Verified Functional Programming

Thèse N° 9479

Présentée le 16 août 2019

à la Faculté informatique et communications
Laboratoire d'analyse et de raisonnement automatisés
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Nicolas Charles Yves VOIROL

Acceptée sur proposition du jury

Prof. R. Guerraoui, président du jury
Prof. V. Kuncak, directeur de thèse
Prof. P. Müller, rapporteur
Dr G. Necula, rapporteur
Prof. M. Odersky, rapporteur

2019

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Acknowledgements

First and foremost, I would like to thank my advisor, Viktor Kuncak, for his guidance and support these past few years. The incredible energy and enthusiasm Viktor devotes to research were both inspiring and communicative. I am grateful for the freedom he gave me in choosing my research topics, as well as his unrelenting attention to detail which allowed my wishful tinkering to become actual research. I would also like to extend my gratitude to my thesis committee members Rachid Guerraoui, Peter Müller, George Necula and Martin Odersky for their time, their constructive questions and remarks during my thesis defense, as well as their invaluable comments on the dissertation.

I would like to thank my colleagues Romain Edelmann, Georg Schmid, Romain Ruetschi, Jad Hamza, Manos Koukoutos, Régis Blanc, Ravichandhran Kandhadai Madhavan, Etienne Kneuss, Mikaël Mayer, Andrew Reynolds, Marco Antognini, Sarah Sallinger, Eva Darulova, Philippe Suter, Andreas Pavlogiannis and Tihomir Gvero for our collaboration, constructive discussions and all-around enjoyable time spent together. I am especially grateful to Jad and Ravi who started (and accompanied) me on the long road to verification soundness. Without their persuasion, Chapter 3 would probably never have seen the light of day. I would also like to thank Yvette Gallay, Sylvie Jankow and Fabien Salvi for their administrative and technical support without which the lab would surely grind to a halt.

I thank my family and friends for their love and support during this endeavour. In particular, I would like to thank my parents for starting me on the road to science, for lending me a helping hand when I needed one along the way, and for showing me that one can do great things with hard work and dedication. I am truly lucky to have such awesome role models.

And of course, I must thank Sarah for sticking with me through thick and thin, and believing in me far more than I do myself. I owe much of this thesis to her unconditional support.

Lausanne, June 2019 N. V.

Abstract

In this thesis, we present Stainless, a verification system for an expressive subset of the Scala language. Our system is based on a dependently-typed language and an algorithmic type checking procedure which ensures total correctness. We rely on SMT solvers to automate the verification process and to provide us with useful counterexamples when considered properties are invalid. We then enable verification in the presence of high-level Scala language features by encoding them into the dependently-typed language.

We introduce an SMT-backed counterexample finding procedure which can also prove that no counterexample exists. The procedure incrementally unfolds function calls and applications in order to progressively explore the space of counterexamples. We present increasingly expressive fragments of our dependently-typed language and establish soundness and completeness properties of the procedure for each fragment. We then describe an extension which introduces support for quantifier reasoning. We discuss syntactic and semantic conditions under which the extended procedure can produce valid counterexamples in the presence of universal quantification.

We present a bidirectional type checking algorithm for our dependent type system. The algorithm relies on our counterexample finding procedure to discharge verification conditions which enables predictable and effective type checking. The type system features a unified treatment of both recursive and corecursive definitions, and further admits mutual recursion between type and function definitions. We establish normalization through a sized types approach. We define a logical relation which associates a set of reducible values to each type, and then show soundness of verification by proving that evaluation of a type checked expression terminates with a reducible value.

We further discuss a set of transformations which encode high-level Scala constructs into our dependently-typed language. These transformations allow our verification system to support object-oriented features such as traits and classes with multiple inheritance, as well as abstract and concrete methods with overriding. We further present a measure inference transformation which enables automated termination checking in our system. In addition to encoding language features, we discuss how Scala can be augmented with natural specification constructs and annotations that empower verification.

Finally, we describe the system implementation and discuss certain practical considerations. We show that the resulting system is effective by evaluating it on a set of benchmarks and case studies comprising over 15K lines of Scala code. These benchmarks showcase both the breadth and flexibility of the system.

Keywords: functional programming, formal verification, dependent type systems, higher-order functions, counterexample finding.

Résumé

Dans cette thèse, nous présentons Stainless, un système de vérification formelle pour un sous-ensemble de Scala. Notre système est basé sur une théorie des types dépendants et un algorithme de typage qui démontre la correction totale des programmes considérés. L'automatisation du processus de vérification est soutenue par un solveur SMT qui sert aussi à générer des contre-exemples lorsque la propriété considérée est invalide. La vérification de programmes Scala qui dépendent de fonctionnalités complexes du langage est rendue possible par l'encodage desdites fonctionnalités dans notre langage à types dépendants.

Tout d'abord, nous présentons une procédure de génération de contre-exemples basée sur un solveur SMT qui permet aussi de démontrer l'absence de contre-exemples. La procédure déroule progressivement les appels de fonctions (nomées ou non) de manière à explorer l'ensemble des contre-exemples. Nous présentons plusieurs fragments de notre language à types dépendants et démontrons certaines propriétés de correction et de complétude de la procédure pour chaque fragment. Nous décrivons ensuite une extension de notre théorie des types qui y introduit des quantificateurs. Nous exposons des conditions syntactiques et sémantiques en vertu desquelles la procédure étendue peut générer des contre-exemples valides malgré la présence de quantificateurs universels.

Ensuite, nous présentons un algorithme de typage bidirectionnel pour notre théorie des types dépendants. L'algorithme s'appuie sur notre procédure de génération de contre-exemples, ce qui permet un processus de vérification prévisible et efficace. Notre système traite de manière unifiée les définitions récursives et corécursives, et admet la récursion mutuelle entre les définitions de types et de fonctions. La terminaison est établie par l'approche des types indexés (sized types). Nous définissons une relation logique associant à chaque type un ensemble de valeurs réducibles, puis prouvons la correction du processus de vérification en démontrant que l'évaluation d'une expression bien typée termine avec une valeur réducible.

Afin de permettre la vérification de programmes Scala, nous exposons plusieurs transformations qui encodent certaines fonctionnalités de haut niveau de Scala dans notre langage à types dépendants. Ces transformations permettent à notre système de vérifier des programmes Scala où figurent des éléments du paradigme orienté-objet tels que des traits et classes avec de l'héritage multiple, ou encore des méthodes abstraites ou concrètes avec de la redéfinition. Nous présentons également un procédé d'inférence de mesures qui permet au système d'établir la terminaison de manière automatique. En sus des encodages, nous décrivons quelques constructions qui permettent de définir naturellement des spécifications et contrats en Scala.

Finalement, nous décrivons l'implémentation du système et exposons quelques considérations pratiques. Nous démontrons l'efficacité du système en vérifiant un ensemble de programmes cumulant plus de 15 mille lignes de code Scala qui mettent en valeur l'envergure et la flexibilité du système.

Mots-clefs : programmation fonctionnelle, vérification formelle, types dépendants, fonctions d'ordre supérieur, génération de contre-exemples.

Contents

Introduction

The amount of software on which the smooth functioning of modern society depends is steadily growing. Software is increasingly appearing in safety-critical systems such as vehicles, medical aparati, power plants, and weapons where the cost of failure can be massive. However, writing bug-free software remains a challenging task. Even in the presence of strict coding guidelines and extensive testing, critical bugs make their way into production software and lead to potentially catastrophic failures.

Given the fundamental difficulty of writing bug-free code, industry has come up with various mitigation techniques. These include improving engineering practices, introducing redundancy in production systems, designing better programming languages and type systems, automating relevant test generation, etc. One important point in this design space of code quality improvement is formal software verification. Formal verification allows users to statically verify that software systems will never crash nor diverge, and will in addition satisfy given functional correctness properties.

Formal verification provides strong guarantees regarding software reliability. However, these guarantees come at the cost of a significantly longer development time [KAE$^+$10]. Furthermore, the advances in industrial programming language design are often poorly supported by verification frameworks which typically either 1) focus on simpler fragments with good theoretical properties [BC04, NPW02, BDN09, VSJ14a], 2) rely on languages that are designed specifically for verification and have low industry adoption [Lei10, SHK$^+$16], or 3) support older programming languages [LM08, CDMV11].

One approach to reducing the time spent building verified software consists in automating the proof effort. Higher levels of automation will reduce the amount of annotation the user must provide in order for the system to prove a property, but will come at the cost of greater verification times. It is therefore important for automation techniques to fail early when given statements that do not hold. In this thesis, we will present a powerful counterexample finding procedure for a higher-order functional language that can furthermore automatically derive proofs of counterexample inexistence. The procedure is backed by a Satisfiability Modulo Theories (SMT) solver and relies on an embedding of relevant properties into a quantifier-free fragment of First-Order Logic. We leverage this procedure in our system to provide both automation and useful feedback to the user when the considered property is invalid.

In general, the inexistence of counterexamples is not a sufficient guarantee when verifying software as it does not preclude crashes or non-termination. In order to ensure these properties while allowing expressive specifications, we introduce a dependent type system with refinement types and present a bidirectional type checking algorithm which establishes both crash-freedom and normalization. In fact, type checking ensures an even stronger property, namely reducibility [Tai67] of the considered programs. The concept of reducibility allows us to extend the notion of termination to programs with higher-order functions where normalization may prove too weak.

The language for which the type checking and counterexample finding procedures are defined consists of a lambda calculus extended with recursive functions and recursive types. We further extend this fragment with support for propositional quantifiers in order to increase the expressivity of specifications and allow additional automation. The resulting verification language is expressive while remaining fairly simple and amenable to automation.

We address the second limitation of formal verification given above, namely the lack of support for modern programming languages, by presenting a verification system for an expressive subset of Scala which includes both the object-oriented and functional aspects of the language. Verification is enabled by encoding Scala programs into the verification language and then performing type checking on the encoded programs. Thanks to the expressivity of our verification language, the encoding is fairly shallow and allows both predictable and scalable verification of Scala programs. Furthermore, counterexamples derived during type checking will generally correspond to real errors at the Scala source level.

The Scala fragment supported by our verification system admits advanced language features such as implicit resolution, multiple inheritance and declaration-site variance. This allows users to write (and verify) idiomatic Scala programs. For example, consider the following covariant List definition with similar methods to the Scala standard library List collection.

```
sealed abstract class List[+T] {
  def ::[T1 >: T](h: T1): List[T1] = new ::(h, this)
  def head: T = {
    require(this != Nil)
    this match { case x :: xs ⇒ x }
  }
  ...
}
case class ::[+T](h: T, t: List[T]) extends List[T]
case object Nil extends List[Nothing]
```

Note that our implementation of the head method introduces a **require** statement [Ode10] which specifies that the list should not be Nil. This property will be statically checked at each call site in order to ensure that it is never violated. Our system will further verify that the match expression is exhaustive (and thus will not crash) which is guaranteed as the List is sealed and the Nil case is excluded by the requirement.

Let us now showcase our Scala verifier by proving that an insertion sort implementation does indeed produce a sorted list. Sorting an instance of the polymorphic type List[T] in Scala relies on an instance of Ordering[T]. This ordering provides a compare method which, given two values, returns an integer whose sign determines the ordering relation between the two values. The Scala API of the Ordering type is therefore given as follows.

```
trait Ordering[T] { def compare(x: T, y: T): Int }
```

However, this interface does not fully specify the behavior expected of Ordering instances. Indeed, the compare method must further satisfy the following constraints (where $\mathsf{sign}(x)$ is either -1, 0 or 1 depending on the sign of x) in order for the Ordering instance to be valid[1]:

inverse : $\forall\, x, y.\ \mathsf{sign}(\mathsf{compare}(x, y)) == -\mathsf{sign}(\mathsf{compare}(y, x))$,
transitive : $\forall\, x, y, z.\ \mathsf{compare}(x, y) > 0 \wedge \mathsf{compare}(y, z) > 0 \implies \mathsf{compare}(x, z) > 0$, and
consistent : $\forall\, x, y, z.\ \mathsf{compare}(x, y) == 0 \implies \mathsf{sign}(\mathsf{compare}(x, z)) == \mathsf{sign}(\mathsf{compare}(y, z))$.

Note that these constraints imply that $\mathsf{compare}(x, y) < 0$ is a total ordering over the equivalence classes given by $\mathsf{compare}(x, y) == 0$. While these requirements are only informally specified in the interface documentation, our system allows them to be made explicit. We rely on an annotation @law which marks a boolean method as being part of the interface specification. Given this annotation, we can precisely define the API of the Ordering type as follows.

```
trait Ordering[T] {
  def compare(x: T, y: T): Int

  @law def inverse(x: T, y: T): Boolean =
    sign(compare(x, y)) == −sign(compare(y, x))

  @law def transitive(x: T, y: T, z: T): Boolean =
    (compare(x, y) > 0 && compare(y, z) > 0) ⟹ (compare(x, z) > 0)

  @law def consistent(x: T, y: T, z: T): Boolean =
    (compare(x, y) == 0) ⟹ (sign(compare(x, z)) == sign(compare(y, z)))

  private final def sign(i: Int): Int = if (i > 0) 1 else if (i < 0) −1 else 0
}
```

Our system will then check that all (concrete) implementations of the Ordering trait satisfy the specified laws. We can therefore rely on these laws when verifying code that depends on some instance of the Ordering type.

Before moving on to the definition of insertion sort, let us consider the notion of list sortedness. We define sortedness through the recursive isSorted function given below which compares the

[1] See https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

adjacent list elements pairwise.

```
def isSorted[T](list: List[T])(implicit ord: Ordering[T]): Boolean = list match {
  case x :: (xs @ (y :: ys)) ⇒ ord.compare(x, y) ≤ 0 && isSorted(xs)
  case _ ⇒ true
}
```

Note that due to the transitivity of the compare method, this definition further implies that all pairs of list elements (taken in the order of appearance in the list) are sorted.

The insertion sort algorithm is then defined as follows. Note that the sort method features an **ensuring** statement which specifies a contract that the associated function must satisfy. In this particular case, we want our system to verify that the output of the sort method is indeed sorted (according to the isSorted predicate).

```
def insert[T](x: T, xs: List[T])(implicit ord: Ordering[T]): List[T] = xs match {
  case y :: ys if ord.compare(x, y) ≤ 0 ⇒ x :: xs
  case y :: ys ⇒ y :: insert(x, ys)
  case Nil ⇒ x :: Nil
}
def sort[T](list: List[T])(implicit ord: Ordering[T]): List[T] = (list match {
  case x :: xs ⇒ insert(x, sort(xs))
  case Nil ⇒ Nil
}) ensuring (isSorted(_))
```

The above implementation is correct, yet the system will fail to verify the sort contract and will timeout (or run forever). Note however that if we had introduced an error in the implementation due to which the contract did not hold, our system would report a counterexample.

In order for verification to succeed, the system needs to know that the insert function preserves sortedness, which leads to the following revised implementation of insert.

```
def insert[T](x: T, xs: List[T])(implicit ord: Ordering[T]): List[T] = {
  require(isSorted(xs))
  xs match {
    case y :: ys if ord.compare(x, y) ≤ 0 ⇒ x :: xs
    case y :: ys ⇒ ord.inverse(x, y); y :: insert(x, ys)
    case Nil ⇒ x :: Nil
  }
} ensuring (isSorted(_))
```

Our verification procedure will consider each branch of the match expression and invoke the counterexample finding procedure to derive a proof of output sortedness. Let us consider the second branch which returns y :: insert(x, ys). The postcondition of insert ensures that the tail of the list is sorted, hence it remains to show that if insert(x, ys) is non-empty, then y precedes the head of insert(x, ys) in the ordering. The system knows that

ord.compare(y, ys.head) ≤ 0	by sortedness of the xs list (if ys is non-empty),
ord.compare(y, x) > 0	by the path condition, and
ord.compare(x, y) ≤ 0	by invocation of the ord.inverse(x, y) law.

If one observes the definition of insert, it becomes clear that the head of insert(x, ys) will either be x or the head of ys. Our counterexample finding procedure unfolds function definitions and will become aware of this fact without needing an explicit specification. Our system can therefore verify that the contracts of the revised insert implementation hold.

Contributions

In this thesis, we present a verifier for an expressive subset of the Scala language. We describe the verifier from the ground up and discuss 1) SMT-based proof automation, 2) verification through algorithmic dependent type checking, and 3) encodings of complex Scala features into the verifiable dependently-typed language.

More concretely, we make the following contributions:

- We present a set of SMT-backed counterexample finding procedures for different functional languages. The procedures rely on embeddings from the considered language into a quantifier-free fragment of SMT. These embeddings under-approximate the operational semantics of certain language constructs to preserve decidability of the embedding target. In order to increase the precision of the approximation, the procedures then perform incremental unfolding of the approximated constructs. We discuss how the procedures can further determine counterexample inexistence and finally prove certain important properties about the procedures.
  - We first present counterexample finding for a first-order language with recursive functions, datatypes and parametric polymorphism. The considered language is Turing complete and features both stuck terms and divergence. We then show that the procedure is sound and complete for counterexamples, as well as sound for proofs of counterexample inexistence.
  - We extend the first-order language with support for higher-order functions by introducing lambdas, applications and function types. We further introduce a notion of structural equality which is decidable for first-class functions. We then extend the counterexample finding procedure to the higher-order setting and show that the properties stated in the first-order case are preserved.
  - We then introduce dependent types with refinements into the language and associate a denotation (or set of admissible values) to each type. We further introduce a notion of *reducibility* for counterexamples that satisfy the relevant denotations. We

  extend our embedding and unfolding procedures with support for the dependently-typed language and show that the resulting counterexample finding procedure is sound for proofs of reducible counterexample inexistence.
  - Finally, we extend the dependently-typed language with impredicative universal and existential quantifiers. We present a quantifier instantiation procedure in the context of the counterexample finder which extends the approach described in [GdM09]. We then discuss some syntactic and semantic fragments for which we conjecture soundness and completeness for counterexamples.

- We present a bidirectional type checking algorithm for the dependently-typed language without quantifiers mentioned above. The algorithm verifies that expressions evaluate to values within the denotation of the expected type, and therefore guarantees both normalization and functional correctness. The type checking procedure relies on the counterexample finding procedure to generate proofs of reducible counterexample inexistence, in particular when checking refinement types.
  - Our system relies on the well-known sized types principle to show termination of recursive functions. We present an algorithmic type generalization procedure which allows sizes to be ignored during type checking under certain conditions. This technique simplifies type checking by allowing irrelevant sizes to be omitted.
  - We introduce sized datatypes in our language which ensure that the denotation is well-formed in the presence of recursive datatype definitions. These further allow verification of corecursive (or productive) function definitions. We assign a denotation to *intersection* (or unsized) datatypes by taking the intersection over all sizes. We then provide type checking rules which enable construction and deconstruction of strictly positive intersection datatypes, thus allowing more natural programs involving recursive datatypes.
  - Finally, we present program formation rules which ensure well-formedness of datatype and function definitions. These rules allow mutual recursion between type and function definitions. We then show that our type checking procedure is sound with respect to the denotation and well-formedness relation. In order to allow mutually recursive definitions, the intermediate lemmas on which the proof relies require significant instrumentation.

- We describe a set of transformations which encode high-level Scala features into the dependently-typed language (with quantifiers) for which verification is defined. (Handling quantifiers during type checking can be left to the counterexample finding procedure.) The transformations are organized in a pipeline which progressively encodes the Scala constructs which are not supported in the dependently-typed language.
  - We present a measure and refinement inference transformation which enables automated termination checking. The transformation considers different candidate ranking functions for (unannotated) function definitions in the program and

    performs partial type checks in order to select a valid measure. The transformation further strengthens function signatures in order to allow the type checking procedure to establish termination.

  - We describe an encoding of Scala type system features including parametric nominal types with subtyping, type bounds, declaration-site variance, higher-kinded types, top and bottom types, as well as union and intersection types. We rely on dependent types and quantified propositions in order to allow a practical encoding for which type checking can be effectively performed.

- Finally, we show that the procedures presented above are effective through a series of benchmarks and case studies. We have evaluated our system on tasks including verifying correctness of datastructure definitions, proving mathematical statements, and checking algebraic laws. Our system has further been used as a backend to verify smart contract implementations.

Thesis. Automated verification and counterexample finding in the presence of dependent types is both feasible and practical. Furthermore, dependently-typed languages are sufficiently expressive to encode high-level design patterns and language constructs featured in industrial functional programming languages. These observations serve as foundations for building an effective verification system for an expressive subset of the Scala language.

1 Counterexamples for Polymorphic Recursive Functions

In this chapter, we present a first-order functional language with parametric polymorphism and describe how SMT solvers can be leveraged to produce proofs and counterexamples for properties stated in this language. The operational semantics of our language and the logical semantics of SMT formulas only differ in certain specific areas, namely in control-flow constructs and recursive function calls. We show how these expressions can be precisely encoded into a fragment of quantifier-free SMT terms, and describe an unfolding procedure that enables counterexample-complete verification for our input language.

Example. Let us consider the following program which defines a generic List type along with the recursive append function which takes two generic lists as arguments and returns their concatenation. The program further defines a lemma rightUnit which states the right unit monoid law on lists with concatenation. However, when stating the law, the programmer inadvertently inserted a mistake and ended up with an invalid definition where the right-hand side of the equality should be list.

```
type List[T] = Cons(head: T, tail: List[T]) | Nil

def append[T](l1: List[T], l2: List[T]): List[T] = l1 match {
  case Cons(x, xs) ⇒ Cons[T](x, append[T](xs, l2))
  case Nil ⇒ l2
}

def rightUnit[T](list: List[T]): Boolean = append[T](list, Nil[T]) ≈ Nil[T]
```

Let us now assume some verification procedure has generated the following verification condition which corresponds to the inductive case of the right unit law.

$$\mathsf{rightUnit}[T](\mathsf{xs}) \implies \mathsf{rightUnit}[T](\mathsf{Cons}[T](\mathsf{x},\mathsf{xs}))$$

$$
\begin{array}{rcl}
\mathit{fdef} & ::= & \textbf{def}\ \mathit{id}[\ \mathit{tdecls}\ ](\ \mathit{id} : \mathit{type}\ ) : \mathit{type} := \mathit{expr} \\
\mathit{tdef} & ::= & \textbf{type}\ \mathit{id}[\ \mathit{tdecls}\ ] := \mathit{id}(\ \mathit{id} : \mathit{type}\ )\ \langle \mid \mathit{id}(\ \mathit{id} : \mathit{type}\ )\ \rangle^{*} \\
\mathit{tdecls} & ::= & \epsilon \mid \mathit{id}\ \langle\ ,\ \mathit{id}\ \rangle^{*} \\
\mathit{tparams} & ::= & \epsilon \mid \mathit{type}\ \langle\ ,\ \mathit{type}\ \rangle^{*} \\
\mathit{expr} & ::= & \textbf{true} \mid \textbf{false} \mid () \mid \mathit{id} \mid \mathit{expr} \approx \mathit{expr} \mid \mathsf{err}[\mathit{type}] \\
 & & \mid \mathit{expr}\ \textbf{match}\ \{\ \langle\ \mathit{id}(\ \mathit{id}\ ) \Rightarrow \mathit{expr}\ \rangle + \} \\
 & & \mid (\ \mathit{expr},\ \mathit{expr}\ ) \mid \pi_i(\mathit{expr}) \\
 & & \mid \textbf{if}\ (\ \mathit{expr}\ )\ \mathit{expr}\ \textbf{else}\ \mathit{expr} \\
 & & \mid \textbf{let}\ \mathit{id} := \mathit{expr}\ \textbf{in}\ \mathit{expr} \\
 & & \mid \mathit{id}[\ \mathit{tparams}\ ](\ \mathit{expr}\ ) \\
\mathit{type} & ::= & \mathsf{Boolean} \mid \mathsf{Unit} \mid \mathit{id} \mid \mathit{id}[\ \mathit{tparams}\ ] \mid (\ \mathit{type},\ \mathit{type}\ ) \\
\mathit{id} & ::= & \mathsf{IDENT}
\end{array}
$$

Figure 1.1 – Syntax of our simple first-order language.

In the remainder of this chapter, we will describe how to build a counterexample-complete procedure that is able to discharge such verification conditions, using this property as a running example. Note that instead of directly showing that the property holds for all inputs x, xs, our procedure will instead either 1) find a counterexample if one exists, or 2) show that no such counterexample exists.

1.1 Language

We will start by considering a simple first-order language with recursive functions, pairs, algebraic datatypes, equality and parametric polymorphism whose syntax is defined in Figure 1.1. We use the syntax $\overline{a}$ to denote a sequence $a_1, \cdots, a_n$ of elements. We denote substitution of x by b in a by $a[x/b]$. We write $\mathcal{C}[\cdot]$ to denote an expression with a hole such that $\mathcal{C}[e]$ is an expression where the hole was filled by e. We write $e_1 \sqsubseteq e_2$ to indicate structural inclusion of the expression e_1 in e_2 (we extend this notation to types as well). In order to improve readability, we will sometimes rely on a more Scala-like syntax (restricted to features supported by our language) when presenting code snippets and examples.

We define a *program* $P \subseteq \mathit{fdef} \cup \mathit{tdef}$ as a set of function and type definitions. The various judgements presented in Figures 1.3, 1.4 and 1.5 are given in the context of some combination of P a program, Θ a set of type variables $T_1, \cdots, T_n$, and Γ a sequence of type bindings $x_1 : \tau_1, \cdots, x_m : \tau_m$. We call the full triplet $P;\Theta;\Gamma$ a *typing environment* and refer to the sequence of type bindings Γ as a *typing context*. The context formation judgement $\vdash \Gamma$ *context* presented in Figure 1.2 depends on P and Θ, the type formation judgement $\vdash \tau$ *type* presented in Figure 1.3 also depends on P and Θ, the typing judgement $\vdash e : \tau$ presented in Figure 1.4

$$\textsc{Empty Context}\quad \dfrac{}{P;\Theta \vdash \ \mathit{context}} \qquad\qquad \textsc{Increase Context}\quad \dfrac{P;\Theta \vdash \tau\ \mathit{type}}{P;\Theta \vdash \Gamma, x : \tau\ \mathit{context}}$$

Figure 1.2 – Context formation rules.

depends on P, Θ and Γ, and finally the definition formation judgement $\vdash \mathsf{d}$ *well-formed* presented in Figure 1.5 depends only on P. When the environment under which the various judgements are performed is clear, given the context, it may be omitted (we simply write $e : \tau$, for example). We generally assume the programs considered are well-formed according to the rules presented in Figure 1.5, and typing environments $P;\Theta;\Gamma$ consist of a well-formed program P and well-formed typing context Γ (note that Θ is always well-formed).

The set of values of our language is defined with respect to some program P and is given by the following grammar where C must correspond to some datatype constructor in P.

$$\mathit{value} \quad ::= \quad \textbf{true} \mid \textbf{false} \mid () \mid (\mathit{value}, \mathit{value}) \mid C[\overline{\tau}](\mathit{value})$$

The call-by-value operational semantics of closed terms are defined in Figure 1.6 and are also given in the context of a (generally omitted) program P. We use *evaluation contexts* $\mathcal{E}$ as an expression with a hole that determines where evaluation will occur next. The expression $\mathcal{E}[e]$ is obtained by substituting a type-compatible expression into the hole. Note that our type system does not enforce progress as the well-typed $\mathsf{err}[\tau]$ expression is stuck. However, we do have preservation. Our type system therefore only imposes a certain *structure* on the programs considered which allows them to be embedded into well-sorted SMT terms.

To each expression and type in our language, we associate a unique static label which we will refer to as $l : e$. During evaluation, labels follow the expression to which they were originally assigned, so an expression in a trace may end up having multiple labels. In the CALL rule, the labels in the inlined function body are computed as a function of the original labels assigned to expressions in the function definition and the label of the function call.

We let $FV(e)$ denote the set of free variables in e and $FT(e)$ the set of free types (*i.e.* type variables). We define *value substitutions* as mappings $\gamma : \mathit{id} \mapsto \mathit{value}$ and *type substitutions* as mappings $\theta : \mathit{id} \mapsto \{\tau \in \mathit{type} \mid FT(\tau) = \emptyset\}$. We consider all variables and identifiers to be fresh in all expressions, and substitutions distribute over sub-expressions. Given a well-formed typing environment $P;\Theta;\Gamma$, an expression e and a type τ such that $P;\Theta;\Gamma \vdash e : \tau$, we will generally be interested in triplets of a program extension P_{in}, type substitution θ, and value substitution γ such that for $d \in P_{in}$, we have $P \cup P_{in} \vdash d$ *well-formed*, for $T \in \Theta$, we have $(T,\tau) \in \theta$ and $P \cup P_{in};\emptyset \vdash \tau$ *type*, and for $(x,\tau) \in \Gamma$, we have $(x, v) \in \gamma$ and $P \cup P_{in};\emptyset;\emptyset \vdash v : \theta(\tau)$. We say that P_{in}, θ, γ are *inputs* for $P;\Theta;\Gamma$, and preservation ensures that $P \cup P_{in};\emptyset;\emptyset \vdash \gamma(\theta(e)) : \theta(\tau)$.

$$\textsc{Boolean Type}\quad P;\Theta \vdash \mathsf{Boolean}\ \mathit{type}$$

$$\textsc{Unit Type}\quad P;\Theta \vdash \mathsf{Unit}\ \mathit{type}$$

$$\textsc{Pair Type}\quad \frac{P;\Theta \vdash \tau_1\ \mathit{type} \qquad P;\Theta \vdash \tau_2\ \mathit{type}}{P;\Theta \vdash (\tau_1, \tau_2)\ \mathit{type}}$$

$$\textsc{Type Variable}\quad \frac{T \in \Theta}{P;\Theta \vdash T\ \mathit{type}}$$

$$\textsc{Datatype}\quad \frac{(\textbf{type}\ d[\overline{\tau}_d] := \cdots) \in P \qquad |\overline{\tau}_d| = |\overline{\tau}| \qquad P;\Theta \vdash \tau\ \mathit{type}\ \text{for}\ \tau \in \overline{\tau}}{P;\Theta \vdash d[\overline{\tau}]\ \mathit{type}}$$

Figure 1.3 – Type formation rules.

$$\textsc{True}\quad \frac{P;\Theta \vdash \Gamma\ \mathit{context}}{P;\Theta;\Gamma \vdash \textbf{true} : \mathsf{Boolean}}$$

$$\textsc{False}\quad \frac{P;\Theta \vdash \Gamma\ \mathit{context}}{P;\Theta;\Gamma \vdash \textbf{false} : \mathsf{Boolean}}$$

$$\textsc{Unit}\quad \frac{P;\Theta \vdash \Gamma\ \mathit{context}}{P;\Theta;\Gamma \vdash () : \mathsf{Unit}}$$

$$\textsc{Var}\quad \frac{P;\Theta \vdash \Gamma\ \mathit{context} \qquad (x, \tau) \in \Gamma}{P;\Theta;\Gamma \vdash x : \tau}$$

$$\textsc{Equals}\quad \frac{P;\Theta;\Gamma \vdash e_1 : \tau \qquad P;\Theta;\Gamma \vdash e_2 : \tau}{P;\Theta;\Gamma \vdash e_1 \approx e_2 : \mathsf{Boolean}}$$

$$\textsc{Let}\quad \frac{P;\Theta;\Gamma \vdash e_1 : \tau_1 \qquad P;\Theta;\Gamma, x : \tau_1 \vdash e_2 : \tau_2}{P;\Theta;\Gamma \vdash \textbf{let}\ x := e_1\ \textbf{in}\ e_2 : \tau_2}$$

$$\textsc{Err}\quad \frac{P;\Theta \vdash \Gamma\ \mathit{context} \qquad P;\Theta \vdash \tau\ \mathit{type}}{P;\Theta;\Gamma \vdash \mathsf{err}[\tau] : \tau}$$

$$\textsc{If}\quad \frac{P;\Theta;\Gamma \vdash c : \mathsf{Boolean} \qquad P;\Theta;\Gamma \vdash e_1 : \tau \qquad P;\Theta;\Gamma \vdash e_2 : \tau}{P;\Theta;\Gamma \vdash \textbf{if}\ (c)\ e_1\ \textbf{else}\ e_2 : \tau}$$

$$\textsc{Pair}\quad \frac{P;\Theta;\Gamma \vdash e_1 : \tau_1 \qquad P;\Theta;\Gamma \vdash e_2 : \tau_2}{P;\Theta;\Gamma \vdash (e_1, e_2) : (\tau_1, \tau_2)}$$

$$\textsc{Projection}\quad \frac{P;\Theta;\Gamma \vdash e : (\tau_1, \tau_2) \qquad 1 \le i \le 2}{P;\Theta;\Gamma \vdash \pi_i(e) : \tau_i}$$

$$\textsc{Constructor}\quad \frac{(\textbf{type}\ d[\overline{\tau}_d] := \cdots \mid C_i(x_i : \tau_i) \mid \cdots) \in P \qquad P;\Theta \vdash d[\overline{\tau}]\ \mathit{type} \qquad P;\Theta;\Gamma \vdash e : \tau_i[\overline{\tau}_d / \overline{\tau}]}{P;\Theta;\Gamma \vdash C_i[\overline{\tau}](e) : d[\overline{\tau}]}$$

$$\textsc{Match}\quad \frac{\begin{array}{c}(\textbf{type}\ d[\overline{\tau}_d] := C_1(x_1 : \tau_1) \mid \cdots \mid C_n(x_n : \tau_n)) \in P \\ P;\Theta;\Gamma \vdash e : d[\overline{\tau}] \qquad P;\Theta;\Gamma, y_i : \tau_i[\overline{\tau}_d / \overline{\tau}] \vdash e_i : \tau\ \text{for}\ 1 \le i \le n\end{array}}{P;\Theta;\Gamma \vdash e\ \textbf{match}\ \{\ C_1(y_1) \Rightarrow e_1 \cdots C_n(y_n) \Rightarrow e_n\ \} : \tau}$$

$$\textsc{Call}\quad \frac{\begin{array}{c}(\textbf{def}\ f[\overline{\tau}_f](x : \tau_1) : \tau_2 := e) \in P \\ |\overline{\tau}_f| = |\overline{\tau}| \qquad P;\Theta \vdash \tau\ \mathit{type}\ \text{for}\ \tau \in \overline{\tau} \qquad P;\Theta;\Gamma \vdash e : \tau_1[\overline{\tau}_f / \overline{\tau}]\end{array}}{P;\Theta;\Gamma \vdash f[\overline{\tau}](e) : \tau_2[\overline{\tau}_f / \overline{\tau}]}$$

Figure 1.4 – Typing rules of our simple first-order language.

$$\textsc{Type}\quad \frac{P;\overline{\tau}_d \vdash \tau_i\ \textit{type}\ \text{for}\ 1 \le i \le n \qquad P;\overline{\tau}_d \vdash d[\overline{\tau}_d]\ \textit{well-defined}}{P \vdash \textbf{type}\ \ d[\overline{\tau}_d]\ :=\ C_1(x_1 : \tau_1)\ \mid \cdots \mid C_n(x_n : \tau_n)\ \textit{well-formed}}$$

$$\textsc{Function}\quad \frac{P;\overline{\tau}_f \vdash \tau_1\ \textit{type} \qquad P;\overline{\tau}_f \vdash \tau_2\ \textit{type} \qquad P;\overline{\tau}_f; x : \tau_1 \vdash e : \tau_2}{P \vdash \textbf{def}\ \ f[\overline{\tau}_f](x : \tau_1) : \tau_2\ :=\ e\ \textit{well-formed}}$$

Figure 1.5 – Program formation rules. The *well-defined* judgement used in the TYPE rule follows from the definition of well-defined datatypes in the SMT theory of datatypes and ensures that a corresponding SMT datatype sort exists for type $d[\overline{\tau}]$.

$$\begin{aligned}\mathcal{E}\ \ ::=\ \ & [\cdot] \mid \textbf{if}\,(\mathcal{E})\ \textit{expr}\ \textbf{else}\ \textit{expr} \mid \mathcal{E} \approx \textit{expr} \mid \textit{value} \approx \mathcal{E} \mid (\mathcal{E}, \textit{expr}) \mid (\textit{value}, \mathcal{E}) \\ & \mid \pi_i(\mathcal{E}) \mid \textbf{let}\ \textit{id} := \mathcal{E}\ \textbf{in}\ \textit{expr} \mid \textit{id}[\overline{\tau}](\mathcal{E}) \mid \mathcal{E}\ \textbf{match}\,\{\cdots\ \textit{id}(\textit{id})\ \Rightarrow\ \textit{expr} \cdots\}\end{aligned}$$

$$\textsc{Context}\quad \frac{e \to e'}{\mathcal{E}[e] \to \mathcal{E}[e']} \qquad \textsc{If-Then}\quad \textbf{if}\,(\textbf{true})\ e_1\ \textbf{else}\ e_2 \to e_1 \qquad \textsc{If-Else}\quad \textbf{if}\,(\textbf{false})\ e_1\ \textbf{else}\ e_2 \to e_2$$

$$\textsc{Let}\quad \frac{v \in \textit{value}}{\textbf{let}\ x\ :=\ v\ \textbf{in}\ e_2 \to e_2[x/v]} \qquad \textsc{Equals-True}\quad \frac{v_1, v_2 \in \textit{value} \qquad v_1 = v_2}{v_1 \approx v_2 \to \textbf{true}} \qquad \textsc{Equals-False}\quad \frac{v_1, v_2 \in \textit{value} \qquad v_1 \neq v_2}{v_1 \approx v_2 \to \textbf{false}}$$

$$\textsc{Project}\quad \frac{v_1, v_2 \in \textit{value} \qquad 1 \le i \le 2}{\pi_i((v_1, v_2)) \to v_i} \qquad \textsc{Call}\quad \frac{(\textbf{def}\ \ f[\overline{\tau}_f](x : \tau_1) : \tau_2\ :=\ e_f) \in P \qquad v \in \textit{value}}{f[\overline{\tau}](v) \to e_f[\overline{\tau}_f/\overline{\tau}][x/v]}$$

$$\textsc{Match}\quad \frac{(\textbf{type}\ \ d[\overline{\tau}_d]\ :=\ \cdots \mid C_i(x_i : \tau_i) \mid \cdots) \in P \qquad v \in \textit{value}}{C_i[\overline{\tau}](v)\ \textbf{match}\,\{\cdots\ C_i(y_i)\ \Rightarrow\ e_i\ \cdots\} \to e_i[y_i/v]}$$

Figure 1.6 – Operational semantics of our simple first-order language, given with respect to some program P (see CALL and MATCH rules).

