\theta'\theta\ a_1\theta\ \cdots\ a_n\theta\ a_{n+1}\ \cdots\ a_m$. As this latter reduct is equivalent to the expression $(a\theta'\ a_1\ \cdots\ a_n)\theta\ a_{n+1}\ \cdots\ a_m$, we are done.

- Consider the definitional expression $\lambda x.\, d$, for which $(\lambda x.\, d)\ a_1\ \cdots\ a_n \Downarrow_{\theta'} a$ just if $d\ a_2\ \cdots\ a_n \Downarrow_{\theta' \cup \{x \mapsto a_1\}} a$. From the induction hypothesis on d, we have that $d\ a_2\theta\ \cdots\ a_n\theta\ a_{n+1}\ \cdots\ a_m \Downarrow_{\theta'\theta \cup \{x \mapsto a_1\theta\}} a\theta\ a_{n+1}\ \cdots\ a_m$ and, therefore, $(\lambda x.\, d)\ a_1\theta\ \cdots\ a_n\theta\ a_{n+1}\ \cdots\ a_m \Downarrow_{\theta'\theta} a\theta\ a_{n+1}\ \cdots\ a_m$ as required.

- In the case of a case expression $d = \texttt{case}\ x\ \texttt{of}\ \{k_i\ \overline{x}_i \mapsto d_i \mid i \leq n\}$, we have that $d\ a_1\ \cdots\ a_n \Downarrow_{\theta'} a$ just if $\theta'(x) = k_i\ b_1\ \cdots\ b_\ell$ for some $i \leq m$ and $d_i\ a_1\ \cdots\ a_n \Downarrow_{\theta' \cup \{\overline{x_i \mapsto b}\}} a$. It follows from the induction hypothesis that $d_i\ a_1\theta\ \cdots\ a_n\theta\ a_{n+1}\ \cdots\ a_m \Downarrow_{\theta'\theta \cup \{\overline{x_i \mapsto b\theta}\}} a\theta\ a_{n+1}\ \cdots\ a_m$ as the pattern variables can be assumed to be disjoint from the free variables of θ. Because $\theta'\theta(x) = k_i\ b_1\theta\ \cdots\ b_\ell\theta$, we have that $d\ a_1\theta\ \cdots\ a_n\theta\ a_{n+1}\ \cdots\ a_m \Downarrow_{\theta'\theta} a\theta\ a_{n+1}\ \cdots\ a_m$ as required.

□

Lemma 2.5. If $P(f)\ a_1\theta\ \cdots\ a_n\theta \Downarrow_\emptyset a$ but $P(f)\ a_1\ \cdots\ a_n \Downarrow_\emptyset$ is undefined, then $\theta(y)$ is of the form $k\ b_1\ \cdots\ b_{\text{arity}(k)}$ for at least one variable $x \in \bigcup_{i \leq n} \mathsf{FV}(a_i)$.

Proof. We will show that if $d\ a_1\theta\ \cdots\ a_n\theta \Downarrow_{\theta'\theta} a$ but $d\ a_1\ \cdots\ a_n \Downarrow_{\theta'}$ is undefined, then $\theta(y)$ is of the form $k\ b_1\ \cdots\ b_{\text{arity}(k)}$ for at least one variable $x \in \mathsf{FV}(\theta') \cup \bigcup_{i \leq n} \mathsf{FV}(a_i)$ by induction on the definitional expression.

- In the base case, there is nothing to show as $d\ a_1\ \cdots\ a_n \Downarrow'_\theta$ is always defined.

- When $(\lambda z.\, d)\ a_1\theta\ \cdots\ a_n\theta \Downarrow_{\theta'\theta} a$ is defined, it must also be case the case that $d\ a_2\theta\ \cdots\ a_n\theta \Downarrow_{(\theta'\theta) \cup \{z \mapsto a_1\theta\}}$ is defined by inversion. As θ does not act on the bound variable z, the substitution $(\theta'\theta) \cup \{z \mapsto a_1\theta\}$ can equivalently be expressed as $(\theta' \cup \{z \mapsto a_1\})\theta$. Furthermore, it cannot be the case that $d\ a_2 \cdots\ a_n \Downarrow_{\theta' \cup \{z \mapsto a_1\}}$ is defined as this would contradict our assumption. By

induction, therefore, $\theta(y)$ is of the form $k\ b_1\ \cdots\ b_m$ for at least one variable $y \in \mathsf{FV}(\theta' \cup \{z \mapsto a_1\}) \cup \bigcup_{2 \leq i \leq n} \mathsf{FV}(a_i)$, which is exactly the set $\mathsf{FV}(\theta') \cup \bigcup_{i \leq n} \mathsf{FV}(a_i)$ as required.

- Finally, if $\texttt{case}\ z\ \texttt{of}\ \{k_i\ \overline{x}_i \mapsto d_i \mid i \leq n\}\ a_1\theta\ \cdots\ a_n\theta \Downarrow_{\theta'\theta}$ is defined, then $\theta'(z)\theta = k_i\ b_1\ \cdots\ b_\ell$ and $d_i\ a_1\theta\ \cdots\ a_n\theta \Downarrow_{(\theta' \circ \theta) \cup \{\overline{x_i \mapsto b}\}} a$. Note that, by inspection, $\theta'(z)$ cannot be an expression other than a variable or a application with a constructor in head position.

  The assumption that $\texttt{case}\ z\ \texttt{of}\ \{k_i\ \overline{x}_i \mapsto d_i \mid i \leq n\}\ a_1\ \cdots\ a_n \Downarrow_{\theta'}$ is undefined can thus be attributed to one of the following cases:

  - Either $\theta'(z) = y$ and $\theta(y) = k_i\ b_1\ \cdots\ b_\ell$, in which case we are done.
  - Else, $\theta'(z) = k_i\ c_1\ \cdots\ c_\ell$ where $c_i\theta = b_i$ for all $i \leq \ell$. In which case, $d_i\ a_1\ \cdots\ a_n \Downarrow_{\theta' \cup \{\overline{x_i \mapsto c}\}}$ must be undefined. By induction, $\theta(x)$ is of the form $k'\ d_1\ \cdots\ d_{\ell'}$ for at least one variable $x \in \mathsf{FV}(\theta' \cup \{\overline{x_i \mapsto c}\}) \cup \bigcup_{i \leq n} \mathsf{FV}(a_i)$. Furthermore, as $\theta'(z) = k_i\ c_1\ \cdots\ c_\ell$, the free variables of each c_i are already present in $\mathsf{FV}(\theta')$. Thus we may conclude $\theta(x)$ is of the form $k'\ d_1\ \cdots\ d_{\ell'}$ for at least one variable $x \in \mathsf{FV}(\theta') \cup \bigcup_{i \leq n} \mathsf{FV}(a_i)$ as required.

□

Corollary 2.6. The least binary relation such that $C[f\ p_1\theta\ \cdots\ p_n\theta] \to_P C[a\theta]$ whenever $P(f)\ p_1\ \cdots\ p_n \Downarrow_\emptyset a$ and $\mathsf{FV}(p_i) \cap \mathsf{FV}(p_j) = \emptyset$ for all $i \neq j$ is exactly the one-step reduction relation.

Proof. We will write $\to'_P$ for the binary relation defined in this corollary's statement.

- First, we show this relation is contained within the true reduction relation defined in Definition 2.21. Suppose $C[f\ p_1\theta \cdots\ p_n\theta] \to'_P C[a\theta]$ is due to the incremental matching $P(f)\ p_1\ \cdots\ p_n \Downarrow_\emptyset a$. By Lemma 2.3, we have that $P(f)\ p_1\theta\ \cdots\ p_n\theta \Downarrow_\emptyset a\theta$ and, therefore, $C[f\ p_1\theta\ \cdots\ p_n\theta] \to_P C[a\theta]$ as required.

- In the converse direction, suppose $C[f\ a_1\ \cdots\ a_n] \to_P C[a]$ is due to the incremental matching $P(f)\ a_1\ \cdots\ a_n \Downarrow_\emptyset a$. By Lemma 2.4, there is a series of linear patterns $p_1, \ldots, p_n$ and a substitution θ such that $p_i\theta = a_i$ for all $i \leq n$ and for which $P(f)\ p_1\ \cdots\ p_n \Downarrow_\emptyset b$ is defined. Furthermore, by Lemma 2.3, $P(f)\ p_1\theta\ \cdots\ p_n\theta \Downarrow_\emptyset b\theta$. Finally, as incremental matching is functional, $b\theta$ is necessarily equal to a and thus $C[f\ a_1\ \cdots\ a_n] \to'_P C[a]$ as required.

□

Theorem 2.7. If $a \to_P b_1$ and $a \to_P b_2$, then there exists an applicative expression c such that $b_1 \to_P^* c$ and $b_2 \to_P^* c$.

Proof. We shall show that it is locally confluent by considering the Critical Pairs Lemma [118]. By Corollary 2.6, we need only consider instances of incremental matching with pattern arguments as rewrite rules.

Suppose, therefore, that $P(f)\ p_1\ \cdots\ p_n \Downarrow_\emptyset a$ and $P(f)\ q_1\ \cdots\ q_m \Downarrow_\emptyset b$ where there exists two substitutions θ_p and θ_q such that $p_i\theta_p = q_i\theta_q$ for all $i \leq n$. Assume, without loss of generality, that $n \leq m$. Note that they can be no other forms of overlap as patterns themselves do not contain function symbols. By Lemma 2.3, we have that $P(f)\ p_1\theta_p\ \cdots\ p_n\theta_p\ q_{n+1}\theta_q\ \cdots\ q_m\theta_q \Downarrow_\emptyset a\theta_p\ q_{n+1}\theta_q\ \cdots\ q_m\theta_q$ and $P(f)\ q_1\theta_q\ \cdots\ q_m\theta_q \Downarrow_\emptyset b\theta_q$. However, as incremental matching is functional, it must be the case that $a\theta_p\ q_{n+1}\theta_q\ \cdots\ q_m\theta_q = b\theta_q$ and thus the critical pair is, in fact, trivial. □

Corollary 2.8. Suppose P is a terminating program. Then, for any applicative expression a, the unique normal form $a{\downarrow_P}$ is well defined.

Proof. By direct application of Newman's Lemma and Theorem 2.7 [118]. □

Lemma 2.9. If P is an exhaustive Σ-program, then the normal form $a{\downarrow_P}$ of a closed expression $\Sigma \vdash a : d\ \tau_1\ \cdots\ \tau_n$ is necessarily of the form $k\ a_1\ \cdots\ a_m$ for some $k \in \mathrm{dom}(\Delta(d))$.

Proof. First, we will show that, assuming any datatype sub-expression of a_i or $\theta(x)$ is of the form $k\ a_1\ \cdots\ a_{\mathrm{arity}(k)}$, $d\ a_1\ \cdots\ a_m \Downarrow_\theta$ is defined when $\Sigma \vdash d\theta\ a_1\ \cdots\ a_m : d\ \tau_1\ \cdots\ \tau_n$ by induction on d:

- When d is an applicative expression, it is already defined.
- In the case of a $\lambda x.\, d$, we know that $m \geq 1$ else it would not be the case that $\Sigma \vdash (\lambda x.\, d)\theta\ a_1\ \cdots\ a_m : d\ \tau_1\ \cdots\ \tau_n$. Therefore, we must show that $d\ a_2\ \cdots\ a_m \Downarrow_{\theta \cup \{x \mapsto a_1\}}$ is defined. However, this obligation follow immediately by induction as no new sub-expressions have been introduced.
- In the case of $\texttt{case}\ x\ \texttt{of}\ \{k_i\ \overline{x}_i \mapsto d_i \mid i \leq n\}$, we have that $\theta(x)$ is of the form $k\ b_1\ \cdots\ b_\ell$ by assumption. Furthermore, as the program is exhaustive there must be some branch d_i corresponding to the constructor k. It remains to show that $d_i\theta\ a_1\ \cdots\ a_m \Downarrow_{\theta \cup \{\overline{x_i \mapsto b}\}}$ is defined. However, as in the previous case, no new sub-expressions have been introduced and thus we may appeal to the induction hypothesis.

Now we will show that any closed normal form $\Sigma \vdash a : d\ \tau_1\ \cdots\ \tau_n$ is of the form $k\ a_1\ \cdots\ a_m$ by structural induction on a:

- If a is an application with a constructor in head position, then there is nothing to prove.
- Instead, suppose it is of the form $f\ a_1\ \cdots\ a_m$ where, by induction, any datatype sub-expression of a_i is of the form $k\ a_1\ \cdots\ a_m$. As we have just shown, however, $P(f)\ a_1\ \cdots\ a_m \Downarrow_\emptyset$ is defined thus contradicting the assumption that it is not in normal form and concluding our proof.

□

Lemma 2.10. For any well-typed program, equivalence is a congruence relation that contains the reduction relation. That is, it is closed under reflexivity, symmetry, transitivity, congruence, and reduction.

Proof.

Reflexivity Clearly, $(C[a])\!\downarrow_P = (C[a])\!\downarrow_P$ for any expression a.

Symmetry Suppose $a \equiv_P b$. Let $C[\cdot]$ be an appropriately typed context. By definition, we have that $(C[a])\!\downarrow_P = (C[b])\!\downarrow_P$ and thus $(C[b])\!\downarrow_P = (C[a])\!\downarrow_P$ as required.

Transitivity Suppose $a \equiv_P b$ and $b \equiv_P c$. Let $C[\cdot]$ be an appropriately typed context. By definition, we have that $(C[a])\!\downarrow_P = (C[b])\!\downarrow_P$ and $(C[b])\!\downarrow_P = (C[c])\!\downarrow_P$. Therefore, $(C[a])\!\downarrow_P = (C[c])\!\downarrow_P$ as required.

Congruence Suppose $a \equiv_P b$ and $C[\cdot]$ is an arbitrary context $\Gamma \cup \{x : \sigma\} \vdash C[x] : \tau$. Now let $\Gamma \cup \{x : \tau\} \vdash C'[\cdot] : \texttt{Bool}$ be an appropriately typed context. By considering the compound context $C'[C[\cdot]]$, we can derive $(C'[C[a]])\!\downarrow_P = (C'[C[b]])\!\downarrow_P$ from the definition of equivalence. Therefore $C[a] \equiv_P C[b]$ as required.