1.2 Embedding the Language

We discuss in this section how our language can be embedded into a decidable fragment of SMT formulas. The fragment on which we rely consists of the usual boolean terms and operators, the theory of uninterpreted functions, the theory of uninterpreted sorts, and the theory of algebraic datatypes. The main challenge here consists in lining up the operational semantics of our language with the logical semantics of SMT. Most of our expressions have straightforward embeddings into SMT terms (for example, datatype constructors, boolean values and variables); however, our named function call semantics differ from those of uninterpreted functions. The precise encoding of our operational semantics is enabled by two key features of our embedding:

1. The resulting SMT formulas are instrumented so that portions of the embedding for which the semantics do not (yet) line up can be disregarded. This aspect is enabled by the embedding of *if*- and *match*-expressions where the control flow of the expression is encoded into special *blocker* instrumentation constant.
2. Function calls are gradually unfolded to increase the set of calls for which the embedding is consistent with the operational semantics of our language.

Expressions from the language presented in Figure 1.1 are embedded into a fragment of quantifier-free SMT terms. We define an embedding relation $\triangleright$ for our types and expressions. Given a type formation context $P;\Theta$ and a well-formed type τ, our type embedding returns an SMT sort σ. Notation wise, we write this as $P;\Theta \vdash \tau \triangleright \sigma$. Our $\mathsf{Boolean}$ type is naturally embedded into its SMT sort counterpart. Datatypes are embedded into the SMT theory of datatypes where the sort is uniquely determined by the type $d[\overline{\tau}]$. Pair types are embedded into SMT datatypes with a single constructor and a field for each projector. For type variables, we use fresh uninterpreted sorts (again uniquely determined by the type variable). The complete type embedding rules are presented in Figure 1.7.

Our expression embedding relies on a boolean-sorted instrumentation constant which encodes the path condition of the expression being embedded. We call these instrumentation constants *blocker constants*. Given a typing context $P;\Theta;\Gamma$, blocker constant b, and well-typed expression e, our embedding returns an SMT term t and a set of SMT clauses Φ under which the term corresponds to the input expression. Notation wise, we write this as $P;\Theta;\Gamma \vdash (b, e) \triangleright (t, \Phi)$. Furthermore, the embedding is such that given $P;\Theta;\Gamma \vdash e : \tau$, the type embedding $P;\Theta \vdash \tau \triangleright \sigma$ corresponds to the sort of t. Some of our expressions have natural embeddings, such as variables and algebraic datatype constructors, but others rely on the clause set and blocker constants to ensure equivalent semantics. The embeddings of *if*- and *match*-expressions introduce blocker constants for each branch, and the embedding of error expressions negates the associated blocker constant (which disallows the relevant branch). Similarly to the embedding of algebraic datatypes, function calls are embedded by introducing uninterpreted function symbols that are uniquely determined by the function identifier and its

$$\dfrac{\textsc{Boolean Type}}{P;\Theta \vdash \mathsf{Boolean} \rhd \mathit{bool}}$$

$$\textsc{Unit Type}\quad \dfrac{\textbf{datatype}\ \delta_{\mathsf{Unit}}\ =\ \mathit{Unit}\ \text{algebraic datatype}}{P;\Theta \vdash \mathsf{Unit} \rhd \delta_{\mathsf{Unit}}} \qquad \textsc{Type Variable}\quad \dfrac{T \in \Theta \qquad \sigma_T\ \text{uninterpreted sort}}{P;\Theta \vdash T \rhd \sigma_T}$$

$$\textsc{Pair Type}\quad \dfrac{P;\Theta \vdash \tau_1 \rhd \sigma_1 \qquad P;\Theta \vdash \tau_2 \rhd \sigma_2 \\ \textbf{datatype}\ \delta_{(\tau_1,\tau_2)}\ =\ \mathit{cons}_{(\tau_1,\tau_n)}(\pi_{(\tau_1,\tau_2),1} : \sigma_1, \pi_{(\tau_1,\tau_2),2} : \sigma_2)\ \text{algebraic datatype}}{P;\Theta \vdash (\tau_1, \tau_2) \rhd \delta_{(\tau_1,\tau_2)}}$$

$$\textsc{Datatype}\quad \dfrac{(\textbf{type}\ \ d[\overline{\tau}_d]\ :=\ C_1(x_1 : \tau_1)\ \mid \cdots \mid\ C_n(x_n : \tau_n)) \in P \qquad P;\Theta \vdash \tau_i[\overline{\tau}_d/\overline{\tau}] \rhd \sigma_i\ \text{ for }\ 1 \le i \le n \\ \textbf{datatype}\ d_{\overline{\tau}}\ =\ C_{\overline{\tau},1}(x_{\overline{\tau},1} : \sigma_1)\ \mid \cdots \mid\ C_{\overline{\tau},n}(x_{\overline{\tau},n} : \sigma_n)\ \text{algebraic datatype}}{P;\Theta \vdash d[\overline{\tau}] \rhd d_{\overline{\tau}}}$$

Figure 1.7 – Type embedding rules.

type parameters. The complete expression embedding rules are presented in Figure 1.8. Note that in order to improve readability, we share certain identifiers between the initial expression and the embedded SMT term.

Similarly to the typing judgement, the embedding environment $P;\Theta;\Gamma$ is generally clear from the context and we write $(b, e) \rhd (t_e, \Phi_e)$. Given some embedding $(b, e) \rhd (t_e, \Phi_e)$, we will often want to discuss the embedding of sub-expressions of e that occurred during the embedding of e. Given some $l' : e' \sqsubseteq e$, we write $\llbracket l' : e' \rrbracket_t$ for the term resulting from the embedding of $l' : e'$, and $\llbracket l' : e' \rrbracket_b$ for the blocker constant under which it was embedded. It is clear given the definition of $\rhd$ that $\llbracket\cdot\rrbracket_t$ and $\llbracket\cdot\rrbracket_b$ are defined for all sub-expressions of e. It is easy to see that for values, the set of clauses obtained by embedding is empty. One should also note that the embeddings of values do not depend on the provided blocker constant. We will therefore generally omit the blocker constant and clause set when discussing the embeddings of values and simply write $v \rhd t_v$ for $v \in \mathit{value}$.

Recalling the right unit law for append stated previously, the embedding of the property is as follows. Note that the implication is considered as an *if*-expression in the presented language.

$$\begin{aligned}
&P; T; \mathsf{x} : T, \mathsf{xs} : \mathsf{List}[T] \vdash (b_e, \mathsf{rightUnit}[T](\mathsf{xs}) \implies \mathsf{rightUnit}[T](\mathsf{Cons}[T](\mathsf{x}, \mathsf{xs}))) \rhd \\
&\quad (r_1, \{\, (b_e \land \mathsf{rightUnit}_T(\mathsf{xs})) \iff b_1, \\
&\qquad (b_e \land \neg\mathsf{rightUnit}_T(\mathsf{xs})) \iff b_2, \\
&\qquad b_1 \implies r_1 \simeq \mathsf{rightUnit}_T(\mathsf{Cons}_T(\mathsf{x}, \mathsf{xs})), \\
&\qquad b_2 \implies r_1 \simeq \mathit{false}\,\})
\end{aligned}$$

$$P;\Theta;\Gamma \vdash (b, \textbf{true}) \rhd (\textit{true}, \emptyset) \qquad P;\Theta;\Gamma \vdash (b, \textbf{false}) \rhd (\textit{false}, \emptyset) \qquad P;\Theta;\Gamma \vdash (b, ()) \rhd (\textit{Unit}, \emptyset)$$

$$\frac{(x,\tau) \in \Gamma}{P;\Theta;\Gamma \vdash (b, x) \rhd (x, \emptyset)} \qquad \frac{P;\Theta;\Gamma \vdash (b, e_1) \rhd (t_1, \Phi_1) \qquad P;\Theta;\Gamma \vdash (b, e_2) \rhd (t_2, \Phi_2)}{P;\Theta;\Gamma \vdash (b, e_1 \approx e_2) \rhd (t_1 \simeq t_2, \Phi_1 \cup \Phi_2)}$$

$$\frac{P;\Theta;\Gamma \vdash (b, e_1) \rhd (t_1, \Phi_1) \qquad P;\Theta;\Gamma \vdash e_1 : \tau \qquad P;\Theta;\Gamma, x : \tau \vdash (b, e_2) \rhd (t_2, \Phi_2)}{P;\Theta;\Gamma \vdash (b, \textbf{let } x := e_1 \textbf{ in } e_2) \rhd (t_2, \Phi_1 \cup \Phi_2 \cup \{b \implies x \simeq t_1\})}$$

$$\frac{r \text{ fresh constant}}{P;\Theta;\Gamma \vdash (b, \mathsf{err}[\tau]) \rhd (r, \{\neg b\})} \qquad \frac{P;\Theta;\Gamma \vdash e : (\tau_1, \tau_2) \qquad P;\Theta;\Gamma \vdash (b, e) \rhd (t, \Phi)}{P;\Theta;\Gamma \vdash (b, \pi_i(e)) \rhd (\pi_{(\tau_1,\tau_2),i}(t), \Phi)}$$

$$\frac{P;\Theta;\Gamma \vdash e_i : \tau_i \text{ for } 1 \le i \le 2 \qquad P;\Theta;\Gamma \vdash (b, e_i) \rhd (t_i, \Phi_i) \text{ for } 1 \le i \le 2}{P;\Theta;\Gamma \vdash (b, (e_1, e_2)) \rhd (\textit{cons}_{(\tau_1,\tau_2)}(t_1, t_2), \Phi_1 \cup \Phi_2)}$$

$$\frac{P;\Theta;\Gamma \vdash (b, e) \rhd (t, \Phi)}{P;\Theta;\Gamma \vdash (b, f[\overline{\tau}](e)) \rhd (f_{\overline{\tau}}(t), \Phi)} \qquad \frac{P;\Theta;\Gamma \vdash (b, e) \rhd (t, \Phi)}{P;\Theta;\Gamma \vdash (b, C_i[\overline{\tau}](e)) \rhd (C_{\overline{\tau},i}(t), \Phi)}$$

$$\frac{\begin{array}{c} P;\Theta;\Gamma \vdash (b, c) \rhd (t_c, \Phi_c) \\ b_1, b_2, r \text{ fresh constants} \qquad P;\Theta;\Gamma \vdash (b_1, e_1) \rhd (t_1, \Phi_1) \qquad P;\Theta;\Gamma \vdash (b_2, e_2) \rhd (t_2, \Phi_2) \\ \Phi_{guard} = \{(b \land t_c) \iff b_1, (b \land \neg t_c) \iff b_2, b_1 \implies r \simeq t_1, b_2 \implies r \simeq t_2\} \end{array}}{P;\Theta;\Gamma \vdash (b, \textbf{if } (c)\ e_1 \textbf{ else } e_2) \rhd (r, \Phi_c \cup \Phi_1 \cup \Phi_2 \cup \Phi_{guard})}$$

$$\frac{\begin{array}{c} P;\Theta;\Gamma \vdash (b, s) \rhd (t_s, \Phi_s) \\ P;\Theta;\Gamma \vdash s : d[\overline{\tau}] \qquad (\textbf{type } d[\overline{\tau}_d] := C_1(x_1 : \tau_1) \mid \cdots \mid C_n(x_n : \tau_n)) \in P \\ b_1, \cdots, b_n, r \text{ fresh constants} \qquad P;\Theta;\Gamma, y_i : \tau_i[\overline{\tau}_d / \overline{\tau}] \vdash (b_i, e_i) \rhd (t_i, \Phi_i) \text{ for } 1 \le i \le n \\ \Phi_{guard} = \bigcup_{1 \le i \le n} \{(b \land \textit{is-}C_{\overline{\tau},i}(t_s)) \iff b_i, b_i \implies y_i \simeq x_{\overline{\tau},i}(t_s), b_i \implies r \simeq t_i\} \end{array}}{P;\Theta;\Gamma \vdash (b, s \textbf{ match } \{\, C_1(y_1) \Rightarrow e_1 \cdots C_n(y_n) \Rightarrow e_n \,\}) \rhd (r, \Phi_s \cup \Phi_1 \cup \cdots \cup \Phi_n \cup \Phi_{guard})}$$

Figure 1.8 – Expression embedding rules. It is important to note that in the rules where identifiers from the expression are used directly in the embedded terms to improve readability (*e.g.* variable rule), the type embedding is leveraged to produce well-sorted terms. The same goes for fresh constants that are introduced, for example in the error expression rule.

An important property of our embedding is that all clauses introduced into the Φ_e set are of the shape $b \implies c$ for some blocker constant b. Hence, by negating the blocker constants, the clause set will always become satisfiable. In the following, given a model M such that $M \models \neg b$, we will assume that M does not require interpretations for constants and function symbols that occur in c to satisfy $b \implies c$. We also assume that models are *minimal* and contain no spurious interpretations.

Lemma 1. *For embedding $P;\Theta;\Gamma \vdash (b_e, e) \rhd (t_e, \Phi_e)$ and model M, if $M \models \neg \llbracket e' \rrbracket_b$ for $e' \sqsubseteq e$, then $M \models \Phi_e$.*

Proof. The proof follows by induction on e. $\square$

Finally, we want some means of extracting SMT models. Given a model M, we want to extract some inputs to the original expression such that it can be evaluated. As the expression may contain free variables for which no value exists (namely if they require values of parametric type), we extract a program extension P_{in} that will contain type definitions corresponding to the type variables in e. Based on these definitions, we construct a type substitution θ from type variables in Θ to concrete types. We then extract a value substitution γ from variables in Γ to values. We therefore want to define an extraction relation $\lhd$ such that $P;\Theta;\Gamma \vdash (P_{in}, \theta, \gamma) \lhd M$.

We start by considering the extraction of type variables in Θ. Note that SMT solvers can produce interpretations for constants of uninterpreted sort, and given a term with uninterpreted sort σ_T, we let its interpretation have the shape $T_i \in id$ for some $i \in \mathbb{N}_+$. Given a type variable $T \in \Theta$ with uninterpreted sort embedding σ_T, we let $M(\sigma_T) = \{T_1, \cdots, T_n\}$ be the set of values terms with sort σ_T that exist in the interpretations of M. We then define an extraction relation $\lhd$ between type variable $T \in \Theta$ and the extracted type definition.

$$\frac{P;\Theta \vdash T \rhd \sigma_T \qquad M(\sigma_T) = \{T_1, \cdots, T_n\}}{M;P;\Theta \vdash (\textbf{type}\ T \ := \ T_1 \ | \ \cdots \ | \ T_n) \lhd T}$$

We now define an extraction relation $\lhd$ between SMT value terms with expected types and expressions. This extraction relation is quite natural for boolean values, the unit constructor, pair constructors and algebraic datatype constructors. Values for type parameters are extracted based on the extracted type definitions described above.

$$P;\Theta \vdash \textbf{true} \lhd (\mathit{true}, \mathsf{Boolean}) \qquad P;\Theta \vdash \textbf{false} \lhd (\mathit{false}, \mathsf{Boolean}) \qquad P;\Theta \vdash () \lhd (\mathit{Unit}, \mathsf{Unit})$$

$$\frac{T \in \Theta}{P;\Theta \vdash T_i \lhd (T_i, T)} \qquad \frac{P;\Theta \vdash v_1 \lhd (t_1, \tau_1) \qquad P;\Theta \vdash v_2 \lhd (t_2, \tau_2)}{P;\Theta \vdash (v_1, v_2) \lhd (\mathit{cons}_{(\tau_1,\tau_2)}(t_1, t_2), (\tau_1, \tau_2))}$$

$$\frac{(\textbf{type}\ d[\overline{\tau}_d] \ := \ \cdots \ | \ C_i(x_i : \tau_i) \ | \ \cdots) \in P \qquad P;\Theta \vdash v \lhd (t, \tau_i[\tau_d/\tau])}{P;\Theta \vdash C_i[\overline{\tau}](v) \lhd (C_{\overline{\tau},i}(t), d[\overline{\tau}])}$$

As in the embedding, we will generally omit the extraction environment and expected type when they are clear from the context and denote the extraction by $e \lhd t$.

Based on the two extraction relations defined above, we define our extraction relation between models and pairs of program extension and value substitution as follows.

$$\frac{\begin{array}{c} P_{in} = \{\mathsf{d} \mid T \in \Theta,\ P;\Theta \vdash T \rhd \sigma_T,\ M;P;\Theta \vdash \mathsf{d} \lhd \sigma_T\} \\ \theta = \{T \mapsto T \mid T \in \Theta\} \qquad \gamma = \{x \mapsto v \mid (x,\tau) \in \Gamma,\ P;\Theta \vdash v \lhd (M(x),\tau)\} \end{array}}{P;\Theta;\Gamma \vdash (P_{in},\theta,\gamma) \lhd M}$$

Again, we write $(P_{in},\theta,\gamma) \lhd M$ when P, Θ and Γ are clear from the context. In the following, when evaluating expressions of the form $\gamma(\theta(e))$, we will assume evaluation is performed under program $P \cup P_{in}$.

The embedding presented above satisfies two important properties. Consider a (typed) expression $e \in \mathit{expr}$ with embedding $(b_e, e) \rhd (t_e, \Phi_e)$. Firstly, the embedding is *sound* with respect to the operational semantics of our language modulo function calls (Lemma 3). In other words, given a model for the embedding of e, we can extract a program extension and value substitution under which e will evaluate to v, the extraction of t_e, *as long as* the model is consisent with the function calls within e. Secondly, the embedding is *complete* with respect to the operational semantics (Lemma 6). We mean by this that given a program extension, type substitution, and value substitution under which e evaluates to v, we can construct a model that satisfies the embedding and is consistent with the function calls in e.

Before moving on to stating these properties, we must clarify some notions. In the context of a given trace, we use the terms *encountered in the trace* to describe expressions that simply appear in the trace, and *evaluated in the trace* to designate expressions that fully reduce to a value in the trace. We can define these notions in a more formal setting as follows.

Definition 1. *Given a trace $e_1 \to^n e_2$ and some expression e', we say that*

- e' *is* encountered in the trace *iff* $e_1 \to^m \mathcal{E}[e'] \to^{n-m} e_2$*, and*
- e' *is* evaluated in the trace *iff* $e_1 \to^{m_1} \mathcal{E}[e'] \to^{m_2} \mathcal{E}[v \in \mathit{value}] \to^{n-m_1-m_2} e_2$.

It is clear that being evaluated in a trace subsumes being encountered.

Let us now more precisely define the notion of *consistency with calls* employed above when defining soundness and completeness (see Definition 2 below). This notion is always defined in relation with a specific set of inputs, model and function call interpretation in the model. At a high level, we say that the model is consistent with the call (interpretation) if the interpretation corresponds to the operational semantics of the extracted call.

In order to consider the extraction of a function call interpretation, we need the ability to extract SMT terms occurring in the embedding into values that may occur during evaluation. The term extraction procedure $\triangleleft$ given above will only allow a specific shape of values to be extracted. However, we want to cover *all* values that may appear during evaluation. Given a program P, set of type variables Θ and type substitution θ, we introduce a pair of embedding/extraction procedures $\overset{ext}{\triangleright}$ and $\overset{ext}{\triangleleft}$ that can embed and extract values that will appear in the trace given some expected type τ. For each $(T,\tau) \in \theta$, we are given an injection $I_T : \mathit{value} \to \mathbb{N}+$ from values with type τ into the (positive) natural numbers. Note that as values are defined as a least fixed point on syntax and are therefore countable, such an injection is guaranteed to exist. We ensure that if τ corresponds to an extracted type definition of the shape **type** $T := T_1 \mid \cdots \mid T_n$, then $I_T(T_i) = i$ for $1 \leq i \leq n$. We can then define the new value embedding procedure as follows.

$$P;\Theta \vdash (\mathbf{true}, \mathsf{Boolean}) \overset{ext}{\triangleright} \mathit{true} \qquad P;\Theta \vdash (\mathbf{false}, \mathsf{Boolean}) \overset{ext}{\triangleright} \mathit{false} \qquad P;\Theta \vdash ((), \mathsf{Unit}) \overset{ext}{\triangleright} \mathit{Unit}$$

$$\frac{T \in \Theta \qquad (v,i) \in I_T}{P;\Theta \vdash (v,\, T) \overset{ext}{\triangleright} T_i} \qquad \frac{P;\Theta \vdash (v_1, \tau_1) \overset{ext}{\triangleright} t_1 \qquad P;\Theta \vdash (v_2, \tau_2) \overset{ext}{\triangleright} t_2}{P;\Theta \vdash ((v_1, v_2), (\tau_1, \tau_2)) \overset{ext}{\triangleright} \mathit{cons}_{(\tau_1,\tau_n)}(t_1, t_2)}$$

$$\frac{(\mathbf{type}\ d[\overline{\tau}_d] := \cdots \mid C_i(x_i : \tau_i) \mid \cdots) \in P \qquad P;\Theta \vdash (v, \tau_i[\overline{\tau}_d/\overline{\tau}]) \overset{ext}{\triangleright} t}{P;\Theta \vdash (C_i[\overline{\tau}'](v), d[\overline{\tau}]) \overset{ext}{\triangleright} C_{\overline{\tau},i}(t)}$$

The value extraction procedure is then further defined as follows.

$$P;\Theta \vdash \mathbf{true} \overset{ext}{\triangleleft} (\mathit{true}, \mathsf{Boolean}) \qquad P;\Theta \vdash \mathbf{false} \overset{ext}{\triangleleft} (\mathit{false}, \mathsf{Boolean}) \qquad P;\Theta \vdash () \overset{ext}{\triangleleft} (\mathit{Unit}, \mathsf{Unit})$$

$$\frac{T \in \Theta \qquad (v,i) \in I_T}{P;\Theta \vdash v \overset{ext}{\triangleleft} (T_i,\, T)} \qquad \frac{P;\Theta \vdash v_1 \overset{ext}{\triangleleft} (t_1, \tau_1) \qquad P;\Theta \vdash v_2 \overset{ext}{\triangleleft} (t_2, \tau_2)}{P;\Theta \vdash (v_1, v_2) \overset{ext}{\triangleleft} (\mathit{cons}_{(\tau_1,\tau_2)}(t_1, t_2), (\tau_1, \tau_2))}$$

$$\frac{(\mathbf{type}\ d[\overline{\tau}_d] := \cdots \mid C_i(x_i : \tau_i) \mid \cdots) \in P \qquad P;\Theta \vdash v \overset{ext}{\triangleleft} (t, \tau_i[\overline{\tau}_d/\overline{\tau}])}{P;\Theta \vdash C_i[\theta(\overline{\tau})](v) \overset{ext}{\triangleleft} (C_{\overline{\tau},i}(t), d[\overline{\tau}])}$$

As previously, we write $(v, \tau) \overset{ext}{\triangleright} t_v$ and $v \overset{ext}{\triangleleft} (t_v, \tau)$ when the environment $P;\Theta$ is clear from context. Note that the type τ on which we rely in the procedures is such that $v : \theta(\tau)$. It is clear by construction that $\overset{ext}{\triangleright}$ and $\overset{ext}{\triangleleft}$ are inverse to each other. One should also note that $\overset{ext}{\triangleright}$ is defined for all values encountered during evaluation, and $\overset{ext}{\triangleleft}$ is defined for all value terms such that the interpretations of σ_T are within the range of I_T. This is in particular the case when the inputs stem from a model extraction by construction of P_{in}.

Based on the new extraction procedures, we can define the notion of *agreement* between a value substitution and a model as follows.

Definition 2. *For well-typed expression $P;\Theta;\Gamma \vdash e : \tau$, model M and inputs P_{in},θ,γ, we say M* agrees *with γ iff for $(x,\tau) \in \Gamma$, we have $\gamma(x) \overset{ext}{\triangleleft} (M(x), \tau)$.*

It is important to realize that the $\overset{ext}{\triangleright}$ / $\overset{ext}{\triangleleft}$ procedures differ from $\triangleright$ / $\triangleleft$ only when the expected type does not match the actual type. This ensures that when θ is empty (or the identity), $\overset{ext}{\triangleright}$ is equivalent to $\triangleright$ and $\overset{ext}{\triangleleft}$ is equivalent to $\triangleleft$, which leads to the following statement.

Lemma 2. *For embedding $P;\Theta;\Gamma \vdash (b_e, e) \rhd (t_e, \Phi_e)$, model $M \models \Phi_e \cup \{b_e\}$ and extraction $(P_{in}, \theta, \gamma) \lhd M$, M agrees with γ.*

Proof. This follows by induction on the term $M(x)$ for each $(x, \tau) \in \Gamma$. □

Given the new embedding and extraction procedures, we can further define the notion of *consistency* between models and function call interpretations as described above.

Definition 3. *For program P, model M, inputs P_{in}, θ, γ and call interpretation $(f_{\overline{\tau}}(t_1) \mapsto t_2) \in M$ with definition (**def** $f[\overline{\tau}_f](x : \tau_1) : \tau_2 := e_f) \in P$, we say M is* consistent *with the interpretation if $v_1 \overset{ext}{\lhd} (t_1, \tau_1[\overline{\tau}_f/\overline{\tau}])$, $v_2 \overset{ext}{\lhd} (t_2, \tau_2[\overline{\tau}_f/\overline{\tau}])$ and $f[\theta(\overline{\tau})](v_1) \to^* v_2$ in $P \cup P_{in}$.*

Note that if the model is consistent with some call interpretation, then the evaluation of the call under the extracted argument must terminate. We will often be interested in knowing whether a given model M is consistent with the interpretation of certain embedded calls. Given some call $l : f[\overline{\tau}](e_1)$ with associated embedding $\llbracket l : f[\overline{\tau}](e_1) \rrbracket_t = f_{\overline{\tau}}(t_1)$, we say M is consistent with the call if either $M \models \neg \llbracket l : f[\overline{\tau}](e_1) \rrbracket_b$ (the embedding appears on the right-hand side of an implication with negated left-hand side), or M is consistent with the interpretation $(f_{\overline{\tau}}(M(t_1)) \mapsto M(f_{\overline{\tau}}(t_1))) \in M$.

The notion of consistency serves two main purposes in the following. First, it ensures that models including interpretations of function call embeddings are consistent with our operational semantics. Second, as $\overset{ext}{\rhd}$, $\overset{ext}{\lhd}$ and our operational semantics are deterministic, consistency ensures that models with consistent function call interpretations (and compatible constant interpretations) can be unified into a single model that satisfies both original clause sets since both models will agree on common constant and function symbol interpretations.

1.2.1 Soundness

We can now state the first property, namely *soundness* of the embedding.

Lemma 3. *For embedding $P;\Theta;\Gamma \vdash (b_e, e) \rhd (t_e, \Phi_e)$, model $M \models \Phi_e \cup \{b_e\}$ and inputs P_{in}, θ, γ, if M agrees with γ and M is consistent with calls in e, then $\gamma(\theta(e)) \to^* v$ for some $v \in value$. Moreover, given $e : \tau$, we have $v \overset{ext}{\lhd} (M(t_e), \tau)$.*

Proof. We show this by induction on e.

Case $e = x \in id$: By definition of the embedding, we have $t_e = x$, by the operational semantics, we have $\gamma(x) = v$, and by agreement of M with γ, we have $\gamma(x) \overset{ext}{\lhd} (M(x), \tau)$.

Case $e = \mathsf{err}[\tau]$: The embedding gives us $\Phi_e = \{\neg b_e\}$ and no model $M \models \{\neg b_e, b_e\}$ exists.

Case $e = f[\overline{\tau}](e_1)$: By definition of the embedding, we have $t_e = f_{\overline{\tau}}(t_1)$. Consider the definition $(\textbf{def}\ f[\overline{\tau}_f](x : \tau_1) : \tau_2 := e_f) \in P$. By induction, we have $\gamma(\theta(e_1)) \to^* v_1$ where $v_1 \overset{ext}{\lhd} (M(t_1), \tau_1[\overline{\tau}_f/\overline{\tau}])$. Given consistency of M with $f_{\overline{\tau}}(M(t_1))$, we have $\gamma(\theta(e)) \to^* v$ and $v \overset{ext}{\lhd} (M(f_{\overline{\tau}}(t_1)), \tau_2[\overline{\tau}_f/\overline{\tau}])$.

Case $e = \textbf{if}\ (c)\ e_1\ \textbf{else}\ e_2$: Let us assume that $M \models t_c$. By $M \models \Phi_{guard}$, we have $M \models b_1$ and by induction, we have $\gamma(\theta(c)) \to^* \textbf{true}$ and $\gamma(\theta(e_1)) \to^* v_1$ where $v_1 \overset{ext}{\lhd} (M(t_1), \tau)$. The embedding further ensures that $M \models t_1 \simeq t_e$ and therefore $v = v_1$. The case where $M \models \neg t_c$ follows by symmetry.

Case $e = \textbf{let}\ x := e_1\ \textbf{in}\ e_2$: Given the typing judgement $P;\Theta;\Gamma \vdash e_1 : \tau_1$, by induction we have $\gamma(\theta(e_1)) \to^* v_1$ where $v_1 \overset{ext}{\lhd} (M(\llbracket e_1 \rrbracket_t), \tau_1)$. Given the inputs $P_{in}, \theta, \gamma \cup \{x \mapsto v_1\}$, we further have $v \overset{ext}{\lhd} (M(\llbracket e_2 \rrbracket_t), \tau)$ by induction again. As $t_e = \llbracket e_2 \rrbracket_t$ by definition of the embedding, we have $v \overset{ext}{\lhd} (M(t_e), \tau)$.

The remaining cases follow using similar techniques. □

Now let us consider some value $v \in \mathit{value}$ with same type as e, as well as its embedding $v \rhd t_v$. If we constrain the embedded term t_e to correspond to t_v we can produce inputs under which evaluation reaches v as long as the model is consistent with calls.

Corollary 1. *For expression e, value v, embeddings* $P;\Theta;\Gamma \vdash (b_e, e) \rhd (t_e, \Phi_e)$ *and* $v \rhd t_v$*, model* $M \models \Phi_e \cup \{b_e, t_e \simeq t_v\}$ *and extraction* $(P_{in}, \theta, \gamma) \lhd M$*, if M is consistent with calls in e, then* $\gamma(\theta(e)) \to^* \theta(v)$.

Based on the above result and the fact that $M \models b_e$, we can further complement this statement by noting that the blocker constants associated to expressions that are encountered during evaluation will hold. The blocker constants on which the embedding relies therefore constitute the *path condition* under which evaluation reaches the associated expression. This property will be the basis of the unfolding strategy which allows us to find inputs under which evaluation can reach function calls.

Lemma 4. *For embedding* $P;\Theta;\Gamma \vdash (b_e, e) \rhd (t_e, \Phi_e)$*, model* $M \models \Phi_e \cup \{b_e\}$*, inputs* P_{in}, θ, γ *and sub-expression* $l_1 : e_1 \sqsubseteq e$*, if* $\gamma(\theta(e)) \to^{n_1} \mathcal{E}[l_1 : e_1']$*, M agrees with* γ *and M is consistent with calls in e that are evaluated in the trace, then* $M \models \llbracket l_1 : e_1 \rrbracket_b$.

Proof. We show this by inverse structural induction on $l_1 : e_1 \sqsubseteq e$ (one can view this as induction on the *depth* of $l_1 : e_1$ in e). In the base case, we have $\llbracket l_1 : e \rrbracket_b = b_e$ and $M \models b_e$.

We then show that for each direct child $l_2 : e_2 \sqsubseteq e_1$, if $\gamma(\theta(e)) \to^{n_2} \mathcal{E}_2[l_2 : e_2']$, then $M \models \llbracket l_2 : e_2 \rrbracket_b$. The interesting cases are the if and match expressions as they introduce new blocker constants. Let us consider the case where $e_1 = \textbf{if}\ (l_c : c)\ l_t : e_t\ \textbf{else}\ l_e : e_e$. If $l_2 = l_c$, then by definition of the embedding we have $\llbracket l_c : c \rrbracket_b = \llbracket l_1 : e_1 \rrbracket_b$ and therefore $M \models \llbracket l_2 : e_2 \rrbracket_b$. If $l_2 = l_t$, then by the evaluation context and IF-THEN rule, we have $\gamma(\theta(e)) \to^{n_1} \mathcal{E}_1[l_1 : \textbf{if } (c')\ e'_t \textbf{ else } e'_e] \to^{n_c}$ $\mathcal{E}_1[\textbf{if } (\textbf{true})\ e'_t \textbf{ else } e'_e]$ for some $n_c < n_2 - n_1$. By Lemma 3, we have $M \models \llbracket l_c : c \rrbracket_t$, and therefore $M \models \llbracket l_2 : e_2 \rrbracket_b$ by definition of the embedding. The case where $l_2 = l_e$ follows by symmetry. The match case goes through following similar techniques, and the remaining cases preserve the blocker constant. □

In the case where the full trace is given, we can further strengthen the above statement to also include expressions that are not encountered during evaluation.

Lemma 5. *For embedding $P; \Theta; \Gamma \vdash (b_e, e) \rhd (t_e, \Phi_e)$, model $M \models \Phi_e \cup \{b_e\}$, inputs P_{in}, θ, γ and sub-expression $l_1 : e_1 \sqsubseteq e$, if $\gamma(\theta(e)) \to^* v \in value$, M agrees with γ and M is consistent with calls in e, then $\gamma(\theta(e)) \to^* \mathcal{E}[l_1 : e'_1]$ iff $M \models \llbracket l_1 : e_1 \rrbracket_b$.*

Proof. As in the proof of Lemma 4, we proceed by inverse structural induction on $l_1 : e_1 \sqsubseteq e$ and the base case is given by $M \models b_e$.

We then show that if $\gamma(\theta(e)) \to^* \mathcal{E}_1[l_1 : e'_1]$, then for each direct child $l_2 : e_2 \sqsubseteq e_1$, we have $\gamma(\theta(e)) \to^* \mathcal{E}_2[l_2 : e'_2]$ iff $M \models \llbracket l_2 : e_2 \rrbracket_b$. The interesting cases are again the if and match expressions, and we consider the case where $e_1 = \textbf{if } (l_c : c)\ l_t : e_t \textbf{ else } l_e : e_e$. We have seen that if $\gamma(\theta(e)) \to^* \mathcal{E}_2[l_2 : e'_2]$, then $M \models \llbracket l_2 : e_2 \rrbracket_b$. Let us therefore consider the case where $l_2 = l_t$ and $\gamma(\theta(e)) \to^* \mathcal{E}_1[l_1 : \textbf{if } (\textbf{false})\ e'_t \textbf{ else } e'_e]$. By Lemma 3, we have $M \models \neg \llbracket l_c : c \rrbracket_t$, and therefore $M \models \neg \llbracket l_2 : e_2 \rrbracket_b$ by definition of the embedding. The remaining cases follow using analogous arguments.

We finally consider the case where $\gamma(\theta(e)) \not\to^* \mathcal{E}[l_1 : e'_1]$. By our operational semantics, it is clear that for each direct child $l_2 : e_2 \sqsubseteq e_1$, we have $\gamma(\theta(e)) \not\to^* \mathcal{E}[l_2 : e'_2]$. We must therefore show that $M \models \neg \llbracket l_2 : e_2 \rrbracket_b$. The interesting cases are again the if and match expressions. Consider the case where $e_1 = \textbf{if } (l_c : c)\ l_t : e_t \textbf{ else } l_e : e_e$. We have $\llbracket l_1 : e_1 \rrbracket_b = \llbracket l_c : c \rrbracket_b$ and Φ_{guard} ensures that $M \models \neg \llbracket l_1 : e_1 \rrbracket_b$ implies both $M \models \neg \llbracket l_t : e_t \rrbracket_b$ and $M \models \neg \llbracket l_e : e_e \rrbracket_b$. The match case follows a similar argument and the other cases again preserve the blocker constant. □

In addition to blocker constant interpretations agreeing with evaluation, the interpretations of sub-expression embeddings agree with the values encountered during the trace.

Corollary 2. *For embedding $P; \Theta; \Gamma \vdash (b_e, e) \rhd (t_e, \Phi_e)$, model $M \models \Phi_e \cup \{b_e\}$, inputs P_{in}, θ, γ and sub-expression $l_1 : e_1 \sqsubseteq e$ where $e_1 : \tau_1$, if $\gamma(\theta(e)) \to^n \mathcal{E}[l_1 : v_1]$, M agrees with γ and M is consistent with calls in e that are evaluated in the trace, then $v_1 \overset{ext}{\lhd} (M(\llbracket l_1 : e_1 \rrbracket_t), \tau_1)$.*

1.2.2 Completeness

We now establish the second property, namely *completeness* of the embedding.

Lemma 6. *For embedding $P;\Theta;\Gamma \vdash (b_e, e) \rhd (t_e, \Phi_e)$ and inputs P_{in}, θ, γ, if $\gamma(\theta(e)) \rightarrow^* v \in value$, then there exists model $M \models \Phi_e \cup \{b_e\}$ such that M agrees with γ, M is consistent with calls in e and given $e : \tau$, we have $v \overset{ext}{\lhd} (M(t_e), \tau)$.*

Proof. Our proof proceeds by induction on e. We rely on the fact that our language and value embedding are deterministic to ensure that interpretations for function symbols and constants agree between sub-models.

Case $e = x \in id$: Given the embedding $(\gamma(x), \Gamma(x)) \overset{ext}{\rhd} t$, we let $M = \{x \mapsto t, b_e \mapsto true\}$. We clearly have $M \models \Phi_e \cup \{b_e\}$ and $\gamma(x) \overset{ext}{\lhd} (M(x), \tau)$ by embedding/extraction inverse. As there are no calls in e, M is also consistent with all calls in e.

Case $e = \mathsf{err}[\tau]$: There exist no inputs such that $\gamma(\theta(e)) \rightarrow^* v$, hence the statement holds.

Case $e = f[\overline{\tau}](e_1)$: Consider value $\gamma(\theta(e_1)) \rightarrow^* v_1$, embedding $(b_e, e_1) \rhd (t_1, \Phi_1)$ and associated definition $(\textbf{def}\ f[\overline{\tau}_f](x : \tau_1) : \tau_2 := e_f) \in P$. By induction, there exists model M_1 such that $M_1 \models \Phi_1 \cup \{b_e\}$, M_1 agrees with γ, M_1 is consistent with calls in e_1 and $v_1 \overset{ext}{\lhd} (M(t_1), \tau_1[\overline{\tau}_f/\overline{\tau}])$. Now consider embeddings $(v_1, \tau_1[\overline{\tau}_f/\overline{\tau}]) \overset{ext}{\rhd} t_1'$ and $(v, \tau) \overset{ext}{\rhd} t_v$. As all constants stemming from variables declared within e_1 or introduced by the embedding are fresh, the model $M = M_1 \cup \{f_{\overline{\tau}}(t_1') \mapsto t_v\}$ is such that $M \models \Phi_e \cup \{b_e\}$, M agrees with γ, M is consistent with calls in e and $v \overset{ext}{\lhd} (M(t_e), \tau)$.

Case $e = \textbf{if}\ (c)\ e_1\ \textbf{else}\ e_2$: Consider the case were $\gamma(\theta(c)) \rightarrow^* \textbf{true}$, and thus $\gamma(\theta(e_1)) \rightarrow^* v$. Further consider embeddings $(b_e, c) \rhd (t_c, \Phi_c)$, $(b_1, e_1) \rhd (t_1, \Phi_1)$, and $(b_2, e_2) \rhd (t_2, \Phi_2)$ (recall b_1, b_2 and r from the embedding definition). By induction, there exist models M_c, M_1 such that $M_c \models \Phi_c \cup \{b_e\}$, $M_1 \models \Phi_1 \cup \{b_1\}$, M_c, M_1 agree with γ, M_c (respectively M_1) is consistent with calls in e_c (respectively e_1), $\textbf{true} \overset{ext}{\lhd} (M_c(t_c), \mathsf{Boolean})$ and $v \overset{ext}{\lhd} (M_1(t_1), \tau)$. By Lemma 1, we know there exists $M_2 \models \Phi_2 \cup \{\neg b_2\}$. As all constants introduced in the embedding are again fresh, the model $M = M_c \cup M_1 \cup M_2 \cup \{r \mapsto M_1(t_1)\}$ satisfies our property.

Case $e = \textbf{let}\ x_1 := e_1\ \textbf{in}\ e_2$: Consider type $e_1 : \tau_1$ and value $\gamma(\theta(e_1)) \rightarrow^* v_1$. Further consider the embeddings $(b_e, e_1) \rhd (t_1, \Phi_1)$ and $(b_e, e_2) \rhd (t_2, \Phi_2)$. By the operational semantics, we have $(\gamma \cup \{x_1 \mapsto v_1\})(\theta(e_2)) \rightarrow^* v$ and by induction there exist models M_1, M_2 such that $M_1 \models \Phi_1 \cup \{b_e\}$, $M_2 \models \Phi_1 \cup \{b_e\}$, M_1, M_2 agree with γ, M_1 (respectively M_2) is consistent with calls in e_1 (respectively e_2), $v_1 \overset{ext}{\lhd} (M_1(t_1), \tau_1)$ and $v \overset{ext}{\lhd} (M_2(t_2), \tau)$. We again have freshness of all constants in the embedding, and we let $M = M_1 \cup M_2 \cup \{x \mapsto M_1(t_1)\}$.

The remaining cases follow using analogous techniques. □

1.3 Blocking Calls

The soundness of our embedding modulo function calls, in conjunction with the blocker constant instrumentation, suggests a procedure for finding generally sound inputs. In this section, we describe how we can produce inputs under which evaluation is guaranteed to reach some expected value.

We saw above in Lemma 4 that $M \models \llbracket l : e_s \rrbracket_b$ corresponds to the condition under which evaluation of $\gamma(\theta(e))$ will reach label l. If we ensure that the blocker constant used during the embedding of each function call $l : f[\overline{\tau}](e_1)$ within e *does not* hold, then only models corresponding to inputs under which evaluation will result in the expected value will satisfy our clause set.

Given an expression $e \in expr$ with embedding $(b_e, e) \rhd (t_e, \Phi_e)$, let $F(e) = \{l : f[\overline{\tau}](e_1) \sqsubseteq e\}$, the set of all function calls within e with their associated labels. Given a set of calls F, we want to disallow all models for which the blocker constants associated to these labels hold, so we further define $\mathsf{block}(F) = \{\neg \llbracket l : f[\overline{\tau}](e_1) \rrbracket_b \mid l : f[\overline{\tau}](e_1) \in F\}$. Note that the set $\mathsf{block}(F)$ is defined with respect to some existing embedding of e. In general this embedding is clear from the context and will be omitted.

Our goal here is to generate SMT formulas such that models satisfying these formulas will correspond to inputs under which e evaluates to the expected value. Therefore, we are interested in the clause set that corresponds to the negation of the blocker constants associated to calls in $F(e)$, namely $\mathsf{block}(F(e))$. If there exists a model $M \models \Phi_e \cup \mathsf{block}(F(e)) \cup \{b_e, t_e \simeq t_v\}$ with extraction $(P_{in}, \theta, \gamma) \lhd M$, then Corollary 1 and Lemma 4 ensure that $\gamma(\theta(e)) \to^* v$ in $P \cup P_{in}$. This observation forms the basis of our model finding procedure.

We will see in Theorems 1 and 2 that these blocker clauses are *precise*, in the sense that satisfying models produce inputs under which evaluation will not reach the associated calls, and if such inputs exist, then so does a satisfying model.

1.4 Unfolding Calls

In this section, we describe how our procedure incrementally extends the clause set to improve the precision of the function call embeddings. The extension is performed by unfolding function calls to allow value substitutions under which evaluation encounters incrementally many function calls.

In this and the following sections, we will sometimes consider labels attached to sub-expressions of inlined function bodies. These labels are considered to be computed as a function of the label at the function call site as well as the label of the sub-expression within the function's body. For distinct call sites, inlining sub-expression labels are therefore assumed to be distinct. Note that this label assignment corresponds to the label assignment that occurs during function call evaluation. Hence, if evaluation reaches a call site and a CALL evaluation step is performed, the labels that will appear within the inlined function body are identical to those that occur within the statically inlined body.

The main insight behind this procedure is that function calls can be progressively inlined within e in order to increase the set of allowed inputs by unblocking these inlined calls (and blocking the newly introduced calls within their inlined bodies). Let us consider some function call $l : f[\overline{\tau}](e_1) \sqsubseteq e$ where $(\textbf{def}\ f[\overline{\tau}_f](x:\tau_1):\tau_2 := e_f) \in P$. We further consider the expression

$$e_{call} = e[l : f[\overline{\tau}](e_1) / \textbf{let}\ x := e_1\ \textbf{in}\ e_f[\overline{\tau}_f/\overline{\tau}]]$$

obtained by replacing the function call in e by a semantically equivalent term that does not contain that call. Note that in practice, some identifier freshening would be required here but we will omit these from the formalism for clarity. Now by applying the procedure discussed above, we can find inputs for e_{call} such that when evaluating e under those inputs, the function call $f[\overline{\tau}](e_1)$ may be encountered.

While the inlining approach presented above does work in practice, we will present a slight variation here that is better suited to certain later extensions. Instead of inlining the body of f at its call site and then taking the embedding, we will instead embed e directly and extend the clause set such that it becomes equisatisfiable with the one obtained from e_{call}.

In order to do this, we leverage the compositionality of our embedding and the determinism of our language. This enables us to embed e and e_f independently, and then consider the union of their clause sets along with a few extra instrumentation clauses. This observation already hints at the incremental nature of our procedure and its ability to explore the space of inputs simply by progressively extending the set of clauses describing the expression.

Let us now concretize these high-level considerations in a more formal setting. Let us first consider the blocker constant b_f under which the call $f[\overline{\tau}](e_1)$ was embedded, namely $b_f = \llbracket f[\overline{\tau}](e_1) \rrbracket_b$. This blocker corresponds to the condition under which the clause set corresponding to the inlining will be relevant. Recall the embedding $(b_e, e) \rhd (t_e, \Phi_e)$ and let $(b_f, e_f[\overline{\tau}_f/\overline{\tau}]) \rhd (t_f, \Phi_f)$. We know by definition of the embedding that $\llbracket f[\overline{\tau}](e_1) \rrbracket_t$ corresponds to $f_{\overline{\tau}}(t_1)$ for some term t_1. Let us now define the clause set

$$\Phi_{inl} = \Phi_f \cup \{ b_f \implies f_{\overline{\tau}}(t_1) \simeq t_f,\ b_f \implies x_1 \simeq t_1 \}$$

The Φ_{inl} clause set ensures that given a model $M \models \Phi_e \cup \Phi_{inl} \cup \{b_e\}$ and extracted inputs $(P_{in}, \theta, \gamma) \lhd M$, the model M will be consistent with the interpretation of $f_{\overline{\tau}}(t_1)$ in M (as long as M is consistent with all other call interpretations in M). Note that we make sure to preserve the $b \implies c$ shape of all clauses in our clause set. We let $\mathsf{unfold}(f[\overline{\tau}](e_1)) = \Phi_{inl}$.

If we consider the embedding of e_{call}, it is clear that embedding of the function call replacement $\textbf{let}\ x := e_1\ \textbf{in}\ e_f[\overline{\tau}_f/\overline{\tau}]$ is performed under the blocker constant b_f. Hence, the embeddings of e_1 and $e_f[\overline{\tau}_f/\overline{\tau}]$ in e_{call} are exactly identical to those occurring in the clause set $\Phi_e \cup \Phi_{inl} \cup \{b_e, t_e \simeq t_v\}$. The single distinguishing factors are that the term t_f in the embedding of e_{call} is replaced by the term $f_{\overline{\tau}}(t_1)$ and the additional constraint that $f_{\overline{\tau}}(t_1) \simeq t_f$ is added to the clause set.

It remains to consider how a blocking clause set equivalent to $\mathsf{block}(e_{call}, F(e_{call}))$ can be obtained without explicitly generating e_{call}. Our goal here is to build a clause set that ensures that the call $f[\overline{\tau}](e_1)$ is unblocked while all new calls introduced in $e_f[\overline{\tau}_f/\overline{\tau}]$ are blocked. Based on the above observations, it is clear that the clause set $\mathsf{block}(F(e) \setminus \{f[\overline{\tau}](e_1)\}) \cup \mathsf{block}(F(e_f[\overline{\tau}_f/\overline{\tau}]))$ corresponds to the blocker clauses obtained for e_{call}.

1.5 Unfolding Procedure

Algorithm 1: Counterexample finding procedure.

```
input  : a boolean expression $e$
output : a counterexample $(P_{in}, \theta, \gamma)$
1  compute embedding $(b_e, e) \rhd (t_e, \Phi_e)$
2  $\Phi_0 \leftarrow \Phi_e \cup \{b_e, \neg t_e\}$
3  $F_0 \leftarrow F(e)$
4  for $i \leftarrow 0, 1, 2, \cdots$ do
5      if $\exists M.\ M \models \Phi_i \cup \mathsf{block}(F_i)$ then
6          extract inputs $(P_{in}, \theta, \gamma) \lhd M$
7          return $(P_{in}, \theta, \gamma)$
8      else
9          select $f[\overline{\tau}](e_1)_i \in F_i$ with associated (def $f[\overline{\tau}_f](x : \tau_1) : \tau_2 := e_f) \in P$
10         $\Phi_{i+1} \leftarrow \Phi_i \cup \mathsf{unfold}(f[\overline{\tau}](e_1)_i)$
11         $F_{i+1} \leftarrow F_i \setminus \{f[\overline{\tau}](e_1)_i\} \cup F(e_f[\overline{\tau}_f/\overline{\tau}])$
12     end
13 end
```

Let us now bring the considerations from the previous sections together and describe the full unfolding procedure. In order to incrementally explore the space of inputs, we keep track of both a clause set Φ_i and the set F_i of known function calls that *have not yet* been unfolded. Note that to each call $f[\overline{\tau}](e_1) \in F_i$ is associated a blocker constant $\llbracket f[\overline{\tau}](e_1) \rrbracket_b$ and an embedded function application $\llbracket f[\overline{\tau}](e_1) \rrbracket_t$. At each step, we select some call $f[\overline{\tau}](e_1)_i \in F_i$ that has yet to be unfolded and perform an unfolding step. This step consists of generating the clauses corresponding to the function unfolding, as well as extending the set of known functions that have yet to be unfolded (while removing the selected function).

We start by defining the sets Φ_0 and F_0 as $\Phi_0 = \Phi_e \cup \{b_e, t_e \simeq t_v\}$ and $F_0 = F(e)$. We will then inductively define the sets Φ_{i+1}, F_{i+1} given Φ_i, F_i as well as some function call $f[\overline{\tau}](e_1)_i \in F_i$. Let us start by considering the clause set extension. As described previously, the new clause set is computed simply as the union of the current clause set and the clauses corresponding to a call unfolding, namely $\Phi_{i+1} = \Phi_i \cup \mathsf{unfold}(f[\overline{\tau}](e_1)_i)$. This incremental extension of the clause set enables us to incrementally constrain models to be consistent with relevant function call interpretations. We keep track of the set of calls that have yet to be unfolded by letting $F_{i+1} = F_i \setminus \{f[\overline{\tau}](e_1)_i\} \cup F(e_f[\overline{\tau}_f/\overline{\tau}])$. Finally, we define the set $U_i = \{f[\overline{\tau}](e_1)_j \mid j < i\}$ of all calls that have been unfolded at step i. We will be interested in models which only contain interpretations for calls in U_i as such models will be consistent with all call interpretations.

This unfolding procedure can be leveraged to produce inputs for properties of interest. At each step i, we query the SMT solver to determine whether a model $M \models \Phi_i \cup \mathsf{block}(F_i)$ exists. If such is the case, then we extract the inputs $(P_{in}, \theta, \gamma) \lhd M$. Evaluation of e under these inputs is guaranteed to result in the value v, namely we have $\gamma(\theta(e)) \rightarrow^* v$. The complete pseudo-code definition of the counterexample finding procedure is given in Algorithm 1. The given pseudo-code relies on the blocking procedure $\mathsf{block}(\cdot)$ defined in Section 1.3 and on the function call unfolding procedure $\mathsf{unfold}(\cdot)$ defined in Section 1.4.

Example unfolding. Recalling the right unit law verification condition presented in this chapter's introduction, a possible unfolding sequence can be found in Figure 1.9. The clause set Φ_0 is constructed based on the embeddings of the verification condition and the value **false** (since we want a counterexample to the given property). The clause sets $\Phi_i \cup \mathsf{block}(F_i)$ are unsatisfiable for $0 \leq i \leq 4$, however there exists $M \models \Phi_5 \cup \mathsf{block}(F_5)$ such that $M(\mathsf{x}) = T_1$ and $M(\mathsf{xs}) = \mathsf{Nil}_T$ where T_1 is a value of uninterpreted sort σ_T. Extraction will then produce the following inputs under which evaluation leads to **false**.

$$P_{in} = \{\textbf{type}\ T := T_1\} \qquad \theta = \{T \mapsto T\} \qquad \gamma = \{\mathsf{x} \mapsto T_1, \mathsf{xs} \mapsto \mathsf{Nil}[T]\}$$

Example of input finding. In addition to the counterexample finding capabilities showcased in the previous example, the unfolding procedure can generate inputs to properties of interest. Let us consider the formulation of single-step evaluation in the untyped lambda calculus given by the program below. Note that we consider here that variables are values and the small-step operational semantics are given by the (partial) eval function.

```
type Nat := Succ(n: Nat) | Zero
type Term := Var(n: Nat) | App(caller: Term, arg: Term) | Abs(x: Nat, body: Term)

def isValue(term: Term): Boolean = term match {
  case Var(_) ⇒ true case Abs(_, _) ⇒ true case App(_, _) ⇒ false
}

def subst(term: Term, n: Nat, v: Term): Term = term match {
  case Var(x) ⇒ if (n ≈ x) v else term
  case App(c, arg) ⇒ App(subst(c, n, v), subst(arg, n, v))
  case Abs(x, body) ⇒ if (x == n) term else Abs(x, subst(body, n, v))
}

def eval(term: Term): Term = term match {
  case App(c, arg) if !isValue(c) ⇒ App(eval(c), arg)
  case App(c, arg) if !isValue(arg) ⇒ App(c, eval(arg))
```

i	$\Phi_i \setminus \Phi_{i-1}$	F_i	$\mathsf{block}(F_i)$
0	$\{b_e, r_1 \simeq \mathit{false},$	$\{\mathsf{rightUnit}[T](\mathsf{xs}),$	$\{\neg b_e, \neg b_1\}$
	$(b_e \wedge \mathsf{rightUnit}_T(\mathsf{xs})) \iff b_1,$	$\mathsf{rightUnit}[T](\mathsf{Cons}[T](\mathsf{x},\mathsf{xs}))\}$	
	$(b_e \wedge \neg\mathsf{rightUnit}_T(\mathsf{xs})) \iff b_2,$		
	$b_1 \implies r_1 \simeq \mathsf{rightUnit}_T(\mathsf{Cons}_T(\mathsf{x},\mathsf{xs})),$		
	$b_2 \implies r_1 \simeq \mathit{false}\}$		
	Unfolding call $\mathsf{rightUnit}[T](\mathsf{xs}) \in F_0$		
1	$\{b_e \implies \mathsf{rightUnit}_T(\mathsf{xs}) \simeq (\mathsf{append}_T(\mathsf{list}_1, \mathsf{Nil}_T) \simeq \mathsf{Nil}_T),$	$\{\mathsf{rightUnit}[T](\mathsf{Cons}[T](\mathsf{x},\mathsf{xs})),$	$\{\neg b_e, \neg b_1\}$
	$b_e \implies \mathsf{list}_1 \simeq \mathsf{xs}\}$	$\mathsf{append}[T](\mathsf{list}_1, \mathsf{Nil}[T])\}$	
	Unfolding call $\mathsf{rightUnit}[T](\mathsf{Cons}[T](\mathsf{x},\mathsf{xs})) \in F_1$		
2	$\{b_1 \implies \mathsf{rightUnit}_T(\mathsf{Cons}_T(\mathsf{x},\mathsf{xs})) \simeq (\mathsf{append}_T(\mathsf{list}_2, \mathsf{Nil}_T) \simeq \mathsf{Nil}_T),$	$\{\mathsf{append}[T](\mathsf{list}_1, \mathsf{Nil}[T]),$	$\{\neg b_e, \neg b_1\}$
	$b_1 \implies \mathsf{list}_2 \simeq \mathsf{Cons}_T(\mathsf{x},\mathsf{xs})\}$	$\mathsf{append}[T](\mathsf{list}_2, \mathsf{Nil}[T])\}$	
	Unfolding call $\mathsf{append}[T](\mathsf{list}_1, \mathsf{Nil}[T]) \in F_2$		
3	$\{b_e \implies \mathsf{append}_T(\mathsf{list}_1, \mathsf{Nil}_T) \simeq r_2,$	$\{\mathsf{append}[T](\mathsf{list}_2, \mathsf{Nil}[T]),$	$\{\neg b_1, \neg b_3\}$
	$b_e \implies \mathsf{l1}_1 \simeq \mathsf{list}_1,$	$\mathsf{append}[T](\mathsf{xs}_1, \mathsf{l2}_1)\}$	
	$b_e \implies \mathsf{l2}_1 \simeq \mathsf{Nil}_T,$		
	$(b_e \wedge \mathit{is\text{-}Cons}_T(\mathsf{l1}_1)) \iff b_3,$		
	$(b_e \wedge \mathit{is\text{-}Nil}_T(\mathsf{l1}_1)) \iff b_4,$		
	$b_3 \implies \mathsf{x}_1 \simeq \mathsf{head}_T(\mathsf{l1}_1),$		
	$b_3 \implies \mathsf{xs}_1 \simeq \mathsf{tail}_T(\mathsf{l1}_1),$		
	$b_3 \implies r_2 \simeq \mathsf{Cons}_T(\mathsf{x}_1, \mathsf{append}_T(\mathsf{xs}_1, \mathsf{l2}_1)),$		
	$b_4 \implies r_2 \simeq \mathsf{l2}_1\}$		
	Unfolding call $\mathsf{append}[T](\mathsf{list}_2, \mathsf{Nil}[T]) \in F_3$		
4	$\{b_1 \implies \mathsf{append}_T(\mathsf{list}_2, \mathsf{Nil}_T) \simeq r_3,$	$\{\mathsf{append}[T](\mathsf{xs}_1, \mathsf{l2}_1),$	$\{\neg b_3, \neg b_5\}$
	$b_1 \implies \mathsf{l1}_2 \simeq \mathsf{list}_2,$	$\mathsf{append}[T](\mathsf{xs}_2, \mathsf{l2}_2)\}$	
	$b_1 \implies \mathsf{l2}_2 \simeq \mathsf{Nil}_T,$		
	$(b_1 \wedge \mathit{is\text{-}Cons}_T(\mathsf{l1}_2)) \iff b_5,$		
	$(b_1 \wedge \mathit{is\text{-}Nil}_T(\mathsf{l1}_2)) \iff b_6,$		
	$b_5 \implies \mathsf{x}_2 \simeq \mathsf{head}_T(\mathsf{l1}_2),$		
	$b_5 \implies \mathsf{xs}_2 \simeq \mathsf{tail}_T(\mathsf{l1}_2),$		
	$b_5 \implies r_3 \simeq \mathsf{Cons}_T(\mathsf{x}_2, \mathsf{append}_T(\mathsf{xs}_2, \mathsf{l2}_2)),$		
	$b_6 \implies r_3 \simeq \mathsf{l2}_2\}$		
	Unfolding call $\mathsf{append}[T](\mathsf{xs}_2, \mathsf{l2}_2) \in F_4$		
5	$\{b_5 \implies \mathsf{append}_T(\mathsf{xs}_2, \mathsf{l2}_2) \simeq r_4,$	$\{\mathsf{append}[T](\mathsf{xs}_1, \mathsf{l2}_1),$	$\{\neg b_3, \neg b_7\}$
	$b_5 \implies \mathsf{l1}_3 \simeq \mathsf{xs}_2,$	$\mathsf{append}[T](\mathsf{xs}_3, \mathsf{l2}_3)\}$	
	$b_5 \implies \mathsf{l2}_3 \simeq \mathsf{l2}_2,$		
	$(b_5 \wedge \mathit{is\text{-}Cons}_T(\mathsf{l1}_3)) \iff b_7,$		
	$(b_5 \wedge \mathit{is\text{-}Nil}_T(\mathsf{l1}_3)) \iff b_8,$		
	$b_7 \implies \mathsf{x}_3 \simeq \mathsf{head}_T(\mathsf{l1}_3),$		
	$b_7 \implies \mathsf{xs}_3 \simeq \mathsf{tail}_T(\mathsf{l1}_3),$		
	$b_7 \implies r_4 \simeq \mathsf{Cons}_T(\mathsf{x}_3, \mathsf{append}_T(\mathsf{xs}_3, \mathsf{l2}_3)),$		
	$b_8 \implies r_4 \simeq \mathsf{l2}_3\}$		

Figure 1.9 – A possible sequence of unfolding steps during counterexample search for the inductive case of the right unit law $\mathsf{rightUnit}[T](\mathsf{xs}) \implies \mathsf{rightUnit}[T](\mathsf{Cons}[T](\mathsf{x},\mathsf{xs}))$. We display at each step i the new clauses $\Phi_i \setminus \Phi_{i-1}$ introduced at step i, the blocked calls F_i, and the associated blocking clauses $\mathsf{block}(F_i)$.

```
    case App(Abs(x, body), arg) ⇒ subst(body, x, arg)
    case _ ⇒ err[Term]
  }
```

It is well-known that evaluation in the untyped lambda calculus can lead to non-termination. Let us consider the problem of finding inputs such that evaluation is idempotent. We can rely on the unfolding procedure to find and assignment for the term free variable such that the following expression evaluates to **true**.

```
eval(term) ≈ term
```

In other words, performing a step of evaluation on the generated term will result again in the same term (and we therefore have non-termination of evaluation). The procedure will output the following assignment for the term variable which corresponds to the familiar omega term $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$ given within our formulation of the lambda calculus.

```
term ↦ App(Abs(Zero, App(Var(Zero), Var(Zero))), Abs(Zero, App(Var(Zero), Var(Zero))))
```

1.5.1 Soundness

We can now state (and prove) the first main result of Chapter 1. Our procedure is *sound*, namely the models it produces correspond to valid inputs.

Theorem 1. *For expression e, value v, unfolding step $i \in \mathbb{N}$ and model $M \models \Phi_i \cup \mathsf{block}(F_i)$, given extraction $(P_{in}, \theta, \gamma) \lhd M$, we have $\gamma(\theta(e)) \to^* \theta(v)$.*

Proof. We will rely here on the notion of *unfolding tree* which we define as the tree where nodes correspond to a root node e or a function call in U_i and edges exist between two nodes $l_1 : e_1, l_2 : e_2$ iff e_2 is a function call and either $l_1 : e_1$ is the root node and $l_2 : e_2 \sqsubseteq e_1$, or $l_1 : e_1$ is a call node and $l_2 : e_2$ occurs within the unfolded body of e_1. Nodes with a same parent are partially ordered by the order in which they would be evaluated in a trace (this is statically known in a language with call-by-value semantics).

For trace $\gamma(\theta(e)) \to^{n_1} e'$, we then show by bottom-up induction on the unfolding tree (and the partial order between sibling nodes) that for each node $l : e_t$ in the tree, if

1. $\gamma(\theta(e)) \to^{n_2} \mathcal{E}[l : e'_t] \to^{n_1 - n_2} e'$,
2. $M \models \llbracket l : e_t \rrbracket_b$, and
3. whenever $e_t = f[\overline{\tau}](e_1)$ with associated definition ($\mathbf{def}\ f[\overline{\tau}_f](x : \tau_1) : \tau_2\ := \ e_f) \in P$ and $\gamma(\theta(e)) \to^{n_3} \mathcal{E}[l : f[\overline{\tau}](v_1)]$ for $n_3 \le n_1$, then we have $v_1 \overset{ext}{\lhd} (M(\llbracket e_1 \rrbracket_t), \tau_1[\overline{\tau}_f / \overline{\tau}])$,

then given the maximal trace $e'_t \to^{n_4} e_v$ for $n_4 \le n_1 - n_2$, M is consistent with all calls evaluated in the trace, and all calls that are encountered in the trace belong to U_i.

In the case where $l : e_t$ is the root node, we can apply Lemmas 2, 4, Corollary 2 and the inductive hypothesis to conclude consistency and inclusion in U_i. We therefore consider the case where $e_t = f[\overline{\tau}](e_1)$. For the trace $\gamma(\theta(e)) \to^{n_2} \mathcal{E}[l : f[\overline{\tau}](e_1')]$, consistency is given by induction on the partial ordering between siblings. Let us consider the interesting case where $\gamma(\theta(e)) \to^{n_3} \mathcal{E}[l : f[\overline{\tau}](v_1)] \to \mathcal{E}[e_f[\overline{\tau}_f/\overline{\tau}][x/v_1]]$ and $n_3 + 1 \leq n_1$. By definition of Φ_i, we have $M \models \llbracket l : f[\overline{\tau}](e_1)\rrbracket_t \simeq \llbracket e_f[\overline{\tau}_f/\overline{\tau}]\rrbracket_t$ and $M \models x \simeq \llbracket e_1 \rrbracket_t$. Given the value substitution $\gamma' = \{x \mapsto v_1\}$, M agrees with γ' by condition 3. Let us now consider the calls that occur within $e_f[\overline{\tau}_f/\overline{\tau}]$ and are evaluated in the trace. We show by induction on the evaluation partial order that M is consistent with each such call. If the call belongs to U_i, then condition 1 of the inductive hypothesis is satisfied (as the call is evaluated), and the inductive hypothesis lets us apply Lemma 4 and Corollary 2 to satisfy the remaining conditions, hence consistency holds. If the call does not belong to U_i, then $M \models \mathsf{block}(F_i)$ and Lemma 4 form a contradiction. Finally, if $\mathcal{E}[f[\overline{\tau}](v_1)] \to^{n_4} \mathcal{E}[v_f \in \mathit{value}]$ then $\gamma'(\theta(e_f[\overline{\tau}_f/\overline{\tau}])) \to^{n_4-1} v_f$ and Lemma 3 ensures that we have $v_f \overset{ext}{\lhd} (M(\llbracket e_f[\overline{\tau}_f/\overline{\tau}]\rrbracket_t), \tau_2[\overline{\tau}_f/\overline{\tau}])$. The model M is therefore consistent with the remaining call $f[\overline{\tau}](v_1)$ which was evaluated in the trace.

It is clear that all three conditions of the above statement hold for the root node e. It therefore remains to show that no infinite trace exists. As all encountered calls belong to U_i and no label can be encountered twice, we know evaluation must terminate. We can then apply Corollary 1 to conclude our proof. $\square$

1.5.2 Completeness

A perhaps more surprising result of this unfolding procedure is that in addition to producing valid inputs, it is *complete* for such inputs. In other words, if a program extension P_{in}, type substitution θ and value substitution γ exist such that $\gamma(\theta(e)) \to^* \theta(v)$ in $P \cup P_{in}$, then the procedure will eventually output a set of inputs. However, the condition for this result is that the selection process for $f[\overline{\tau}](e_1)_i$ at each step i is *fair*. Let us define more clearly what we mean by *fairness* in this context.

Definition 4. *The selection process for the call to unfold at each step is* fair *iff for each i and $f[\overline{\tau}](e_1) \in F_i$, there exists a j such that $f[\overline{\tau}](e_1) \in U_j$.*

In other words, *fairness* implies that any function call in the program will eventually be unfolded by the procedure. Based on this definition, we can state the following completeness result.