Reduction Suppose $a \to_P b$ and let $C[\cdot]$ be an appropriately typed context. By confluence, we have that $(C[a])\!\downarrow_P = (C[b])\!\downarrow_P$. Therefore, $a \equiv_P b$ as required.

□

Lemma 2.11. Let P be an exhaustive Σ-program. If $\Sigma \vdash a,\ b : d\ \tau_1\ \cdots\ \tau_n$ are two P-equivalent applicative expressions, then $a \to_P^* k\ a_1\ \cdots\ a_m$ and $b \to_P^* k\ b_1\ \cdots\ b_m$ for some $k \in \mathrm{dom}(\Delta(d))$ where $a_i \equiv_P b_i$ for all $i \leq m$. Furthermore, the converse holds by congruence.

Proof. By Lemma 2.9, there exists some constructors $k,\ k' \in \mathrm{dom}(\Delta(d))$ such that $a \to_P k\ a_1\ \cdots\ a_m$ and $b \to_P k'\ b_1\ \cdots\ b_\ell$. If $k \neq k'$, then it is trivial to see that a and b are distinguished by the combinatorial context that corresponds to the expression $\lambda x.\, \texttt{case}\ x\ \texttt{of}\ \{k\ \overline{x} \mapsto \texttt{True};\ k'\ \overline{x}' \mapsto \texttt{False};\ \ldots\}$.

Now suppose $a_i \not\equiv_P b_i$ for some $i \leq m$. That is, there exist some Boolean valued context $C[\cdot]$ that can distinguish a_i and b_i. In which case, the combinatorial context that corresponds to the expression $\lambda x.\, \texttt{case}\ x\ \texttt{of}\ \{k\ \overline{x} \mapsto C[x_i];\ \ldots\}$ can distinguish a and b. As they are assumed to be equivalent, we have a contradict and thus may conclude $a_i \equiv_P b_i$ for all $i \leq m$ as required. □

B Proofs for Section 3.2 (Declarative System)

Lemma 3.1. Subtyping is a preorder, i.e. it is reflexive and transitive.

Proof. It is straightforward to see by induction on the complement of the subtyping relation that $\tau \not\sqsubseteq \tau$ leads to a contradiction. Thus, subtyping is reflexive.

We will show that the relation $R = \{(\tau_1,\, \tau_2) \mid \forall \sigma.\, \tau_1 \not\sqsubseteq \sigma \vee \sigma \not\sqsubseteq \tau_2\}$ contains the complement of the subtyping relation, i.e. the complement of R is a model, by induction over the complement of the subtyping relation.

- (SShape) Suppose $\mathcal{U}(\tau_1) \neq \mathcal{U}(\tau_2)$. Then, for any σ, it cannot be the case that both $\tau_1 \sqsubseteq \sigma$ and $\sigma \sqsubseteq \tau_2$, else we would also have $\mathcal{U}(\tau_1) = \mathcal{U}(\sigma) = \mathcal{U}(\tau_2)$, which is contradictory. Therefore, we have that $(\tau_1,\, \tau_2) \in R$ as required.

- (SMis) Suppose $\Delta^*(d_1) \not\subseteq \Delta^*(d_2)$ and, for the sake of contradiction, that $(d_1\ \overline{\tau}_1,\, d_2\ \overline{\tau}_2) \notin R$. That is, there exists some type σ such that both $d_1\ \overline{\tau}_1 \sqsubseteq \sigma$ and $\sigma \sqsubseteq d_2\ \overline{\tau}_2$ hold. By (SShape), it must be the case that σ is of the form $d_3\ \overline{\tau}_3$. Furthermore, by (SMis), $\Delta^*(d_1) \subseteq \Delta^*(d_3)$ and similarly $\Delta^*(d_3) \subseteq \Delta^*(d_2)$. However, this would imply that $\Delta^*(d_1) \subseteq \Delta^*(d_2)$, thus giving us a contradiction as required.

- (SSim) Suppose $d_1\ \overline{\tau}_1 \not\sqsubseteq d_2\ \overline{\tau}_2$ is witnessed by some $k \in \Delta^*(d_1)$ and $i \leq \text{arity}(k)$ for which $\Delta^*(d_1)(k)_i[\overline{\tau_1/\alpha}] \not\sqsubseteq \Delta^*(d_2)(k)_i[\overline{\tau_2/\alpha}]$. By induction, we have that $(\Delta^*(d_1)(k)_i[\overline{\tau_1/\alpha}],\, \Delta^*(d_2)(k)_i[\overline{\tau_2/\alpha}]) \in R$. Now suppose, for the sake of contradiction, that σ is a type such that $d_1\ \overline{\tau}_1 \sqsubseteq \sigma$ and $\sigma \sqsubseteq d_2\ \overline{\tau}_2$. By (SShape), it must be the case that σ is of the form $d_3\ \overline{\tau}_3$ and thus, by (SMis), we have that $\Delta^*(d_1) \subseteq \Delta^*(d_3)$. Therefore, the subtypings $\Delta^*(d_1)(k)_i[\overline{\tau_1/\alpha}] \sqsubseteq \Delta^*(d_3)(k)_i[\overline{\tau_3/\alpha}]$ and $\Delta^*(d_3)(k)_i[\overline{\tau_3/\alpha}] \sqsubseteq \Delta^*(d_2)(k)_i[\overline{\tau_2/\alpha}]$ hold. As this would imply, by definition, that the pair $(\Delta^*(d_1)(k)_i[\overline{\tau_1/\alpha}],\, \Delta^*(d_2)(k)_i[\overline{\tau_2/\alpha}])$ is not an element of R, we have a contradiction as required.

- (SArrL) Suppose $\tau_1 \to \sigma_1 \not\sqsubseteq \tau_2 \to \sigma_2$ is due to $\tau_2 \not\sqsubseteq \tau_1$. By induction, we have that $(\tau_2,\, \tau_1) \in R$. Then suppose, for the sake of contradiction, that σ is a type such that both $\tau_1 \to \sigma_1 \sqsubseteq \sigma$ and $\sigma \sqsubseteq \tau_2 \to \sigma_2$ hold. By (SShape), it must be the case that σ is of the form $\tau_3 \to \sigma_3$. Therefore, by (SArrL), we have that $\tau_2 \sqsubseteq \tau_3$ and $\tau_3 \sqsubseteq \tau_1$. As this would imply that $(\tau_2,\, \tau_1) \notin R$, we have a contradiction as required.

- (SArrR) Analogous to the previous case.

Therefore, subtyping is also transitive. □

Lemma 3.2 (Simulation)**.** Suppose $R \subseteq \mathsf{Dt}^* \times \mathsf{Dt}^*$ is a binary relation on intensional datatypes such that, for any pair of intensional datatypes $(d_1\ \overline{\tau},\ d_2\ \overline{\sigma}) \in R$ and constructor $k \in \mathrm{dom}(\Delta^*(d_1))$, the following properties hold:

1. $k \in \mathrm{dom}(\Delta^*(d_2))$
2. $\mathcal{U}(d_1\ \overline{\tau}) = \mathcal{U}(d_2\ \overline{\sigma})$.
3. $\mathsf{Ty}(R)(\Delta(d_1)(k)[\overline{\tau/\alpha}]_i,\ \Delta(d_2)(k)[\overline{\sigma/\alpha}]_i)$ for all $i \leq \mathrm{arity}(k)$.

Then it follows that $\mathsf{Ty}(R)$ is included in the subtyping relation.

Proof. Again let us proceed by induction on the complements, showing that $\tau_1 \not\sqsubseteq \tau_2$ implies $(\tau_1,\ \tau_2) \notin \mathsf{Ty}(R)$:

- (SShape) Suppose $\tau \not\sqsubseteq \sigma$ is due to $\mathcal{U}(\tau) \neq \mathcal{U}(\sigma)$. It is straightforward to see by induction that $\mathcal{U}(\tau) = \mathcal{U}(\sigma)$ for any $(\tau,\ \sigma) \in \mathsf{Ty}(R)$. Therefore, it cannot be the case that $(\tau,\ \sigma) \in \mathsf{Ty}(R)$ as required.
- (SMis) Suppose $d_1\ \overline{\tau} \not\sqsubseteq d_2\ \overline{\sigma}$ is due to $\mathrm{dom}(\Delta^*(d_1)) \not\subseteq \mathrm{dom}(\Delta^*(d_2))$. By assumption (1), it cannot be the case $(d_1\ \overline{\tau},\ d_2\ \overline{\sigma}) \in R$. Furthermore, there are no other ways to relate datatypes under $\mathsf{Ty}(R)$ and thus $(d_1\ \overline{\tau},\ d_2\ \overline{\sigma}) \notin \mathsf{Ty}(R)$ as required.
- (SSim) Suppose $d_1\ \overline{\tau} \not\sqsubseteq d_2\ \overline{\sigma}$ is due to $\Delta^*(d_1)(k)_i[\overline{\tau/\alpha}] \not\sqsubseteq \Delta^*(d_2)(k)_i[\overline{\sigma/\alpha}]$ for some $i \leq \mathrm{arity}(k)$. Then, it follows from the induction hypothesis that $(\Delta^*(d_1)(k)_i[\overline{\tau/\alpha}],\ \Delta^*(d_2)(k)_i[\overline{\sigma/\alpha}]) \notin \mathsf{Ty}(R)$. The only way to relate datatypes under $\mathsf{Ty}(R)$ is if they are related by R. Thus, it cannot be the case that the $(d_1\ \overline{\tau},\ d_2\ \overline{\tau})$ is an element of $\mathsf{Ty}(R)$, else we would have contradicted assumption (3).
- (SArrL) Suppose $\tau \to \sigma \not\sqsubseteq \tau' \to \sigma'$ is due to $\tau' \sqsubseteq \tau$. By induction, we have that $(\tau',\ \tau) \notin \mathsf{Ty}(R)$. Suppose $(\tau \to \sigma,\ \tau' \to \sigma') \in \mathsf{Ty}(R)$. Then, by inversion, we would have that $(\tau',\ \tau) \in \mathsf{Ty}(R)$, giving us a contradiction. Thus, the pair $(\tau \to \sigma,\ \tau' \to \sigma')$ is not an element of $\mathsf{Ty}(R)$ as required.
- (SArrR) Analogous to the previous case.

□

Theorem 3.3. Suppose $(\underline{\Delta},\ \underline{\Sigma},\ P)$ is a positive instance of the refinement typeability problem witnessed by the refinement environment Σ. Then the normal form $a{\downarrow_P}$ of an applicative expression $\Sigma \Vdash a : d\ \tau_1\ \cdots\ \tau_n$ is necessarily of the form $k\ a_1\ \cdots\ a_{\mathrm{arity}(k)}$ for some $k \in \mathrm{dom}(\Delta^*(d))$.

Proof. First, we will show, whenever $\Sigma \Vdash d\theta\ a_1\ \cdots\ a_m : d\ \overline{\tau}$ and any datatype expression $\Sigma \Vdash a : d'\ \overline{\sigma}$ appearing as a sub-expression of a_i or $\theta(x)$ is of the form $k\ a_1\ \cdots\ a_{\text{arity}(k)}$ for some $k \in \text{dom}(\Delta^*(d'))$, that $d\ a_1\ \cdots\ a_m \Downarrow_\theta$ is defined. Let us proceed by induction on d:

- When d is an applicative expression, it is already defined.
- In the case of a $\lambda x.\ d$, we know that $m \geq 1$ else it would not be the case that $\Sigma \Vdash (\lambda x.\ d)\theta\ a_1\ \cdots\ a_m : d\,\overline{\tau}$. Therefore, we must show that $d\,a_2\ \cdots\ a_m \Downarrow_{\theta \cup \{x \mapsto a_1\}}$ is defined. However, this obligation follow immediately from the induction hypothesis as no new sub-expressions have been introduced.
- In the case of $\texttt{case}\ x\ \texttt{of}\ \{k_i\ \overline{x}_i \mapsto d_i \mid i \leq n\}$, we have that $\theta(x)$ is of the form $k\ b_1\ \cdots\ b_{\text{arity}(k)}$ by assumption. From the typing assumptions, we have that $\Sigma \Vdash \theta(x) : d'\ \overline{\sigma}$ where $\text{dom}(\Delta^*(d')) \subseteq \{k_1, \ldots, k_n\}$. Furthermore, as $k \in \text{dom}(\Delta^*(d'))$ by assumption, there must be some branch d_i corresponding to the constructor k. It remains to show that $d_i\ a_1\ \cdots\ a_m \Downarrow_{\theta \cup \{\overline{x_i \mapsto b}\}}$ is defined. However, as in the previous case, no new sub-expressions have been introduced, and thus we may appeal to the induction hypothesis.

Now we will show that any closed normal form $\Sigma \Vdash a : d\ \overline{\tau}$ is of the form $k\ a_1\ \cdots\ a_m$ for some $k \in \text{dom}(\Delta^*(d))$ by structural induction on a:

- If a is an application with a constructor in head position then we're done by inversion of the typing judgement.
- If, on the other hand, it is of the form $f\ a_1\ \cdots\ a_m$ for some program variable f where, by induction, any datatype sub-expression of a_i is of the form $k\ a_1\ \cdots\ a_m$. As we have just shown, however, $P(f)\ a_1\ \cdots\ a_m \Downarrow_\emptyset$ is defined thus contradicting the assumption that it is not in normal form and concluding our proof.

□

C Proofs for Section 3.3 (Algorithmic System)

Lemma 3.4. Suppose $H \Vdash \tau \sqsubseteq \sigma \Longrightarrow C$ is a subtyping inference and θ is an assignment such that $\theta \vDash_j H$ and $\theta \vDash C$. Then we have that $\tau\theta \sqsubseteq_j \sigma\theta$.

Proof. We proceed by induction on the subtyping inference derivation $H \Vdash \tau \sqsubseteq \sigma \Longrightarrow C$ showing that, for all $j \geq 0$, if $\theta \vDash_j H$ and $\theta \vDash C$, then $\tau\theta \sqsubseteq_j \sigma\theta$.

- (ISTyVar) is trivial as the stratified subtyping always holds.

- (ISArr) In this case, C is the union of C_1 and C_2 that are derived from the subtyping inferences $H \Vdash \tau' \sqsubseteq \tau \implies C_1$ and $H \Vdash \sigma \sqsubseteq \sigma' \implies C_1$ respectively.

  By assumption $\theta \vDash C_1 \cup C_2$ and, in particular, $\theta \vDash C_1$ and $\theta \vDash C_2$. Therefore, by induction, we have that $\tau'\theta \sqsubseteq_j \tau\theta$ and $\sigma\theta \sqsubseteq_j \sigma'\theta$. Hence, the subtyping $\tau\theta \to \sigma\theta \sqsubseteq_j \tau'\theta \to \sigma'\theta$ satisfies the (SArrL) and (SArrR) rules as required.

- (ISData) In this case, C consists of $X(\underline{d}) \subseteq Y(\underline{d})$ and $k \in X(\underline{d})\ ?\ C_{ki}$ where each C_{ki} is derived from the subtyping inference $H' \Vdash \text{inj}_X(\Delta(\underline{d})(k)_i)[\overline{\tau/\alpha}] \sqsubseteq \text{inj}_Y(\Delta(\underline{d})(k)_i)[\overline{\sigma/\alpha}] \implies C_{ki}$ where $H' = H \cup \{(\text{inj}_X(\underline{d})\ \overline{\tau},\ \text{inj}_Y(\underline{d})\ \overline{\sigma})\}$. Suppose, therefore, that $\theta \vDash X(\underline{d}) \subseteq Y(\underline{d})$ and $\theta \vDash C_{ki}$ for each $k \in \theta(X)(\underline{d})$ and $i \leq \text{arity}(k)$.

  In order to derive a contradiction, suppose that $j \geq 1$ is a minimal stratification for which $\text{inj}_X(\underline{d})\ \overline{\tau} \not\sqsubseteq_j \text{inj}_Y(\underline{d})\ \overline{\sigma}$ is satisfied by θ. Note that the base case is trivially absurd.

  - Suppose $\text{inj}_X(\underline{d})\ \overline{\tau} \not\sqsubseteq_j \text{inj}_Y(\underline{d})\ \overline{\sigma}$ is derived via (SMis). However, this would imply that $\theta(X)(\underline{d}) \not\subseteq \theta(Y)(\underline{d})$, contradicting our assumptions.
  - Suppose, therefore, that there exists some $k \in \theta(X)(\underline{d})$ and $i \leq \text{arity}(k)$ such that $\text{inj}_X(\Delta(\underline{d})(k)_i)[\overline{\tau/\alpha}] \not\sqsubseteq_{j-1} \text{inj}_Y(\Delta(\underline{d})(k)_i)[\overline{\sigma/\alpha}]$ is not satisfied by θ. Note that $\theta \vDash C_{ki}$ must hold in this case. By induction, we have that, if $\theta \vDash_{j-1} H'$ then $\text{inj}_X(\Delta(\underline{d})(k)_i)[\overline{\tau/\alpha}] \not\sqsubseteq_{j-1} \text{inj}_Y(\Delta(\underline{d})(k)_i)$. From which it follows that $\theta \nvDash_{j-1} H'$. As we have already assumed that $\theta \vDash_j H$ and, therefore, $\theta \vDash_{j-1} H$, it must be the case that $\text{inj}_X(\underline{d})\ \overline{\tau} \not\sqsubseteq_{j-1} \text{inj}_Y(\underline{d})\ \overline{\sigma}$. However, this contradicts our assumption that j was minimal.

  Therefore, there cannot exist a stratification for which the subtyping does not hold.

- (ISStop) is trivial as the assumption that the history is satisfied immediately implies the subtyping is satisfied.

□

Lemma 3.5. Suppose $H \Vdash \tau \sqsubseteq \sigma \implies C$ is a subtyping inference and θ is an assignment such that $\theta \vDash_j H$ for all $j \geq 0$ and $\tau\theta \sqsubseteq \sigma\theta$. Then we have that $\theta \vDash C$.

Proof. We proceed by induction on the subtyping inference derivation $H \Vdash \tau \sqsubseteq \sigma \implies C$ showing that, if $\theta \vDash_j H$ for all $j \geq 0$ and $\tau\theta \sqsubseteq \sigma\theta$, then $\theta \vDash C$.

- (ISTyVar) is trivial as the constraints are empty and thus always satisfied.

- (ISArr) In this case, C is the union of C_1 and C_2 that are derived from the subtyping inferences $H \Vdash \tau' \sqsubseteq \tau \implies C_1$ and $H \Vdash \sigma \sqsubseteq \sigma' \implies C_1$ respectively.

Suppose that $\tau\theta \to \sigma\theta \sqsubseteq \tau'\theta \to \sigma'\theta$. By inversion, it follows that $\tau'\theta \sqsubseteq \tau\theta$ and $\sigma\theta \sqsubseteq \sigma'\theta$. Therefore, by induction, we have that $\theta \vDash C_1$ and $\theta \vDash C_2$. In particular, $\theta \vDash C_1 \cup C_2$ as required.

- (ISData) In this case, C consists of $X(\underline{d}) \subseteq Y(\underline{d})$ and $k \in X(\underline{d})\ ?\ C_{ki}$ where each C_{ki} is derived from the subtyping inference $H' \Vdash \mathrm{inj}_X(\Delta(\underline{d})(k)_i)[\overline{\tau/\alpha}] \sqsubseteq \mathrm{inj}_Y(\Delta(\underline{d})(k)_i)[\overline{\sigma/\alpha}] \Longrightarrow C_{ki}$ where $H' = H \cup \{(\mathrm{inj}_X(\underline{d})\ \overline{\tau},\ \mathrm{inj}_Y(\underline{d})\ \overline{\sigma})\}$.

  Suppose that $\mathrm{inj}_X(\underline{d})\ \overline{\tau} \sqsubseteq \mathrm{inj}_Y(\underline{d})\ \overline{\sigma}$. By inversion, we have that $\theta \vDash X(\underline{d}) \subseteq Y(\underline{d})$ and, for each $k \in \theta(X)(\underline{d})$ and $i \leq \mathrm{arity}(k)$, we have that $\mathrm{inj}_X(\Delta(\underline{d})(k)_i)[\overline{\tau/\alpha}] \sqsubseteq \mathrm{inj}_Y(\Delta(\underline{d})(k)_i)[\overline{\sigma/\alpha}]$ is satisfied by θ. Therefore, by induction, $\theta \vDash C_{ki}$ for each $k \in \theta(X)(\underline{d})$ and $i \leq \mathrm{arity}(k)$. It follows that $\theta \vDash C$ as required.

- (ISStop) is trivial as the constraints are empty and thus always satisfied.

□

Corollary 3.6. Suppose $\vDash \tau \sqsubseteq \sigma \Rightarrow C$ is a subtyping inference and θ is an assignment. Then $\tau\theta \sqsubseteq \sigma\theta$ if, and only if, $\theta \vDash C$.

Proof. Suppose that $\vdash \tau \sqsubseteq \sigma \Rightarrow C$ is a subtyping inference and θ is an assignment such that $\tau\theta \sqsubseteq \sigma\theta$. As the empty hypotheses are trivially satisfied at all stratification, it follows by Lemma 3.5 that $\theta \vDash C$ as required.

Conversely, suppose that $\theta \vDash C$. By Lemma 3.4, we have that $\tau\theta \sqsubseteq_j \sigma\theta$ for all stratifications as, again, the empty hypotheses are trivially satisfied. Therefore, $\tau\theta \sqsubseteq \sigma\theta$ as required. □

Lemma 3.7. Suppose $\Gamma \Vdash e \Longrightarrow \tau$, C is an instance of type inference and θ is an assignment such that $\theta \vDash C$. Then $(\!|\Gamma\theta|\!) \Vdash e : \tau\theta$.

Proof. Let us proceed by induction on the inference judgement:

- (IVar) Suppose $x : \forall\overline{X}.\,\overline{\alpha}.\,C \supset \tau \in \Gamma$ is instantiated by $[\overline{Y/X}]$ and $[\overline{\tau/\alpha}]$, producing the constraints $C[\overline{Y/X}]$. By assumption $\theta \vDash C[\overline{Y/X}]$ and, therefore, that $x : \forall\overline{\alpha}.\,\tau[\overline{Y/X}]\theta \in (\!|\Gamma\theta|\!)$. Thus, we can derive $(\!|\Gamma\theta|\!) \Vdash x : \tau[\overline{Y/X}][\overline{\tau/\alpha}]\theta$ via the (TVar) rule as required.

- (ICon) In the case of a constructor k, the constraints set C is exactly the set $\{k \in X(\underline{d})\}$. Let $\underline{d} \in \underline{D}$ be the datatype identifier for which $k : \forall\overline{\alpha}.\,\tau \in \Delta(\underline{d})$. As C must be satisfied by θ, we have that $k \in \theta(X)(\underline{d})$. Therefore, $k : \forall\overline{\alpha}.\,\mathrm{inj}_{\theta(X)}(\tau) \in \Delta^*(\mathrm{inj}_{\theta(X)}(\underline{d}))$ and we can derive, via the (TCon) rule, $(\!|\Gamma\theta|\!) \Vdash k : \mathrm{inj}_{\theta(X)}(\tau)[\overline{\sigma/\alpha}]$ for any types $\overline{\sigma}$ as required.

- (IAbs) In this case, the constraint set is derived from the expression inference $\Gamma \cup \{x : \tau\} \Vdash e \Longrightarrow \sigma,\ C$ where e is the body of the λ-expression. By induction, we have that $\Gamma\theta \cup \{x : \tau\theta\} \Vdash_{(\!|\Sigma|\!)} e : \sigma\theta$. Therefore, $\Gamma\theta \vdash_{(\!|\Sigma|\!)} \lambda x.\, e : \tau\theta \to \sigma\theta$ follows by (TAbs) as required.

- (IApp) For an application expression $e_1\ e_2$, the constraint set consists of three components: C_1 which is derived from the expression inference $\Gamma \Vdash e \Longrightarrow \tau \to \sigma,\ C_1$, C_2 which is derived from the expression inference $\Gamma \Vdash e_2 \Longrightarrow \tau',\ C_2$, and C_3 which is derived from the subtyping inference $\Vdash \tau' \sqsubseteq \tau \Longrightarrow C_3$.

  As θ must satisfy each of these constraint sets, we have that $(\!|\Gamma\theta|\!) \Vdash e_1 : \tau_1\theta \to \tau_2\theta$ and $(\!|\Gamma\theta|\!) \Vdash e_2 : \tau_1'\theta$ by induction. Furthermore, $\tau_1'\theta \sqsubseteq \tau_1\theta$ follows from Corollary 3.6. Therefore, $(\!|\Gamma\theta|\!) \Vdash e_2 : \tau_1\theta$ by (TSub) and, consequently, $(\!|\Gamma|\!) \Vdash e_1\ e_2 : \tau_2\theta$ as required.

- (ICase) If the expression is of the form $\texttt{case}\ e\ \texttt{of}\ \{k_i\ \overline{x}_i \mapsto e_i \mid i \leq n\}$ where $\Gamma \Vdash e \Longrightarrow \mathrm{inj}_X(\underline{d})\ \overline{\tau},\ C_0$ and, for each $i \leq m$, we have that $\Gamma \cup \Gamma_i \Vdash e_i \Longrightarrow \sigma_i,\ C_i$ and $\Vdash \sigma_i \sqsubseteq \sigma \Longrightarrow C_i'$, the constraint set is $C_0 \cup \{X(\underline{d}) \subseteq \{k_1, \ldots, k_m\}\}$ and, for each $i \leq m$, $k_i \in X(\underline{d})\ ?\ C_i \cup C_i'$.

  It follows from the induction hypotheses that $(\!|\Gamma\theta \cup \Gamma_i\theta|\!) \Vdash e_i : \sigma_i\theta$ for each $i \in \theta(X)(\underline{d})$. Furthermore, by Corollary 3.6, we have that $\sigma_i\theta \sqsubseteq \sigma\theta$ for each $i \in \theta(X)(\underline{d})$. In which case, we can derive $(\!|\Gamma\theta \cup \Gamma_i\theta|\!) \Vdash e_i : \sigma\theta$ via (TSub).

  As θ satisfies these constraints, we have that $(\!|\Gamma\theta|\!) \Vdash e : \mathrm{inj}_{\theta(X)}(\underline{d})\ \overline{\tau}\theta$. Finally, as $\theta(X)(\underline{d}) \subseteq \{k_1, \ldots, k_m\}$ and each reachable branch is appropriately typed, we have that $(\!|\Gamma\theta|\!) \Vdash \texttt{case}\ e\ \texttt{of}\ \{k_i\ \overline{x}_i \mapsto e_i \mid i \leq n\} : \sigma\theta$ as required.

□

Lemma 3.8. Suppose $\Gamma' \Vdash e : \tau'$ and $\Gamma \Vdash e \Longrightarrow \tau,\ C$ is an instance of type inference for which there exists some assignment θ such that $(\!|\Gamma\theta|\!) \sqsubseteq \Gamma'$. Then there exists an assignment θ' that (1) satisfies C, (2) agrees with θ on the refinement variables of Γ, i.e. $\theta' \equiv_{\mathsf{FRV}(\Gamma)} \theta$, and such that (3) $\tau\theta' \sqsubseteq \tau'$.

Proof. Let us proceed by induction on the typing $\Gamma' \Vdash e : \tau'$:

- (TVar) In this case, we have that $\Gamma' \Vdash x : \tau'[\overline{\sigma'/\alpha}]$ is due to $x : \forall\overline{\alpha}.\, \tau' \in \Gamma'$. From the assumption $(\!|\Gamma\theta|\!) \sqsubseteq \Gamma'$, we have that $x : \forall\overline{\alpha}.\, \forall\overline{X}.\, C \supset \tau \in \Gamma$ such that $\tau\theta\theta_x \sqsubseteq \tau'$ for some assignment θ_x such that $\theta_x \vDash C\theta$, where θ is assumed to not apply to the bound refinement variables $\overline{X}$. Inference then produces the type $\tau[\overline{Y/X}][\overline{\sigma/\alpha}]$ and the constraints $C[\overline{Y/X}]$ where both $\overline{Y}$ and $\overline{\sigma}$ are fresh.

  As $\overline{\sigma}$ and $\overline{\sigma'}$ refine the same underlying type and $\overline{\sigma}$ is fresh, we may take θ' to be such that $\overline{\sigma}\theta' = \overline{\sigma'}$ without violating the condition (2) that $\theta' \equiv_{\mathsf{FRV}(\Gamma)} \theta$. Likewise, let us assume that $\overline{Y}\theta' = \overline{X}\theta\theta_x$. Then it is straightforward to see that (1) $\theta' \vDash C[\overline{Y/X}]$ and (3) $\tau[\overline{Y/X}][\overline{\sigma/\alpha}]\theta' \sqsubseteq \tau'[\overline{\sigma'/\alpha}]$ as required.