Theorem 2. *For expression e and value v, if there exist inputs P_{in}, θ, γ such that $\gamma(\theta(e)) \to^* \theta(v)$ and the unfolding selection process is fair, then there exists a $k \in \mathbb{N}$ and model M such that $M \models \Phi_k \cup \mathsf{block}(F_k)$.*

Proof. First, select k such that for each call $l : f[\overline{\tau}'](e_1')$ that is encountered in $\gamma(\theta(e)) \to^* \theta(v)$, we have $l : f[\overline{\tau}](e_1) \in U_k$. Note that this k is guaranteed to exist. Indeed, for each call that is encountered in the trace, there exists a sequence of CALL evaluation steps that will reach it and the unfolding selection process is fair, hence the relevant unfoldings will eventually occur.

We then show by induction on $0 \leq i \leq k$ that there exists a model M_i such that

1. $M_i \models \Phi_i$,
2. M_i agrees with γ,
3. for $l : f[\overline{\tau}](e_1) \in F_i \cup U_i$ we have $\gamma(\theta(e)) \to^* \mathcal{E}[l : f[\overline{\tau}'](e_1')]$ iff $M \models \llbracket l : f[\overline{\tau}](e_1) \rrbracket_b$, and
4. for $l : f[\overline{\tau}](e_1) \in F_i \cup U_i$ with $(\textbf{def}\ f[\overline{\tau}_f](x : \tau_1) : \tau_2 := e_f) \in P$, if $\gamma(\theta(e)) \to^* \mathcal{E}[l : f[\overline{\tau}'](e_1')]$, then given evaluation results $e_1' \to^* v_1$ and $f[\overline{\tau}'](e_1') \to^* v_f$ where $v_1, v_f \in \mathit{value}$, we have $v_1 \overset{ext}{\lhd} (M(\llbracket e_1 \rrbracket_t), \tau_1[\overline{\tau}_f / \overline{\tau}])$ and $v_f \overset{ext}{\lhd} (M(\llbracket f[\overline{\tau}](e_1) \rrbracket_t), \tau_2[\overline{\tau}_f / \overline{\tau}])$.

Note that items 3 and 4 above imply consistency with all calls in $F_i \cup U_i$.

The base case is given by Lemmas 5, 6 and Corollary 2. For the inductive case, we can assume a model M_i for which the hypothesis holds. Given the call $l : f[\overline{\tau}](e_1)$ selected at step i to produce Φ_{i+1}, F_{i+1}, consider its definition $(\textbf{def}\ f[\overline{\tau}_f](x : \tau_1) : \tau_2 := e_f) \in P$ and the embeddings $(\llbracket l : f[\overline{\tau}](e_1) \rrbracket_b, e_f[\overline{\tau}_f / \overline{\tau}]) \rhd (t_f, \Phi_f)$ and $f_\tau(t_1) = \llbracket l : f[\overline{\tau}](e_1) \rrbracket_t$. We then consider the following two cases:

1. $\gamma(\theta(e)) \to^* \mathcal{E}[l : f[\theta(\overline{\tau})](v_1)] \to^* \mathcal{E}[v_f]$: We have $M_i \models \llbracket l : f[\overline{\tau}](e_1) \rrbracket_b$ by item 3 of the inductive hypothesis. Given value substitution $\gamma' = \{x \mapsto v_1\}$, Lemma 6 ensures that a model $M_f \models \Phi_f \cup \{\llbracket l : f[\overline{\tau}](e_1) \rrbracket_b, \llbracket l : f[\overline{\tau}](e_1) \rrbracket_t \simeq t_f\}$ exists such that M_f is consistent with the calls in $e_f[\overline{\tau}_f / \overline{\tau}]$. Lemma 5 and Corollary 2 then give us items 3 and 4 of the statement. Item 4 of the inductive hypothesis finally ensures that M_i and M_f agree on all common constant and function interpretations, hence they can be unified into M_{i+1} which satisfies all four items.
2. $\gamma(\theta(e)) \not\to^* \mathcal{E}[l : f[\overline{\tau}'](e_1')]$: By induction, we have $M_i \models \neg \llbracket l : f[\overline{\tau}](e_1) \rrbracket_b$, and if we extend M_i to M_{i+1} by setting all blocker constants associated to sub-expressions of $e_f[\overline{\tau}_f / \overline{\tau}]$ to *false*, Lemma 1 ensures that $M_{i+1} \models \Phi_{i+1}$ and the remaining items are satisfied.

Finally, item 3 of the statement gives us $M_k \models \mathsf{block}(F_k)$, thus concluding the proof. □

1.5.3 Procedure Termination

In the case where no inputs P_{in}, θ, γ exist such that $\gamma(\theta(e)) \to^* \theta(v)$ (and e calls at least one recursive function), our unfolding procedure will never terminate. However, the incremental nature of the Φ_i clause sets implies that if Φ_i is unsatisfiable, then for all $j \geq i$, the clause set $\Phi_j \cup \mathsf{block}(F_j)$ will be unsatisfiable. This observation leads to the following Theorem.

Theorem 3. *For expression e, value v and unfolding index $i \in \mathbb{N}$, if there exists no model $M \models \Phi_i$, then there exist no inputs P_{in}, θ, γ such that $\gamma(\theta(e)) \to^* \theta(v)$.*

Proof. This follows from definition of Φ_i and Theorem 2. □

The extended counterexample finding procedure including termination when a proof of counterexample inexistence is produced can be found in Algorithm 2. It is important to realize here that the inexistence of a counterexample does not imply that the property must evaluate to **true** for all inputs. Indeed, our language contains both stuck and diverging expressions for which evaluation will not terminate to a value. The notion of *correctness* of a program will be explored in more details in Chapter 3.

Algorithm 2: Counterexample finding procedure with inexistence proofs.

```
input  : a boolean expression $e$
output : a counterexample $(P_{in}, \theta, \gamma)$ or counterexample inexistence
compute embedding $(b_e, e) \rhd (t_e, \Phi_e)$
$\Phi_0 \leftarrow \Phi_e \cup \{b_e, \neg t_e\}$
$F_0 \leftarrow F(e)$
for $i \leftarrow 0, 1, 2, \cdots$ do
    if $\exists M.\ M \models \Phi_i \cup \mathsf{block}(F_i)$ then
        extract inputs $(P_{in}, \theta, \gamma) \lhd M$
        return $(P_{in}, \theta, \gamma)$
    else if $\nexists M.\ M \models \Phi_i$ then
        return inexistence
    else
        select $f[\overline{\tau}](e_1)_i \in F_i$ with associated $(\mathbf{def}\ f[\overline{\tau}_f](x : \tau_1) : \tau_2 := e_f) \in P$
        $\Phi_{i+1} \leftarrow \Phi_i \cup \mathsf{unfold}(f[\overline{\tau}](e_1)_i)$
        $F_{i+1} \leftarrow F_i \setminus \{f[\overline{\tau}](e_1)_i\} \cup F(e_f[\overline{\tau}_f / \overline{\tau}])$
    end
end
```

Given the counterexample we obtained for our right unit law formulation, we can fix the property statement as follows.

```
def rightUnit[T](list: List[T]): Boolean = append[T](list, Nil[T]) ≈ list
```

Given this new definition, the differences in the unfolding procedure with respect to the steps listed in Figure 1.9 can be found in Figure 1.10. The clause set Φ_4 thus obtained is already unsatisfiable and the procedure can terminate as no counterexample exists.

i	$\Phi_i \setminus \Phi_{i-1}$	F_i	$\mathsf{block}(F_i)$
	Unfolding call $\mathsf{rightUnit}[T](\mathsf{xs}) \in F_0$		
1	$\{b_e \implies \mathsf{rightUnit}_T(\mathsf{xs}) \simeq (\mathsf{append}_T(\mathsf{list}_1, \mathsf{Nil}_T) \simeq \mathsf{list}_1),$ $b_e \implies \mathsf{list}_1 \simeq \mathsf{xs}\}$	$\{\mathsf{rightUnit}[T](\mathsf{Cons}[T](\mathsf{x}, \mathsf{xs})),$ $\mathsf{append}[T](\mathsf{list}_1, \mathsf{Nil}[T])\}$	$\{\neg b_e, \neg b_1\}$
	Unfolding call $\mathsf{rightUnit}[T](\mathsf{Cons}[T](\mathsf{x}, \mathsf{xs})) \in F_1$		
2	$\{b_1 \implies \mathsf{rightUnit}_T(\mathsf{Cons}_T(\mathsf{x}, \mathsf{xs})) \simeq (\mathsf{append}_T(\mathsf{list}_2, \mathsf{Nil}_T) \simeq \mathsf{list}_2),$ $b_1 \implies \mathsf{list}_2 \simeq \mathsf{Cons}_T(\mathsf{x}, \mathsf{xs})\}$	$\{\mathsf{append}[T](\mathsf{list}_1, \mathsf{Nil}[T]),$ $\mathsf{append}[T](\mathsf{list}_2, \mathsf{Nil}[T])\}$	$\{\neg b_e, \neg b_1\}$

Figure 1.10 – The changed steps that would result from applying the unfolding steps listed in Figure 1.9 with the corrected rightUnit function definition.

2 Counterexample Finding in the Presence of Higher-Order Functions

In this chapter, we will discuss how our language can be extended with higher-order functions (namely by introducing function types, lambdas and function applications). We will extend both the embedding and unfolding procedure to handle these new language constructs in order to allow counterexample finding in the presence of first-class functions. We do not rely on a closed-world assumption and can produce counterexamples (and proofs of their inexistence) which involve new lambdas which do not belong to the considered program.

We embed first-class functions (*i.e.* lambdas) into algebraic datatype constructors and rely on special uninterpreted dispatch functions in order to embed applications. The unfolding procedure is extended to handle applications through a form of dynamic dispatch. Since our language supports parametric polymorphism, the set of (typed) lambdas observed during unfolding is not necessarily bounded. Our procedure therefore incrementally extends the set of constructors associated to the algebraic datatype used for lambda embeddings. By relying on a datatype at the embedding level and progressively introducing constructors, our approach can handle programs where even whole-program defunctionalization into records (*i.e.* datatypes in our case) would fail [Rey98].

In order to handle open-world programs, we rely on a special $\mathsf{Else}(n : \mathbb{N})$ constructor in the algebraic datatype associated to first-class functions which does not correspond to any lambda observed during unfolding. Given an SMT value term $\mathsf{Else}(t)$, we extract a first-class function value by considering the interpretation of the special dispatch function symbol. The function value is constructed by generating an *if*-expression based on the interpretations of the dispatch symbol for each embedded application. This approach supposes some means of differentiating between inputs in order to generate the *if*-expression conditions. In other words, we need a notion of equality defined for all types in our language in order to extract function-typed values. Although such a feature presented no issue in the language discussed in Chapter 1, in the presence of first-class functions, it becomes a non-trivial requirement.

Extensional equality, which intuitively seems like a desirable notion of equality, is not well suited to operationally defined languages as it is an undecidable property. Syntactic equality also has its shortcomings as it allows some level of introspection into the *definition* of a function. For example, it allows the language to differentiate between the functions $\lambda.\ \mathsf{plus}(\mathsf{Zero},\mathsf{Zero})$ and $\lambda.\ \mathsf{Zero}$, while no difference can be observed through only applications. However, as expression enumeration is not possible in our language, the potential for introspection remains quite limited and the decidability of syntactic equality led us to select it as our notion of function equality.

We will show in this chapter that our embedding and unfolding procedure can be extended to the higher-order setting while preserving soundness and completeness for counterexamples. We will further show that the procedure may soundly terminate when no counterexample exists, thus resulting in a proof of counterexample inexistence. Finally, we will present some important optimizations which greatly improve the efficiency of our procedure.

Example. Let us consider an example program to demonstrate the new higher-order language constructs. We again define a program containing a generic List type definition, as well as the generic recursive higher-order exists and forall functions on List. The program finally defines a lemma existsForall which states the correspondence between existential and universal quantification in lists. However, the absent-minded programmer has again inserted an error in the lemma definition, forgetting a negation.

```
type List[T] = Cons(head: T, tail: List[T]) | Nil

def exists[T](l: List[T], p: T → Boolean): Boolean = l match {
  case Cons(x, xs) ⇒ p(x) || exists[T](xs, p)
  case Nil ⇒ false
}

def forall[T](l: List[T], p: T → Boolean): Boolean = l match {
  case Cons(x, xs) ⇒ p(x) && forall[T](xs, p)
  case Nil ⇒ true
}

def existsForall[T](l: List[T], p: T → Boolean): Boolean =
  exists[T](l, p) ≈ !forall[T](l, λ x: T. p(x))
```

Some verification procedure is then invoked and generates the following verification condition corresponding to the inductive case of the exists-forall correspondance property.

$$\mathsf{existsForall}[T](\mathsf{xs},\mathsf{p}) \implies \mathsf{existsForall}[T](\mathsf{Cons}[T](\mathsf{x},\mathsf{xs}),\mathsf{p})$$

In the remainder of this chapter, we will see how our procedure can be used to generate counterexamples for such property statements, as well as show their inexistence.

$$\textsc{Function Type}\quad \frac{P;\Theta \vdash \tau_1\ \mathit{type} \qquad P;\Theta \vdash \tau_2\ \mathit{type}}{P;\Theta \vdash \tau_1 \to \tau_2\ \mathit{type}}$$

$$\textsc{App}\quad \frac{P;\Theta;\Gamma \vdash e_1 : \tau_2 \to \tau \qquad P;\Theta;\Gamma \vdash e_2 : \tau_2}{P;\Theta;\Gamma \vdash e_1\ e_2 : \tau} \qquad \textsc{Abs}\quad \frac{P;\Theta;\Gamma, x : \tau_1 \vdash e : \tau_2}{P;\Theta;\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \to \tau_2}$$

Figure 2.1 – Type formation and typing rules of the higher-order language extension.

$$\mathcal{E} \ ::= \ \cdots \mid \mathcal{E}\ \mathit{expr} \mid \mathit{value}\ \mathcal{E} \qquad \textsc{App}\quad \frac{v \in \mathit{value}}{(\lambda x : \tau.\ e)\ v \to e[x/v]}$$

Figure 2.2 – Operational semantics of the higher-order language extension.

2.1 Language

Let us now extend the language presented in Chapter 1 with higher-order functions. This is done fairly simply at the syntax level by extending the expression, value, and type grammars with the following rules:

$$\begin{array}{rcl} \mathit{expr} & ::= & \cdots \mid \lambda \mathit{id} : \mathit{type}.\ \mathit{expr} \mid \mathit{expr}\ \mathit{expr} \\ \mathit{value} & ::= & \cdots \mid (\lambda x : \tau.\ e) \in \mathit{expr}\ \text{ if }\ FV(e) \subseteq \{x\} \\ \mathit{type} & ::= & \cdots \mid \mathit{type} \to \mathit{type} \end{array}$$

In order to improve readability, we will sometimes omit the type associated to the lambda binding in the following and denote lambdas by $\lambda x.\ e$.

When considering lambda values in the following, we will consider two values containing lambdas to be equal if they are equivalent modulo alpha renaming. In other words, given two lambdas $\lambda x_1 : \tau.\ e_1$ and $\lambda x_2 : \tau.\ e_2$, we either have $\lambda x_1 : \tau.\ e_1 = \lambda x_2 : \tau.\ e_2$ or there is no identifier bijection θ such that $\lambda x_1 : \tau.\ e_1 = \theta(\lambda x_2 : \tau.\ e_2)$. Such a requirement can be efficiently implemented using, for example, normalization through de Bruijn indices. The static and operational semantics are then extended with the rules shown in Figure 2.1 and Figure 2.2.

2.1.1 Structural Equality

The operational semantics we attribute to our equality predicate rely on the *structure* of values to determine whether they are equal. This may seem surprising at a first glance, however this approach allows for efficiently decidable equality which leads to completeness for model finding in the presence of higher-order functions.

We should recall here that the evaluation rules LET, MATCH, CALL and APP perform value substitutions in the expression that is being evaluated, as well as the initial value substitution needed to evaluate open expressions. This means that although the exact static structure of two lambdas may differ, they may end up being equal during evaluation. Consider for example the following closed expression

$$e = (\textbf{let}\ x\ :=\ \mathsf{Zero}\ \textbf{in}\ \lambda y : \tau.\ x) \approx \lambda y : \tau.\ \mathsf{Zero}$$

It is clear that $e \rightarrow^* \textbf{true}$ since $\textbf{let}\ x\ :=\ \mathsf{Zero}\ \textbf{in}\ \lambda y : \tau.\ x \rightarrow \lambda y : \tau.\ \mathsf{Zero}$. However, the static structure of the two lambdas are clearly not equal.

To clarify the relation between static and operational equality, let us introduce the notion of *pseudo-value* which can be defined using the following grammar:

$$pvalue \quad ::= \quad x \mid \textbf{true} \mid \textbf{false} \mid () \mid (pvalue, pvalue) \mid C[\overline{\tau}](pvalue) \mid \lambda x : \tau.\ expr$$

One can see that $value \subseteq pvalue$ and for any $v \in pvalue$ and (partial) value substitution γ, $\gamma(v) \in pvalue$. This also holds for pseudo-value substitutions, namely mappings from identifiers to pseudo-values. We define $normalize(\lambda x : \tau.\ e) = (\lambda x : \tau.\ e', \gamma)$ the *normalization* of the lambda $\lambda x : \tau.\ e$ such that the following three properties hold:

1. for all $l_1 : e_1 \sqsubseteq e'$ such that $e_1 \in pvalue$, we either have
   (a) $e_1 \in FV(\lambda x : \tau.\ e')$ and there exists no $l_2 : e_1 \sqsubseteq e'$ where $l_1 \neq l_2$, or
   (b) $FV(e_1) \nsubseteq FV(\lambda x : \tau.\ e')$;
2. γ is a pseudo-value substitution such that $\mathsf{dom}(\gamma) = FV(\lambda x : \tau.\ e')$ and $\gamma(e') = e$;
3. the identifiers in γ are *normalized*, namely for any two lambdas $\lambda x : \tau.\ e_1$ and $\lambda x : \tau.\ e_2$ with $normalize(\lambda x : \tau.\ e_1) = (\lambda x : \tau.\ e'_1, \gamma_1)$ and $normalize(\lambda x : \tau.\ e_2) = (\lambda x : \tau.\ e'_2, \gamma_2)$, if there exists an identifier bijection ρ such that $\lambda x : \tau.\ e'_1 = \rho(\lambda x : \tau.\ e'_2)$, then we have $\lambda x : \tau.\ e'_1 = \lambda x : \tau.\ e'_2$.

We say that $\lambda x : \tau.\ e'$ is the *structure* of the lambda $\lambda x : \tau.\ e$, and we will refer to the expressions in the domain of γ as its *structural closures*. Based on this normalization, we can determine statically whether two lambdas may become equal during evaluation simply by considering their structure.

Lemma 7. *For lambdas $\lambda x_1 : \tau_1.\ e_1$, $\lambda x_2 : \tau_2.\ e_2$ with $normalize(\lambda x_i : \tau_i.\ e_i) = (\lambda x_i : \tau_i.\ e'_i, \gamma_i)$ for $i \in \{1,2\}$, and for inputs P_{in}, θ, γ, we have*

1. $\lambda x_1 : \tau_1.\ e'_1 = \lambda x_2 : \tau_2.\ e'_2$ *and*

2. for $x \in FV(\lambda x_1 : \tau_1.\ e'_1)$, $\gamma(\theta(\gamma_1(x) \approx \gamma_2(x))) \rightarrow$ **true**

iff $\gamma(\theta(\lambda x_1 : \tau_1.\ e_1 \approx \lambda x_2 : \tau_2.\ e_2)) \rightarrow$ **true**.

Proof. We start by showing that the left-hand side implies the right-hand side. For free variable $x \in FV(\lambda x_1 : \tau_1.\ e'_1)$, we have $\gamma(\theta(\gamma_1(x))) = \gamma(\theta(\gamma_2(x)))$ by distributivity of θ and γ and the EQUALS-TRUE evaluation rule. Hence, we have $\gamma(\theta(\gamma_1(e_1))) = \gamma(\theta(\gamma_2(e_2)))$ and therefore $\gamma(\theta(\lambda x_1 : \tau_1.\ e_1)) = \gamma(\theta(\lambda x_2 : \tau_2.\ e_2))$ and the EQUALS-TRUE evaluation rule applies.

For the other direction, we know by the EQUALS-TRUE rule and substitution distributivity that $\gamma(\theta(\lambda x_1 : \tau_1.\ e_1)) = \gamma(\theta(\lambda x_2 : \tau_2.\ e_2))$. Now consider the following recursive transformation function applied to e_1 and e_2:

```
def transform($n_1$: $expr$, $n_2$: $expr$): $(expr, id \mapsto pvalue, id \mapsto pvalue)$ =
  if ($n_1 \in pvalue$ && $FV(n_1) \subseteq FV(e_1)$) $(y, \{y \mapsto n_1\}, \{y \mapsto n_2\})$
  else merge(children($n_1$) zip children($n_2$) map transform)
```

In each recursive call, the invariant that $\gamma(\theta(n_1)) = \gamma(\theta(n_2))$ is maintained by distributivity of substitutions. This implies both that $n_1 \in pvalue$ iff $n_2 \in pvalue$ and $FV(n_1) \subseteq FV(e_1)$ iff $FV(n_2) \subseteq FV(e_2)$. We can then show by induction on $\gamma(\theta(e_1))$ that the result $(n', \gamma'_1, \gamma'_2)$ of the call transform(n_1,n_2) is such that n' satisfies condition 1 of normalization for both n_1 and n_2, and γ'_1 and γ'_2 satisfy condition 2. It is also clear that each y can be chosen to satisfy condition 3, for example by using a shared counter in transform. □

Given a normalization *normalize*$(\lambda x : \tau.\ e) = (\lambda x : \tau.\ e', \gamma)$, it is useful to be able to refer to some deterministic ordering of the sequence $y_1, \cdots, y_n$ of free variables within $\lambda x : \tau.\ e'$. Note that such an ordering can be defined by considering the order in which a deterministic traversal of the body would encounter each free variable (for the first time). It is also useful to be able to refer to the type τ_i associated to each y_i, and we will therefore write $y_i : \tau_i$ to signify this. It is important to realize that the formulation of Lemma 7 ensures that the unification of lambda structures is only relevant if the τ_i associated to each free variable agree (otherwise, the equality on the structural closures cannot evaluate to **true**).

We have seen that equality between lambdas has two components, namely the lambda's structure which is statically known, and the structural closures of the lambda, which are dynamic. We can therefore compile closures into tagged structures with an environment and function pointer, where equality is determined through equality of the tags and environments. Note however that structural equality requires type parameter equality as well, which can be ensured either by specialization of named functions, or through runtime type information.

2.2 Extending the Embedding

In this section, we discuss how the embedding and extraction procedures we presented in the previous chapter can be extended to the higher-order language. We assume in this section the existence of Λ the set of all known (and embedded) lambdas and A the set of all known (and embedded) applications. We will see in the next section how these sets are computed and assume for now that these contain exactly all lambdas and applications that appear in the set of embedded expressions. Note that these sets contain *labeled* lambdas and applications and may therefore contain multiple instances of syntactically identical elements. For readability, we generally omit the label when discussing elements of these sets.

The considerations regarding first-class function equality discussed in the previous section show that syntactic equality between lambdas reduces to some statically known component (namely the normalized structure) and a dynamic one (namely the values that will fill the holes in the normalized structure). These considerations hint at a natural embedding of lambdas using algebraic datatypes where the lambda's type determines which algebraic datatypes should be used, the structure determines the constructor, and the pseudo-value substitution determines the arguments to be passed to the constructor.

The algebraic datatype associated to the function type $\tau = \tau_1 \to \tau_2$ can be generated as follows. Consider the set N_τ of lambda structures that correspond to lambdas in Λ with type τ. Note that we drop the labels here and N_τ contains only syntactic expressions (hence no duplicates).

$$N_\tau = \{\lambda x.\, e'_b \mid \lambda x.\, e_b \in \Lambda,\ \lambda x.\, e_b : \tau,\ \mathit{normalize}(\lambda x.\, e_b) = (\lambda x.\, e'_b, \gamma)\}$$

Now consider the elements $\cdots, \lambda x.\, e'_i, \cdots$ of N_τ, the (typed) free variables $\cdots, y_{i,j} : \tau_{i,j}, \cdots$ of each $\lambda x.\, e'_i$, and the type embeddings $\tau_{i,j} \rhd \sigma_{i,j}$. The algebraic datatype is then defined as

$$\textbf{datatype}\ \delta_{\tau_1 \to \tau_2} := \cdots \mid C_{\lambda x.\, e'_i}(\cdots, y_{i,j} : \sigma_{i,j}, \cdots) \mid \cdots \mid \mathsf{Else}(n : \mathbb{N})$$

The $\mathsf{Else}(n : \mathbb{N})$ constructor allows for unknown lambdas whose structure does not fall within the N_τ set. We rely here on some encoding of the natural numbers (either through algebraic datatypes or native support by the SMT solver) to ensure an unbounded number of values are admissible for the Else constructor.

Let us now extend our embedding to handle the new language features presented above. Function types are embedded as the synthetic algebraic datatype presented above which gives us the type embedding rule $P; \Theta \vdash \tau_1 \to \tau_2 \rhd \delta_{\tau_1 \to \tau_2}$. Lambdas are then embedded as algebraic datatype constructors and applications as dispatches through special $\mathit{dispatch}_\tau$ functions that are parametric in τ, the caller's type.

$$\frac{\begin{array}{c} \mathit{normalize}(\lambda x.\ e) = (\lambda x.\ e', \{y_1 \mapsto e_1, \cdots, y_n \mapsto e_n\}) \\ P;\Theta;\Gamma \vdash (b, e_i) \rhd (t_i, \Phi_i) \ \text{ for } 1 \le i \le n \end{array}}{P;\Theta;\Gamma \vdash (b, \lambda x.\ e) \rhd (C_{\lambda x.\ e'}(t_1, \cdots, t_n), \Phi_1 \cup \cdots \cup \Phi_n)}$$

$$\frac{P;\Theta;\Gamma \vdash (b, e_1) \rhd (t_1, \Phi_1) \qquad P;\Theta;\Gamma \vdash (b, e_2) \rhd (t_2, \Phi_2) \qquad P;\Theta;\Gamma \vdash e_1 : \tau}{P;\Theta;\Gamma \vdash (b, e_1\ e_2) \rhd (\mathit{dispatch}_\tau(t_1, t_2), \Phi_1 \cup \Phi_2)}$$

Note that the term and clause set that result from embedding are only well-formed if all embedded lambdas belong to the set Λ that is used to generate the $\delta_{\tau_1 \to \tau_2}$ datatypes. We will show later on that our unfolding procedure ensures that this is indeed always the case.

One should note at this point that at a purely structural level, the set of lambdas that exist within a program P can be computed in advance and is finite. However, polymorphic types allow for an unbounded number of *typed structures*, which compose N_τ. This property disallows a program-wide defunctionalization approach which would encode the program into an equivalent first-order program.

Extraction of a lambda value from the term t with expected type $\tau_1 \to \tau_2$ is handled by distinguishing the cases where 1) there exists a $\lambda x.\ e \in \Lambda$ such that the lambda has type $\tau_1 \to \tau_2$, $M \models \llbracket \lambda x.\ e \rrbracket_b$ and $M \models \llbracket \lambda x.\ e \rrbracket_t \simeq t$, and 2) no such lambda expression exists. In the first case, we have $\llbracket \lambda x.\ e \rrbracket_t = C_{\lambda x.\ e'_b}(t_1, \cdots, t_n)$ and we extract t based on the structure $\lambda x.\ e'_b$ and the extractions of $t_1, \cdots, t_n$. In the second case, we extract t as a special value v such that given another lambda term t' and extraction v' in the model, we have $v = v'$ iff $M \models t \simeq t'$. We therefore introduce a set of embedded lambdas Λ as part of the extraction environment, and handle the first case described above through the following extraction rule.

$$\frac{\begin{array}{c} \lambda x.\ e \in \Lambda \qquad \lambda x.\ e : \tau_1 \to \tau_2 \\ M \models \llbracket \lambda x.\ e \rrbracket_b \qquad M \models \llbracket \lambda x.\ e \rrbracket_t \simeq t \qquad \llbracket \lambda x.\ e \rrbracket_t = C_{\lambda x.\ e'_b}(t_1, \cdots, t_n) \\ FV(\lambda x.\ e'_b) = \{y_1 : \tau_1, \cdots, y_n : \tau_n\} \qquad P;\Theta;\Lambda \vdash v_i \lhd (t_i, \tau_i) \ \text{ for } 1 \le i \le n \end{array}}{P;\Theta;\Lambda \vdash \lambda x.\ e'_b[y_1/v_1, \cdots, y_n/v_n] \lhd (t, \tau)}$$

For the second case, we rely on the fact that terms within the model returned by the underlying SMT solver are in *normal form*, namely given $t_1, t_2 \in M$, we have $M \models t_1 \simeq t_2$ iff $t_1 = t_2$. Consider a term t to be extracted with expected type τ such that the previous extraction rule does not apply. We introduce an identifier f_t which will correspond to a synthetic function definition constructed based on the model M. We can then handle the second case described above through the following extraction rule.

$$\frac{\nexists \lambda x.\ e \in \Lambda.\ \lambda x.\ e : \tau_1 \to \tau_2 \land M \models \llbracket \lambda x.\ e \rrbracket_b \land M \models \llbracket \lambda x.\ e \rrbracket_t \simeq t}{P;\Theta;\Lambda \vdash \lambda x.\ f_t(x) \lhd (t, \tau)}$$

It is important to realize that, by construction, these extensions ensure that given any $t_1, t_2 \in M$ with sort $\delta_{\tau_1 \to \tau_2}$ and extractions $v_1 \lhd t_1, v_2 \lhd t_2$, we have $M \models t_1 \simeq t_2$ iff $v_1 = v_2$.

It remains to discuss the construction of the synthetic f_t function obtained when extracting a function-typed term with expected type $\tau = \tau_1 \to \tau_2$ that did not correspond to a known lambda. Similarly to how extracting lambda values relies on the set of known lambdas Λ, we rely here on the set A of known function applications. Based on this set and the interpretation of $dispatch_\tau$ in M, we construct the definition of f_t. We compute the set of applications corresponding to type τ for which the associated blocker constant holds, and extract the interpretation of $dispatch_\tau$ for each such application into the set I_t.

$$\begin{aligned} I_t = \{\, v_2 \mapsto v \mid\ & (e_1\ e_2) \in A,\ e_1 : \tau, \\ & M \models \llbracket e_1\ e_2 \rrbracket_b,\ M \models \llbracket e_1 \rrbracket_t \simeq t, \\ & v_2 \lhd M(\llbracket e_2 \rrbracket_t),\ v \lhd M(\llbracket e_1\ e_2 \rrbracket_t) \} \end{aligned}$$

Then, given the contents $v_{2,1} \mapsto v_1, \cdots, v_{2,n} \mapsto v_n$ of I_t, we can construct the function f_t.

$$\textbf{def}\ \ f_t(x : \tau_1)\ :\ \tau_2\ \ := \ \textbf{if}\ (x \approx v_{2,1})\ v_1\ \textbf{else}\ \cdots\ \textbf{else if}\ (x \approx v_{2,n})\ v_n\ \textbf{else}\ f_t(x)$$

Note that we have defined f_t as a recursive (and potentially non-terminating) function. However, we will see below that this choice does not impact the validity of value substitutions generated by our procedure. An alternative approach would be to simply use some well-typed value instead of the recursive call $f_t(x)$. Given our current language, such a value is guaranteed to exist for every type.

Based on the new extraction rules presented above, we extend the input extraction to produce program extensions that contain the synthetic f_t function definitions. Note that input extraction now depends on both a set Λ of known lambdas and a set A of known applications.

$$\frac{\begin{array}{c} P_{type} = \{\mathsf{d} \mid T \in \Theta,\ P;\Theta \vdash T \rhd \sigma_T,\ M;P;\Theta \vdash \mathsf{d} \lhd \sigma_T\} \\ P_{def} = \{(\textbf{def}\ \ f_t(x)\ :\ \tau\ \ := \cdots) \mid (x, v) \in \gamma,\ \lambda x.\ f_t(x) \sqsubseteq v\} \\ \theta = \{T \mapsto T \mid T \in \Theta\} \qquad \gamma = \{x \mapsto v \mid (x, \tau) \in \Gamma,\ P;\Theta;\Lambda \vdash v \lhd (M(x), \tau)\} \end{array}}{P;\Theta;\Gamma;\Lambda;A \vdash (P_{type} \cup P_{def}, \theta, \gamma) \lhd M}$$

Example of lambda extraction. In order to demonstrate how lambdas are embedded and extracted, let us consider the following expression that involves lambdas, function-typed variables, as well as applications.

$$e \ = \ \mathsf{f} \not\approx \lambda\mathsf{x} : \mathsf{Nat}.\,\mathsf{Succ}(\mathsf{x}) \ \ \&\& \ \ \mathsf{f}(\mathsf{x}) \approx \mathsf{Succ}(\mathsf{x}) \ \ \&\& \ \ \mathsf{f}(\mathsf{Zero}) \not\approx \mathsf{Zero}$$

Further consider the following program P, set of type variables Θ and typing context Γ. Note that the resulting typing environment $P;\Theta;\Gamma$ is such that $P;\Theta;\Gamma \vdash e : \mathsf{Boolean}$.

$$P = \{\textbf{type}\ \ \mathsf{Nat}\ := \ \mathsf{Succ}(\mathsf{n} : \mathsf{Nat})\ \mid\ \mathsf{Zero}\} \qquad \Theta = \emptyset \qquad \Gamma = \mathsf{x} : \mathsf{Nat}, \mathsf{f} : \mathsf{Nat} \to \mathsf{Nat}$$

Consider the embedding $P;\Theta;\Gamma \vdash (b_e,\, e) \rhd (t_e,\, \Phi_e)$ and clause set $\Phi = \Phi_e \cup \{\, b_e,\, t_e \simeq \mathit{true}\}$. The clause set Φ is given as follows. Note that && is treated as syntactic sugar for an *if*-expression and expression disequality is simply embedded as term disequality.

$$\begin{aligned}
\Phi = \{\, & (b_e \land \mathsf{f} \not\simeq C_{\lambda x\,:\,\mathsf{Nat}.\ \mathsf{Succ}(x)}()) \iff b_1, && (b_1 \land \mathit{dispatch}_{\mathsf{Nat}\to\mathsf{Nat}}(\mathsf{f},\mathsf{x}) \simeq \mathsf{Succ}(\mathsf{x})) \iff b_3,\\
& (b_e \land \mathsf{f} \simeq C_{\lambda x\,:\,\mathsf{Nat}.\ \mathsf{Succ}(x)}()) \iff b_2, && (b_1 \land \mathit{dispatch}_{\mathsf{Nat}\to\mathsf{Nat}}(\mathsf{f},\mathsf{x}) \not\simeq \mathsf{Succ}(\mathsf{x})) \iff b_4,\\
& b_1 \implies t_e \simeq r_2, && b_3 \implies r_2 \simeq \mathit{dispatch}_{\mathsf{Nat}\to\mathsf{Nat}}(\mathsf{f},\mathsf{Zero}) \not\simeq \mathsf{Zero},\\
& b_2 \implies t_e \simeq \mathit{false}, && b_4 \implies r_2 \simeq \mathit{false},\\
& b_e, && t_e \simeq \mathit{true}\,\}
\end{aligned}$$

The sets of known lambdas and applications clearly correspond to $\Lambda = \{\lambda\mathsf{x} : \mathsf{Nat}.\ \mathsf{Succ}(\mathsf{x})\}$ and $A = \{\mathsf{f\ x},\, \mathsf{f\ Zero}\}$. Based on the set of known lambdas, the sort $\delta_{\mathsf{Nat}\to\mathsf{Nat}}$ with respect to which satisfiability of Φ will be considered is constructed as follows.

$$\textbf{datatype}\ \ \delta_{\mathsf{Nat}\to\mathsf{Nat}} := C_{\lambda x\,:\,\mathsf{Nat}.\ \mathsf{Succ}(x)}() \mid \mathsf{Else}(n\colon \mathsf{Nat})$$

The following model M satisfies the clause set Φ.

$$\begin{aligned}
M = \{\, & \mathsf{x} \mapsto \mathsf{Succ}(\mathsf{Zero}), \quad b_e, b_1, b_3, t_e, r_2 \mapsto \mathit{true},\\
& \mathsf{f} \mapsto \mathsf{Else}(\mathsf{Zero}), \quad b_2, b_4 \mapsto \mathit{false},\\
& \mathit{dispatch}_{\mathsf{Nat}\to\mathsf{Nat}}(\mathsf{Else}(\mathsf{Zero}), \mathsf{Succ}(\mathsf{Zero})) \mapsto \mathsf{Succ}(\mathsf{Succ}(\mathsf{Zero})),\\
& \mathit{dispatch}_{\mathsf{Nat}\to\mathsf{Nat}}(\mathsf{Else}(\mathsf{Zero}), \mathsf{Zero}) \mapsto \mathsf{Succ}(\mathsf{Zero})\,\}
\end{aligned}$$

Let us now consider input extraction from the model M. We clearly have $\mathsf{Succ}(\mathsf{Zero}) \lhd M(\mathsf{x})$. For f, the second extraction rule applies as $M \not\models \mathsf{f} \simeq \llbracket \lambda\mathsf{x} : \mathsf{Nat}.\ \mathsf{Succ}(\mathsf{x}) \rrbracket_t$, and we therefore have $\lambda\mathsf{x} : \mathsf{Nat}.\ f_{\mathsf{Else(Zero)}}(\mathsf{x}) \lhd M(\mathsf{f})$. The function $f_{\mathsf{Else(Zero)}}$ is then extracted as follows based on the known applications in A and the interpretation of $\mathit{dispatch}_{\mathsf{Nat}\to\mathsf{Nat}}$ in M.

```
def $f_{\mathsf{Else(Zero)}}(x : \mathsf{Nat}) : \mathsf{Nat}$ :=
  if $(x \approx \mathsf{Succ(Zero)})$ $\mathsf{Succ(Succ(Zero))}$
  else if $(x \approx \mathsf{Zero})$ $\mathsf{Succ(Zero)}$
  else $f_{\mathsf{Else(Zero)}}(x)$
```

We will see later that inputs resulting from procedure are such that if evaluation under these inputs encounters an application $l : (e_1'\ e_2')$, then we have $l : (e_1\ e_2) \in A$ and the associated blocker constant holds in M. As the body of $f_{\mathsf{Else(Zero)}}$ is constructed so that a value will be produced for each application in A whose blocker constant holds, it is clear that the final **else** branch (namely the recursive call) will never be reached during evaluation.

Example with streams. An interesting feature of our lambda extraction through named functions is that it allows for recursion under lambdas in the extracted function. Consider the following generic Stream type definition.

$$\textbf{type}\ \ \mathsf{Stream}[T] := \mathsf{SCons}(\mathsf{head} : T,\ \mathsf{tail} : \mathsf{Unit} \to \mathsf{Stream}[T]) \mid \mathsf{SNil}$$

Given a program P which contains this Stream type definition, we consider the embedding

$$P; T; \mathsf{s} : \mathsf{Stream}[T] \vdash (b_e, \mathsf{s}\ \textbf{match}\ \{\ \mathsf{SCons(h,t)}\ \Rightarrow\ \mathsf{s} \approx \mathsf{t}\ \ \mathsf{SNil}\ \Rightarrow\ \textbf{false}\ \}) \rhd (t_e, \Phi_e)$$

as well as the clause set $\Phi = \Phi_e \cup \{b_e,\ t_e \simeq \mathit{true}\}$ which is given as follows.

$$\begin{aligned}
\Phi = \{\ & (b_e \land \mathit{is\text{-}SCons}_T(\mathsf{s})) \iff b_1, & b_1 &\implies \mathsf{h}_1 \simeq \mathsf{head}_T(\mathsf{s}),\\
& (b_e \land \mathit{is\text{-}SNil}_T(\mathsf{s})) \iff b_2, & b_1 &\implies \mathsf{t}_1 \simeq \mathsf{tail}_T(\mathsf{s}),\\
& b_1 \implies t_e \simeq (\mathsf{s} \simeq \mathit{dispatch}_{\mathsf{Unit}\to\mathsf{Stream}[T]}(\mathsf{t}_1, \mathit{Unit})), & b_2 &\implies t_e \simeq \mathit{false},\\
& b_e, & t_e &\simeq \mathit{true}\ \}
\end{aligned}$$

The clause set Φ ensures that given model $M \models \Phi$, we have $M \models \mathsf{s} \simeq \mathit{dispatch}_{\mathsf{Unit}\to\mathsf{Stream}[T]}(\mathsf{t}_1)$. As the set of known lambdas is empty, we have $M(\mathsf{s}) = \mathsf{SCons}_T(T_1, \mathsf{Else}(n))$ for some T_1 with sort σ_T and n with sort Nat. Recall from Chapter 1 that the type variable T is extracted into some datatype in the resulting program extension and is a concrete type in extraction results. We therefore have $\mathsf{SCons}[T](T_1, \lambda.\ f_{\mathsf{Else}(n)}()) \lhd M(\mathsf{s})$. The set of known applications only contains the application $\mathsf{t}_1\ ()$ and extraction of $f_{\mathsf{Else}(n)}$ results in the following definition.

$$\textbf{def}\ \ f_{\mathsf{Else}(n)}()\ :\ \mathsf{Stream}[T]\ \ :=\ \ \mathsf{SCons}[T](T_1, \lambda u.\ f_{\mathsf{Else}(n)}())$$

Note that we have simplified the *if*-expression structure as the condition is simply **true**. The stream extracted for s thus corresponds to an infinite stream of T_1 values. However, it is clear that evaluation under the extracted inputs will terminate to the value **true** and the function $f_{\mathsf{Else}(n)}$ will be invoked only once (when evaluating t ()).

It is interesting to note here that our embedding and extraction procedures remain applicable when the SNil base case constructor is dropped from the Stream datatype definition, namely

$$\textbf{type}\ \ \mathsf{Stream}[T]\ :=\ \mathsf{SCons}(\mathsf{head} : T,\ \mathsf{tail} : \mathsf{Unit} \to \mathsf{Stream}[T])$$

Indeed, the extraction of function-typed values into synthetic recursive functions allows us to naturally handle infinite structures, as shown in the example above.

Example with negative datatypes. It is important to note at this point that we have imposed no requirement on datatypes to be well-founded. In other words, we allow both non-strictly positive datatypes (as seen above in the stream example) and negative datatypes in our language, and model finding remains applicable.

Consider the program given in Figure 2.3 which defines the syntax of the lambda calculus in the Expr datatype along with an eval function that implements call-by-value evaluation under some given environment. Evaluation results in a Value which consists of a language-level first-class function value. Note that this Value type is not well-founded as the FunVal constructor introduces negative recursion in the type definition. However, this definition remains quite useful since it allows a very natural implementation of lambda calculus evaluation.

```
type Nat := Succ(n: Nat) | Zero
type Expr := Var(n: Nat) | App(caller: Expr, arg: Expr) | Abs(bound: Nat, body: Expr)
type Value := FunVal(f: Value → Value)

def update(env: Nat → Value, key: Nat, value: Value): Nat → Value =
  if (env(key) ≈ value) env else λv: Nat. if (v ≈ key) value else env(v)

def eval(expr: Expr, env: Nat → Value): Value = expr match {
  case Var(n) ⇒ env(n)
  case App(c, arg) ⇒ eval(c, env) match {
    case FunVal(f) ⇒ f(eval(arg, env))
  }
  case Abs(x, body) ⇒ FunVal(λn: Value. eval(body, update(env, x, n)))
}
```

Figure 2.3 – Implementation of lambda calculus evaluation with first-class function values.

Let us first consider some simple property defined in the context of the above program. Consider the following boolean expression with free variables expr, env, v and f, as well as the corresponding (simplified) embedded clause set Φ.

```
eval(expr, env) ≈ v && v ≈ FunVal(f) && f(v) ≈ v
```

$$\Phi = \{\mathsf{eval}(\mathsf{expr},\mathsf{env}) \approx \mathsf{v}, \mathsf{v} \approx \mathsf{FunVal}(\mathsf{f}), \mathit{dispatch}_{\mathsf{Value}\to\mathsf{Value}}(\mathsf{f},\mathsf{v}) \approx \mathsf{v}\}$$

Since the property contains a function call, a certain number of unfolding steps will be required in order to adequately constrain satisfying models. We will discuss unfolding in the higher-order language later in the chapter and simply assume here that we are given some satisfying model $M \models \Phi$ which contains the following interpretations.

$$\begin{aligned} M \supseteq \{ & \mathsf{expr} \mapsto \mathsf{Var(Zero)}, \mathsf{env} \mapsto \mathsf{Else(Zero)}, \mathsf{v} \mapsto \mathsf{FunVal(Else(Zero))}, \mathsf{f} \mapsto \mathsf{Else(Zero)}, \\ & \mathit{dispatch}_{\mathsf{Value}\to\mathsf{Value}}(\mathsf{Else(Zero)}, \mathsf{FunVal(Else(Zero))}) \mapsto \mathsf{FunVal(Else(Zero))}, \\ & \mathit{dispatch}_{\mathsf{Nat}\to\mathsf{Value}}(\mathsf{Else(Zero)}, \mathsf{Zero}) \mapsto \mathsf{FunVal(Else(Zero))} \} \end{aligned}$$

Extraction will then produce the following value substitutions and synthetic definitions.

$$\mathsf{expr} \mapsto \mathsf{Var(Zero)} \qquad \mathsf{env} \mapsto \lambda x.\, f_2(x) \qquad \mathsf{v} \mapsto \mathsf{FunVal}(\lambda y.\, f_1(y)) \qquad \mathsf{f} \mapsto \lambda y.\, f_1(y)$$

$$\textbf{def } f_1(x : \mathsf{Value}) : \mathsf{Value} := \textbf{if } (x \approx \mathsf{FunVal}(\lambda y.\, f_1(y)))\ \mathsf{FunVal}(\lambda y.\, f_1(y))\ \textbf{else } f_1(x)$$

$$\textbf{def } f_2(x : \mathsf{Nat}) : \mathsf{Value} := \textbf{if } (x \approx \mathsf{Zero})\ \mathsf{FunVal}(\lambda y.\, f_1(y))\ \textbf{else } f_2(x)$$

Evaluation of the original boolean expression under the extracted inputs will terminate to **true**, as expected. Clearly, our embedding and extraction procedures do not depend on the well-foundedness of relevant datatype definitions. It is important to note here that allowing negative recursion in datatypes is a non-trivial property that is only rarely featured by counterexample finding and verification techniques.

Let us now consider model finding in the context of a more interesting property. Say we are (rightfully) worried about the termination of the eval function and are trying to find some non-terminating inputs. In this endeavour, assume that following some static analysis, we have generated the boolean-typed expression below based on which non-terminating inputs for the eval function can be extracted.

```
env(x) ≈ FunVal(λn: Value. eval(App(Var(x), arg), update(env, x, n))) &&
env ≈ update(env, x, eval(arg, env))
```

Inputs for which this expression evaluates to **true** can be used to generate an infinite trace involving the eval functions. Indeed, given inputs P_{in}, θ, γ such that the given property evaluates to **true**, evaluation of the function call eval(App(Var(x), arg), env) under these inputs will lead to a trace of the following shape where $n > 0$

$$\gamma(\theta(\mathsf{eval}(\mathsf{App}(\mathsf{Var}(\mathsf{x}),\mathsf{arg}),\mathsf{env}))) \rightarrow^n \gamma(\theta(\mathsf{eval}(\mathsf{App}(\mathsf{Var}(\mathsf{x}),\mathsf{arg}),\mathsf{env}))) \rightarrow^n \cdots$$

Similarly to the previous example, by applying the unfolding procedure we can obtain a satisfying model which leads to the following extracted value substitution.

$$\mathsf{x} \mapsto \mathsf{Zero} \qquad \mathsf{arg} \mapsto \mathsf{Var}(\mathsf{Zero}) \qquad \mathsf{env} \mapsto \lambda x.\ f_{\mathsf{env}}(x)$$

The extracted synthetic function definition associated to f_{env} is then such that

$$f_{\mathsf{env}}(\mathsf{Zero}) = \mathsf{FunVal}(\lambda \mathsf{n}.\ \mathsf{eval}(\mathsf{App}(\mathsf{Var}(\mathsf{Zero}),\mathsf{Var}(\mathsf{Zero})),\mathsf{update}(\lambda y.\ f_{\mathsf{env}}(y),\mathsf{Zero},\mathsf{n})))$$

If we evaluate the substituted expression $\mathsf{eval}(\mathsf{App}(\mathsf{Var}(\mathsf{Zero}),\mathsf{Var}(\mathsf{Zero})),\lambda x.\ f_{\mathsf{env}}(x))$, we will obtain an infinite trace of the shape described above. It is interesting to note the resemblance of the extracted inputs with the well-known omega term (modulo an evaluation context).

2.2.1 Soundness and Completeness

The embedding and extraction rules we presented for our extended language preserve the soundness and completeness properties discussed in the previous Chapter. Similarly to the first-order case, we need a notion of consistency for function application interpretations in order to state these properties. Let us start by extending the $\overset{ext}{\triangleright}$ embedding and $\overset{ext}{\triangleleft}$ extraction procedures to handle lambdas encountered during evaluation. These procedures are again defined with respect to inputs P_{in}, θ, γ. As the $\triangleleft$ extraction procedure relies on the model M and the set of known embedded lambdas Λ, we also rely on these in $\overset{ext}{\triangleleft}$ to ensure that Lemma 2 is preserved. We start by introducing the rules dealing with embedding and extraction of known lambdas.

$$\frac{\begin{array}{c}\lambda x.\, e_b \in \Lambda \qquad M \models \llbracket \lambda x.\, e_b \rrbracket_b \\ \mathit{normalize}(\lambda x.\, e_b) = (\lambda x.\, e'_b, \gamma') \qquad FV(\lambda x.\, e'_b) = \{y_1 : \tau_1, \cdots, y_n : \tau_n\} \\ \mathit{normalize}(\lambda x.\, e_v) = (\theta(\lambda x.\, e'_b), \{y_1 \mapsto v_1, \cdots, y_n \mapsto v_n\}) \qquad P;\Theta \vdash (v_i, \tau_i) \overset{ext}{\rhd} t_i \text{ for } 1 \le i \le n\end{array}}{P;\Theta \vdash (\lambda x.\, e_v, \tau) \overset{ext}{\rhd} C_{\lambda x.\, e'_b}(t_1, \cdots, t_n)}$$

$$\frac{\begin{array}{c}\lambda x.\, e_b \in \Lambda \qquad M \models \llbracket \lambda x.\, e_b \rrbracket_b \qquad \mathit{normalize}(\lambda x.\, e_b) = (\lambda x.\, e'_b, \gamma') \\ FV(\lambda x.\, e'_b) = \{y_1 : \tau_1, \cdots, y_n : \tau_n\} \qquad P;\Theta \vdash v_i \overset{ext}{\lhd} (t_i, \tau_i) \text{ for } 1 \le i \le n\end{array}}{P;\Theta \vdash \theta(\lambda x.\, e'_b)[y_1/v_1, \cdots, y_n/v_n] \overset{ext}{\lhd} (C_{\lambda x.\, e'_b}(t_1, \cdots, t_n), \tau)}$$

In order to ensure that $\overset{ext}{\rhd}$ and $\overset{ext}{\lhd}$ correspond to $\rhd$ and $\lhd$ when dealing with model extractions, we introduce the following rules that deal with lambdas extracted through $\lhd$. We let these rules take precedence over the ones given above when applicable.

$$\frac{P;\Theta;\Lambda \vdash \lambda x.\, f_t(x) \lhd (t, \tau)}{P;\Theta \vdash (\lambda x.\, f_t(x), \tau) \overset{ext}{\rhd} t} \qquad \frac{P;\Theta;\Lambda \vdash \lambda x.\, f_t(x) \lhd (t, \tau)}{P;\Theta \vdash \lambda x.\, f_t(x) \overset{ext}{\lhd} (t, \tau)}$$

Finally, we want to handle embedding and extraction of lambdas with unknown structure. We again rely on an injection $I_\lambda : \mathit{value} \mapsto \mathbb{N}$ into the natural numbers. As with the I_T injections discussed in the previous chapter, such an injection is guaranteed to exist since the set of all lambdas is defined as a syntactic least fixed point and is therefore enumerable. In order to ensure that embeddings are distinct, we require that if we have an extracted function of the shape $f_{\mathsf{Else}(i)}$ in P_{in}, then $i \notin \mathit{range}(I_\lambda)$. Based on this injection, and if no other embedding or extraction rule is applicable, we apply the following rules when dealing with lambdas with unknown structure.

$$\frac{(\lambda x.\, e_v, i) \in I_\lambda}{P;\Theta \vdash (\lambda x.\, e_v, \tau) \overset{ext}{\rhd} \mathsf{Else}_\tau(i)} \qquad \frac{(\lambda x.\, e_v, i) \in I_\lambda}{P;\Theta \vdash \lambda x.\, e_v \overset{ext}{\lhd} (\mathsf{Else}_\tau(i), \tau)}$$

The extended embedding and extraction procedures $\overset{ext}{\rhd}/\overset{ext}{\lhd}$ remain inverse to each other. Furthermore, Lemma 2 still holds and can again be shown by induction on the term being extracted. The embedding and extraction procedures $\rhd/\lhd$ and $\overset{ext}{\rhd}/\overset{ext}{\lhd}$ are no longer equivalent when $\theta = \emptyset$ as $\overset{ext}{\rhd}/\overset{ext}{\lhd}$ can handle unknown lambdas. Note however that once all lambdas that occur during evaluation are known (*i.e.* either they correspond to some lambda in Λ or they were extracted from the model), then equivalence is again given.

Agreement for value substitutions and consistency with function call interpretations remain unchanged in their formulation and are given with respect to the extended $\overset{ext}{\rhd}/\overset{ext}{\lhd}$ procedures. We now introduce a notion of consistency between models and function application interpretations, which is now given with respect to a specific set of inputs, model, set of known lambdas (as $\overset{ext}{\rhd}/\overset{ext}{\lhd}$ now depend on this set) and dispatch symbol interpretation in the model. Again, consistency is given when the interpretation corresponds to the evaluation of the extracted application.

Definition 5. *For program P, model M, inputs P_{in},θ,γ, known lambdas Λ and dispatch symbol interpretation $(dispatch_{\tau_2\to\tau}(t_1,t_2)\mapsto t')\in M$, we say M is consistent with the interpretation if $v_1 \overset{ext}{\lhd} (t_1,\tau_2\to\tau)$, $v_2 \overset{ext}{\lhd} (t_2,\tau_2)$, $v \overset{ext}{\lhd} (t',\tau)$ and $v_1\ v_2 \to^* v$ in $P\cup P_{in}$.*

Similarly to the function call case, we are generally interested in establishing consistency with the embeddings of function applications. Given a model M and function application $l:(e_1\ e_2)$ with associated embedding $\llbracket e_1\ e_2\rrbracket_t = dispatch_\tau(t_1,t_2)$, we say M is consistent with the function application if either $M\models\neg\llbracket e_1\ e_2\rrbracket_b$ or M is consistent with the interpretation $dispatch_\tau(M(t_1),M(t_2))\mapsto M(dispatch_\tau(t_1,t_2))$.

The embedding of our higher-order language remains sound with respect to extracted inputs. Based on the embedding and extraction extensions, as well as consistency with function applications, we can restate Lemmas 3, 4 and 5 for the extended language. As (most) expressions within lambda bodies are not considered in our embedding, our formulations restrict certain expressions to those that have been embedded and thus do not occur under a lambda.

Lemma 3. *For embedding $P;\Theta;\Gamma\vdash(b_e,e)\rhd(t_e,\Phi_e)$, model $M\models\Phi_e\cup\{b_e\}$, inputs P_{in},θ,γ and known lambdas Λ, if M agrees with γ and M is consistent with embedded calls and applications in e, then $\gamma(\theta(e))\to^* v$ for some $v\in value$ and given $e:\tau$, we have $v\overset{ext}{\lhd}(M(t_e),\tau)$.*

Proof. We extend the inductive proof of Lemma 3 given in Chapter 1 with the new cases.

Case $e=e_1\ e_2$: Consider the embedding $t_e = dispatch_{\tau_2\to\tau}(t_1,t_2)$. Given $\gamma(\theta(e_1))\to^*\lambda x.\ e_v$ and $\gamma(\theta(e_2))\to^* v_2$ where $\lambda x.\ e_v, v_2\in value$, we have $\lambda x.\ e_v \overset{ext}{\lhd}(M(t_1),\tau_2\to\tau)$ and $v_2\overset{ext}{\lhd}(M(t_2),\tau_2)$ by induction. Consistency of M with $dispatch_{\tau_2\to\tau}(M(t_1),M(t_2))$ then ensures that $\gamma(\theta(e))\to^* v$ and $v\overset{ext}{\lhd}(M(dispatch_{\tau_2\to\tau}(t_1,t_2)),\tau)$.

Case $e=\lambda x.\ e_b$: We have $normalize(\lambda x.\ e_b) = (\lambda x.\ e'_b, \{y_1\mapsto e_1,\cdots,y_n\mapsto e_n\})$ and given embeddings $(b_e,e_i)\rhd(t_i,\Phi_i)$ for $1\le i\le n$, we have $t_e = C_{\lambda x.\ e'_b}(t_1,\cdots,t_n)$. By induction, and given the type τ_i associated to y_i, we have $\gamma(\theta(e_i))\to^* v_i$ where $v_i\overset{ext}{\lhd}(M(t_i),\tau_i)$ for $1\le i\le n$. By Lemma 7, we have $\gamma(\theta(\lambda x.\ e_b))\to^*\lambda x.\ e'_b[y_1/v_1,\cdots,y_n/v_n]$. As e was embedded, we have $e\in\Lambda$, and by definition of extraction, we therefore have $\lambda x.\ e'_b[y_1/v_1,\cdots,y_n/v_n]\overset{ext}{\lhd}(M(t_e),\tau)$. □

Lemma 4. *For embedding $P;\Theta;\Gamma\vdash(b_e,e)\rhd(t_e,\Phi_e)$, model $M\models\Phi_e\cup\{b_e\}$, inputs P_{in},θ,γ, known lambdas Λ and embedded sub-expression $l_1:e_1\sqsubseteq e$, if $\gamma(\theta(e))\to^n\mathcal{E}[l_1:e'_1]$, M agrees with γ and M is consistent with embedded calls and applications in e that are evaluated in the trace, then $M\models\llbracket l_1:e_1\rrbracket_b$.*

Proof. The proof is analogous to the one given for Lemma 4 in Chapter 1. □

Lemma 5. *For embedding $P;\Theta;\Gamma\vdash(b_e,e)\rhd(t_e,\Phi_e)$, model $M\models\Phi_e\cup\{b_e\}$, inputs P_{in},θ,γ, known lambdas Λ and embedded sub-expression $l_1:e_1\sqsubseteq e$, if $\gamma(\theta(e))\to^* v\in value$, M agrees* *with γ and M is consistent with embedded calls and applications in e, then $\gamma(\theta(e)) \to^* \mathcal{E}[l_1 : e'_1]$ iff $M \models \llbracket l_1 : e_1 \rrbracket_b$.*

Proof. The proof is again analogous to the one given for Lemma 5 in Chapter 1. □

Finally, the extension to our higher-order language preserves the completeness property of our embedding.

Lemma 6. *For embedding $P;\Theta;\Gamma \vdash (b_e, e) \rhd (t_e, \Phi_e)$, inputs P_{in}, θ, γ and known lambdas Λ, if $\gamma(\theta(e)) \to^* v \in value$, then there exists model $M \models \Phi_e \cup \{b_e\}$ such that M agrees with γ, M is consistent with embedded calls and applications in e and given $e : \tau$, we have $v \overset{ext}{\lhd} (M(t_e), \tau)$.*

Proof. As before, we follow the proof of Lemma 6 exposed in Chapter 1. The induction is extended with the following new cases.

Case $e = e_1\ e_2$: Consider values $\gamma(\theta(e_1)) \to^* \lambda x.\ e_v$ and $\gamma(\theta(e_2)) \to^* v_2$. Further consider embeddings $(b_e, e_1) \rhd (t_1, \Phi_1)$ and $(b_e, e_2) \rhd (t_2, \Phi_2)$. Given the type $e_1 : \tau_2 \to \tau$, by induction there exist models $M_i \models \Phi_i \cup \{b_e\}$ for $1 \le i \le 2$ such that M_1 and M_2 agree with γ, M_1 (respectively M_2) is consistent with embedded calls and applications in e_1 (respectively e_2), $\lambda x.\ e_v \overset{ext}{\lhd} (M_1(t_1), \tau_2 \to \tau)$ and $v_2 \overset{ext}{\lhd} (M_2(t_2), \tau_2)$. Now consider embeddings $(\lambda x.\ e_v, \tau_2 \to \tau) \overset{ext}{\rhd} t'_c$, $(v_2, \tau_2) \overset{ext}{\rhd} t'_2$ and $(v, \tau) \overset{ext}{\rhd} t_v$. As the models M_1 and M_2 agree on common interpretations, the union of models $M = M_1 \cup M_2 \cup \{dispatch_{\tau_2 \to \tau}(t'_1, t'_2) \mapsto t_v\}$ is such that $M \models \Phi_e \cup \{b_e\}$, M agrees with γ, M is consistent with embedded calls and applications in e, and we have $v \overset{ext}{\lhd} (M(t_e), \tau)$.

Case $e = \lambda x.\ e_b$: Consider $normalize(\lambda x.\ e_b) = (\lambda x.\ e'_b, \{y_1 \mapsto e_1, \cdots, y_n \mapsto e_n\})$. Given the type τ_i associated to y_i and value $\gamma(\theta(e_i)) \to^* v_i$, consider embeddings $(b_e, e_i) \rhd (t_i, \Phi_i)$ for $1 \le i \le n$. By induction, there exists a model $M_i \models \Phi_i \cup \{b_e\}$ such that M_i agrees with γ, M_i is consistent with embedded calls and applications in e_i and $v_i \overset{ext}{\lhd} (M_i(t_i), \tau_i)$ for $1 \le i \le n$. Again, all constants introduced during embedding are fresh and we can unify $M_1, \cdots, M_n$ into M. By definition of $\overset{ext}{\lhd}$ for known lambdas, we have $v \overset{ext}{\lhd} (M(t_e), \tau)$. □

2.3 Unfolding Applications

We have already seen how to relax the constraint on function calls in our approach to generating value substitutions, and we shall now present a transformation that allows one to find value substitutions in the presence of function applications as well. We will then show that this procedure, in conjunction with the one presented in Section 1.4, remains complete for finding value substitutions in the higher-order language.

Similarly to how labels within inlined function bodies are computed based on the call site label and the original label in the function's body, labels within inlined lambda bodies are computed based on the application label and original label in the lambda's body. This again ensures that labels which will occur during evaluation can be statically determined by considering the application label and labels within some lambda expression's body. If that particular lambda ends up flowing into the caller position during evaluation, then the statically computed labels will match up with those encountered during evaluation.

We discussed how function calls can be progressively inlined to increase the space of the program that can be visited during evaluation under value substitutions. However, the same technique cannot be directly applied to first-class function applications. Indeed, the callee function is not statically known at application point and it's body is therefore unknown. What we do know statically is the set of all lambdas within the current expression. Let us consider some application $l : (e_1\ e_2) \sqsubseteq e$ and inputs P_{in}, θ, γ under which evaluation encounters l and terminates, namely $\gamma(\theta(e)) \to^* \mathcal{E}[l : (e'_1\ e'_2)] \to^* v \in \mathit{value}$. Given the context evaluation rules, we therefore have $\mathcal{E}[e'_1\ e'_2] \to^* \mathcal{E}[(\lambda x.\ e_v)\ e'_2]$, or, more precisely, $e'_1 \to^* \lambda x.\ e_v$. If evaluation of $\gamma(\theta(e))$ encounters no function call and no other function application, then clearly there must exist either a $l_b : \lambda x.\ e_b \sqsubseteq e$ such that $\gamma(\theta(e)) \to^* \mathcal{E}[l_b : \lambda x.\ e_v]$, or a variable $x \in FV(e)$ such that $\lambda x.\ e_v \sqsubseteq \gamma(x)$. In other words, function applications can be correctly dispatched by considering the existing lambdas in e and carefully extracting those within the value substitutions.

We will present the inlining procedure directly at the level of the SMT terms and formulas. Recall Λ the set of known lambdas and A the set of lambda applications. Let us assume we are given some lambda $\lambda x.\ e_b \in \Lambda$ with type τ and some application $(e_1\ e_2) \in A$ where the caller e_1 has type τ as well. We then compute the structural normalization of the lambda $\mathit{normalize}(\lambda x.\ e_b) = (\lambda x.\ e'_b, \{y_1 \mapsto e'_1, \cdots, y_n \mapsto e'_n\})$, and given a fresh constant b_b, we compute the embedding of the normalized lambda's body $(b_b, e'_b) \rhd (t_b, \Phi_b)$. Note that the lambdas and applications that occur within e'_b must be added to the sets Λ and A of known lambdas and applications. The δ_τ datatype definitions will thus ensure that constructors for lambdas in e'_b exist. Let us now define the clause set

$$\begin{aligned}\Phi_{disp} = \Phi_b \cup \{\, & b_b \iff (\llbracket \lambda x.\ e_b \rrbracket_b \wedge \llbracket e_1\ e_2 \rrbracket_b \wedge \llbracket \lambda x.\ e_b \rrbracket_t \simeq \llbracket e_1 \rrbracket_t), \\ & b_b \implies \llbracket e_1\ e_2 \rrbracket_t \simeq t_b,\ b_b \implies x \simeq \llbracket e_2 \rrbracket_t, \\ & b_b \implies y_1 \simeq \llbracket e'_1 \rrbracket_t, \cdots, b_b \implies y_n \simeq \llbracket e'_n \rrbracket_t \}\end{aligned}$$

Similarly to Φ_{inl} defined in the previous chapter, this Φ_{disp} clause set ensures that satisfying models will be consistent with the function application $e_1\ e_2$ (as long as M is consistent with all other applications). We let $\mathsf{unfold}(e_1\ e_2, \lambda x.\ e_b) = \Phi_{disp}$.

It is important to note at this point that it is not possible to generalize this dispatch procedure to all lambdas within Λ as each dispatch may increase Λ and a fixpoint does not necessarily exist. It is therefore not possible to *completely* dispatch a given function application, which motivates to the following section on blocking applications.

2.4 Blocking Applications

As in the case of function calls, we would like to define a clause for which satisfying models ensure that evaluation under the extracted value substitution will only encounter applications with which the model is consistent. While the set of relevant applications is statically known, it is impossible to statically determine when sufficient unfolding has occurred and the application can be unblocked. Indeed, as seen above, the set Λ of potential targets keeps growing as more unfoldings occur. We must therefore rely on a more *dynamic* property to determine when a certain function application may be unblocked.

The key insight behind our application blocking approach is that the set of *potential* dispatch targets is known. If the caller corresponds to some lambda within Λ which has not yet been unfolded (for the given application), then the application should be blocked. Note that there are in fact four possible cases to consider:

1. The *right* lambda has already been unfolded: in this case, there is no need to block the application as we have already ensured potential models are consistent with the application.
2. The *right* lambda is known but has not yet been unfolded: the application will be blocked and evaluation under value substitutions extracted from satisfying models will not encounter the application.
3. The *right* lambda is not yet known: if the lambda does not yet belong to the set of known lambdas, then some function call or application must be unfolded for it to appear. There must therefore exist some (blocked) call or application which disallows models that would lead to evaluation of the application.
4. The caller corresponds to a free variable in the expression: if the caller does not correspond to any known lambdas and will appear in the extracted value substitution, our extraction procedure will produce a lambda for which the model is consistent with the application.

In a more formal sense, given an application $(e_1\ e_2) \in A$ where the caller e_1 has type τ and the set $\Lambda_u \subseteq \Lambda$ of lambdas that have been unfolded for the application, consider the definition:

$$\begin{array}{rcl} \mathsf{block}(e_1\ e_2, \Lambda, \Lambda_u) & = & \bigvee_{\lambda x.\ e_b \in \Lambda,\ \lambda x.\ e_b\ :\ \tau} \llbracket \lambda x.\ e_b \rrbracket_b \wedge \\ & & \llbracket e_1 \rrbracket_t \simeq \llbracket \lambda x.\ e_b \rrbracket_t \wedge \\ & & \bigwedge_{\lambda y.\ e_u \in \Lambda_u} \neg(\llbracket \lambda y.\ e_u \rrbracket_b \wedge \llbracket \lambda x.\ e_b \rrbracket_t \simeq \llbracket \lambda y.\ e_u \rrbracket_t) \\ & \implies & \neg \llbracket e_1\ e_2 \rrbracket_b \end{array}$$

Conceptually, this clause ensures that $(\llbracket e_1 \rrbracket_t \in \llbracket \Lambda \rrbracket_t \setminus \llbracket \Lambda_u \rrbracket_t) \implies \neg \llbracket e_1\ e_2 \rrbracket_b$. In other words, the blocker constant for $e_1\ e_2$ will be falsified as long as e_1 corresponds to a known lambda which we have not yet unfolded for the application. Given the sets A of known applications, Λ

of known lambdas, and $D \subseteq A \times \Lambda$ of lambda unfoldings that have occurred, we extend the block function to the triplet A, Λ, D as follows:

$$\mathsf{block}(A, \Lambda, D) = \bigcup_{(e_1\ e_2) \in A} \mathsf{block}(e_1\ e_2, \Lambda, \{e_\lambda \mid (e_1\ e_2, e_\lambda) \in D\})$$

2.5 Unfolding Procedure

At this point, we have set up all the necessary components of the complete unfolding procedure. As in the previous chapter, our procedure is incremental and relies on a clause set and some instrumentation that is used to generate further clauses. We again keep track of the clause set Φ_i and the set F_i of known function calls that have yet to be unfolded. We additionally introduce the sets A_i of known function applications, Λ_i of known lambdas, and $D_i \subseteq A_i \times \Lambda_i$ of function application - lambda pairs that have already been unfolded. Given a function type τ, we will write $A_{i,\tau}$ for the subset of A_i where the caller has type τ, and $\Lambda_{i,\tau}$ for the subset of Λ_i where the lambda has type τ. Note that as with function calls in F_i, embedding has been performed for each application in A_i and lambda in Λ_i, so $\llbracket\cdot\rrbracket_b$ and $\llbracket\cdot\rrbracket_t$ are well defined on members of these sets.

The complete procedure alternates between unfolding function calls within F_i and function application - lambda pairs within $A_i \times \Lambda_i$. This enables us to incrementally explore the space of models that are consistent with an increasing number of calls and applications. At each step i, we can construct a set of *blocking clauses* that ensures satisfying models will lead to the expected evaluation trace.