- (TCon) In this case, we have that $\Gamma' \Vdash k : \mathrm{inj}_{\phi}(\underline{\tau})\ [\overline{\sigma'/\alpha}]$ where $k \in \phi(\underline{d})$ and $k : \forall\overline{\alpha}.\underline{\tau} \in \underline{\Delta}(\underline{d})$. Likewise, inference for expressions produces the type $\mathrm{inj}_X(\underline{\tau})\ [\overline{\sigma/\alpha}]$ where X is fresh and $\overline{\sigma}$ and $\overline{\sigma'}$ refine the same underlying types.

  As each the refinement variables in $\overline{\sigma} \cup \{X\}$ is fresh, we can pick θ' such that $\theta'(X)$ is ϕ and $\overline{\sigma\theta} = \overline{\sigma'}$ without violating (2). It immediately follows that (1) $\theta' \vDash k \in X(\underline{d})$ and (3) $\mathrm{inj}_{\theta'(X)}(\underline{\tau})\ \overline{\sigma\theta'} \sqsubseteq \mathrm{inj}_{\phi}(\underline{\tau})\ \overline{\sigma'}$ by reflexivity.

- (TSub) Suppose $\Gamma \Vdash e \Longrightarrow \tau,\ C$ and, in this case, $\Gamma' \Vdash e : \tau'$ is due to $\Gamma' \Vdash e : \tau''$ where $\tau'' \sqsubseteq \tau'$ and $(\!|\theta\Gamma|\!) \sqsubseteq \Gamma'$.

  By induction, there exists some assignment θ' such that (1) $\theta' \vDash C$, (2) $\theta' \equiv_{\mathsf{FRV}(\Gamma)} \theta$, and (3) $\tau\theta' \sqsubseteq \tau''$. Using the same witness, we need only show that (3) $\tau\theta' \sqsubseteq \tau'$, which follows immediately by transitivity.

- (TAbs) In this case, $\Gamma' \Vdash \lambda x.\, e : \tau' \to \sigma'$ is due to $\Gamma' \cup \{x : \tau'\} \Vdash e : \sigma'$. Likewise, $\Gamma \Vdash \lambda x.\, e \Longrightarrow \tau \to \sigma,\ C$ is due to $\Gamma \cup \{x : \tau\} \Vdash e \Longrightarrow \sigma,\ C$ by inversion.

  As the free refinement variables of τ are fresh, we can pick θ' such that $\tau\theta' = \tau$ without violating (2). Note that, this implies $(\!|(\Gamma \cup \{x : \tau\})\theta'|\!)$ is a sub-environment of $\Gamma' \cup \{x : \tau'\}$ as $\Gamma\theta' = \Gamma\theta$. Therefore, by induction, there exists an assignment θ'' such that (1) $\theta'' \equiv_{\mathsf{FRV}(\Gamma)\cup\mathsf{FRV}(\tau)} \theta'$, (2) $\theta'' \vDash C$, and (3) $\sigma\theta'' \sqsubseteq \sigma'$. Using the same witness, we have that (1) $\theta'' \equiv_{\mathsf{FRV}(\Gamma)} \theta$ by transitivity, (2) $\theta'' \vDash C$, and (3) $\tau\theta'' \to \sigma\theta'' \sqsubseteq \tau' \to \sigma'$ as required.

- (TApp) Suppose $\Gamma' \Vdash e_1\ e_2 : \sigma'$ is due to $\Gamma' \Vdash e_1 : \tau' \to \sigma'$ and $\Gamma' \Vdash e_2 : \tau'$. Likewise, by inversion, $\Gamma \Vdash e_1 \Longrightarrow \tau_1 \to \sigma,\ C_1$ and $\Gamma \Vdash e_2 \Longrightarrow \tau_2,\ C_2$ and $\Vdash \tau_2 \sqsubseteq \tau_1 \Longrightarrow C_3$. By induction, there exists a substitution $\theta_1 \equiv_{\mathsf{FRV}(\Gamma)} \theta$ such that (1a) $\theta_1 \vDash C_1$ and (3a) $\tau_1\theta_1 \to \sigma\theta_1 \sqsubseteq \tau' \to \sigma'$. Likewise, there exists a substitution $\theta_2 \equiv_{\mathsf{FRV}(\Gamma)} \theta$ such that (1b) $\theta_2 \vDash C_2$ and (3b) $\tau_2\theta_1 \sqsubseteq \tau'$.

  Due to the introduction of fresh refinement variables, the inference system guarantee that the only common variables between sibling branches are those found in the type environment. Therefore, we can define:

$$\theta'(Z) = \begin{cases} \theta_1(Z) & \text{if } Z \in \mathsf{FRV}(\tau_1 \to \sigma) \\ \theta_2(Z) & \text{if } Z \in \mathsf{FRV}(\tau_2) \\ \theta(Z) & \text{otherwise} \end{cases}$$

  without violating (2).

  Furthermore, as $\tau_2\theta' \sqsubseteq \tau'$ and $\tau' \sqsubseteq \tau_1\theta'$, we have that $\tau_2\theta' \sqsubseteq \tau_1\theta'$. Therefore, by Corollary 3.6, (1) $\theta' \vDash C_3$ (and $\theta' \vDash C_1 \cup C_2$ by (1a) and (1b)). Finally, (3) $\sigma\theta' \sqsubseteq \sigma'$ follows from (3a).

- (TCase) In this case, $\Gamma' \Vdash \texttt{case}\ e\ \texttt{of}\ \{k_i\ \overline{x}_i \mapsto e_i \mid i \leq n\} : \sigma'$ is due to $\Gamma' \Vdash e : \mathrm{inj}_\phi(\underline{d})\ \overline{\tau'}$ and $\Gamma' \cup \{x_i : \mathrm{inj}_\phi(\Delta^*(\underline{d})(k_i))\ \overline{[\tau'/\alpha]}\} \Vdash e_i : \sigma'$, for each $k_i \in \phi(\underline{d})$. Furthermore, we may assume that $\phi(\underline{d}) \subseteq \{k_1, \ldots, k_m\}$. Likewise, by inversion, we have that $\Gamma \Vdash e \Longrightarrow \mathrm{inj}_X(\underline{d})\ \overline{\tau}, C_0$ and, for each $k_i \in \underline{\Delta}(\underline{d})$, $\Gamma \cup \{x_i : \mathrm{inj}_X(\Delta^*(\underline{d})(k_i))\ \overline{[\tau/\alpha]}\} \Vdash e_i \Longrightarrow \sigma_i, C_i$ and $\Vdash \sigma_i \sqsubseteq \sigma \Longrightarrow C_i'$ where σ is fresh.

  By induction, there exists an assignment θ_0 such that (1) $\theta_0 \vDash C_0$, (2) $\theta_0 \equiv_{\mathsf{FRV}(\Gamma)} \theta$ and (3) $\mathrm{inj}_{\theta_0(X)}(\underline{d})\ \overline{\tau\theta_0} \sqsubseteq \mathrm{inj}_\phi(\underline{d})\ \overline{\tau'}$. From which, we can derive the subtypings $\vdash \mathrm{inj}_{\theta_0(X)}(\Delta^*(\underline{d})(k_i)_j)\ \overline{[\tau\theta/\alpha]} \sqsubseteq \mathrm{inj}_\phi(\Delta^*(\underline{d})(k_i)_j)\ \overline{[\tau'/\alpha]}$ for each $k_i \in \theta(X)$ and $j \leq \mathrm{arity}(k)$ by (SSim). Furthermore, there exists assignments $\theta_i \equiv_{\mathsf{FRV}(\Gamma) \cup \{X\} \cup \mathsf{FRV}(\overline{\tau})} \theta$ such that (1i) $\theta_i \vDash C_i$ and (3i) $\sigma_i\theta_i' \sqsubseteq \sigma$ for each $k_i \in \phi(\underline{d})$.

  As σ is fresh, we may also construct some assignment θ' such that (3) $\sigma\theta' = \sigma'$ and $\theta' \equiv_{\mathsf{FRV}(\Gamma)} \theta$. We use the same argument as the previous case (i.e. the sibling branches only share refinement variables through the context) to combine these substitutions:

$$\theta''(Z) = \begin{cases} \theta'(Z) & \text{if } Z \in \mathsf{FRV}(\sigma) \\ \theta_0(Z) & \text{if } Z = X \text{ or } Z \in \mathsf{FRV}(C_0) \\ \theta_i(Z) & \text{if } Z \in \mathsf{FRV}(\sigma_i) \cup \mathsf{FRV}(C_i) \\ \theta(Z) & \text{otherwise} \end{cases}$$

  without violating (1)

  By definition, we have that:

  - $\sigma\theta'' = \sigma'$
  - $\theta'' \vDash C_0$
  - For each $k_i \in \phi(\underline{d})$, $\theta'' \vDash C_i$ and $\theta'' \vDash C_i'$.
  - Furthermore, in each of these cases, we have $\sigma_i\theta'' \sqsubseteq \sigma\theta''$ and thus $\theta_i' \vDash C_i$ by Corollary 3.6.
  - Finally, as $\theta''(X) \subseteq \phi(X)$, we have that $\theta''(X) \subseteq \{k_1, \ldots, k_m\}$.

  Therefore, (1) the constraints C_0, $\bigcup_{i \leq m} k_i \in X(\underline{d})\ ?\ C_i \cup C_i'$, and the constraint $X(\underline{d}) \subseteq \{k_1, \ldots, k_m\}$ are satisfied as required.

□

Theorem 3.9. Let P be a program such that $\vdash P \Longrightarrow \Sigma$. Then, $(\underline{\Delta}, \underline{\Sigma}, P)$ is a positive instance of the refinement typeability problem if, and only if, $(\!|\Sigma|\!)$ provides at least one type to every program variable.

Proof. First, we shall show that if $\Vdash P \Longrightarrow \Sigma$ and $(\!|\Sigma|\!)$ isn't empty for any program variable, then P is typeable under $(\!|\Sigma|\!)$. Our proof is by induction on the program:

- In the base case, the program is empty and thus is trivially well-typed.

- Consider a program of the form $P; f = e$. By inversion, we have that Σ is of the form $\Sigma' \cup \{f : \rho\}$ where $\Vdash P \Longrightarrow \Sigma'$, $\Sigma' \cup \{f : \tau\} \Vdash e \Longrightarrow \tau'$, C_1, and $\Vdash \tau' \sqsubseteq \tau \Longrightarrow C_2$ where ρ is defined as $\forall\overline{\alpha}.\, \forall\overline{X}.\, C_1 \cup C_2 \supset \tau$. As $(\!|\Sigma|\!)$ is not empty of any program variable by assumption, it clearly is the case that $(\!|\Sigma'|\!)$ is also non-empty. Therefore, by induction, P is typeable under $(\!|\Sigma'|\!)$.

  As the preceding sub-program P does not depend on f, it is plain to see that P is also typeable under $(\!|\Sigma|\!)$ by weakening. It remains to show that $(\!|\Sigma|\!) \Vdash e : \tau\theta$ for each $\theta \vDash C_1 \cup C_2$. It follows from the expression inference judgement and Lemma 3.7 that $(\!|\Sigma' \cup \{f : \tau\theta\}|\!) \Vdash e : \tau'\theta$ (note that, as Σ' is a closed environment, it is unaffected by θ). Likewise from the subtyping inference, it then follows by Corollary 3.6 that $(\!|\Sigma' \cup \{f : \tau\theta\}|\!) \Vdash e : \tau\theta$, which implies $(\!|\Sigma|\!) \Vdash e : \tau'\theta$ by weakening as required.

In the converse direction, we will show that, if P is a typeable under the refinement environment Σ' and $\Vdash P \Longrightarrow \Sigma$, then $(\!|\Sigma|\!) \sqsubseteq \Sigma'$. Let us again proceed by induction on P:

- In the base case, we have that Σ' and $(\!|\Sigma|\!)$ are empty environments. Therefore, $(\!|\Sigma|\!) \sqsubseteq \Sigma'$ is trivial.

- Consider a program of the form $P; f = e$. By inversion, we have that Σ' is of the form $\Sigma'_1 \cup \Sigma'_2$ where P is Σ'_1 program and Σ'_2 contains a series of typings for f such that $\Sigma' \Vdash e : \sigma$ for each $f : \forall\overline{\alpha}.\, \sigma \in \Sigma'_2$. Likewise, by inversion, we have that Σ is of the form $\Sigma_1 \cup \{f : \rho\}$ where $\Vdash P \Longrightarrow \Sigma_1$, $\Sigma_1 \cup \{f : \tau\} \Vdash e \Longrightarrow \tau'$, C_1, and $\Vdash \tau' \sqsubseteq \tau \Longrightarrow C_2$ where ρ is defined as $\forall\overline{\alpha}.\, \forall\overline{X}.\, C_1 \cup C_2 \supset \tau$. We have that $(\!|\Sigma_1|\!) \sqsubseteq \Sigma'_1$ by induction.

  It remains to show that, for each typing $f : \forall\overline{\alpha}.\, \sigma \in \Sigma'_2$, there exists a substitution θ such that $\theta \vDash C_1 \cup C_2$ and $\tau\theta \sqsubseteq \sigma$. As τ is fresh, we can find a substitution θ' such that $\tau\theta' = \sigma$, from which it follows that $(\!|\Sigma_1 \cup \{f : \tau\theta'\}|\!) \sqsubseteq \Sigma'$. Therefore, by Lemma 3.8, we have some assignment θ that satisfies $\theta \vDash C_1$ and for which $\tau'\theta \sqsubseteq \sigma$ or, equivalently, $\tau'\theta \sqsubseteq \tau\theta$. It follows from Corollary 3.6 that θ also solves C_2. Thus, θ is a substitution that satisfies $C_1 \cup C_2$ and for which $\tau\theta \sqsubseteq \sigma$ as required.

□

D Proofs for Section 3.4 (Solving Constraints)

Theorem 3.10. For any assignment θ and atomic set of constraints, we have that $\theta \vDash C$ if, and only, if $\theta \vDash \mathsf{Sat}(C)$.