For expression e, value v, and fresh constant b, consider the embedding $(b, e \approx v) \rhd (t, \Phi)$. We rely on the equality expression here to ensure that lambdas within v are known during embedding and extraction. Similarly to the function $F(\cdot)$ defined in the previous chapter, we define the following helper functions that respectively collect all embedded applications and all embedded lambdas within a given expression:

$$\Lambda(e) = \{\lambda x.\ e_b \mid \lambda x.\ e_b \sqsubseteq e, \llbracket\lambda x.\ e_b\rrbracket_t \text{ defined}\} \qquad A(e) = \{e_1\ e_2 \mid e_1\ e_2 \sqsubseteq e, \llbracket e_1\ e_2\rrbracket_t \text{ defined}\}$$

We then start by defining $\Phi_0 = \Phi \cup \{b, t\}$, $F_0 = F(e \approx v)$, $A_0 = A(e \approx v)$, $\Lambda_0 = \Lambda(e \approx v)$, and $D_0 = \emptyset$. The sets $\Phi_{i+1}, F_{i+1}, A_{i+1}, \Lambda_{i+1}, D_{i+1}$ are then inductively defined given $\Phi_i, F_i, A_i, \Lambda_i, D_i$ by either performing a function call or application unfolding step.

The call unfolding step is performed as discussed in the previous chapter. First, some function call $f[\overline{\tau}](e_1)_i \in F_i$ with corresponding definition $(\textbf{def}\ \ f[\overline{\tau}](x : \tau_1) : \tau_2\ := \ e_f) \in P$ is selected. Second, the clause set is extended with the new unfolding clauses, namely $\Phi_{i+1} = \Phi_i \cup \mathsf{unfold}(f[\overline{\tau}](e_1)_i)$. The unfolded call is then removed from the function call set and the new calls in the body e_f are added, giving us $F_{i+1} = F_i \setminus \{f[\overline{\tau}](e_1)_i\} \cup F(e_f[\overline{\tau}_f/\overline{\tau}])$. For the sets A_{i+1} and Λ_{i+1}, we simply record the new applications and lambdas that appeared in e_f, namely $A_{i+1} = A_i \cup A(e_f[\overline{\tau}_f/\overline{\tau}])$ and $\Lambda_{i+1} = \Lambda_i \cup \Lambda(e_f[\overline{\tau}_f/\overline{\tau}])$. Finally, the set of unfolded application - lambda pairs remains unchanged as no application unfolding has occurred in this step: $D_{i+1} = D_i$.

For the alternative application unfolding step, we select a pair of application $(e_1\ e_2) \in A_i$ and lambda $\lambda x.\ e_b \in \Lambda_i$ such that $e_1 : \tau$ and $\lambda x.\ e_b : \tau$ for some function type τ. The clause set is then extended with the application unfolding clauses, $\Phi_{i+1} = \Phi_i \cup \mathsf{unfold}(e_1\ e_2, \lambda x.\ e_b)$. Recall that application unfolding proceeds by embedding the normalized body e'_b of $\lambda x.\ e_b$, and the known calls, applications, and lambdas are therefore updated accordingly, giving us $F_{i+1} = F_i \cup F(e'_b)$, $A_{i+1} = A_i \cup A(e'_b)$ and $\Lambda_{i+1} = \Lambda_i \cup \Lambda(e'_b)$. Finally, the new application - lambda pair is added to the set of unfolded pairs: $D_{i+1} = D_i \cup \{(e_1\ e_2, \lambda x.\ e_b)\}$.

It is important to note that the extended unfolding procedure preserves the structure presented in Algorithm 1 of Section 1.5. Indeed, supporting higher-order functions only requires tracking a little more state and alternating application unfolding steps with the previous call unfoldings.

Example unfolding. The exists-forall correspondance property stated in the beginning of the chapter leads to the following embedding.

$$\begin{array}{l}
P; T; \mathsf{p} : T \rightarrow \mathsf{Boolean}, \mathsf{x} : T, \mathsf{xs} : \mathsf{List}[T] \vdash \\
\quad (b_e, \mathsf{existsForall}[T](\mathsf{xs}, \mathsf{p}) \implies \mathsf{existsForall}[T](\mathsf{Cons}(\mathsf{x}, \mathsf{xs}), \mathsf{p})) \rhd \\
\qquad (r_1, \{\, (b_e \land \mathsf{existsForall}_T(\mathsf{xs}, \mathsf{p})) \iff b_1, \\
\qquad\quad (b_e \land \neg\mathsf{existsForall}_T(\mathsf{xs}, \mathsf{p})) \iff b_2, \\
\qquad\quad b_1 \implies r_1 \simeq \mathsf{existsForall}_T(\mathsf{Cons}_T(\mathsf{x}, \mathsf{xs}), \mathsf{p}), \\
\qquad\quad b_2 \implies r_1 \simeq \mathit{false}\,\})
\end{array}$$

Note that the constant p which appears in the embedded clause set has sort $\delta_{T\rightarrow\mathsf{Boolean}}$ and given that the initial set of known lambdas $\Lambda_{T\rightarrow\mathsf{Boolean}}$ is empty, the datatype is defined as

$$\textbf{datatype}\ \delta_{T\rightarrow\mathsf{Boolean}} := \mathsf{Else}(n : \mathbb{N})$$

A possible sequence of unfolding steps can then be found in Figure 2.4. After a few additional unfoldings, a satisfying model will exist from which a counterexample can be extracted. Similarly to the example given in Chapter 1, if we fix the existsForall property definition, then the unfolding procedure will terminate and show that no counterexample exists.

2.5.1 Soundness

The unfolding procedure we presented for our higher-order language preserves soundness of reported inputs. Indeed, the clause sets given by $\mathsf{block}(U_i)$ and $\mathsf{block}(A_i, \Lambda_i, D_i)$ again ensure that traces resulting from model extractions will evaluate to the expected value v. This leads to the following re-formulation of Theorem 1.

Theorem 1. *For expression e, value v, unfolding step $i \in \mathbb{N}$ and model $M \models \Phi_i \cup \mathsf{block}(F_i) \cup \mathsf{block}(A_i, \Lambda_i, D_i)$, given extraction $(P_{in}, \theta, \gamma) \lhd M$, we have $\gamma(\theta(e)) \rightarrow^* \theta(v)$.*

i	$\Phi_i \setminus \Phi_{i-1}$	F_i	A_i	Λ_i
0	$\{b_e, r_1 \simeq \mathit{false},$ $(b_e \wedge \mathsf{existsForall}_T(\mathsf{xs},\mathsf{p})) \iff b_1,$ $(b_e \wedge \neg\mathsf{existsForall}_T(\mathsf{xs}),\mathsf{p}) \iff b_2,$ $b_1 \implies r_1 \simeq \mathsf{existsForall}_T(\mathsf{Cons}_T(\mathsf{x},\mathsf{xs}),\mathsf{p}),$ $b_2 \implies r_1 \simeq \mathit{false}\}$	$\{\mathsf{existsForall}[T](\mathsf{xs},\mathsf{p}),$ $\mathsf{existsForall}[T](\mathsf{Cons}[T](\mathsf{x},\mathsf{xs}),\mathsf{p})\}$	$\emptyset$	$\emptyset$
	Unfolding call $\mathsf{existsForall}[T](\mathsf{xs},\mathsf{p}) \in F_0$			
1	$\{b_e \implies \mathsf{existsForall}_T(\mathsf{xs},\mathsf{p}) \simeq$ $(\mathsf{exists}_T(\mathsf{l}_1,\mathsf{p}_1) \simeq \neg\mathsf{forall}_T(\mathsf{l}_1, C_\lambda(\mathsf{p}_1))),$ $b_e \implies \mathsf{l}_1 \simeq \mathsf{xs},$ $b_e \implies \mathsf{p}_1 \simeq \mathsf{p}\}$	$\{\mathsf{existsForall}[T](\mathsf{Cons}[T](\mathsf{x},\mathsf{xs}),\mathsf{p}),$ $\mathsf{exists}[T](\mathsf{l}_1,\mathsf{p}_1),$ $\mathsf{forall}[T](\mathsf{l}_1,\lambda x : T.\ \mathsf{p}_1(x))\}$	$\emptyset$	$\{\lambda x : T.\ \mathsf{p}_1(x)\}$
	Unfolding call $\mathsf{existsForall}[T](\mathsf{Cons}[T](\mathsf{x},\mathsf{xs}),\mathsf{p}) \in F_1$			
2	$\{b_1 \implies \mathsf{existsForall}_T(\mathsf{Cons}[T](\mathsf{x},\mathsf{xs}),\mathsf{p}) \simeq$ $(\mathsf{exists}_T(\mathsf{l}_2,\mathsf{p}_2) \simeq \neg\mathsf{forall}_T(\mathsf{l}_2, C_\lambda(\mathsf{p}_2))),$ $b_1 \implies \mathsf{l}_2 \simeq \mathsf{Cons}_T(\mathsf{x},\mathsf{xs}),$ $b_1 \implies \mathsf{p}_2 \simeq \mathsf{p}\}$	$\{\mathsf{exists}[T](\mathsf{l}_1,\mathsf{p}_1),$ $\mathsf{forall}[T](\mathsf{l}_1,\lambda x : T.\ \mathsf{p}_1(x)),$ $\mathsf{exists}[T](\mathsf{l}_2,\mathsf{p}_2),$ $\mathsf{forall}[T](\mathsf{l}_2,\lambda x : T.\ \mathsf{p}_2(x))\}$	$\emptyset$	$\{\lambda x : T.\ \mathsf{p}_1(x),$ $\lambda x : T.\ \mathsf{p}_2(x)\}$
	Unfolding call $\mathsf{forall}[T](\mathsf{l}_1,\lambda x : T.\ \mathsf{p}_1(x)) \in F_2$			
3	$\{b_e \implies \mathsf{forall}_T(\mathsf{l}_1, C_\lambda(\mathsf{p}_1)) \simeq r_2,$ $b_e \implies \mathsf{l}_3 \simeq \mathsf{l}_1,$ $b_e \implies \mathsf{p}_3 \simeq C_\lambda(\mathsf{p}_1),$ $(b_e \wedge \mathit{is\text{-}Cons}_T(\mathsf{l}_3)) \iff b_3,$ $(b_e \wedge \mathit{is\text{-}Nil}_T(\mathsf{l}_3)) \iff b_4,$ $b_3 \implies \mathsf{x}_1 \simeq \mathsf{head}_T(\mathsf{l}_3),$ $b_3 \implies \mathsf{xs}_1 \simeq \mathsf{tail}_T(\mathsf{l}_3),$ $b_3 \implies r_2 \simeq r_3,$ $b_4 \implies r_2 \simeq \mathit{true},$ $(b_3 \wedge \mathit{dispatch}_{T\to\mathsf{Boolean}}(\mathsf{p}_3,\mathsf{x}_1)) \iff b_5,$ $(b_3 \wedge \neg\mathit{dispatch}_{T\to\mathsf{Boolean}}(\mathsf{p}_3,\mathsf{x}_1)) \iff b_6,$ $b_5 \implies r_3 \simeq \mathsf{forall}_T(\mathsf{xs}_1,\mathsf{p}_3),$ $b_6 \implies r_3 \simeq \mathit{false}\}$	$\{\mathsf{exists}[T](\mathsf{l}_1,\mathsf{p}_1),$ $\mathsf{exists}[T](\mathsf{l}_2,\mathsf{p}_2),$ $\mathsf{forall}[T](\mathsf{l}_2,\lambda x : T.\ \mathsf{p}_2(x)),$ $\mathsf{forall}[T](\mathsf{xs}_1,\mathsf{p}_3)\}$	$\{\mathsf{p}_3(\mathsf{x}_1)\}$	$\{\lambda x : T.\ \mathsf{p}_1(x),$ $\lambda x : T.\ \mathsf{p}_2(x)\}$
	Unfolding pair $(\mathsf{p}_3(x), \lambda x : T.\ \mathsf{p}_1(x)) \in A_3 \times \Lambda_3$			
4	$\{b_7 \iff (b_e \wedge b_3 \wedge \mathsf{p}_3 \simeq C_\lambda(\mathsf{p}_1)),$ $b_7 \implies \mathit{dispatch}_{T\to\mathsf{Boolean}}(\mathsf{p}_3,\mathsf{x}_1) \simeq$ $\mathit{dispatch}_{T\to\mathsf{Boolean}}(\mathsf{y}_1,\mathsf{x}_2),$ $b_7 \implies \mathsf{y}_1 \simeq \mathsf{p}_1,$ $b_7 \implies \mathsf{x}_2 \simeq \mathsf{x}_1\}$	$\{\mathsf{exists}[T](\mathsf{l}_1,\mathsf{p}_1),$ $\mathsf{exists}[T](\mathsf{l}_2,\mathsf{p}_2),$ $\mathsf{forall}[T](\mathsf{l}_2,\lambda x : T.\ \mathsf{p}_2(x)),$ $\mathsf{forall}[T](\mathsf{xs}_1,\mathsf{p}_3)\}$	$\{\mathsf{p}_3(\mathsf{x}_1),$ $\mathsf{y}_1(\mathsf{x}_2)\}$	$\{\lambda x : T.\ \mathsf{p}_1(x),$ $\lambda x : T.\ \mathsf{p}_2(x)\}$

Figure 2.4 – A possible sequence of unfolding steps during counterexample search for $\mathsf{existsForall}[T](\mathsf{xs},\mathsf{p}) \implies \mathsf{existsForall}[T](\mathsf{Cons}[T](\mathsf{x},\mathsf{xs}),\mathsf{p})$. Note that an extra constructor $C_{\lambda x.\ y(x)}(y : \delta_{T\to\mathsf{Boolean}})$ will be introduced in the $\delta_{T\to\mathsf{Boolean}}$ algebraic datatype definition from step 1 onwards since a new lambda structure appears in the set of known lambdas.

Proof. The proof follows a similar structure to the proof of Theorem 1 in Chapter 1.

First, we extend our notion of *unfolding tree* to function applications. We start by adding the function applications in A_i to the set of nodes. For function call nodes, we then introduce edges between the call and the nodes which occur within the function's unfolded body, for application nodes, we introduce edges between the application and the nodes which occur within the bodies of all lambdas that have been unfolded for that application, and for the root node, we introduce edges between the root and all calls and applications that occur within e. We again rely on the partial ordering of sibling nodes based on evaluation order.

For trace $\gamma(\theta(e)) \rightarrow^{n_1} e'$, we then show by bottom-up induction on the unfolding tree and on the partial order between sibling nodes that for each node $l : e_t$ in the tree, if

1. $\gamma(\theta(e)) \rightarrow^{n_2} \mathcal{E}[l : e'_t] \rightarrow^{n_1 - n_2} e'$,
2. $M \models \llbracket l : e_t \rrbracket_b$,
3. whenever $e_t = f[\overline{\tau}](e_1)$ with associated definition $(\mathbf{def}\ f[\overline{\tau}_f](x : \tau_1) : \tau_2 \ := \ e_f) \in P$ and $\gamma(\theta(e)) \rightarrow^{n_3} \mathcal{E}[l : f[\overline{\tau}](v_1)] \rightarrow^{n_1 - n_3} e'$, then $v_1 \overset{ext}{\triangleleft} (M(\llbracket e_1 \rrbracket_t), \tau_1[\overline{\tau}_f / \overline{\tau}])$, and
4. whenever $e_t = e_1\ e_2$ and $\gamma(\theta(e)) \rightarrow^{n_3} \mathcal{E}[l : (\lambda x.\ e_v)\ v_2] \rightarrow^{n_1 - n_3} e'$, then given the function type $e_1 : \tau_2 \rightarrow \tau$, we have $\lambda x.\ e_v \overset{ext}{\triangleleft} (M(\llbracket e_1 \rrbracket_t), \tau_2 \rightarrow \tau)$ and $v_2 \overset{ext}{\triangleleft} (M(\llbracket e_2 \rrbracket_t), \tau_2)$,

then given the maximal trace $e'_t \rightarrow^{n_4} e_v$ for $n_4 \le n_1 - n_2$, M is consistent with all calls and applications that are evaluated in the trace, each call that is encountered in the trace belongs to U_i, and each application that is encountered in the trace belongs to A_i.

The cases for the root and call nodes follow as in the proof presented in Chapter 1. We therefore consider the case where $e_t = e_c\ e_2$. For the trace $\gamma(\theta(e)) \rightarrow^{n_2} \mathcal{E}[l : e'_1\ e'_2]$, consistency is again given by induction on the partial order. We then consider the interesting case where $\gamma(\theta(e)) \rightarrow^{n_3} \mathcal{E}[l : (\lambda x.\ e_v)\ v_2] \rightarrow \mathcal{E}[e_v[x / v_2]]$ and $n_3 < n_1$, separating the analysis into three cases: 1) there exists $\lambda x.\ e_b \in \Lambda_i$ such that $M \models \llbracket \lambda x.\ e_b \rrbracket_b \wedge \llbracket \lambda x.\ e_b \rrbracket_t \simeq \llbracket e_1 \rrbracket_t$, 2) we have $\lambda x.\ f_t(x) \overset{ext}{\triangleleft} (M(\llbracket e_1 \rrbracket_t), \tau_2 \rightarrow \tau)$, and 3) neither case 1) nor 2) holds.

Case 1) By $M \models \mathsf{block}(A_i, \Lambda_i, D_i)$, we have $(e_1\ e_2, \lambda x.\ e_b) \in D_i$. By definition of Φ_i, given *normalize*$(\lambda x.\ e_b) = (\lambda x.\ e'_b, \{y_1 \mapsto e'_1, \cdots, y_n \mapsto e'_n\})$, we have $M \models \llbracket e_1\ e_2 \rrbracket_t \simeq \llbracket e'_b \rrbracket_t$, $M \models x \simeq \llbracket e_2 \rrbracket_t$ and $M \models y_j \simeq \llbracket e'_j \rrbracket_t$ for $1 \le j \le n$. Given $FV(\lambda x.\ e'_b) = \{y_1 : \tau'_1, \cdots, y_n : \tau'_n\}$ and $v'_j \overset{ext}{\triangleleft} (M(\llbracket e'_j \rrbracket_t), \tau'_j)$ for $1 \le j \le n$, we have $\lambda x.\ e_v = \lambda x.\ e'_b[y_1 / v'_1, \cdots, y_n / v'_n]$ by definition of $\overset{ext}{\triangleleft}$. Given the value substitution $\gamma' = \{x \mapsto v_2\} \cup \{y_j \mapsto v'_j \mid 1 \le j \le n\}$, M therefore agrees with γ' by condition 4 and $\gamma(\theta(e)) \rightarrow^{n_3 + 1} \mathcal{E}[\gamma'(\theta(e'_b))]$ by the operational semantics. Similarly to the proof in the previous chapter, we then rely on the sibling partial order induction to apply Lemmas 3, 4 and Corollary 2 in order to ensure both consistency and inclusion.

Case 2) By construction of $\lambda x.\ f_t(x)$, M is consistent with the application. As no call or application exists within the extracted body of f_t, we also have both consistency and inclusion of all calls and applications in the trace.

Case 3) The remaining possibility is that $\lambda x.\ e_v$ corresponds to a lambda that is defined within the program but is not yet known. By definition of unfolding, all non-nested lambdas which appear in the body of each call and application unfolding are known. If $\lambda x.\ e_v$ is defined as a non-nested lambda within a function body, then there must exist some function call in the trace which has not been unfolded (*i.e* it does not belong to U_i) which is in contradiction with the inductive hypothesis. A similar argument applies if $\lambda x.\ e_v$ is defined within another lambda's body.

As before, we see that all four conditions of the above statement hold for the root node e. Again, no infinite trace exists as U_i and A_i are finite, hence evaluation must terminate. We then apply Corollary 1 to conclude the proof. □

2.5.2 Completeness

Even in the higher-order setting, our model finding procedure remains complete. This property again requires a *fair* unfolding strategy and we therefore extend the notion of *fairness*, discussed in Chapter 1 in the context of call unfolding, to the situation with both function calls and applications.

Definition 4. *The selection process for the call or application - lambda pair to unfold at each step is* fair *iff for each i, for each $f[\overline{\tau}](e_1) \in F_i$, there exists a j such that $f[\overline{\tau}](e_1) \in U_j$, and for each $(e_1\ e_2) \in A_i$ and $\lambda x.\ e_b \in \Lambda_i$, there exists a k such that $(e_1\ e_2, \lambda x.\ e_b) \in D_k$.*

Fairness in this setting thus implies that not only will any function call eventually be unfolded, but also that for any application and any type-compatible lambda that is discovered during unfolding, we will unfold the corresponding application-lambda pair. Based on this definition, we can re-formulate the completeness theorem as follows.

Theorem 2. *For expression e and value v, if there exist inputs P_{in}, θ, γ such that $\gamma(\theta(e)) \to^* \theta(v)$ and the unfolding selection process is fair, then there exists $k \in \mathbb{N}$ and a model M such that $M \models \Phi_k \cup \mathsf{block}(U_k) \cup \mathsf{block}(A_k, \Lambda_k, D_k)$.*

Proof. First, let us define the (potentially infinite) sets $F = \bigcup_i F_i$, $A = \bigcup_i A_i$ and $\Lambda = \bigcup_i \Lambda_i$. We then select k such that

1. for $l : f[\overline{\tau}](e_1) \in F$, if $\gamma(\theta(e)) \to^* \mathcal{E}[l : f[\overline{\tau}](e_1')]$, then $f[\overline{\tau}](e_1) \in U_k$,

2. for $l_b : \lambda x.\ e_b \in \Lambda$, if $\gamma(\theta(e)) \to^* \mathcal{E}[l_b : \lambda x.\ e_b']$, then $\lambda x.\ e_b \in \Lambda_k$,

3. for $l : (e_1\ e_2) \in A$, if $\gamma(\theta(e)) \to^* \mathcal{E}[l : (e'_1\ e'_2)]$, then $(e_1\ e_2) \in A_k$, and given $e'_1 \to^* \lambda x.\ e_v$, if there exists $l_b : \lambda x.\ e_b \in \Lambda$ such that $\gamma(\theta(e)) \to^* \mathcal{E}[l_b : \lambda x.\ e_v]$, then $(e_1\ e_2, \lambda x.\ e_b) \in D_k$.

As the trace is finite and our unfolding selection strategy is fair, such a k is guaranteed to exist.

We then show by induction on $0 \le i \le k$ that there exists a model M_i such that

1. $M_i \models \Phi_i$,
2. M_i agrees with γ,
3. for $l_s : e_s \in F_i \cup U_i \cup A_i \cup \Lambda_i$ we have $\gamma(\theta(e)) \to^* \mathcal{E}[l_s : e'_s]$ iff $M \models \llbracket l_s : e_s \rrbracket_b$,
4. for $l : f[\overline{\tau}](e_1) \in F_i \cup U_i$ with $(\textbf{def}\ f[\overline{\tau}_f](x : \tau_1) : \tau_2 := e_f) \in P$, if $\gamma(\theta(e)) \to^* \mathcal{E}[l : f[\overline{\tau}'](e'_1)]$, then given evaluation results $e'_1 \to^* v_1$ and $f[\overline{\tau}'](e'_1) \to^* v_f$ where $v_1, v_f \in \mathit{value}$, we have $v_1 \overset{ext}{\lhd} (M(\llbracket e_1 \rrbracket_t), \tau_1[\overline{\tau}_f / \overline{\tau}])$ and $v_f \overset{ext}{\lhd} (M(\llbracket f[\overline{\tau}](e_1) \rrbracket_t), \tau_2[\overline{\tau}_f / \overline{\tau}])$, and
5. for $l : (e_1\ e_2) \in A_i$, if $\gamma(\theta(e)) \to^* \mathcal{E}[l : (e'_1\ e'_2]$, then given typing judgement $e_1 : \tau_2 \to \tau$ and evaluation results $e'_1 \to^* \lambda x.\ e_v$, $e'_2 \to^* v_2$ and $e'_1\ e'_2 \to^* v_c$ where $\lambda x.\ e_v, v_2, v_c \in \mathit{value}$, we have $\lambda x.\ e_v \overset{ext}{\lhd} (M(\llbracket e_1 \rrbracket_t), \tau_2 \to \tau)$, $v_2 \overset{ext}{\lhd} (M(\llbracket e_2 \rrbracket_t), \tau_2)$ and $v_c \overset{ext}{\lhd} (M(\llbracket e_1\ e_2 \rrbracket_t), \tau)$.

Note that again, items 3, 4 and 5 above imply consistency with all calls in $F_i \cup U_i$ and all applications in A_i.

The base case is given by Lemmas 5, 6 and Corollary 2. For the inductive case, we can assume a model M_i for which the hypothesis holds. If step $i+1$ is a call unfolding step, the proof is analogous to the one presented in Chapter 1. If the step is an application unfolding step, consider the pair $(l : e_1\ e_2, l_b : \lambda x.\ e_b) \in D_{i+1} \setminus D_i$ that was unfolded. Further consider normalization $\mathit{normalize}(\lambda x.\ e_b) = (\lambda x.\ e'_b, \{y_1 \mapsto e'_1, \cdots, y_n \mapsto e'_n\})$, (typed) free variables $FV(\lambda x.\ e'_b) = \{y_1 : \tau'_1, \cdots, y_n : \tau'_n\}$, the blocker constant b_b introduced when unfolding the application-lambda pair, and embedding $(b_b, e'_b) \rhd (t_b, \Phi_b)$. We then consider the following three cases:

$\gamma(\theta(e)) \to^* \mathcal{E}[l : (\lambda x.\ e_v)\ v_2]$ and $\gamma(\theta(e)) \to^* \mathcal{E}_b[l_b : \lambda x.\ e_v]$: By item 3 of the inductive hypothesis, we have $M_i \models \llbracket e_1\ e_2 \rrbracket_b$ and $M_i \models \llbracket \lambda x.\ e_b \rrbracket_b$. As $\overset{ext}{\rhd} / \overset{ext}{\lhd}$ are deterministic and inverse, we also have $M_i \models \llbracket e_1 \rrbracket_t \simeq \llbracket \lambda x.\ e_b \rrbracket_t$. By definition of $\overset{ext}{\lhd}$, given $v'_j \overset{ext}{\lhd} (e'_j, \tau'_j)$ for $1 \le j \le n$, we have $\lambda x.\ e_v = \theta(\lambda x.\ e'_b)[y_1 / v'_1, \cdots, y_n / v'_n]$. Given the value substitution $\gamma' = \{x \mapsto v_2\} \cup \{y_j \mapsto v'_j \mid 1 \le j \le n\}$, Lemma 6 ensures that there exists a model $M_b \models \Phi_b \cup \{b_b, \llbracket e_1\ e_2 \rrbracket_t \simeq t_b\}$ such that M_b is consistent with the calls and applications in e'_b. Lemma 5 and Corollary 2 then ensure a valid M_{i+1} exists by a similar argument to the one presented in Chapter 1.

$\gamma(\theta(e)) \to^* \mathcal{E}[l : (\lambda x.\ e_v)\ v_2]$ and $\gamma(\theta(e)) \not\to^* \mathcal{E}_b[l_b : \lambda x.\ e_v]$: There are two possibilities here. If $\gamma(\theta(e)) \not\to^* \mathcal{E}[l_b : \lambda x.\ e'_b]$, then we have $M_i \models \neg\llbracket \lambda x.\ e_b \rrbracket_b$ and thus $M_i \models \neg b_b$. If

$\gamma(\theta(e)) \to^* \mathcal{E}[l_b : \lambda x.\ e'_v]$ where $\lambda x.\ e_v \neq \lambda x.\ e'_v$, then $M_i \models \llbracket e_1 \rrbracket_t \neq \llbracket \lambda x.\ e_b \rrbracket_t$ as $\overset{ext}{\rhd} / \overset{ext}{\lhd}$ are deterministic and inverse, and therefore $M_i \models \neg b_b$. If we extend M_i to M_{i+1} by setting all introduced blocker constants to *false*, Lemma 1 ensures $M_{i+1} \models \Phi_{i+1}$ and no label within the unfolded e'_b can be encountered by evaluation, hence the remaining items are satisfied.

$\gamma(\theta(e)) \not\to^* \mathcal{E}[l : e'_1\ e'_2]$: By item 3 of the inductive hypothesis, we have $M \models \neg \llbracket e_1\ e_2 \rrbracket_b$, and if we extend M_i to M_{i+1} by setting all introduced blocker constants to *false*, Lemma 1 ensures $M_{i+1} \models \Phi_{i+1}$ and the remaining items are satisfied.

We again have $M_k \models \mathsf{block}(F_k)$ by item 3 of the statement. By the selection criterion of k and items 3 and 5, we have $M_k \models \mathsf{block}(A_k, \Lambda_k, D_k)$ which concludes our proof. □

2.5.3 Procedure Termination

As in the first-order case, our unfolding procedure incrementally extends the Φ_i clause set during model search. If at some point this clause set becomes unsatisfiable, then we would again like to terminate the procedure and report a proof.

Although the high-level principle remains the same, it is important to note here that there exists a subtle difference between the embeddings of the first-order and higher-order languages which affects the proof of the termination property. Indeed, although we have $\Phi_i \subseteq \Phi_{i+1}$ for all i, the δ_τ datatype definitions can change at each step. Hence, satisfiability of the clause sets is not quite as incremental as in the first-order case.

Theorem 3. *For expression e, value v and unfolding index $i \in \mathbb{N}$, if there exists no model $M \models \Phi_i$, then there exist no inputs P_{in}, θ, γ such that $\gamma(\theta(e)) \to^* \theta(v)$.*

Proof. We show this by contradiction. Let us assume that Φ_i is unsatisfiable (with respect to Λ_i), yet there exists a model $M \models \Phi_{i+1}$ (with respect to Λ_{i+1}). For each $C_{\lambda x.\ e'_b}(\bar{t})$ value term in M where the $C_{\lambda x.\ e'_b}$ is a *new* constructor, we can select some $\mathsf{Else}(n)$ value term that does not exist in M. Given $M' = M[C_{\lambda x.\ e'_b}(\bar{t}) / \mathsf{Else}(n)]$, we have $M' \models \Phi_i$ (with respect to Λ_{i+1}) as there exist no tester or selector applications on terms with a δ_τ sort in Φ_i by our embedding definition. By repeatedly removing each new constructor term in M, we obtain a model $M_i \models \Phi_i$ with respect to Λ_i, hence a contradiction. □

It is clear that the structure of the procedure which can produce counterexample inexistence proofs remains similar to the one given in Algorithm 2 of Section 1.5.

2.6 Optimizations

The embedding and unfolding procedures presented in this chapter feature strong theoretical properties, however it is important to discuss certain practical considerations. In this section, we present two important optimizations which enable effective counterexample finding in the presence of higher-order functions.

2.6.1 Incremental Embedding

Modern SMT solvers support incremental solving, a technique which improves performance when multiple satisfiability checks are performed on an increasing clause set. It is clear that our unfolding procedure follows such a schema. However, the embedding of first-class functions into SMT terms and sorts requires knowing in advance the set of all lambda structures that may occur during embedding, and the discussed *staged* embedding approach disallows incremental solving. In this section, we present an incremental embedding strategy for first-class functions which is compatible with incremental solving.

We propose an alternative embedding that preserves certain important properties that algebraic datatypes give us but allows an unbounded set of typed structures. Consider two lambdas $\lambda x.\ e_i$ for $i \in \{1,2\}$ with type $\lambda x.\ e_i : \tau$. For $1 \leq i \leq 2$, further consider normalizations $(\lambda x.\ e'_i, \{y_{i,1} \mapsto e_{i,1}, \cdots, y_{i,n_i} \mapsto e_{i,n_i}\})$, and embeddings $t_i = \llbracket \lambda x.\ e_i \rrbracket_t$ and $t_{i,j} = \llbracket e_{i,j} \rrbracket_t$ for $1 \leq j \leq n_i$. The properties we want are:

Distinctness : $t_1 \not\simeq t_2$ if $\lambda x.\ e'_1 \neq \lambda x.\ e'_2$.

Injectivity : if $t_1 \simeq t_2$, then $n_1 = n_2$ and $t_{1,1} \simeq t_{2,1}, \cdots, t_{1,n_1} \simeq t_{2,n_2}$.

Surjectivity : if $\lambda x.\ e'_1 = \lambda x.\ e'_2$ and $t_{1,1} \simeq t_{2,1}, \cdots, t_{1,n_1} \simeq t_{2,n_2}$, then $t_1 \simeq t_2$.

Inductiveness : value extraction cannot lead to cycles.

The embedding of function types into algebraic datatypes ensures all four properties are satisfied, along with the additional property of **exhaustivity** which limits the set of interpretations to the exhaustively given constructors. Recall however that we introduced an Else constructor explicitly to avoid this particular property.

We propose in our alternative embedding to embed the function type τ into an uninterpreted sort σ_τ. Lambdas are then embedded into fresh constants with sort σ_τ. Recall that we already track the set of known lambdas in Λ. We rely on this set to enforce the distinctness, injectivity and surjectivity constraints described above. Given the two lambdas above and blocker constants $b_i = \llbracket \lambda x.\ e_i \rrbracket_b$ for $i \in \{1,2\}$, we define the following clause set

$$\Phi_{eq} = \begin{cases} \{(b_1 \wedge b_2) \implies (t_1 \simeq t_2 \iff t_{1,1} \simeq t_{2,1} \wedge \cdots \wedge t_{1,n_1} \simeq t_{2,n_2})\} & \text{if } \lambda x.\ e'_1 = \lambda x.\ e'_2 \\ \{(b_1 \wedge b_2) \implies t_1 \not\simeq t_2\} & \text{if } \lambda x.\ e'_1 \neq \lambda x.\ e'_2 \end{cases}$$

which ensures distinctness, injectivity and surjectivity of our embedding. Such clauses must be generated for all pairs of known lambdas in Λ.

Ensuring inductiveness is slightly more involved. The high-level idea is to impose an ordering on values such that closures within a lambda have strictly lower order than the lambda itself. This restriction corresponds to the restriction imposed by inductiveness of algebraic datatypes and can be viewed as a consequence of linear scopes as featured by our language. This ordering can be imposed by introducing a new special function *order* with integer domain and generating clauses that impose the correct ordering relation. We define the function isOrdered which takes the type τ of the closure as argument:

```
def isOrdered($\tau$: type, $e_c$: expr, $e_\lambda$: expr): Boolean = $\tau$ match {
  case $\tau_1 \rightarrow \tau_2$ ⇒ $order(e_c) < order(e_\lambda)$
  case $d[\overline{\tau}]$ where (type $d[\overline{\tau}_d]$ := $\cdots$ | $C_i(x_i : \tau_i)$ | $\cdots) \in P$ ⇒
    $e_c$ match { $\cdots$ case $C_i(y_i)$ ⇒ isOrdered($\tau_i[\overline{\tau}_d/\overline{\tau}]$, $y_i$, $e_\lambda$) $\cdots$ }
  case _ ⇒ true
}
```

It is clear that this function cannot be represented as such in our language as it takes a *type* as argument. However, much like in the case of function type parameters, the set of instantiations for τ in a given expression is finite and statically known. Hence, we can handle the type-parametricity of isOrdered in the same way as type parameters by inlining the correct body depending on τ during the unfolding procedure. Given a lambda $\lambda x.\ e \in \Lambda_\tau$ and its normalization $normalize(\lambda x.\ e) = (\lambda x.\ e', \{y_1 : \tau_1 \mapsto e_1, \cdots, y_n : \tau_n \mapsto e_n\})$, consider the associated blocking constant $b = \llbracket \lambda x.\ e \rrbracket_b$ and embeddings $t = \llbracket \lambda x.\ e \rrbracket_t$, and $t_i = \llbracket e_i \rrbracket_t$ for $1 \le i \le n$. We can then define the clause set

$$\Phi_{order} = \{b \implies (\text{isOrdered}(\tau_1, t_1, t) \wedge \cdots \wedge \text{isOrdered}(\tau_n, t_n, t))\}$$

which ensures the inductiveness of our embedding. Note that the isOrdered(τ_i, t_i, t) function calls must be handled in a similar manner to named functions in the expression, with static blocking and term-level inlining.

It is interesting to note here that in addition to enforcing inductiveness, these ordering clauses enable a useful optimization during unfolding. Given the pair $(e_1\ e_2, \lambda x.\ e_b) \in A_i \times \Lambda_i$, if a clause $order(\llbracket e_1 \rrbracket_t) < order(\llbracket \lambda x.\ e_b \rrbracket_t)$ has been generated during embedding, then there is no need to unfold the application - lambda pair, as the blocking condition of $\llbracket e_1 \rrbracket_t \simeq \llbracket \lambda x.\ e_b \rrbracket_t$ can never hold. This observation enables us to skip unfoldings in the presence of lambdas of the shape $\lambda x.\ \mathcal{C}[f\ x]$ where f has same type as the lambda itself.

Value extraction from model interpretations of function-typed terms is performed similarly to the setting with datatypes. If there exists a lambda in Λ which corresponds to the term we're extracting, we extract the embedding of each structural closure and substitute them in the lambda's normalized structure. If no such lambda exists, we apply the previously defined extraction rule and introduce a synthetic named function.

Function types and lambdas can therefore be embedded in an incremental manner without having to rely on a deferred datatype definition to allow for (as of yet) unknown lambdas. This incremental embedding produces the relevant Φ_{eq} and Φ_{order} clauses whenever a new lambda is embedded. While the number of generated Φ_{eq} clauses is quadratic in the number of known lambdas, this value remains quite tractable in practice.

2.6.2 Lambda Tracking

The unfolding procedure for function applications described in the previous sections shows that a quadratic blowup of unfoldings occurs in terms of known applications and lambdas, even when factoring in the optimization based on orderings described in the previous section. However, the flow of lambdas through a program is generally less obscure than that of arbitrary data, and it is often possible to determine which lambda exactly is being applied during unfolding. We give a high-level description of a tracking procedure, and of how this information can be leveraged during application unfolding.

Tracking is performed by a lightweight flow analysis during which we compute a global mapping pointers : *Pointers* from embedded terms to lambdas, *Pointers* = *term* $\mapsto$ *lambda*. This mapping is populated by establishing known lambda targets based on the identifiers in *let*-expression embeddings, as well as call and application unfoldings. Given an embedded receiving identifier, an expression, and the current global pointers mapping, an extension to the mapping is computed by finding datatype selector chains from the identifier to known lambdas (or known pointers) in the expression.

```
def getPointers($t$: $term$, $e$: $expr$): $Pointers$ = $e$ match {
  case $C_i(e_1)$ where $e : d[\overline{\tau}]$ $\Rightarrow$ getPointers($selector_{d[\overline{\tau}],i}(t)$, $e_1$)
  case $(e_1, e_2)$ where $e : (\tau_1, \tau_2)$ $\Rightarrow$ $\bigcup_{1\leq i\leq 2}$ getPointers($\pi_{(\tau_1,\tau_2),i}(t)$, $e_i$)
  case $x \in id$ if $x \in$ pointers $\Rightarrow$ $\{t \mapsto \text{pointers}(x)\}$
  case $\lambda x.\ e_b$ $\Rightarrow$ $\{t \mapsto \lambda x.\ e_b\}$
  case _ $\Rightarrow$ $\{\}$
}
```

Based on this target computation, the global pointers mapping is populated by recursively traversing the embedded expressions that have occurred during unfolding until step i. Given an embedded expression, the pointers mapping is constructed as follows.

Case $\lambda x.\ e_b$: the mapping $\llbracket \lambda x.\ e_b \rrbracket_t \mapsto \lambda x.\ e_b$ is added to pointers.

Case let $x := e_1$ **in** e_2 : the pointers obtained by getPointers(x, e_1) are added to the mapping. Note that we know that x is unique in the program, so any application where the caller depends on x must correspond to this let binding.

Case $f[\overline{\tau}](e_1) \in U_i$: given (**def** $f[\overline{\tau}_f](x : \tau_1) : \tau_2 := e_f) \in P$, we register the pointers obtained through getPointers(x, e_1). We further consider returned lambdas and add getPointers$(\llbracket f[\overline{\tau}](e_1) \rrbracket_t, e_f[\overline{\tau}_f/\overline{\tau}])$ to pointers. We then recursively traverse $e_f[\overline{\tau}_f/\overline{\tau}]$ in order to register the lambda pointers that appear within the unfolded body.

Case $e_1\ e_2$ **if** $\lfloor\!\lfloor e_1 \rfloor\!\rfloor_t \in \mathsf{pointers}$: given the target lambda $\mathsf{pointers}(\lfloor\!\lfloor e_1 \rfloor\!\rfloor_t) = \lambda x.\ e_b$, as well as normalization $\mathit{normalize}(\lambda x.\ e_b) = (\lambda x.\ e'_b, \{y_1 \mapsto e'_1, \cdots, y_n \mapsto e'_n\})$, we register the pointers $\mathsf{getPointers}(x, e_2)$ and $\mathsf{getPointers}(y_j, e'_j)$ for $1 \le j \le n$. We again further register the pointers $\mathsf{getPointers}(\lfloor\!\lfloor e_1\ e_2 \rfloor\!\rfloor_t, e'_b)$ and recursively traverse e'_b to locate pointers in the unfolded body.

In all other cases, we recursively traverse the children of the given expression and continue establishing pointers. It is important to realize that this lightweight flow analysis is efficiently performed through a single top-down traversal of the initial expression and the unfolded bodies. While there are many lambda targets that are not captured by this approach, it has shown effective in practice.

Based on the term to lambda mapping resulting from the flow analysis, we can avoid certain unnecessary application - lambda unfoldings. Indeed, given a pair $(e_1\ e_2, \lambda x.\ e_b) \in A_i \times \Lambda_i$, if $\lfloor\!\lfloor e_1 \rfloor\!\rfloor_t \in \mathsf{pointers}$ and $\mathsf{pointers}(\lfloor\!\lfloor e_1 \rfloor\!\rfloor_t) \neq \lambda x.\ e_b$, then we know that this particular pair can be skipped during unfolding. Indeed, even when there exists $M \models \Phi_i \cup \{\lfloor\!\lfloor e_1 \rfloor\!\rfloor_t \simeq \lfloor\!\lfloor \lambda x.\ e_b \rfloor\!\rfloor_t\}$, the clause set resulting from the unfolding of $(e_1\ e_2, \lambda x.\ e_b)$ will be made redundant by the unfolding of the pair $(e_1\ e_2, \mathsf{pointers}(\lfloor\!\lfloor e_1 \rfloor\!\rfloor_t))$ for which $M \models \lfloor\!\lfloor e_1 \rfloor\!\rfloor_t \simeq \lfloor\!\lfloor \mathsf{pointers}(\lfloor\!\lfloor e_1 \rfloor\!\rfloor_t) \rfloor\!\rfloor_t$ is guaranteed to hold in any satisfying model.

3 Verification with Dependent Types

In this chapter, we will discuss how the counterexample finding procedures presented in the previous two chapters can be leveraged to build a verification system based on dependent types. We will first extend our language with dependent types such as dependent function types, dependent pair types and refinement types. We will then define a denotation for all types in a program, and extend our counterexample finding procedure to leverage these denotations. Finally, we will present an algorithmic type checking procedure that relies on the embedding and unfolding procedures we described.

Our approach to program verification is based on a type system which ensures that well-typed expressions reduce to expected values. Our type system is loosely based on System T [Gir90, Chapter 7]. We define a *reducible* logical relation that associates to each type a denotation, namely a set of "good" values of the given types. We then show that our type checking procedure only accepts expressions that, given well-formed inputs, will evaluate to a value in the denotation of the expected type.

In collaboration with Dr. Jad Hamza, we have developed a fully mechanized proof in Coq of a variant of the type system presented here which includes fixpoint operators, parametric polymorphism (à la System F), sized recursive types and refinement types. Unlike the system we present here, equality is not considered decidable in the mechanized system and we therefore rely on equality types and judgements instead. Another important distinction is that the mechanized system does not allow mutual recursion between function and type definitions. We have found that formally defining and proving this type system variant was invaluable in building a sound verification algorithm.

3.1 Language

We extend the language presented in Chapter 2 with dependent types and a stuck expression. As mentioned above, we introduce dependent function types (pi-types), dependent pair types (sigma-types) and refinement types. We further introduce dependent sized recursive types in order to ensure that our denotation is well-defined in the presence of algebraic datatypes. We therefore extend the type grammar as follows:

$$\begin{array}{lll} \mathit{type} & ::= & \cdots \mid \Pi\, \mathit{id} : \mathit{type}.\ \mathit{type} \mid \Sigma\, \mathit{id} : \mathit{type}.\ \mathit{type} \\ & & \mid \{\, \mathit{id} : \mathit{type} \mid \mathit{expr} \,\} \mid \mathit{id}[\, \mathit{tparams} \,](\, \mathit{expr} \,) \end{array}$$

We consider $\tau_1 \to \tau_2$ to be a special form of $\Pi\, x : \tau_1.\ \tau_2$ where τ_2 does not depend on x. We similarly consider (τ_1, τ_2) to be a special case of $\Sigma\, x : \tau_1.\ \tau_2$. Given a typing context Γ and boolean expression p, we denote the context $\Gamma, u : \{\, u : \mathsf{Unit} \mid p \,\}$ where u is fresh by Γ, p.

A sized datatype $d[\overline{\tau}](e_m)$ relies on the well-order Nat to ensure it is well defined. We assume hereafter that the type Nat and an associated ordering relation $<$ are available in all considered programs. Contrary to certain definitions of sized types which rely on ordinals, we restrict ourselves to the natural numbers in order to simplify the inference of size expressions and leverage the automation given by the theory of linear arithmetic in the underlying SMT solver. We therefore modify the datatype definition grammar to include the size binding (given after the type parameters) as follows:

$$\mathit{tdef} \ ::= \ \textbf{type}\ \mathit{id}[\, \mathit{tdecls} \,](\, \mathit{id} \,) := \mathit{id}(\, \mathit{id} : \mathit{type} \,) \,\langle \mid \mathit{id}(\, \mathit{id} : \mathit{type} \,) \,\rangle^*$$

We will see later that we rely on a syntactic constraint to ensure that references to mutually recursive types in constructor parameters will decrease the size expression. The datatype $d[\overline{\tau}]$ will then correspond to the intersection over all n of $d[\overline{\tau}](n)$, as we will see below when defining the denotation.

In order to ensure our recursive function definitions are well-founded, we again rely on a *size binding* which is shown to decrease during type checking. We do not restrict this binding to the natural numbers but instead rely on some arbitrary well-order τ_m. We again assume the existence of a $<$ relation on the order τ_m in our language. We thus introduce a size parameter given before the type parameters and modify the function definition grammar as follows:

$$\mathit{fdef} \ ::= \ \textbf{def}\ \mathit{id}(\, \mathit{id} : \mathit{type} \,)[\, \mathit{tdecls} \,](\, \mathit{id} : \mathit{type} \,) : \mathit{type} := \mathit{expr}$$

We impose a syntactic restriction on sized function definitions which only allows the size binding to appear within types or size expression positions. This will ensure that evaluation is unaffected by the size expressions that appear in the program.

At the expression level, we extend function calls and datatype constructors with an extra *size* expression parameter which is used during type checking. However, we also allow function calls and datatype constructors *without* these size expressions in cases where they are not necessary to ensure well-foundedness. The expression grammar is thus extended as follows:

$$expr \quad ::= \quad \cdots \mid id\,(\,expr\,)[\,tparams\,](\,expr\,)$$

These language extensions will be leveraged to perform program verification. However, they constitute *annotations* and should not influence the operational semantics of our language. We therefore introduce the notion of *erasure* for types, expressions, and programs. Type erasure drops refinements and size predicates. Expression erasure drops the size expressions in function calls and datatype constructors, as well as erases all types within the expression. Program erasure erases all types and expressions within the type and function definitions, as well as the size binding for function definitions. Note that type erasure effectively drops pi- and sigma-types as well since all dependencies will have been removed. It is clear that erased types, expressions and programs belong to the language discussed in the previous chapter. We will rely on an $erase(\cdot)$ function hereafter which computes the erasure of the given type, expression or program.

We extend the simple typing judgement presented in the previous chapters to the dependently-typed language by letting $P;\Theta;\Gamma \vdash e : \tau$ hold iff the judgement holds for the erased typing environment, expression and type. Similarly, we rely on erasure to extend our operational semantics to the dependently-typed language. We therefore say $e \to e'$ in P iff the erasure of e evaluates to e' in the erasure of P.

For certain considerations about well-foundedness, it is useful to determine the dependencies between definitions in this dependently-typed language. Given a program P, we assume the existence of some total ordering relation $\preceq \in P \times P$ such that given two definitions $d_1, d_2 \in P$, if the identifier of d_1 appears in d_2, then we have $d_1 \preceq d_2$. If $d_1 \preceq d_2$ and $d_2 \preceq d_1$, then both definitions are in the same equivalence class and we write $d_1 \sim d_2$. We say $d_1 \prec d_2$ when $d_1 \preceq d_2$ and $d_1 \not\sim d_2$. Note that the definition of $\preceq$ ensures that if the definitions of d_1 and d_2 are mutually recursive, then we have $d_1 \sim d_2$. It is clear that $\preceq$ always exists, as in the worst case we have $d_1 \sim d_2$ for all $d_1, d_2 \in P$.

Given the $\sim$ relation on definitions, we can describe the syntactic restriction we impose on datatype definitions as mentioned above. Given some datatype definition

$$(\textbf{type}\ \ d[\overline{\tau}_d](m)\ :=\ C_1(x_1 : \tau_1)\ \mid \cdots \mid\ C_n(x_n : \tau_n)) \in P$$

we ensure that for $1 \leq i \leq n$, for each intersection datatype $d'[\overline{\tau}'] \sqsubseteq \tau_i$ (namely $d'[\overline{\tau}']$ occurs within τ_i), we have $d \not\sim d'$, and for each sized datatype $d'[\overline{\tau}'](e_m) \sqsubseteq \tau_i$, if $d \sim d'$, then $e_m = m-1$ (or m **match** $\{\,\mathsf{Succ}(x)\ \Rightarrow\ x\ \ \mathsf{Zero}\ \Rightarrow\ \mathsf{err[Nat]}\,\}$ in our language). Note that at the source level, datatypes can be declared as usual and the size bindings can be automatically inserted.

We introduce further syntactic restrictions on function definitions that are mutually recursive with function or datatype definitions. Given some function definition

$$(\textbf{def}\ \ f(m:\tau_m)[\overline{\tau}_f](x_1:\tau_1):\tau_2\ :=\ e_f \in P$$

we require that each other function definition $f' \in P$ such that $f \sim f'$ be associated the same well-order τ_m. If there exists a datatype definition $d \in P$ such that $f \sim d$, then we require τ_m to be Nat. Furthermore, for $1 \leq i \leq 2$, for intersection datatype $d[\overline{\tau}] \sqsubseteq \tau_i$ we have $f \not\sim d$, and for sized datatype $d[\overline{\tau}]e_m \sqsubseteq \tau_i$, we have $e_m = n$. Note that we do not enforce a size decrease here but allow same-sized datatypes in the function signature.

3.2 Reducibility

In this section, we describe a logical relation that corresponds to expressions that will evaluate to some value (*i.e.* expressions where evaluation neither gets stuck nor diverges). This relation is inspired by the notion of *reducibility* (and other similar notions) described for example in [Tai67, Gir90, Har16]. We will therefore say a value, closed or open expression, type or program is *reducible* when it satisfies the relevant logical relation.

Reducibility for closed expressions. We start by defining the notion of reducibility for closed values and expressions, *i.e.* values and expressions that contain no type or expression variable. We define two mutually recursive relations. For each type τ, we define $[\![\tau]\!]_v$ the set of *reducible values* and $[\![\tau]\!]_e$ the set of *reducible expressions*. Intuitively, the set $[\![\tau]\!]_v$ contains the set of values of type τ, and $[\![\tau]\!]_e$ contains the set of expressions that evaluate to some value of type τ. Both definitions can be found in Figure 3.1. We call the set $[\![\tau]\!]_v$ the *denotation* of the type τ.

If a closed expression is reducible, *i.e.* belongs to some set $[\![\tau]\!]_e$, then by definition we know that e will evaluate to some value in a finite number of steps. Furthermore, if τ is a function type, for example $\mathsf{Nat} \to \mathsf{Nat}$, then any application of e to a value in $[\![\mathsf{Nat}]\!]_v$ will terminate.

Recursive types are known to introduce logical inconsistencies into type systems such as Russell's paradox, and to lead to non-termination. Sized types are a common approach to resolving this issue. Our definition of $[\![d[\overline{\tau}](e_m)]\!]_v$ corresponds to a variation of sized types where sizes belong to Nat (as opposed to the more general ordinals [HPS96, Par98, Abe07]). Note however that we allow arbitrary expressions in size positions.

During type checking, it will sometimes be useful to determine that an expression will reduce to some (not necessarily reducible) value. This will namely be the case when type checking a datatype constructor with size Zero. We therefore introduce the following special type $\mathsf{val}[\mathit{type}]$ with associated denotation $[\![\mathsf{val}[\tau]]\!]_v = \{ v \in \mathit{value} \mid v : \tau \}$. The erasure of $\mathsf{val}[\tau]$ is simply given as the erasure of τ. Note that this new type is internal to our type checking algorithm and will never appear in our input programs.

$$
\begin{aligned}
&[\![\mathsf{Unit}]\!]_v \;=\; \{()\} \\
&[\![\mathsf{Boolean}]\!]_v \;=\; \{\textbf{true}, \textbf{false}\} \\
&[\![\mathsf{Nat}]\!]_v \;=\; \{\mathsf{Zero}, \mathsf{Succ(Zero)}, \mathsf{Succ(Succ(Zero))}, \cdots\} \\
&[\![d[\overline{\tau}](e_m)]\!]_v \;=\; \{v \in \mathit{value} \mid \exists\, v_m \in [\![\mathsf{Nat}]\!]_v.\ v : d[\overline{\tau}] \;\wedge\; e_m \to^* v_m \;\wedge \\
&\qquad v_m > \mathsf{Zero} \implies (\exists\, v_1 \in [\![\tau_i[m/v_m][\overline{\tau}_d/\overline{\tau}]]\!]_v.\ v = C_i[\mathit{erase}(\overline{\tau})](v_1) \\
&\qquad\qquad\qquad\qquad \text{where } (\textbf{type}\ d[\overline{\tau}_d](m) := \cdots \mid C_i(x_i : \tau_i) \mid \cdots) \in P)\} \\
&[\![d[\overline{\tau}]]\!]_v \;=\; \{v \in \mathit{value} \mid \forall\, v_m \in [\![\mathsf{Nat}]\!]_v.\ v \in [\![d[\overline{\tau}](v_m)]\!]_v\} \\
&[\![\{x : \tau \mid p\}]\!]_v \;=\; \{v \in [\![\tau]\!]_v \mid p[x/v] \to^* \textbf{true}\} \\
&[\![\Pi\, x : \tau_1.\, \tau_2]\!]_v \;=\; \{v \in \mathit{value} \mid \forall\, v_1 \in [\![\tau_1]\!]_v.\ v\ v_1 \in [\![\tau_2[x/v_1]]\!]_e\} \\
&[\![\Sigma\, x : \tau_1.\, \tau_2]\!]_v \;=\; \{(v_1, v_2) \mid v_1 \in [\![\tau_1]\!]_v \;\wedge\; v_2 \in [\![\tau_2[x/v_1]]\!]_v\} \\
\\
&[\![\tau]\!]_e \;=\; \{e \in \mathit{expr} \mid e : \tau \;\wedge\; \exists\, v \in [\![\tau]\!]_v.\ e \to^* v\}
\end{aligned}
$$

Figure 3.1 – Definition of reducibility for values and expressions for each type with respect to some program P under which evaluation and (simple) typing is performed. Note that we define a denotation for the Nat type in order to ensure that the denotation is well-formed. However, we will generally assume hereafter that a Nat datatype is available in the program and treat it similarly to any other datatype.

Reducibility for open expressions. We now describe reducibility in the context of open expressions, namely expressions with free type or expression variables. Reducibility in the context of open expressions denotes expressions which, when given a set of reducible inputs, belong to the set of reducible closed expressions.

Given a typing environment $P;\Theta;\Gamma$, we say the inputs P_{in},θ,γ are *reducible* (or *reducible inputs*) for $P;\Theta;\Gamma$ if for each $(x,\tau) \in \Gamma$, we have $\gamma(x) \in [\![\gamma(\theta(\tau))]\!]_v$ in program $P \cup P_{in}$. We denote the set of *reducible inputs* for $P;\Theta;\Gamma$ as $[\![P;\Theta;\Gamma]\!]_v$. We say an open expression e is reducible, denoted by $P;\Theta;\Gamma \models e \in [\![\tau]\!]$, if $\forall\,(P_{in},\theta,\gamma) \in [\![P;\Theta;\Gamma]\!]_v.\ \gamma(\theta(e)) \in [\![\gamma(\theta(\tau))]\!]_e$ and $P;\Theta;\Gamma \vdash e : \tau$.

Reducibility for contexts, types and programs. Reducibility in this case becomes more of a well-formedness relation and simply ensures that all expressions that appear within typing contexts, types and program definitions are reducible. Given a program P, set of type variables Θ and typing context Γ, we denote reducibility of Γ as $P;\Theta \models \Gamma$ *context*, reducibility of type τ as $P;\Theta;\Gamma \models \tau$ *type* and reducibility of type or function definition d as $P \models d$ *well-formed*. Each reducibility notion is then respectively defined in Figures 3.2, 3.3 and 3.4. For program P, we define program reducibility $\models P$ *program* as $\forall\, d \in P.\ P \models d$ *well-formed*. It is important to note that these notions of reducibility imply well-formedness as defined in the previous chapters.

Bounded reducibility. Before concluding our presentation of the different reducibility relations, let us touch upon the notion of *bounded reducibility*. The type checking procedure we will present later in this chapter aims to establish reducibility for expressions, types, contexts

Empty Context

$$P;\Theta \models \ \textit{context}$$

Increase Context

$$\frac{P;\Theta;\Gamma \models \tau \ \textit{type}}{P;\Theta \models \Gamma, x:\tau \ \textit{context}}$$

Figure 3.2 – Context reducibility rules.

Boolean Type

$$\frac{P;\Theta \models \Gamma \ \textit{context}}{P;\Theta;\Gamma \models \mathsf{Boolean} \ \textit{type}}$$

Unit Type

$$\frac{P;\Theta \models \Gamma \ \textit{context}}{P;\Theta;\Gamma \models \mathsf{Unit} \ \textit{type}}$$

Type Variable

$$\frac{P;\Theta \models \Gamma \ \textit{context} \qquad T \in \Theta}{P;\Theta;\Gamma \models T \ \textit{type}}$$

Datatype

$$\frac{\begin{array}{c} P;\Theta \models \Gamma \ \textit{context} \qquad (\textbf{type} \ d[\overline{\tau}_d](m) \ := \ \cdots) \in P \\ |\overline{\tau}_d| = |\overline{\tau}| \qquad P;\Theta;\Gamma \models \tau \ \textit{type} \text{ for } \tau \in \overline{\tau} \qquad P;\Theta;\Gamma \models e_m \in [\![\mathsf{Nat}]\!] \end{array}}{P\Theta;\Gamma \models d[\overline{\tau}](e_m) \ \textit{type}}$$

Refinement Type

$$\frac{P;\Theta;\Gamma \models \tau \ \textit{type} \qquad P;\Theta;\Gamma, x:\tau \models p \in [\![\mathsf{Boolean}]\!]}{P;\Theta;\Gamma \models \{x:\tau \ | \ p\} \ \textit{type}}$$

Pi-Type

$$\frac{P;\Theta;\Gamma \models \tau_1 \ \textit{type} \qquad P;\Theta;\Gamma, x:\tau_1 \models \tau_2 \ \textit{type}}{P;\Theta;\Gamma \models \Pi x:\tau_1.\ \tau_2 \ \textit{type}}$$

Sigma-Type

$$\frac{P;\Theta;\Gamma \models \tau_1 \ \textit{type} \qquad P;\Theta;\Gamma, x:\tau_1 \models \tau_2 \ \textit{type}}{P;\Theta;\Gamma \models \Sigma x:\tau_1.\ \tau_2 \ \textit{type}}$$

Figure 3.3 – Type reducibility rules. The Datatype rule further allows a variant for $d[\overline{\tau}]$ where the $P;\Theta;\Gamma \models e_m \in [\![\mathsf{Nat}]\!]$ reducibility check is dropped.

Type

$$\frac{P;\overline{\tau}_f; m:\mathsf{Nat}, m > \mathsf{Zero} \models \tau_i \ \textit{type} \text{ for } 1 \le i \le n \qquad \textit{erase}(P);\overline{\tau}_d \vdash d[\overline{\tau}_d] \ \textit{well-defined}}{P \models \textbf{type} \ d[\overline{\tau}_f](m) \ := \ C_1(x_1:\tau_1) \ | \ \cdots \ | \ C_n(x_n:\tau_n) \ \textit{well-formed}}$$

Function

$$\frac{\begin{array}{c} P;\emptyset;\emptyset \models \tau_m \ \textit{type} \qquad \tau_m \text{ well-order} \\ P;\overline{\tau}_f; m:\tau_m \models \tau_1 \ \textit{type} \qquad P;\overline{\tau}_f; m:\tau_m, x:\tau_1 \models \tau_2 \ \textit{type} \qquad P;\overline{\tau}_f; m:\tau_m, x:\tau_1 \models e_f \in [\![\tau_2]\!] \end{array}}{P \models \textbf{def} \ f(m:\tau_m)[\overline{\tau}_f](x:\tau_1):\tau_2 \ := \ e_f \ \textit{well-formed}}$$

Figure 3.4 – Program reducibility rules. Recall that the syntactic constraints discussed in the context of both datatype and function definitions further apply.

and programs. The proof of soundness for this procedure will proceed by induction over the ordering $\prec$ between definitions and the well-order τ_m. The first induction over $\prec$ implies that our type checking procedure will be defined in the context of a reducible program P_1 and a set of mutually recursive definitions P_2 which are currently being checked. The second induction will then give us some $n \in [\![\tau_m]\!]_v$ such that types and definitions in P_2 are reducible "below" n. Intuitively, this value n gives us a bound below which reducibility is known to hold.

We start by defining the notion of bounded reducibility for function definitions in P_2. Consider some definition $(\textbf{def}\ f(m:\tau_m)[\overline{\tau}_f](x:\tau_1):\tau_2 \ := \ e_f) \in P_2$. We say that f is *reducible below n* iff for $n' \in [\![\tau_m]\!]_v$ such that $n' < n$, we have

$$P_1 \cup P_2; \overline{\tau}_f; \emptyset \models \tau_1[m/n']\ \mathit{type}\ \wedge\ P_1 \cup P_2; \overline{\tau}_f; x:\tau_1[m/n'] \models \tau_2[m/n']\ \mathit{type}\ \wedge \\ P_1 \cup P_2; \overline{\tau}_f; x:\tau_1[m/n'] \models e_f[m/n'] \in [\![\tau_2[m/n']]\!]$$

Note that reducibility below n is therefore weaker than reducibility. Furthermore, if f is reducible below n for all $n \in [\![\tau_m]\!]_v$ (and τ_m is a reducible well-order), then f is reducible.

Next, consider the datatype definition $(\textbf{type}\ d[\overline{\tau}_d](m)\ :=\ C_1(x_1:\tau_1)\ \mid \cdots \mid\ C_r(x_r:\tau_r)) \in P_2$. Similarly to the function case, we say that d is *reducible below n* iff the type τ_m is Nat, and for $n' \in [\![\mathsf{Nat}]\!]_v$ such that $\mathsf{Zero} < n' < n$, we have

$$\mathit{erase}(P_1 \cup P_2); \overline{\tau}_d \vdash d[\overline{\tau}_d]\ \mathit{well\text{-}defined}\ \wedge\ \forall 1 \leq i \leq r.\ P_1 \cup P_2; \overline{\tau}_d; \emptyset \models \tau_r[m/n']\ \mathit{type}$$

Again, reducibility below n is a weaker notion than reducibility for datatype definitions, and if d is reducible below n for all $n \in [\![\mathsf{Nat}]\!]_v$, then d must also be reducible.

In the remainder of this chapter, we will be dealing with a reducible program P_1 and set of mutually recursive definitions P_2. In order to improve readability, we will generally write P instead of $P_1 \cup P_2$ when the distinction is not relevant.

3.2.1 Type Parameter Polarity

When considering the relations between the denotations of datatypes with different type parameter instantiations, it is useful to rely on the notions of *type parameter polarity*. We rely on the following four different polarities:

$\circ$	:	arbitrary or mixed polarity,	$+$	:	positive or *covariant* polarity,
$\bullet$	:	constant or *invariant* polarity,	$-$	:	negative or *contravariant* polarity.

These polarities form a lattice, and we further define a *composition* operation $(p_1 * p_2)$ and a *negation* operation $(\neg p)$ on the lattice elements.

$$\begin{tikzcd}
 & \bullet & \\
- \arrow[ur] & & + \arrow[ul] \\
 & \circ \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

$p_1 * p_2$	$\circ$	$+$	$-$	$\bullet$
$\circ$	$\circ$	$+$	$-$	$\bullet$
$+$	$\circ$	$+$	$-$	$\bullet$
$-$	$\circ$	$-$	$+$	$\bullet$
$\bullet$	$\bullet$	$\bullet$	$\bullet$	$\bullet$

$\neg p$	$\circ$	$+$	$-$	$\bullet$
	$\circ$	$-$	$+$	$\bullet$

The polarity lattice naturally implies the existence of least-upper-bound ($p_1 \sqcap p_2$) and greatest-lower-bound ($p_1 \sqcup p_2$) operations. We further assume a $\preceq$ operator such that given polarities p_1, p_2, we have $p_1 \preceq p_2$ iff $p_1 \sqcap p_2 = p_2$.

We are interested in establishing a *type parameter polarity assignment* which associates some polarity from the lattice to each type parameter of each datatype in a program. Moreover, we restrict ourselves to *valid* assignments which can be defined as follows. It is clear based on this definition that a valid type parameter polarity assignment can be leveraged to define a subtyping relation on datatypes with same size.

Definition 6. *We say a type parameter polarity assignment $pol_T : id \mapsto polarity$ is* valid *if for* $(\textbf{type}\ \ d[\overline{\tau}_d](m)\ :=\ \cdots) \in P$, $v_m \in [\![\mathsf{Nat}]\!]_v$ *and* $(P_1,\theta_1),(P_2,\theta_2) \in [\![P;\overline{\tau}_d;\emptyset]\!]_v$ *such that*

$$\forall\, T_i \in \overline{\tau}_d.\ + \preceq pol_T(T_i) \implies [\![\theta_1(T_i)]\!]_v \subseteq [\![\theta_2(T_i)]\!]_v \wedge - \preceq pol_T(T_i) \implies [\![\theta_1(T_i)]\!]_v \supseteq [\![\theta_2(T_i)]\!]_v$$

we have $[\![\theta_1(d[\overline{\tau}_d](v_m))]\!]_v \subseteq [\![\theta_2(d[\overline{\tau}_d](v_m))]\!]_v$.

In practice, a valid type parameter polarity assignment can be computed as a fixpoint based on the erasure of datatype definitions in the program. The assignment $pol_T : id \mapsto$ polarity is first initialized to the bottom of the lattice, namely $pol(T) = \circ$ for each datatype type parameter in P. For each (erased) datatype definition $(\textbf{type}\ \ d[\overline{\tau}_d]\ :=\ C_1(x_1 : \tau_1)\ \mid \cdots \mid\ C_n(x_n : \tau_n)) \in erase(P)$, type parameter $T \in \overline{\tau}_d$ and constructor index $1 \leq i \leq n$, we then compute $polarity(T, \tau_i) = p_i$ the polarity of T in τ_i given the current polarity assignment. Finally, we update the assignment $pol_T(T) = pol_T(T) \sqcap p_i$ and iterate until a fixpoint has been reached. The computation of the polarity of T in τ_i is performed according to the following rules given the current pol_T.

$$polarity(T, T) = + \qquad\qquad \frac{T \not\sqsubseteq \tau}{polarity(T, \tau) = \circ}$$

$$\frac{polarity(T, \tau_1) = p_1 \qquad polarity(T, \tau_2) = p_2}{polarity(T, (\tau_1, \tau_2)) = p_1 \sqcap p_2} \qquad \frac{polarity(T, \tau_1) = p_1 \qquad polarity(T, \tau_2) = p_2}{polarity(T, \tau_1 \to \tau_2) = \neg p_1 \sqcap p_2}$$

$$\frac{(\textbf{type}\ \ d[T_1, \cdots, T_r]\ :=\ \cdots) \in P \qquad polarity(T, \tau_i) = p_i\ \text{ for }\ 1 \leq i \leq r}{polarity(T, d[\tau_1, \cdots, \tau_r]) = pol_T(T_1) * p_1 \sqcap \cdots \sqcap pol_T(T_r) * p_r}$$

We will assume hereafter that a valid type parameter polarity assignment is available for every program we consider.

3.2.2 Datatype Polarity

In our system, we are interested in two polarities for datatypes, namely positive and strictly positive. Positivity allows us to relate the denotations of datatypes with identical type parameter instantiations but different size expressions. Strict positivity further enables construction and deconstruction of intersection datatypes by dropping the size binding from constructor field types. Similarly to the notion of type parameter polarity, we rely on semantic definitions for datatype polarities. Indeed, precisely defining the polarity computations is fairly complex in our fragment due to the size bindings in datatype definitions. However, we have defined the computations in the variant formalized in Coq and shown that they imply the semantic definitions given here.

Let us start by considering the notion of positive polarity. This notion enables a second dimension of subtyping for datatypes in addition to the one given by type parameter polarity by relating sized datatypes with distinct size expressions.

Definition 7. *We say a datatype* $(\textbf{type}\ \ d[\overline{\tau}_d](m)\ :=\ \cdots) \in P$ *is* positive *if for each* $v_1, v_2 \in [\![\mathsf{Nat}]\!]_v$ *such that* $v_1 < v_2$ *and* $(P_{in}, \theta) \in [\![P; \overline{\tau}_d; \emptyset]\!]_v$*, we have* $[\![\theta(d[\overline{\tau}_d](v_1))]\!]_v \subseteq [\![\theta(d[\overline{\tau}_d](v_2))]\!]_v$.