Proof. We shall show that the resolution rules preserve solutions (naturally they reflect solutions too, since they do not remove constraints). In each case, we shall assume that the derived guard holds, otherwise there is nothing to show.

- (Trans) Suppose $\phi \mathbin{?} S_1 \subseteq S_2$ and $\psi \mathbin{?} S_2 \subseteq S_3$ and both ϕ and ψ are satisfied by θ. Therefore, $S_1\theta \subseteq S_2\theta$ and $S_2\theta \subseteq S_3\theta$. It immediately follows that $S_1\theta \subseteq S_3\theta$ as required.
- (Sat) Suppose $\phi \mathbin{?} k \in X(\underline{d})$ and $\psi \cup \{k \in X(\underline{d})\} \mathbin{?} S_1 \subseteq S_2$ and both ϕ and ψ are satisfied by θ. Thus, $k \in \theta(X)(\underline{d})$ and, therefore, $S_1\theta \subseteq S_2\theta$ as required.
- (Weak) Suppose $\phi \mathbin{?} X(\underline{d}) \subseteq Y(\underline{d})$ and $\psi \cup \{k \in Y(\underline{d})\} \mathbin{?} S_1 \subseteq S_2$ and that ϕ, ψ, and $k \in X(\underline{d})$ are satisfied by θ. It follows that $\theta(X)(\underline{d}) \subseteq \theta(Y)(\underline{d})$ and $k \in \theta(X)(\underline{d})$, therefore, $k \in \theta(Y)(\underline{d})$. Thus, $S_1\theta \subseteq S_2\theta$ as required.

□

Theorem 3.11. A set of atomic constraints C is satisfiable if, and only if, $\mathsf{Sat}(C)$ does not have any trivially unsatisfiable constraints.

Proof. Clearly, if $\mathsf{Sat}(C)$ has a trivially unsatisfiable constraint, then it is unsatisfiable. Therefore, by Theorem 3.10, C is unsatisfiable.

Suppose this is not the case. We will show that the assignment $X \mapsto \phi_X$ where $\phi_X(\underline{d}) \coloneqq \{k \mid k \in X(\underline{d}) \in \mathsf{Sat}(C)\}$ is a solution to $\mathsf{Sat}(C)$ and, therefore, to C. Let $\phi \mathbin{?} S_1 \subseteq S_2$ be an atomic constraint in $\mathsf{Sat}(C)$ such that $\theta \vDash \phi$. As each inclusion $k \in X(\underline{d}) \in \phi$ is satisfied by θ just if $k \in X(\underline{d}) \in \mathsf{Sat}(C)$, we have that $S_1 \subseteq S_2 \in \mathsf{Sat}(C)$ by (Weak). Consider the form of atomic constraints that this constraint may take:

- If we have that $k \in Y(\underline{d})$, then $k \in \theta(Y)(\underline{d})$ by construction as this constraint appears unguarded in the saturated set.
- $Y(\underline{d}) \subseteq Z(\underline{d})$. In this case, any $k \in \theta(Y)(\underline{d})$ arises from a constraint $k \in Y(\underline{d})$ which transitively induces $k \in Z(\underline{d})$. Therefore, $k \in \theta(Z)(\underline{d})$ by construction.
- The case of $Y(\underline{d}) \subseteq \{k_1, \ldots, k_m\}$ is similar.

□

Theorem 3.14. Under the assumption that the size of types and the size of each function definition is bounded, the complexity of type inference is $\mathcal{O}(N)$ where N is the number of function symbols.

Proof. To perform type inference on a program $f_1 = e_1; \ldots; f_N = e_N$, we must perform N-instance of inference for expressions. By induction on the structure of an expression, If we assume a fixed bound on the number of constructors and the number of datatypes, there are $\mathcal{O}(2^I \cdot I^2)$ possible constraints associated with a constrained type scheme with I refinement variables. As we have also assumed that the size of types is fixed, the number of refinement variables it concerns is constant, and we may conclude that each instance of inference has a constant-time complexity. Therefore, the whole-program analysis is $\mathcal{O}(N)$ as required. □

E Proofs for Section 4.2 (The CycleQ Proof System)

Theorem 4.1 (Local Soundness)**.** Let $v_1 \in V \setminus \mathrm{Ax}$ be a non-axiomatic node within the pre-proof $(V,\ E,\ \lambda,\ \rho)$ such that $\theta_1 \not\models \lambda(v_1)$ for some valuation θ_1. Then, there exist a child node $v_2 \in E(v_1)$ with a necessary precursor θ_2, i.e. where $(v_1,\ \theta_1) \rightarrowtriangle (v_2,\ \theta_2)$, for which $\theta_2 \not\models \lambda(v_2)$.

Proof. Suppose $v_1 \in V \setminus \mathrm{Ax}$ and consider the possible inference rules $\rho(v_1)$:

- (Refl) This case is absurd as reflexivity is valid.
- (Reduce) Suppose that $\theta_1 \not\models a = b$, $a \to_P a'$, and $b \to_P b'$. We have that θ_1 is a necessary precursor of the sole premise $v_1 1$ by definition. Furthermore, as a result of the definition of equivalence via normalisation, $\theta_1 \not\models a' = b'$ and thus is not satisfied by a necessary precursor as required.
- (Cong) Suppose that $\theta_1 \not\models k\ a_1\ \cdots\ a_n = k\ b_1\ \cdots\ b_n$. We have that θ_1 is a necessary precursor of any premise $v_1 i$. If $\theta_1 \vDash a_i = b_i$ for all $i \leq n$, then $\theta_1 \not\models k\ a_1\ \cdots\ a_n = k\ b_1\ \cdots\ b_n$ would follow from congruence. Thus, there exists some $i \leq n$ such that $\theta_1 \not\models a_i = b_i$, i.e. it is not satisfied by a necessary precursor as required.
- (FunExt) Suppose that $\theta_1 \not\models a = b$ where $\Gamma \vdash a,\ b : \tau_1 \to \tau_2$. By the function extensionality Lemma 2.12, there exists some c such that $\theta_1 \cup \{x \mapsto c\} \not\models a\ c = b\ c$. As this valuation is a necessary precursor of θ_1, we are done.
- (Subst) Suppose that $\theta_1 \not\models C[a\theta] = b$ where θ is the substitution instance of the lemma. If $\theta \circ \theta_1 \not\models a = c$, then we are done as $\theta \circ \theta_1$ is a necessary precursor for the lemma $v_1 1$. Otherwise, we have that $a\theta\theta_1 \equiv_P c\theta\theta_1$. It cannot be the case, therefore, that $C[c\theta]\theta_1 \equiv_P b\theta_1$ else we would have a contradicted. Thus, the continuation (the right-hand premise) is not satisfied by θ. As $(v_1,\ \theta) \rightarrowtriangle (v_2,\ \theta)$ where v_2 is the lemma, we are done.
- (Case) Finally, suppose $\theta_1 \not\models a = b$ and x is the subject of case analysis. By Lemma 2.9, $\theta_1(x) \to_P k\ c_1\ \cdots\ c_n$ for some $k \in \Delta$. Let vi be the premise

associated with the constructor k. The substitution $\theta_1 \cup \{\overline{x \mapsto c}\}$ is a necessary precursor for this premise. Furthermore, by Lemma 2.11, it cannot be the case that $a[k\ x_1\ \cdots\ x_n/x]\theta_1 \equiv_P b[k\ x_1\ \cdots\ x_n/x]\theta_1$ as this would imply that $a\theta_1 \equiv_P b\theta_1$. Thus, we have a necessary precursor that does not satisfy the equation of its respective node.

□

Lemma 4.2. Let (V, E, λ, ρ) be a cyclic pre-proof with some path $(v)_{i\in\mathbb{N}}$ and suppose $(t)_{i\in\mathbb{N}}$ is a $\leq$-trace along this path. If θ_i is a valuation of some node $v_i \in V$ and $(v_i, \theta_i) \rightarrowtriangle (v_{i+1}, \theta_{i+1})$, then $t_{i+1}\theta_{i+1} \leq t_i\theta_i$ and $t_{i+1}\theta_{i+1} < t_i\theta_i$ when i is a progress-point.

Proof. Let us consider the possible justifications for v_i:

- Suppose $\rho(v_i)$ is (Case) where $x : d\ \overline{\tau}$ is the variable upon which case analysis is performed. By Lemma 2.9 and the definition of a necessary precursor, $\theta(x) \rightarrow_P k\ c_1\ \cdots\ c_n$ for some $k \in \Delta$ and v_{i+1} is the premise associated with this constructor. Furthermore, $\theta_{i+1} = \theta_i \cup \{\overline{x \mapsto c}\}$. By definition, $t_{i+1} \leq t_i[k\ x_1\ \cdots\ x_n/x]$ and thus $t_{i+1}\theta_{i+1} \leq t_i[k\ x_1\ \cdots\ x_n/x]\theta_{i+1} = t_i\theta$ as required.

- Suppose $\rho(v_i)$ is (Subst) with substitution θ and v_{i+1} is the lemma. We have that $t_{i+1}\theta \leq t_i$. The only necessary precursor to θ_i in this case is $\theta \circ \theta_i$. Thus, by stability, $t_{i+1}\theta_{i+1} \leq t_i\theta_i$ as required.

- If $\rho(v_i)$ is (FunEx), then $\theta_{i+1}(y) = \theta_i(y)$ for all y other than the fresh variable x. As t_i cannot depend on x, we have that $t_i\theta_{i+1} = t_i\theta_i$. And thus $t_{i+1}\theta_{i+1} \leq t_i\theta_i$ as required.

- In all other cases, $\theta_{i+1} = \theta_i$ and $t_{i+1} \leq t_i$. Thus, $t_{i+1}\theta_{i+1} \leq t_i\theta_i$ as required.

Furthermore, in each of the above cases, we can replace the inequality by strict inequality in the case of a progress point. □

Lemma 4.3. Let (V, E, λ, ρ) be a cyclic pre-proof and $\leq$ a stable, well-founded partial order. If, for every path $(v_i)_{i\in\mathbb{N}}$, there is some index $j \in \mathbb{N}$ and a $\leq$-trace $(t_i)_{i\in\mathbb{N}}$ along the corresponding suffix $(v_{i+j})_{i\in\mathbb{N}}$ with infinitely many progress points. Then, the precursor $\rightarrowtriangle$ relation is well-founded.

Proof. Suppose $(\theta_i)_{i\in\mathbb{N}}$ is an infinite sequence of precursory valuations along a path $(v_i)_{i\in\mathbb{N}}$. By assumption, there exists a $\leq$-trace $(t_i)_{i\in\mathbb{N}}$ along this path with infinitely many progress points. By Lemma 4.2, $t_{i+1}\theta_{i+1} \leq t_i\theta_i$ for all $i \in \mathbb{N}$. Furthermore, as there are infinitely many progress points, there is an infinite sub-sequence $(i_j)_{j\in\mathbb{N}}$ such that $t_{i_{j+1}}\theta_{i_{j+1}} < t_{i_j}\theta_{i_j}$. Thus, we contradicted the fact that $\leq$ is well-founded

and shown there are no such infinite sequences, i.e. precursor relation is well-founded. □

Theorem 4.4 (Global soundness)**.** Let (V, E, λ, ρ) be a cyclic proof such that, for every axiom $v \in \text{Ax}$, the associated equation $\lambda(v)$ is valid. Then, for every other node $v \in V \setminus \text{Ax}$, the associated equation $\lambda(v)$ is also valid.

Proof. We will show by well-founded induction on the necessary precursor relation that all nodes are valid. Let U be the set $\{(v, \theta) \mid v \in V, \theta \not\models \lambda(v)\}$ of nodes with a valuation that doesn't satisfy their associated equation. Suppose U is non-empty.

By Lemma 4.3, there exists a minimal element $(v_1, \theta_1) \in U$ which cannot be an axiom by assumption. However, by Theorem 4.1, there exists node $v_2 \in T$ and a valuation θ_2 that is a necessary precursor, i.e. $(v_1, \theta_1) \rightarrowtriangle (v_2, \theta_2)$, and $\theta_2 \not\models \lambda(v_2)$. In which case, (v_1, θ_1) is not a minimal element of U, and we have a contradiction. Therefore, U must be empty, and no nodes are invalid. □

Lemma 4.5. The substructural order $\trianglelefteq$ on applicative expressions, defined as follows, is well-founded and stable.

$$a \trianglelefteq b \iff \exists C[\cdot].\, C[a] = b$$

Proof. It is straightforward to show the substructural order is well-founded by induction on the structure of applicative expressions. Furthermore, suppose $C[a] = b$. It clearly follows that $C\theta[a\theta] = b\theta$ and thus $a\theta \trianglelefteq b\theta$ as required. □

F Proofs for Section 4.3 (Rewriting Induction)

Lemma 4.6. For any equation $\Gamma \vdash C[a] = b$ where $\Gamma \vdash a : d\,\overline{\tau}$ is a basic expression, there is a finite derivation tree using only instances of the (Case) rule and exactly one instance of the (Reduce) rule per branch where the leaves are labelled by equations from the set $\mathsf{Expand}(C[a] = b)$ and the root is labelled $\Gamma \vdash C[a] = b$.

Proof. Suppose $\Gamma \vdash a : d\,\overline{\tau}$ is $f\ p_1\ \cdots\ p_n$ for some $f \in \Sigma$ and patterns $p_1, \ldots, p_n$. Each expanded equation arises from the most general unifier between the patterns $p_1, \ldots, p_n$ and $p'_1\ \ldots, p'_n$ where $P(f)\ p'_1\ \ldots, p'_n \Downarrow_\emptyset$ is defined. As these unifiers are between patterns, they clearly only map variables to other patterns. Furthermore, for any closed instance of a where the variables are replaced by patterns, there is at least one incremental matching; otherwise, we would have contradicted Lemma 2.9.

It follows that we can construct a derivation tree using (Case) where the leaf of each branch corresponds to a unifier θ. As the unifiers ensure a is an instance of an incremental match, and incremental matching is closed under substitution by

Lemma 2.3, we know that $a\theta \to_P a'$ for some a'. Thus, each leaf can be extended with this reduction step as required. □

Lemma 4.7. If $\vdash (H, G)$ is a rewriting induction derivation, then there exists a cyclic pre-proof (V, E, λ, ρ) where $\lambda(\text{Ax}) \subseteq H$ and, for each goal $\Gamma \vdash a \doteq b$ in the set G, there is a node $v \in V \setminus \text{Ax}$ such that $\lambda(v) = \Gamma \vdash a \doteq b$ modulo orientation.

Proof.

- Suppose $\vdash (H, \emptyset)$ is derived from the (End) rule. Then consider the cyclic pre-proof $(\{1, \ldots, n\}, \emptyset, \beta, \lambda, \rho)$ where n is the number of hypotheses, ρ is empty, and $\lambda(i)$ is the i^{th} hypothesis. Clearly this pre-proof meets the above requirements.

- Suppose $\vdash (H, G \cup \{\Gamma \vdash a \doteq a\})$ is derived from the (Delete) rule and, by induction, there exists a cyclic pre-proof (V, E, λ, ρ) where $\lambda(\text{Ax}) \subseteq H$ and, for all goals $g \in G$, there is a node $v \in T \setminus \text{Ax}$ such that $\lambda(v) = g$ modulo orientation. Then we can construct the larger pre-proof $(V \cup \{v\}, E', \lambda', \rho')$ with an additional leaf node $v \notin V$ such that $E'(v) = \varepsilon$, $\lambda'(v) = \Gamma \vdash a \doteq a$, and $\rho'(v)$ is (Refl).

- Suppose $\vdash (H, G \cup \{\Gamma \vdash a \doteq b\})$ is derived from the (Simplify) rule with $a \to_P^* a'$ and, by induction, there exists a cyclic pre-proof (V, E, λ, ρ) where $\lambda(\text{Ax}) \subseteq H$ and, for all goals $g \in G \cup \{\Gamma \vdash a' \doteq b\}$, there is a node $v \in V \setminus \text{Ax}$ such that $\lambda(v) = g$ modulo orientation.

  In particular, there exist a node $v \in V \setminus \text{Ax}$ such that $\lambda(v) = \Gamma \vdash a' \doteq b$. Then we can construct the larger pre-proof $(V \cup \{u\}, E', \beta, \lambda', \rho')$ with an additional node $u \notin V$ with a single child v, i.e. $E'(u) = v$, such that $\lambda'(u) = \Gamma \vdash a \doteq b$ and $\rho'(u)$ is (Reduce).

- Suppose $\vdash (H, G \cup \{\Gamma \vdash C[a\theta] \doteq C[b\theta]\})$ is derived from the (Hypothesis) rule with $\Delta \vdash a = c \in H$ and, by induction, there exists a cyclic pre-proof (V, E, λ, ρ) where $\lambda(\text{Ax}) \subseteq H$ and, for all goals $g \in G \cup \{\Gamma \vdash C[c\theta] \doteq b\}$, there is a node $v \in T \setminus \text{Ax}$ such that $\lambda(v) = g$ modulo orientation.

  In particular, there exist a node $v \in V \setminus \text{Ax}$ such that $\lambda(v) = \Gamma \vdash C[c\theta] \doteq b$ and $u \in \text{Ax}$ such that $\lambda(u) = \Delta \vdash a = c$. Then we can construct the larger pre-proof $(V \cup \{w\}, E', \beta, \lambda', \rho')$ with an additional node w that has two children u and v, i.e. $E'(w) = uv$, such that $\lambda'(w) = \Gamma \vdash C[a\theta] \doteq b$ and $\rho'(w)$ is (Subst) using u as the lemma node and v as the continuation.

- Finally, suppose $\vdash (H, G \cup \{\Gamma \vdash a = b\})$ is derived from the (Expand) rule where $a > b$ and, by induction, there exists a cyclic pre-proof (V, E, λ, ρ)

where $\lambda(\text{Ax}) \subseteq H \cup \{\Gamma \vdash a = b\}$ and, for all goals $g \in \mathsf{Expand}(\Gamma \vdash a \doteq b)$, there is an axiom $u_g \in Ax$ such that $\lambda(u_g) = g$ modulo orientation.

In particular, there exists a node $v \in \text{Ax}$ such that $\lambda(v) = \Gamma \vdash a = b$ and, for each equation $g \in \mathsf{Expand}(\Gamma \vdash a \doteq b)$, there is some node $u \in V$ such that $\lambda(u) = g$. By Lemma 4.6, there is a finite derivation tree $(V', E', \lambda', \rho')$, such that $V \cap V' = \{v\} \cup \{u_g \mid g \in \mathsf{Expand}(\Gamma \vdash a \doteq b)\}$, built from only the (Case) and (Reduce) rules with each $u_g \in \text{Ax}$ as an axiom and where v is still labelled $\Gamma \vdash a = b$. By combining these two derivation trees, we can thus construct a cyclic pre-proof where v is not an axiom as required.

□

Lemma 4.8. If $\leq$ is a reduction order, then $\leq_{\text{sub}}$ is stable and well-founded.

Proof.

Reflexivity, Transitivity By definition, $\leq_{\text{sub}}$ is reflexive and transitive.

Antisymmetry This property follows immediately from well-foundedness

Well-foundedness See Definition 9 [46].

Stability As both $\leq$ and $\lhd$ are stable, it is straightforward to show by induction on the length of a derivation $a_1 \leq a_2 \lhd a_3 \cdots a_n$ that $a_1\theta \leq_{\text{sub}} a_n\theta$ for any substitution θ.

□

Lemma 4.9. For any rewriting induction derivation $\vdash (H, G)$, the pre-proof constructed in Lemma 4.7 satisfies the following invariants:

- Along a path $(v_i)_{i\in\mathbb{N}}$, let $\lambda(v_i) = \Gamma_i \vdash a_i = b_i$ be the corresponding equation. The sequence of left-hand sides $(a_i)_{i\in\mathbb{N}}$ is monotonically decreasing with respect to the substructural extension of the reduction order $\leq$.
- Within every cycle $v_1, \ldots, v_n$, there is at least one $i \leq n$ for which $\rho(v_i)$ is an instance of (Reduce) and where the trace expression has a progress point.

Proof. This lemma is witnessed by extending the construction of Lemma 4.7 with the aforementioned invariants as induction hypotheses.

- For the first invariant, we will consider the instance of each rule introduced by Lemma 4.7.
  - (Refl) There are no infinite paths as this rule has no premises.

- (Reduce) As the substructural extension of a reduction order includes the reduction order, and we have assumed $\to_P \subseteq \leq$, this rule is also monotonic.

- (Subst) During construction, the lemma used by this rule is an axiom $\Delta \vdash a = b$ replacing an instance of its left-hand side with its right-hand side. As the hypotheses are necessarily oriented in rewriting induction derivations, we have that $a > b$ by definition.

  Therefore, as $>$ is a reduction order, the path following the continuation has a decrease $C[a\theta] > C[b\theta]$ and thus $C[a\theta] >_{\text{sub}} C[b\theta]$ by extension. The path following the lemma has the trace expressions $C[a\theta]$ and $a\theta$ as the respective left-hand sides, recall the trace condition for the (Subst) rule instantiates the lemma's trace expression with the substitution used. As the substructural extension include the sub-expression relation, we have $C[a\theta] \leq_{\text{sub}} a\theta$ as required.

- (Case) Suppose case analysis is applied to the variable $x : d\,\overline{\tau}$ in the equation $\Gamma \vdash a = b$ to produce $\Gamma \cup \Delta \vdash a[k\ x_1\ \cdots\ x_n] = b[k\ x_1\ \cdots\ x_n]$. The path has trace expressions $a[k\ x_1\ \cdots\ x_n]$ and $a[k\ x_1\ \cdots\ x_n]$, recall the trace condition for the (Case) rule instantiates the conclusions trace expression with the substitution. Thus the trace along this path is still monotonic.

- Now consider the cycles of a pre-proof constructed by Lemma 4.7. These arise from instances of (Subst) where the lemma was originally an axiom but later justified using existing nodes. As the axioms correspond to hypotheses of a rewriting induction derivation, they are introduced by the (Expand) rules, which is translated into a case tree where each branch has a reduction step. In particular, all newly introduced cycles thus pass through a proper reduction step before returning to pre-existing nodes in the pre-proof. As the pre-proofs that result from Lemma 4.6 are themselves trees, and thus have no internal cycles, the global soundness condition is maintained.

□

Theorem 4.10. For any rewriting induction derivation $\vdash (\emptyset, G)$, there exists a cyclic proof (V, E, λ, ρ) where, for all goals $\Gamma \vdash a \doteq b \in G$, there is a node $v \in T \setminus \text{Ax}$ such that $\lambda(v) = \Gamma \vdash a \doteq b$ modulo orientation.

Proof. This theorem is an immediate corollary of Lemma 4.7 and Lemma 4.9. □

G Proofs for Section 4.4 (Efficient Proof Search)

Lemma 4.11. Suppose $\Gamma \vdash a : \tau$ and $b \in \mathsf{Need}(a)$, then b is a proper datatype sub-expression, i.e. $b \triangleleft a$ and $\Gamma \vdash b : d\,\overline{\tau}$, and is not an application with a constructor in head position.

Proof. Needed sub-expressions are only derived from the final case where case analysis is performed on a variable that does not have a constructor in leading position. It follows immediately that any needed sub-expression must be a proper sub-expression. Therefore, it is sufficient to show that the accumulated substitution only maps variables to datatype sub-expressions. This invariant follows straightforwardly by induction on the definition, see Lemma 2.9 for a similar property. □

Lemma 4.12. If $a \to_P^* k\ a_1\ \cdots\ a_n$, then $\mathsf{Need}(a)$ is empty.

Proof. It is straightforward to show that $\mathsf{Need}(a) \subseteq \mathsf{Need}(b)$ whenever $a \to_P^* b$ due to the close correspondence between needed sub-expressions and reduction. From which, it follows by definition that $\mathsf{Need}(a)$ is empty. □

Lemma 4.13. Let $(V,\ E,\ \lambda,\ \rho)$ be a cyclic pre-proof with nodes $v_1,\ v_2 \in V$ where $v_2 \in E(v_1)$. Then, whenever there is an edge $(x_1,\ \ell,\ x_2) \in G_{v_1,\,v_2}$, there is also a trace $x_1,\ x_2$ along this path segment. Furthermore, if labelled $\triangleright$, this trace segment has a progress point.

Proof. Let us consider the cases of $\rho(v_1)$:

- (Reduce), (Cong), (FunEx) Across these proof rules the trace condition stipulates that $t_1 \trianglerighteq t_2$. As the size-change graph has edges $(x,\ =,\ x)$ for any variable x, clearly all edges correspond to valid trace segments.
- (Case) If v_2 is the premise associated with the substitution $\{x \mapsto k\ x_1\ \cdots\ x_n\}$, then a valid trace is of the form $t_1[k\ x_1\ \cdots\ x_n/x] \trianglerighteq t_2$. The size-change graph $G_{v_1,\,v_2}$ has an edge $(x,\ \triangleright,\ x_i) \in G$ for all $i \leq n$ and $(y,\ =,\ y) \in G$ for any other variable. Clearly, both meet the criteria for a valid trace.
- (Subst) If v_2 is the continuation, then this case is analogous to that of (Reduce). Otherwise, a valid trace must be of the form $t_1 \trianglelefteq t_2\theta$. The size-change graph $G_{v_1,\,v_2}$ has an edge $(\theta(y),\ =,\ y) \in G$ just if $\theta(y)$ is a variable for some y in the type environment of the lemma's equation. Clearly, $\theta(y) \trianglelefteq \theta(y)$ by reflexivity.

□

Corollary 4.14. Let $(V,\ E,\ \lambda,\ \rho)$ be a cyclic pre-proof with a path segment $v_1,\ \ldots,\ v_n$ where $n > 1$. Then, whenever there is an edge $(x_1,\ \ell,\ x_n)$ in the composition of size-change graphs along this path segment $G_{v_1,\,v_2} \circ \cdots \circ G_{v_{n-1},\,v_n}$, there is also a trace

$x_1, \ldots, x_n$ consisting solely of variables along this path segment. Furthermore, if labelled $\triangleright$, this trace segment has a progress point.

Proof. Straightforward proof by induction on the length of the path segment. □

H Proofs for Section 4.5 (Implementation)

```
data Tm α
  = Var α
  | Cst Int
  | App (Exp α) (Exp α)

data Exp α
  = MkExp (Tm α) Int
```

```
argsE :: Exp α → [Exp α]
argsE (MkExp t n) = argsT t

argsT :: Tm α → [Exp α]
argsT (Var x) = []
argsT (Cst n) = []
argsT (App e1 e2) =
  e2 : argsE e1
```

```
mapE :: (α → β) → Exp α → Exp β
mapE f (MkExp t n) =
  MkExp (mapT f t) n

mapT :: (α → β) → Tm α → Tm β
mapT f (Var n) = Var (f n)
mapT f (Cst n) = Cst n
mapT f (App e1 e2) =
  App (mapE f e1) (mapE f e2)

mapA :: (Int → Int) → Exp α → Exp α
mapA f (MkExp t n) =
  MkExp (mapA' f t) (f n)

mapA' :: (Int → Int) → Exp α → Exp α
mapA' f (Var x) = Var x
mapA' f (Cst x) = Cst x
mapA' f (App e1 e2) =
  App (mapA f e1) (mapA f e2)

headE :: Exp α → Exp α
headE (MkExp (Var x) n) = MkExp (Var x) n
headE (MkExp (Cst x) n) = MkExp (Cst x) n
headE (MkExp (App e1 e2) n) = headE e1
```

Figure 1: Definitions for mutual induction benchmarks.

```
{-# ANN prop_1 defaultParams #-}
prop_1 :: Exp α → Formula
prop_1 e = mapE id e ≡ e

  {-# ANN prop_2 defaultParams #-}
prop_2 :: (β → γ) → (α → β) → Exp α → Formula
prop_2 f g e =
  mapE (f ∘ g) e ≡ mapE f (mapE g e)

{-# ANN prop_3 defaultParams #-}
prop_3 :: (α → β) → Exp α → Formula
prop_3 f e =
  argsE (mapE f e) ≡ map (mapE f) (argsE e)

{-# ANN prop_4 defaultParams #-}
prop_4 :: (α → β) → Tm α → Formula
prop_4 f e =
  argsT (mapT f e) ≡ map (mapE f) (argsT e)

{-# ANN prop_5 defaultParams #-}
prop_5 :: (α → β) → Exp α → Formula
prop_5 f e =
  headE (mapE f e) ≡ mapE f (headE e)
```

Figure 2: Mutual induction benchmark problems.

I Proofs for Section 5.2 (Working with Hypotheses)

Lemma 5.1. Suppose $\Gamma \vdash H$ wf and $\Gamma \vdash a : \tau$. If $H \vdash a \rightsquigarrow b$, then $\Gamma \vdash b : \tau$.

Proof. Let us proceed by case analysis on the hypothetical reduction.

- Suppose $H \vdash C[a] \rightsquigarrow C[b]$ due to a hypothesis $a = b \in H$. By induction on the context $C[\cdot]$, we have that $\Gamma,\ x : \sigma \vdash C[x] : \tau$ for some type σ such that $\Gamma \vdash a : \sigma$. As the hypotheses are well-formed, we have that $\Gamma \vdash b : \sigma$ as well. Therefore, by the substitution principle, $\Gamma \vdash C[b] : \tau$ as required.
- If hypothetical reduction is merely an instance of the program's reduction relation, preservation follows from Lemma 2.2.

□

Lemma 5.2. If $H \vdash a \rightsquigarrow^* b$, then $H \Rightarrow a = b$ is valid.

Proof. Suppose $H \vdash a \rightsquigarrow^n b$ for some $n \in \mathbb{N}$. We will show that $H \Rightarrow a = b$ by induction on $n \in \mathbb{N}$:

- In the base case, $a = b$. Therefore, $a\theta \equiv_P b\theta$ for any appropriate valuation θ by the reflexivity of contextual equivalence.
- Otherwise, suppose $H \vdash C[a] \rightsquigarrow C[c]$ and $H \vdash C[c] \rightsquigarrow^{n-1} b$. Let θ be a valuation such that $\theta \vDash H$. By induction, we have that $H \Rightarrow C[c] = b$ is valid and, therefore, $C[c]\theta \sim b\theta$. Consider the cases for $H \vdash C[a] \rightsquigarrow C[c]$:
  - Under the hypothesis $a = c \in H$, we have that $a\theta \equiv_P c\theta$ by the definition of satisfaction for conjunction. Thus, $C[a]\theta \equiv_P b\theta$ follows from the transitivity and congruence of equivalence.
  - For the second rule that is derived from incremental matching, we can clearly assume that $a \to_P^* c$. Therefore, $C[a]\theta \equiv_P c\theta$ and again $C[a\theta] \equiv_P b\theta$ by the transitivity and congruence of equivalence.

□

Lemma 5.3. Suppose $\Gamma \vdash H$ wf and $H \vdash a \rightsquigarrow^* b$. Then $H \vdash C[a\theta] \rightsquigarrow^* C[b\theta]$ for any context $C[\cdot]$ and any substitution θ such that $\mathrm{dom}(\theta) \cap \mathrm{dom}(\Gamma) = \emptyset$.

Proof. We will show that this property holds of the one-step hypothetical reduction relation, from which it is plain to see that it holds for the many-step reduction relation by induction. Let us proceed by case analysis:

- Suppose $H \vdash C[a] \rightsquigarrow C[b]$ is due to $a = b \in H$ and consider $C'[(C[a])\theta]$ where θ is such that $\mathrm{dom}(\theta) \cap \mathrm{dom}(\Gamma) = \emptyset$. As both $\mathsf{FV}(a) \subseteq \mathrm{dom}(\Gamma)$ and $\mathsf{FV}(b) \subseteq \mathrm{dom}(\Gamma)$, we have that $C'[(C[a])\theta] = C'[C\theta[a]]$ and likewise for b. Therefore, $H \vdash C'[C\theta[a]] \rightsquigarrow C'[C\theta[b]]$, which is equivalent to $H \vdash C'[(C[a])\theta] \rightsquigarrow C'[(C[b])\theta]$ as required.
- Suppose $H \vdash C[(f\ p_1\ \cdots\ p_n)\theta] \rightsquigarrow C[b\theta]$ is due to the incremental matching $P(f)\ p_1\ \cdots\ p_n \Downarrow_\emptyset b$. It immediately follows that $H \vdash C'[(C[(f\ p_1\ \cdots\ p_n)\theta])\theta'] \rightsquigarrow C'[(C[b\theta])\theta']$ is due to the incremental matching $P(f)\ p_1\ \cdots\ p_n \Downarrow_\emptyset b$ for any context $C'[\cdot]$ and substitution θ' as required.

□

Lemma 5.4 (The Critical Pairs Lemma [118])**.** Suppose every critical pair $a = b$ of a hypothesis set H is joinable, i.e. there exists some applicative expression c such that $H \vdash a \rightsquigarrow^* c$ and $H \vdash b \rightsquigarrow^* c$. Then the applicative reduction relation $H \vdash \cdot \rightsquigarrow^* \cdot$ is locally confluent. That is, if $H \vdash a \rightsquigarrow b_1$ and $H \vdash a \rightsquigarrow b_2$, then $H \vdash b_1 \rightsquigarrow^* c$ and $H \vdash b_2 \rightsquigarrow^* c$ for some applicative expression c.

Proof. We have defined critical pairs to be between hypotheses or between a hypothesis and the program. This excludes critical pairs within the program itself, however, they are necessarily joinable as its reduction relation is confluent. □

Corollary 5.5. Suppose H is a hypothesis set satisfying the pre-condition of the Critical Pairs Lemma and such that hypothetical reduction terminates, i.e. there does not exist an infinite chain of applicative expressions $H \vdash a_1 \rightsquigarrow a_2 \rightsquigarrow \cdots$, then hypothetical reduction is confluent.

Proof. Newman's Lemma dictates that locally confluent relations are necessarily globally confluent if they are terminating [118]. □

Lemma 5.6. Let $\Gamma \vdash H$ wf be a set of hypotheses and suppose there is some hypothesis $a = b \in H$ that has a type 3 critical overlap. Then a is unstable.

Proof. Suppose $C[\cdot]$, a', and θ are the context, sub-expression, and substitution according to the definition of a type 3 critical overlap. As $C[\cdot]$ is non-trivial such that $C[a']$ is of the form $f\ p_1\ \cdots\ p_n$, it can take one of two forms: either $\cdot\ p_j\ \cdots\ p_n$, or $f\ p_1\ \cdots\ p_{i-1}\ C'[\cdot]\ p_{i+1}\ \cdots\ p_n$. In the former case, we have that a' is of the form $f\ a_1\ \cdots\ a_n$ and is a function type as the context is non-trivial. By definition, applications of a function type with a program variable in head position are unstable expression. Otherwise, $C'[a']$ must be a pattern and, as a' cannot be a variable, it must be the case that a' is of the form $k\ a'_1\ \cdots\ a'_n$ for some constructor k. Therefore, as $a = a'\theta$, it also the case that a is of the form $k\ a_1\ \cdots\ a_n$, which is an unstable expression as required. □

Lemma 5.7 (Soundness)**.** Let $\langle E, R\rangle \vdash \langle E', R'\rangle$ be an inference of the hypothesis completion procedure. Then, the set of satisfying instances is preserved, i.e. the clause $E \wedge R \Rightarrow a = b$ is valid for any hypothesis $a = b \in E' \cup R'$.

Proof. We may assume, without loss of generality, that the hypothesis in question is not an element of $E \cup R$. Let us consider the cases for the entailment.

- (Delete) As $E' \subset E$ and $R' = R$, this case is trivial.
- (Simplify), (Compose), (Collapse) Let $a = b \in R \cup E$ be the equation such that $a' = b' \in R' \cup E'$ where $R \vdash a \rightsquigarrow^* a'$ and $R \vdash b \rightsquigarrow^* b'$. By Lemma 5.2, we have that the clauses $R \Rightarrow a = a'$ and $R \Rightarrow b = b'$ are valid. Furthermore, $E \Rightarrow a = b$ is valid. It follows by transitivity that, $E \wedge R \Rightarrow a' = b'$ as required.
- (Orient) Trivial.
- (Match) Let $k\ a_1\ \cdots\ a_n = k\ b_1\ \cdots\ b_n \in E$ and $a_i = b_i \in E'$ for some $k \in K$. If $\theta \vDash k\ a_1\ \cdots\ a_n = k\ b_1\ \cdots\ b_n$ for some valuation θ, then it must be the case that $\theta \vDash a_i = b_i$ by Lemma 2.11. Thus, $E \cup R \vdash a_i = b_i$ as required.
- (Fail) Not applicable.

□

Lemma 5.8 (Soundness)**.** If $\langle E,\ R\rangle \vdash \lightning$ is an inference of the hypothesis completion procedure, then there is no valuation θ such that $\theta \vDash E \wedge R$.

Proof. By inversion, there must be some equation $k\ a_1\ \cdots\ a_n = k'\ b_1\ \cdots\ b_m \in E \cup R$ where $k \neq k' \in \mathbb{K}$. It is straightforward to see that $(k\ a_1\ \cdots\ a_n)\theta \not\equiv_P (k'\ b_1\ \cdots\ b_m)\theta$ for any valuation θ. Therefore, there is no valuation satisfying these hypotheses. □

Lemma 5.9 (Correctness)**.** Suppose $\langle E,\ R\rangle$ is a terminal configuration that is well-oriented. Then R has no critical overlaps, i.e. hypothetical reduction under R is confluent and terminating.

Proof. As the configuration is well-oriented, for any equation $a = b \in R$, it is the case that $a > b$. Therefore, either $a \to_P b$ or a is a stable, normal form such that $a > b$. As this configuration is terminal, however, it must be the latter case else we could apply the (Collapse) rule.

Consider the cases for critical overlaps:

- If there were a type 1 with a hypothesis $C[a] = c \in R$, then we would again be able to apply the (Collapse) rule and derive a new equation $C[b] = c$. In which case, the configuration would not be terminal.
- A type 2 is impossible as a is in normal form.
- Finally, by Lemma 5.6, the only type 3 critical overlaps arise when a is not stable, which contradicts our assumptions.

□

Lemma 5.10. Suppose $\langle E,\ R\rangle \vdash \langle E',\ R'\rangle$ is an inference of the hypothesis completion procedure where $\langle E,\ R\rangle$ is well-oriented. Then $\langle E',\ R'\rangle$ is also well-oriented.

Proof. Let us proceed by case analysis on the hypothesis completion inference:

- (Orient) The side-condition of this inference rule requires that the only additional $a = b \in R$ is oriented so that $a > b$ as required.
- (Compose) Suppose $a = b \in R$ and $R \vdash b \rightsquigarrow^+ b'$ so that $a = b' \in R'$. It is straightforward to show by induction that $b > b'$. Therefore, by transitivity, $a > b'$ as required.
- (Delete), (Simplify), (Collapse), (Match), (Fail) Under these inference rules $R' \subseteq R$, and thus the invariant is clearly maintained.

□

Lemma 5.11. $>_{\mathtt{u/o}}$ is well-founded.

Proof. By Lemma 4.8, we know that $>_{\text{sub}}$ is well-founded. Furthermore, as $>_{\mathfrak{u}/\mathfrak{o}}$ is its length-bound lexicographical extension, so is $>_{\mathfrak{u}/\mathfrak{o}}$. □

Lemma 5.13. Suppose $\langle E,\ R\rangle$ is a well-oriented configuration and $\langle E,\ R\rangle \vdash \langle E',\ R'\rangle$ is an inference of the hypothesis completion. Then, $\langle E,\ R\rangle \gg \langle E',\ R'\rangle$.

Proof. Let us proceed by case analysis on the inference rule:

- (Delete) This rule results in the elimination of an equation and thus is a strict decrease under the multiset interpretation.
- (Orient) The multiset interpretation of configurations under this rule replaces certain unoriented equation with oriented equation, thus resulting in a strict decrease.
- (Simplify), (Compose), (Collapse) These rules results in an expression being reduced by hypothetical reduction. As R is assumed to be well-oriented, we have that the labelled pair in question is smaller under $>_{\mathfrak{u}/\mathfrak{o}}$, regardless of orientation. Therefore, the multiset interpretation of configurations has also decreased.
- (Match) In this case, an instance of equation $(k\,a_1\ \cdots\ a_n,\ \mathfrak{u})$ and $(k\,v_1\ \cdots\ b_n,\ \mathfrak{u})$ are replaced by equation $(a_i,\ \mathfrak{u})$ and $(b_i,\ \mathfrak{u})$ for all $i \leq n$. As we are using the substructural extension, we have that $k\ a_1\ \cdots\ a_n, >_{\text{sub}}\ a_i$ and $k\ b_1\ \cdots\ b_n, >_{\text{sub}} b_i$. Thus, we have a decrease in this case as well.
- (Fail) Finally, this rule holds vacuously as it doesn't result in a non-contradictory configuration.

□

Corollary 5.14. There is no infinite run of the hypothesis completion starting from a well-oriented configuration $\langle E,\ R\rangle$.

Proof. Clearly, an infinite run would require the infinite evolution of a hypothesis as hypothesis completion creates a finitely branching evolution tree. However, this would contradict the preceding lemmas. □

Lemma 5.15. Suppose $\Gamma_1 \vdash H\ \mathsf{wf}$ is a well-formed set of hypotheses and Γ_2 is a type environment that is disjoint from Γ_1, i.e. $\mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \emptyset$. If $\Gamma_2,\ H \vdash a \overset{?}{\rightsquigarrow}^*_\theta b$ is an instance of hypothetical narrowing, then $\mathrm{dom}(\theta) \subseteq \Gamma_2$ and $H \vdash a\theta \rightsquigarrow^* b$ is an instance of hypothetical reduction.

Proof. Let us proceed by induction on the many-step hypothetical narrowing relation:

- In the base case, θ is the empty substitution and, therefore, $\mathrm{dom}(\theta) \subseteq \Gamma_2$. Furthermore, we clearly have that $H \vdash a \rightsquigarrow^* a$ as required.

- Suppose $\Gamma_2,\ H \vdash a \overset{?}{\rightsquigarrow}_\theta b$ and $\Gamma_2,\ H \vdash b \overset{?}{\rightsquigarrow}^*_{\theta'} c$ where $\mathrm{dom}(\theta) \cap \mathrm{dom}(\theta') = \emptyset$. Consider the cases of the one-step hypothetical narrowing:
  - (Reduce) In this case, θ is the empty substitution, and we have that $H \vdash a \rightsquigarrow b$ by assumption. Furthermore, by induction, $\mathrm{dom}(\theta') \subseteq \Gamma_2$ and $H \vdash b\theta' \rightsquigarrow^* c$. Therefore, $H \vdash a\theta' \rightsquigarrow^* c$ by stability (See Lemma 5.3) and transitivity.
  - (Unify) In this case, θ is such that $a\theta = b \in H$ where $\mathrm{dom}(\theta) \subseteq \mathrm{dom}(\Gamma_2)$ and $H \vdash C\theta[a\theta] \rightsquigarrow C\theta[b]$ by assumption. Furthermore, by induction, $\mathrm{dom}(\theta') \subseteq \Gamma_2$ and $H \vdash (C\theta[b])\theta' \rightsquigarrow^* c$. Therefore, $\mathrm{dom}(\theta \cup \theta') \subseteq \Gamma_2$ as required. Finally, we must show that $H \vdash (C\theta[a\theta])\theta' \rightsquigarrow c$, which again follows by stability.

□

Lemma 5.16. Let $>$ be a reduction order that is compatible with the program's reduction relation. Suppose $\Gamma_1 \vdash H$ wf are well-oriented hypotheses, i.e. $a > b$ for all equations $a = b \in H$, and Γ_2 is a type environment that is disjoint from Γ_1. Then, for any hypothetical narrowing $\Gamma_2,\ H \vdash a \overset{?}{\rightsquigarrow}_\theta b$, either $\mathrm{dom}(\theta)$ is non-empty or $a > b$.

Proof. Let us proceed by case analysis on hypothetical narrowing:

- (Reduce) In this case, we clearly have that $a > b$ as required.
- (Unify) Suppose θ is empty and thus $a = b \in H$. As H is well-oriented and $>$ is stable under context, we have that $C[a] > C[b]$ as required.

□

Theorem 5.17. Suppose $\Gamma_1 \vdash H$ wf are well-oriented hypotheses, Γ_2 is a type environment that is disjoint from Γ_1, and a is an applicative expression. Then there is no infinite sequence of expressions $(a_i)_{i\in\mathbb{N}}$ and substitutions $(\theta_i)_{i\in\mathbb{N}}$ such that $\Gamma_2,\ H \vdash a_i \overset{?}{\rightsquigarrow}_{\theta_i} a_{i+1}$ for all $i \in \mathbb{N}$.

Proof. Suppose there were such sequences $(a_i)_{i\in\mathbb{N}}$ and $(\theta_i)_{i\in\mathbb{N}}$. Consider the following two cases:

- If there exists some $i \in \mathbb{N}$ such that θ_j is empty for all $j \geq i$, then the subsequence $(a_{j+i})_{i\in\mathbb{N}}$ is monotonically decreasing by Lemma 5.16. As $>$ is well-founded, we have a contradiction.
- Otherwise, there are infinitely many $i \in \mathbb{N}$ such that θ_i is non-empty. However, as $\mathrm{dom}(\theta_i) \subseteq \mathrm{dom}(\Gamma_2)$ for all i, there must be two indices n and m such that $\mathrm{dom}(\theta_n) \cap \mathrm{dom}(\theta_m) \neq \emptyset$. Without loss of generality, assume $n < m$. However, as $\mathrm{dom}(\theta_n) \cap \mathsf{FV}(a_i) = \emptyset$ for all $i > n$ and $\mathrm{dom}(\theta_m) \subseteq \mathsf{FV}(a_m)$ we have a contradiction.

For any a, therefore, there are finitely many such sequences. Hence, finitely many substitutions θ and expressions b such that $\Gamma_2,\ H \vdash a \overset{?}{\rightsquigarrow}^*_\theta b$. □

Corollary 5.18. Suppose $\Gamma_1 \vdash H$ wf are well-oriented hypotheses, Γ_2 is a type environment that is disjoint from Γ_1, and a is an applicative expression. Then there are finitely many substitutions θ and expressions b such that $\Gamma_2,\ H \vdash a \overset{?}{\rightsquigarrow}^*_\theta b$.

Proof. Clearly, for any a, there are only finitely many b and θ such that $\Gamma_2,\ H \vdash a \overset{?}{\rightsquigarrow}_\theta b$. It then follows from Theorem 5.17 that there are finitely many narrowing sequences. Hence, finitely many substitutions θ and expressions b such that $\Gamma_2,\ H \vdash a \overset{?}{\rightsquigarrow}^*_\theta b$. □

Theorem 5.19. Suppose $\Gamma_2,\ H \vdash a \overset{?}{\rightsquigarrow}^*_\theta a'$ and $\Gamma_2,\ H \vdash b\theta \overset{?}{\rightsquigarrow}^*_{\theta'} b'$ are two instances of hypothetical narrowing such that $a'\theta'\theta'' = b'\theta''$ for some substitution θ'' where $\mathrm{dom}(\theta'') \subseteq \mathrm{dom}(\Gamma_2)$. Then the composite substitution $\theta_{\mathrm{sol}} = \theta\theta'\theta''$ is a solution to the equation $a = b$ under the hypotheses H, i.e. the clause $H \Rightarrow a\theta_{\mathrm{sol}} = b\theta_{\mathrm{sol}}$ is valid.

Proof. We have that $H \vdash a\theta \rightsquigarrow^* a'$ and $H \vdash b\theta\theta' \rightsquigarrow^* b'$ by Lemma 5.15. Furthermore, by Lemma 5.3, $H \vdash a\theta_{\mathrm{sol}} \rightsquigarrow^* a'\theta'\theta''$ and $H \vdash b\theta_{\mathrm{sol}} \rightsquigarrow^* b'\theta''$. By assumption, $a'\theta'\theta''$ and $b'\theta''$ are the same expression c. By Lemma 5.2, we have that both $H \Rightarrow a\theta_{\mathrm{sol}} = c$ and $H \Rightarrow b\theta_{\mathrm{sol}} = c$ are valid. And, therefore, $H \Rightarrow a\theta_{\mathrm{sol}} = b\theta_{\mathrm{sol}}$ is valid by transitivity. □

J Proofs for Section 5.3 (Extended Proof System)

Theorem 5.20 (Local Soundness)**.** Let $v_1 \in V \setminus \mathrm{Ax}$ be a non-axiom node within the pre-proof $(V,\ E,\ \lambda,\ \rho)$ such that $\theta_1 \nVdash \lambda(v_1)$ for some valuation θ_1. Then there exists a necessary precursor $(v_2,\ \theta_2)$ such that $\theta_2 \nVdash \lambda(v_2)$.

Proof. If $v_1 \in \mathrm{dom}(\beta)$, i.e. it is a bud, then let v_2 be its companion $\beta(v_2)$. In this case, $\lambda(v_1) = \lambda(v_2)$ and so we have that $\theta_1 \nVdash \lambda(v_2)$. Furthermore, θ_1 is a necessary precursor as required.

Now suppose $v_1 \in T \setminus \mathrm{dom}(\beta)$ and consider the possible inference rules $\rho(v_1)$:

- (Refl) This case is absurd as reflexivity is valid.
- (Refute) Suppose that $\theta_1 \not\Vdash H \Rightarrow a \doteq b$, i.e. $\theta_1 \Vdash H$ and $\theta_1 \not\Vdash a \doteq b$. We have that θ_1 is a necessary precursor of the sole premise $v_1 1$ by definition and clearly $\theta_1 \not\Vdash H \Rightarrow \bot$ as required.
- (Absurd) Suppose that $\theta_1 \not\Vdash H \Rightarrow \phi$ and $\langle H,\ \emptyset\rangle \vdash^* \lightning$. By Lemma 5.8, there are no valuation that satisfy H. Thus, we have a contradiction.
- (Subst)$_\bot$ Suppose that $\theta_1 \not\Vdash H_1 \Rightarrow \phi$, i.e. $\theta_1 \Vdash H_1$ and $\theta_1 \not\Vdash \phi$, and $\lambda(v_2) = H_2 \Rightarrow \bot$ is the lemma. We have that $H_1 \Rightarrow H_2\theta$ and $\Gamma_1 \vdash \theta : (\Gamma_2 \setminus \Sigma)\Theta$ where

Γ_1 and Γ_2 are the type environments of the conclusion and lemma respectively. Therefore, $\theta \circ \theta_1 \not\Vdash H_2 \Rightarrow \bot$ as required.

- (Subst) Suppose that $\theta_1 \not\Vdash H_1 \Rightarrow C[a] \doteq b$, i.e. $\theta_1 \Vdash H_1$ and $\theta_1 \not\Vdash \phi$, and $\lambda(v_2) = H_2 \Rightarrow \bot$ is the lemma. As in the previous case, we have that $H_1 \Rightarrow H_2\theta$ and $\Gamma_1 \vdash \theta : (\Gamma_2 \setminus \Sigma)\Theta$ where Γ_1 and Γ_2 are the type environments of the conclusion and lemma respectively. Therefore, either $\theta \circ \theta_1 \Vdash a \doteq b$, in which case we have found a necessary precursor that is not satisfied as required, or $a\theta\theta_1 \equiv_P c\theta\theta_1$. Therefore, the continuation $H_1 \Rightarrow C[c\theta]\theta_1 \equiv_P b\theta_1$ cannot be satisfied by the necessary precursor θ_1 else we would have a contradiction.

- (Reduce) Suppose that $\theta_1 \not\Vdash H \Rightarrow a \doteq b$ and $\langle H, \emptyset\rangle \vdash^* R$. By Lemma 5.7 and Lemma 5.2, $H \Rightarrow a \doteq a'$ and $H \Rightarrow a \doteq b'$ are valid. Therefore, $\theta_1 \not\Vdash H \Rightarrow a' \doteq b'$ and as θ_1 is a necessary precursor we are done.

- (Cong), (FunExt) Analogous to the corresponding cases in Theorem 4.1.

- (Case) Finally, suppose $\theta_1 \not\Vdash H \Rightarrow \phi$ and a is the subject of case analysis. By Lemma 2.9, it must be the case that $\theta_1(x) \to_P k\ a_1\ \cdots\ a_n$ for some $k \in \Delta(d)$. Let v_2 be the premise associated with the constructor k. The substitution $\theta_1 \cup \{\overline{x \mapsto a}\}$ is a necessary precursor for this premise. As $\theta_1 \cup \{\overline{x \mapsto a}\} \vDash a \doteq k\ a_1\ \cdots\ a_n$, we have that $\theta_1 \cup \{\overline{x \mapsto a}\} \not\Vdash H \wedge a = k\ a_1\ \cdots\ a_n \Rightarrow \phi$ as required.

□

Lemma 5.21. Let (V, E, λ, ρ) be a cyclic pre-proof with a path $(v)_{i\in\mathbb{N}}$ and suppose $(t)_{i\in\mathbb{N}}$ is a $\leq$-trace along this path. If θ_i is a valuation of some node $v_i \in T$ and $(v_i, \theta_i) \twoheadrightarrow (v_{i+1}, \theta_{i+1})$, then $t_{i+1}\theta_{i+1} \leq t_i\theta_i$ and $t_{i+1}\theta_{i+1} < t_i\theta_i$ when i is a progress-point.

Proof. Let us consider the possible justifications for v_i:

- Suppose $\rho(v_i)$ is (Case) where $x : d\ \overline{\tau}$ is the variable upon which case analysis is performed. By Lemma 2.9 and the definition of a necessary precursor, $\theta(x) \to_P k\ c_1\ \cdots\ c_n$ for some $k \in \Delta(d)$ and v_{i+1} is the premise associated with this constructor. Furthermore, $\theta_{i+1} = \theta_i \cup \{\overline{x \mapsto c}\}$. By definition, $t_{i+1} \leq t_i[k\ x_1\ \cdots\ x_n/x]$ and thus $t_{i+1}\theta_{i+1} \leq t_i[k\ x_1\ \cdots\ x_n/x]\theta_{i+1} = t_i\theta$ as required.

- Suppose $\rho(v_i)$ is (Subst) or $(\text{Subst})_\bot$ with substitution θ and v_{i+1} is the lemma. We have that $t_{i+1}\theta \leq t_i$. The only necessary precursor to θ_i in this case is $\theta \circ \theta_i$. Thus, by stability, $t_{i+1}\theta_{i+1} \leq t_i\theta_i$ as required.

- If $\rho(v_i)$ is (FunEx), then $\theta_{i+1}(y) = \theta_i(y)$ for all y other than the fresh variable x. As t_i cannot depend on x, we have that $t_i\theta_{i+1} = t_i\theta_i$. And thus $t_{i+1}\theta_{i+1} \leq t_i\theta_i$ as required.
- In all other cases, $\theta_{i+1} = \theta_i$ and $t_{i+1} \leq t_i$. Thus, $t_{i+1}\theta_{i+1} \leq t_i\theta_i$ as required.

Furthermore, in each of the above cases, we can replace the inequality by strict inequality in the case of a progress point. □

Theorem 5.22 (Global Soundness)**.** Let $(V,\ E,\ \lambda,\ \rho)$ be a cyclic proof such that, for every axiom $v \in \mathrm{Ax}$, the associated equation $\lambda(v)$ is valid. Then, for every other node $v \in V \setminus \mathrm{Ax}$, the associated equation $\lambda(v)$ is also valid.

Proof. First, we shall show that if, for every path $(v_i)_{i\in\mathbb{N}}$, there is some $j \in \mathbb{N}$ and $\leq$-trace $(t_i)_{i\in\mathbb{N}}$ along the suffix $(v_{i+j})_{i\in\mathbb{N}}$ with infinitely many progress points, then the precursor $\rightarrowtriangle$ relation is well-founded. This result is analogous to Lemma 4.3.

Suppose $(\theta_i)_{i\in\mathbb{N}}$ is an infinite sequence of precursory valuations along a path $(v_i)_{i\in\mathbb{N}}$. By assumption, there exists a $\leq$-trace $(t_i)_{i\in\mathbb{N}}$ along this path with infinitely many progress points. By Lemma 5.21, $t_{i+1}\theta_{i+1} \leq t_i\theta_i$ for all $i \in \mathbb{N}$. Furthermore, as there are infinitely many progress points, there is an infinite sub-sequence $(i_j)_{j\in\mathbb{N}}$ such that $t_{i_{j+1}}\theta_{i_{j+1}} < t_{i_j}\theta_{i_j}$. Thus, we contradicted the fact that $\leq$ is well-founded and shown there are no such infinite sequences, i.e. precursor relation is well-founded.

With this result and Lemma 5.20, it is then straightforward to see that every node is valid by analogy to Theorem 4.4. □