We then consider the notion of strictly positive polarity. Strict positivity ensures that the datatype intersection can be pushed down into the constructor field types. Given a datatype definition $(\textbf{type}\ \ d[\overline{\tau}_d](m)\ :=\ C_1(x_1 : \tau_1)\ \mid\ \cdots\ \mid\ C_n(x_n : \tau_n)) \in P$, we introduce a helper function $\mathsf{intersect}_d : \mathit{type} \rightarrow \mathit{type}$ which takes a type τ and returns a type τ' obtained by replacing all occurrences in τ of sized datatypes which are mutually recursive with d by the corresponding intersection datatypes, namely $\tau' = \tau[d'[\overline{\tau}'](e_m) \text{ s.t. } d \sim d' / d'[\overline{\tau}']]$. Given an intersection datatype $d[\overline{\tau}]$, the type $\mathsf{intersect}_d(\tau_i)[\overline{\tau}_d/\overline{\tau}]$ effectively corresponds to pushing the intersection down to the (mutually) recursive occurrences of the datatype in the constructor field type. Strict positivity then ensures that the denotation of the intersection datatype can be equivalently computed by ignoring the quantification on $[\![\mathsf{Nat}]\!]_v$ and considering instead the type $\mathsf{intersect}_d(\tau_i)[\overline{\tau}_d/\overline{\tau}]$ associated to each field type τ_i.

Definition 8. *We say a datatype* $(\textbf{type}\ \ d[\overline{\tau}_d](m)\ :=\ C_1(x_1 : \tau_1)\ \mid\ \cdots\ \mid\ C_r(x_r : \tau_r)) \in P$ *is* strictly positive *if for* $1 \leq i \leq n$*, the type* $\mathsf{intersect}_d(\tau_i)$ *is reducible, and for value* $C_i[\overline{\tau}](v)$*, we have* $C_i[\overline{\tau}](v) \in [\![d[\overline{\tau}]]\!]_v$ *iff* $v \in [\![\mathsf{intersect}_d(\tau_i)[\overline{\tau}_d/\overline{\tau}]]\!]_v$.

For datatypes where the size binding has been automatically injected, (strict) positivity can be computed through a recursive traversal of the constructor field types defined by similar rules to ones exposed in the case of type parameter polarity. We will again assume hereafter that datatypes are validly marked as (strictly) positive (or not) in every considered program.

3.2.3 Type Generalization

In order to show reducibility of recursive functions, we rely on the size binding in the functions definition. However, once reducibility has been shown, the size expression that appears within the function parameter and return types often becomes an unnecessary burden. We will therefore be interested in *generalizing* such types to allow omitting the size expression.

Let us consider the reducible function definition (**def** $f(m:\tau_m)[\overline{\tau}_f](x:\tau_1):\tau_2 \ := \ e_f) \in P$. It is clear by definition of reducibility that we have

$$\forall\,(P_{in},\theta,\gamma) \in [\![P;\overline{\tau}_f;m:\tau_m,x:\tau_1]\!]_v.\ \gamma(\theta(f(m)[\overline{\tau}_f](x))) \in [\![\gamma(\theta(\tau_2))]\!]_v$$

As the well-order τ_m does not depend on $\overline{\tau}_f$ and evaluation of a reducible function call is independent of the size expression, this property holds iff

$$\forall\, v_m \in [\![\tau_m]\!]_v.\ \forall\,(P_{in},\theta,\gamma) \in [\![P;\overline{\tau}_f;x:\tau_1[m/v_m]]\!]_v.\ \gamma(\theta(f[\overline{\tau}_f](x))) \in [\![\gamma(\theta(\tau_2[m/v_m]))]\!]_v$$

Let us consider a type parameter substitution θ, a value substitution γ, and a set $S \subseteq [\![\tau_m]\!]_v$ of values from the well-order τ_m such that for $v_s \in S$, we have $\gamma(x) \in [\![\gamma(\theta(\tau_1[m/v_s]))]\!]_v$. If it is the case that for $v_m \in [\![\tau_m]\!]_v$, we have $[\![\gamma(\theta(\tau_2[m/v_m]))]\!]_v \subseteq \bigcap_{v_s\in S}[\![\gamma(\theta(\tau_2[m/v_s]))]\!]_v$, then the call $f[\theta(\overline{\tau}_f)](\gamma(x))$ is such that for $v_m \in [\![\tau_m]\!]_v$, we have $f[\theta(\overline{\tau}_f)](\gamma(x)) \in [\![\gamma(\theta(\tau_2[m/v_m]))]\!]_e$. If we had $\tau_2 = d[\overline{\tau}](m)$, we could thus successfully generalize the type inferred for the call to the intersection datatype $d[\overline{\tau}]$ following the definition of $[\![d[\overline{\tau}]]\!]_v$.

Let us now clarify these considerations with a concrete procedure to establish sound type generalizations. Our generalization approach is specialized to removing measure annotations which appear as refinements of the shape $\{x:\tau \mid e \leq m\}$ and size expressions in datatypes. We introduce the following generalization relations $\vdash (\tau, m)\ \mathsf{gen}_{\subseteq}\ \tau'$ and $\vdash (\tau, m)\ \mathsf{gen}_{\supseteq}\ \tau'$ which generalize type τ by removing references to the size binding m. We rely on two distinct generalization relations in order to handle contra-variance in function parameters. In order to allow dropping measure annotations, generalization relates the resulting type with a union of intersections. Namely, if $\vdash (\tau, m)\ \mathsf{gen}_{\subseteq/\supseteq}\ \tau'$, then the set $\bigcup_{n_1\in[\![\mathsf{Nat}]\!]_v}\bigcap_{n_2\in[\![\mathsf{Nat}]\!]_v,n_1\leq n_2}[\![\tau[m/n_2]]\!]_v$ will be a subset, respectively superset of $[\![\tau']\!]_v$ (modulo some type and value substitutions). Similarly to $\mathsf{intersect}_d$, the generalization procedures therefore eliminate the size binding from the given type, but have a different consequence on the denotation of the generalized type. We can then define the two generalization procedures as follows.

$$\frac{m \notin FV(\tau)}{\vdash (\tau, m)\ \mathsf{gen}_{\subseteq/\supseteq}\ \tau} \qquad \frac{\vdash (\tau_1, m)\ \mathsf{gen}_{\subseteq/\supseteq}\ \tau_1' \qquad \vdash (\tau_2, m)\ \mathsf{gen}_{\subseteq/\supseteq}\ \tau_2'}{\vdash (\Sigma x:\tau_1.\,\tau_2, m)\ \mathsf{gen}_{\subseteq/\supseteq}\ \Sigma x:\tau_1'.\,\tau_2'}$$

$$\frac{m \notin FV(e) \qquad \vdash (\tau, m)\ \mathsf{gen}_{\subseteq/\supseteq}\ \tau'}{\vdash (\{x:\tau \mid e\leq m\}, m)\ \mathsf{gen}_{\subseteq/\supseteq}\ \tau'} \qquad \frac{m \notin FV(p) \qquad \vdash (\tau, m)\ \mathsf{gen}_{\subseteq/\supseteq}\ \tau'}{\vdash (\{x:\tau \mid p\}, m)\ \mathsf{gen}_{\subseteq/\supseteq}\ \{x:\tau' \mid p\}}$$

$$\frac{\vdash (\tau_1, m)\ \mathsf{gen}_{\supseteq}\ \tau_1' \qquad \vdash (\tau_2, m)\ \mathsf{gen}_{\subseteq}\ \tau_2'}{\vdash (\Pi x:\tau_1.\,\tau_2, m)\ \mathsf{gen}_{\subseteq}\ \Pi x:\tau_1'.\,\tau_2'} \qquad \frac{m \notin FV(\tau_i)\ \text{ for }\ \tau_i \in \overline{\tau}}{\vdash (d[\overline{\tau}](e_m), m)\ \mathsf{gen}_{\supseteq}\ d[\overline{\tau}]}$$

$$\frac{m \notin FV(\tau_i)\ \text{ for }\ \tau_i \in \overline{\tau} \qquad \bigcup_{n_1\in[\![\mathsf{Nat}]\!]_v}\bigcap_{n_2\in[\![\mathsf{Nat}]\!]_v,n_1\leq n_2}[\![d[\overline{\tau}](e_m[m/n_2])]\!]_v \subseteq [\![d[\overline{\tau}]]\!]_v}{\vdash (d[\overline{\tau}](e_m), m)\ \mathsf{gen}_{\subseteq}\ d[\overline{\tau}]}$$

Note that the two relations only differ for pi-types and sized datatypes. Further note that $\mathsf{gen}_{\supseteq}$ is not defined for pi-types which depend on m and therefore $\vdash (\tau,\, m)\ \mathsf{gen}_{\subseteq}\ \tau'$ will only hold when m appears in strictly positive positions in τ.

Lemma 8. *For typing environment $P;\Theta;\Gamma$, size binding $m : \tau_m$, reducible type τ_1, type τ_2 and reducible inputs $(P_{in}, \theta, \gamma) \in [\![P;\Theta;\Gamma]\!]_v$, if we have $\vdash (\tau_1,\, m)\ \mathsf{gen}_{\subseteq/\supseteq}\ \tau_2$, then*

$$\bigcup_{n_1 \in [\![\mathsf{Nat}]\!]_v} \bigcap_{n_2 \in [\![\mathsf{Nat}]\!]_v, n_1 \leq n_2} [\![\gamma(\theta(\tau_1[m/n_2]))]\!]_v \ \subseteq / \supseteq\ [\![\gamma(\theta(\tau_2))]\!]_v$$

Proof. The proof proceeds by induction on the derivation of $\vdash (\tau_1,\, m)\ \mathsf{gen}_{\subseteq/\supseteq}\ \tau_2$. □

The condition on sized datatype generalization in $\mathsf{gen}_{\subseteq}$ is hard to show in practice. However, in the case of positive datatypes, we have $[\![d[\overline{\tau}](\mathsf{Succ}(e_m))]\!]_v \subseteq [\![d[\overline{\tau}](e_m)]\!]_v$. Hence, it suffices to show that $\forall\, n_1 \in [\![\mathsf{Nat}]\!]_v.\ \exists\, n_2 \in [\![\mathsf{Nat}]\!]_v.\ e_m[m/n_2] \geq n_1 \to^* \mathbf{true}$, which is a much simpler property. This property is again implied by monotonicity of e_m. In practice, one can therefore rely on a template-based approach that allows generalization when e_m is a polynomial in n with positive greatest degree factor.

3.3 Embedding Reducibility

In this section, we will define an embedding procedure for the reducibility relation. In the context of verification, we are mostly interested in *proofs*, namely in showing that there exist no (reducible) inputs such that evaluation can reach some value. Indeed, if we can show that some expression is reducible in $[\![\mathsf{Boolean}]\!]_v$ *and* that there exist no reducible inputs such that the expression evaluates to **false**, then we know it will evaluate to **true** for all reducible inputs. The main purpose of the reducibility relation embedding will therefore be to restrict considered inputs to only those that are reducible. As we are interested here in showing input inexistence, we will focus on the *completeness* property of the embedding.

Let us first consider the embedding, extraction and model finding procedures presented in the previous chapters. Similarly to the erased typing judgement, we extend these to the dependent type setting embedding erasures. We therefore write $P;\Theta;\Gamma \vdash (b,\, e) \rhd (t,\, \Phi)$ when the erasure of e is embedded into t under the erasure of $P;\Theta;\Gamma$ (and similarly for $\lhd$, $\overset{ext}{\rhd}$ and $\overset{ext}{\lhd}$). It is important to realize here that reducible inputs are a subset of the general inputs considered in the previous chapters as the erased typing judgement holds for reducible values by definition of $[\![\cdot]\!]_v$. As our model finding procedures are rooted in the operational semantics, they remain applicable in the presence of dependent types and can therefore be used directly to show the inexistence of reducible inputs. However, such an approach would completely disregard the constraints that dependent types impose on valid inputs. We therefore want to leverage these types in order to improve the precision of model finding with respect to reducible inputs.

Similarly to the lambda embedding presented in the previous chapter, we rely on a type normalization procedure for the reducibility relation embedding. However, we require weaker conditions on the normalization as it will not participate in equality evaluation. We therefore define $\mathit{normalize}(\tau) = (\tau', \gamma')$ the *normalization* of type τ such that

1. γ' is a pseudo-value substitution such that $\mathsf{dom}(\gamma') = FV(\tau')$,
2. for inputs P_{in}, θ, γ, we have $[\![\gamma(\theta(\tau))]\!]_v = [\![\gamma(\theta(\gamma'(\tau')))]\!]_v$, and
3. the identifiers in γ' are *normalized* as defined in the previous chapter.

Note that the second condition above is satisfied when $\gamma'(\tau') = \tau$, but more aggressive normalizations are also possible. Based on this normalization procedure, the embedding of the reducibility relation then relies on a special $\mathit{reducible}_{\tau'}$ predicate that is parametric in τ', the normalized structure of τ.

$$P;\Theta;\Gamma \vdash (b,\ t,\ \mathsf{Boolean})\ \overset{red}{\rhd}(\mathit{true}, \emptyset) \qquad\qquad P;\Theta;\Gamma \vdash (b,\ t,\ \mathsf{Unit})\ \overset{red}{\rhd}(\mathit{true}, \emptyset)$$

$$P;\Theta;\Gamma \vdash (b,\ t,\ \mathsf{val}[\tau])\ \overset{red}{\rhd}(\mathit{true}, \emptyset)$$

$$\frac{P;\Theta;\Gamma \vdash (b,\ t,\ \tau)\ \overset{red}{\rhd}(t_\tau, \Phi_\tau) \qquad P;\Theta;\Gamma, x:\tau \vdash (b,\ p) \rhd (t_p, \Phi_p)}{P;\Theta;\Gamma \vdash (b,\ t,\ \{x:\tau \mid p\})\ \overset{red}{\rhd}(t_\tau \wedge t_p, \Phi_\tau \cup \Phi_p \cup \{b \implies x \simeq t\})}$$

$$\frac{\begin{array}{c}P;\Theta;\Gamma \vdash (b,\ x,\ \tau_1)\ \overset{red}{\rhd}(t_1, \Phi_1) \qquad P;\Theta;\Gamma, x:\tau_1 \vdash (b,\ \pi_{(\mathit{erase}(\tau_1), \mathit{erase}(\tau_2)),2}(t),\ \tau_2)\ \overset{red}{\rhd}(t_2, \Phi_2)\\ \Phi_{\mathit{proj}} = \{b \implies x \simeq \pi_{(\mathit{erase}(\tau_1), \mathit{erase}(\tau_2)),1}(t)\}\end{array}}{P;\Theta;\Gamma \vdash (b,\ t,\ \Sigma x:\tau_1.\ \tau_2)\ \overset{red}{\rhd}(t_1 \wedge t_2, \Phi_1 \cup \Phi_2 \cup \Phi_{\mathit{proj}})}$$

$$\frac{\mathit{normalize}(\tau) = (\tau', \{y_1 \mapsto e_1, \cdots, y_n \mapsto e_n\}) \qquad P;\Theta;\Gamma \vdash (b,\ e_i) \rhd (t_i, \Phi_i)\ \text{ for }\ 1 \le i \le n}{P;\Theta;\Gamma \vdash (b,\ t,\ \tau)\ \overset{red}{\rhd}(\mathit{reducible}_{\tau'}(t, t_1, \cdots, t_n)), \Phi_1 \cup \cdots \cup \Phi_n)}$$

Figure 3.5 – Reducibility relation embedding rules. Note that the final rule which relies on the type normalization only applies if no other rules does.

Let us now define the embedding of the reducibility relation $P;\Theta;\Gamma \models e \in [\![\tau]\!]$ into SMT terms and clauses. Given a blocker constant b_τ and the embedding t_e of expression e, the reducibility relation embedding will produce an SMT term t_τ and set of SMT clauses Φ_τ such that t_τ and Φ_τ constrain t_e to be in the denotation of τ. Notation wise, we write this embedding as $P;\Theta;\Gamma \vdash (b_\tau,\ t_e,\ \tau)\ \overset{red}{\rhd}(t_\tau, \Phi_\tau)$. The reducibility relation embedding rules are presented in Figure 3.5. Certain types, namely booleans, refinement types and sigma-types imply straightforward constraints on the given term. The constraints stemming from the remaining types are approximated through the use of the $\mathit{reducible}_{\tau'}$ predicates described above. Note that we preserve the $b \implies c$ shape of the generated clauses and Lemma 1 holds for this embedding procedure as well. Given an embedding $(b_\tau,\ t_e,\ \tau)\ \overset{red}{\rhd}(t_\tau, \Phi_\tau)$, we will want to discuss the embedding of sub-expressions and reducibility relations associated to sub-types in τ. We therefore extend the notations $[\![\lfloor\cdot\rfloor]\!]_t$ and $[\![\lfloor\cdot\rfloor]\!]_b$ to each embedded $l' : e'/\tau' \sqsubseteq \tau$. Note that similarly to how the blocker constant is uniquely determined by the sub-expression, both the associated term and blocker constant under which the reducibility relation embedding occurs are uniquely given by the sub-type.

This embedding will not *precisely* encode the reducibility relation. Indeed, as pi-types can encode arbitrary functional dependencies, model finding in this context becomes as hard as recursive function synthesis over unbounded domains. Instead, the embedding will constitute a sound *under-approximation* of the reducibility relation which has shown effective in practice. More precisely, if $P;\Theta;\Gamma \models e \in [\![\tau]\!]$ and $P;\Theta;\Gamma \vdash (b_\tau,\ t_e,\ \tau)\ \overset{red}{\rhd} (t_\tau, \Phi_\tau)$, then the clause set $\Phi_\tau \cup \{b_\tau, t_\tau\}$ should be satisfiable.

We are interested in showing completeness of the reducibility relation embedding. Similarly to model consistency with function call interpretations and function application interpretations, we need a notion of consistency for reducible predicates. Note that as the set of values has not changed since the previous chapter, we extend the $\overset{ext}{\rhd}$ and $\overset{ext}{\lhd}$ procedures to the dependently-typed language simply by embedding erased programs, types and values. We can therefore introduce a notion of consistency between models and reducible predicate interpretations.

Definition 9. *For program P, model M, inputs P_{in},θ,γ, known lambdas Λ and reducibility symbol interpretation $(reducible_{\tau'}(t, t_1, \cdots, t_n) \mapsto t') \in M$, given the set of (typed) free variables $FV(\tau') = \{y_1 : \tau_1, \cdots, y_n : \tau_n\}$, we say M is consistent with the interpretation if $v \overset{ext}{\lhd} (M(t), \tau')$, $v_i \overset{ext}{\lhd} (M(t_i), \tau_i)$ for $1 \le i \le n$, the substituted type $\theta(\tau')[y_1/v_1, \cdots, y_n/v_n]$ is reducible, and we have $\models t'$ iff $v \in [\![\theta(\tau')[y_1/v_1, \cdots, y_n/v_n]]\!]_v$.*

Similarly to function calls and applications, we are generally interested in establishing consistency with embedding results. Given a model M and embedded reducibility relation associated to $l : \tau$ such that $\lfloor\!\lfloor \tau \rfloor\!\rfloor_t = reducible_{\tau'}(t, t_1, \cdots, t_n)$, we say M is consistent with the reducibility relation embedding if $M \models \neg\lfloor\!\lfloor \tau \rfloor\!\rfloor_b$ or M is consistent with the interpretation $reducible_{\tau'}(M(t), M(t_1), \cdots, M(t_n)) \mapsto M(reducible_{\tau'}(t, t_1, \cdots, t_n))$. Given these considerations, we can now state completeness of our reducibility relation embedding.

Lemma 9. *For embedded expression e, typing environment $P;\Theta;\Gamma$, reducible type τ such that $P;\Theta;\Gamma \vdash e : \tau$, reducibility relation embedding $P;\Theta;\Gamma \vdash (b_\tau, \lfloor\!\lfloor e \rfloor\!\rfloor_t, \tau)\ \overset{red}{\rhd} (t_\tau, \Phi_\tau)$, reducible inputs $(P_{in}, \theta, \gamma) \in [\![P;\Theta;\Gamma]\!]_v$ and known lambdas Λ, if $\gamma(\theta(e)) \rightarrow^* v$ where $v \in value$, then there exists model $M \models \Phi_\tau \cup \{b_\tau\}$ such that M agrees with γ, M is consistent with embedded calls and applications in e, M is consistent with embedded calls, applications and types in τ, $v \overset{ext}{\lhd} (M(\lfloor\!\lfloor e \rfloor\!\rfloor_t), \tau)$, and $M \models t_\tau$ iff $v \in [\![\gamma(\theta(\tau))]\!]_v$.*

Proof. Given the clause set Φ_e generated when embedding e, Lemma 6 gives us an initial model $M_e \models \Phi_e \cup \{\lfloor\!\lfloor e \rfloor\!\rfloor_b\}$ such that M_e agrees with γ, M_e is consistent with all embedded calls and applications in e, and $v \overset{ext}{\lhd} (M_e(\lfloor\!\lfloor e \rfloor\!\rfloor_t), \tau)$. We then show that M exists by induction on τ.

Case $\tau = \mathsf{Boolean}$**,** $\tau = \mathsf{Unit}$ **or** $\tau = \mathsf{val}[\tau']$: By preservation, we have $v \in [\![\tau]\!]_v$ and $M = M_e$.

Case $\tau = \{x : \tau' \mid p\}$: Consider embeddings $(b_\tau, \lfloor\!\lfloor e \rfloor\!\rfloor_t, \tau') \overset{red}{\rhd} (t'_\tau, \Phi'_\tau)$ and $(b_\tau, p) \rhd (t_p, \Phi_p)$. By reducibility of τ, induction and Lemma 6, there exist models $M' \models \Phi'_\tau \cup \{b_\tau\}$ and $M_p \models \Phi_p \cup \{b_\tau\}$. As all models agree on common interpretations, we let $M = M_e \cup M' \cup M$. If $v \in \llbracket \gamma(\theta(\tau)) \rrbracket_v$, then by definition of $\llbracket\cdot\rrbracket_v$, we have $M' \models t'_\tau$, $M_p \models t_p$ and therefore $M \models t_\tau$. If $v \notin \llbracket \gamma(\theta(\tau)) \rrbracket_v$, then either $M' \models \neg t'_\tau$ or $M_p \models \neg t_p$, and we have $M \models \neg t_\tau$.

Case $\tau = \Sigma\, x : \tau_1.\, \tau_2$: This case follows by induction on τ_1 and τ_2.

Case $\tau \in \Theta$**,** $\tau = d[\overline{\tau}]$**,** $\tau = d[\overline{\tau}](e_m)$ **or** $\tau = \Pi\, x : \tau_1.\, \tau_2$: Consider normalization $\mathit{normalize}(\tau) = (\tau', \{y_1 \mapsto e_1, \cdots, y_n \mapsto e_n\})$, (typed) free variables $FV(\tau') = \{y_1 : \tau_1, \cdots, y_n : \tau_n\}$ and embeddings $(b_\tau, e_i) \rhd (t_i, \Phi_i)$ for $1 \le i \le n$. We know by definition of normalization that each $e_i \in \mathit{pvalue}$, and therefore $\gamma(\theta(e_i)) \in \mathit{value}$. By Lemma 6, for $1 \le i \le n$ there exists $M_i \models \Phi_i \cup \{b_\tau\}$ such that $\gamma(\theta(e_i)) \overset{ext}{\lhd} (M_i(t_i), \tau_i)$. Now consider embeddings $(v, \tau) \overset{ext}{\rhd} t_v$ and $(\gamma(\theta(e_i)), \tau_i) \overset{ext}{\rhd} t'_i$ for $1 \le i \le n$. As each M_i agrees with γ and e_i contains no function call or application, we let $t_b = \mathit{true}$ if $v \in \llbracket \gamma(\theta(\tau)) \rrbracket_v$ and $t_b = \mathit{false}$ otherwise, and have $M = M_e \cup M_1 \cup \cdots \cup M_n \cup \{\mathit{reducible}_{\tau'}(t_v, t'_1, \cdots, t'_n) \mapsto t_b\}$. Finally, as $P; \Theta; \Gamma \models \tau$ *type* and $\tau = \tau'[y_1/e_1, \cdots, y_n/e_n]$, we have $\theta(\tau')[y_1/\gamma(\theta(e_1)), \cdots, y_n/\gamma(\theta(e_n))]$ reducible by distributivity of substitutions and definition of reducibility for types. □

3.4 Unfolding Procedure

In this section, we will discuss how the reducibility relation embedding can be integrated into the general unfolding procedure we presented in the previous chapters. We will then show that the extended model finding procedure remains sound for proofs.

The model finding procedure will take as input a program $P = P_1 \cup P_2$ where P_1 is a reducible program and P_2 is a set of mutually recursive definitions, as discussed in the context of bounded reducibility. Recall that the set P_2 will have an associated well-order τ_m. Now assume we are further given some $v_n \in \llbracket \tau_m \rrbracket_v$ such that all datatype definitions in P_2 are reducible below v_n. (Note that if there is at least one datatype definition in P_2, then we have $\tau_m = \mathsf{Nat}$.) It is important to note here that although we consider the value v_n to be given, our procedure will not rely on this bound in any way, and it simply constitutes an instrumentation on which our statements *about* the procedure will depend.

Let us now introduce a notion of bounded types. Consider an inclusion relation $\sqsubseteq_\tau$ such that for $\tau_1, \tau_2 \in \mathit{type}$, we have $\tau_1 \sqsubseteq_\tau \tau_2$ iff $\tau_1 \sqsubseteq \tau_2$ *and* there exists no $e_\tau \in \mathit{expr}$ such that $\tau_1 \sqsubseteq e_\tau \sqsubseteq \tau_2$. We say a type τ is *bounded by* v_n iff for $d[\overline{\tau}](e_m) \sqsubseteq_\tau \tau$ where $d \in P_2$, we have $e_m \in \llbracket \{n' : \mathsf{Nat} \mid n' < v_n\} \rrbracket_e$, and for $d[\overline{\tau}] \sqsubseteq_\tau \tau$, we have $d \in P_1$. Similarly to consistency, we extend this notion to embedded reducibility relations as follows.

Definition 10. *For reducible program P_1, set of mutually recursive definitions P_2, model M, inputs P_{in}, θ, γ, known lambdas Λ and type τ with embedding $\lfloor\!\lfloor \tau \rfloor\!\rfloor_t = \mathit{reducible}_{\tau'}(t, t_1, \cdots, t_r)$, given the set of (typed) free variables $FV(\tau') = \{y_1 : \tau_1, \cdots, y_r : \tau_r\}$, we say M bounds τ by v_n if $v_i \overset{ext}{\lhd} (M(t_i), \tau_i)$ for $1 \le i \le r$ and the type $\theta(\tau')[y_1/v_1, \cdots, y_r/v_r]$ is bounded by v_n.*

We then introduce bounded typing contexts and say that Γ is bounded by v_n iff for $(x,\tau) \in \Gamma$, the type τ is bounded by v_n. The model finding procedure will take as further input a reducible typing context $\Gamma = m : \tau_m, \Gamma'$ and a value $v'_n \in [\![\tau_m]\!]_v$ such that the substituted context $\Gamma'[m/v'_n]$ is bounded by v_n. Again, this value v'_n corresponds to a proof instrumentation.

In addition to the program P, bounds v_n, v'_n and typing context Γ described above, the model finding procedure is finally given a set of type variables Θ, an expression e and a value v. The unfolding procedure again tracks a clause set Φ_i, a set F_i of known function calls that have yet to be unfolded, a set A_i of known applications, a set Λ_i of known lambdas and a set D_i of application - lambda pairs that have already been dispatched. We further track the set R_i of types associated to reducibility predicates that have yet to be unfolded. We define the following helper function that collects all sub-types which have an associated reducibility predicate embedding within a given type:

$$R(\tau) = \{\tau' \mid \tau' \sqsubseteq \tau,\ (\tau' \in \Theta \vee \tau' = d[\overline{\tau}] \vee \tau' = d[\overline{\tau}](e_m) \vee \tau' = \Pi\, x : \tau_1.\, \tau_2),\ \lfloor\!\lfloor \tau' \rfloor\!\rfloor_t \text{ defined}\}$$

We also extend the helper functions $F(\cdot)$, $A(\cdot)$ and $\Lambda(\cdot)$ to operate on types as well. In the following, due to similarities between how calls yet to be unfolded, known applications and known lambdas are handled, we will sometimes use a placehoder S instead of F, A, or Λ when some procedure or definition is identical for all three sets.

Before moving on to the procedure itself, let us discuss the main property we want it to satisfy. Our end goal is to ensure that if there exist reducible inputs to the original expression, then the clause set Φ_i is satisfiable. However, in order to enable induction, we want a slightly stronger property. Based on the notions of agreement and consistency in relation with call, application and reducibility relation interpretations, we introduce a general notion of consistency with inputs at unfolding step i.

Definition 11. *For reducible program P_1, set of mutually recursive definitions P_2, model M_i, inputs P_{in}, θ, γ and unfolding step i, we say M_i is* consistent *with P_{in}, θ, γ at step i iff 1) $M_i \models \Phi_i$, 2) M_i agrees with γ, 3) M_i is consistent with calls in $F_i \cup U_i$, applications in A_i and reducibility relations associated to types in R_i, and 4) for $\tau \in R_i$, M_i bounds τ by v_n.*

We say an unfolding step from state i to state $i+1$ *preserves consistency* when given inputs P_{in}, θ, γ and model M_i, if M_i is consistent with the inputs at step i, then there exists some M_{i+1} that is consistent with the inputs at step $i+1$.

Let us start by defining the initial unfolding state at step $i = 0$. We first introduce a fresh boolean constant b and compute the expression embedding $(b, e \approx v) \triangleright (t, \Phi)$. Given the typing context bindings $\Gamma = m : \tau_m, x_1 : \tau_1, \cdots, x_l : \tau_l$, we further compute the reducibility relation embeddings $(b, x_j, \tau_j) \overset{red}{\triangleright} (t'_j, \Phi'_j)$ for $1 \le j \le l$. We then define the state as follows.

$$\Phi_0 = \Phi \cup \{b, t, t'_1, \cdots, t'_l\} \cup \Phi'_1 \cup \cdots \cup \Phi'_l \qquad S_0 = S(e \approx v) \cup S(\tau_1) \cup \cdots \cup S(\tau_l) \qquad D_0 = \emptyset$$

$$R_0 = R(\tau_1) \cup \cdots \cup R(\tau_l)$$

The state at step $i+1$ is then inductively defined given the state at step i by either performing a call unfolding, an application unfolding, or a reducibility predicate unfolding. The call and application unfolding steps follow the same approach as presented in the previous chapters. Note that the call in F_i that was selected to be unfolded may present a size expression argument. As this argument is erased both during evaluation and embedding, we ignore it as well during unfolding. The set R_{i+1} of unfolded reducibility relations remains equal to R_i during call and application unfolding steps.

Let us now describe how the reducible predicates introduced during embedding can be unfolded to increase the precision of the reducibility relation embedding. Consider some type $\tau_i \in R_i$ with associated embedded reducible predicate $\llfloor \tau_i \rrfloor_t = \mathit{reducible}_{\tau'}(t, t_1, \cdots, t_r)$. Considering the reducibility relation embedding given in the previous section, we know that τ' is either 1) a sized datatype $d[\overline{\tau}'](e'_m)$, 2) an intersection datatype $d[\overline{\tau}']$, 3) a pi-type $\Pi\, x : \tau'_1.\, \tau'_2$, or 4) a type variables $T \in \Theta$. For the type variable $T \in \Theta$, the reducible predicate is already a precise embedding of the reducibility relation and no unfolding needs to be performed. Unfolding a reducible predicate therefore consists of three distinct procedures depending on the shape of τ'. We will rely in the following on the typing environment $P;\Theta;\Gamma_i$ and blocker constant b under which the reducibility relation associated to τ_i was embedded, as well as the typed free variables $FV(\tau') = \{y_1 : \tau_{y_1}, \cdots, y_r : \tau_{y_r}\}$.

It is important to realize here that given the denotations of intersection datatypes and pi-types, a precise unfolding which exactly captures the reducibility relation would need to rely on some form of universal quantification. This necessity of quantification is what disallows sound and complete model finding in the presence of dependent types. We have opted for an unfolding strategy that avoids generating quantified formulas. While this strategy leads to an under-approximation of the reducibility relation, it increases the predictability of the procedure as the generated clause sets remains within a decidable fragment.

Sized datatype. Given that τ' corresponds to a sized datatype $d[\overline{\tau}'](e'_m)$, consider the associated definition (**type** $d[\overline{\tau}_d](m) := C_1(x_1 : \tau_1) \mid \cdots \mid C_n(x_n : \tau_n)) \in P$. The denotation of $d[\overline{\tau}'](e'_m)$ tells us that for reducible inputs $(P_{in}, \theta, \gamma) \in [\![P;\Theta;\Gamma_i, y_1 : \tau_{y_1}, \cdots, y_r : \tau_{y_r}]\!]_v$, some value $v : d[\overline{\tau}']$ belongs to $[\![\gamma(\theta(d[\overline{\tau}'](e'_m)))]\!]_v$ iff $\gamma(\theta(e'_m)) \rightarrow^* v_m \in [\![\mathsf{Nat}]\!]_v$ and if $v_m > \mathsf{Zero}$, then there exists some constructor $C_j(x_j : \tau_j)$ of d such that $v = C_j[\mathit{erase}(\gamma(\theta(\overline{\tau}')))](v_1)$ and $v_1 \in [\![\gamma(\theta(\tau_j[m/v_m][\overline{\tau}_d/\overline{\tau}']))]\!]_v$. Our unfolding of the reducible predicate will therefore rely on embedding the size expression e'_m and the reducibility relation associated to each τ_j.

Let us start by computing the embedding $P;\Theta;\Gamma_i, y_1 : \tau_{y_1}, \cdots, y_r : \tau_{y_r} \vdash (b, e'_m) \triangleright (t_m, \Phi_m)$. Similarly to the match expression embedding presented in Chapter 1, we then embed the reducibility relation by splitting on the constructor cases. We start by introducing fresh boolean constants $b_1, \cdots, b_n$. For $1 \leq j \leq n$, we then compute the reducibility relation embeddings

$$P;\Theta;\Gamma_i, y_1 : \tau_{y_1}, \cdots, y_r : \tau_{y_r}, m : \mathsf{Nat} \vdash (b_j, x_j, \tau_j[\overline{\tau}_d/\overline{\tau}']) \overset{red}{\triangleright} (t_{\tau_j}, \Phi_{\tau_j})$$

Based on these considerations, we can define the unfolding result as follows.

$$\begin{aligned}\Phi_{i+1} = \Phi_i \;\cup\; & \Phi_m \cup (\textstyle\bigcup_{1\le j\le n} \Phi_{\tau_j}) \cup \{b \implies m \simeq t_m\} \cup \{b \implies y_j \simeq t_j \mid 1 \le j \le r\} \\ \cup\; & \{(b \wedge m > \mathsf{Zero}) \implies (\mathit{reducible}_{\tau'}(t, t_1, \cdots, t_r) \iff \textstyle\bigwedge_{1\le j\le n}(b_j \implies t_{\tau_j}))\} \\ \cup\; & \{(b \wedge m > \mathsf{Zero} \wedge \mathit{is}\text{-}C_{\mathit{erase}(\overline{\tau}'),j}(t)) \iff b_j \mid 1 \le j \le n\} \\ \cup\; & \{b_j \implies x_j \simeq x_{\mathit{erase}(\overline{\tau}'),j}(t) \mid 1 \le j \le n\}\end{aligned}$$

$$S_{i+1} = S_i \cup S(e'_m) \cup \textstyle\bigcup_{1\le j\le n} S(\tau_j[\overline{\tau}_d/\overline{\tau}']) \qquad\qquad D_{i+1} = D_i$$

$$R_{i+1} = R_i \setminus \{\tau_i\} \cup \textstyle\bigcup_{1\le j\le n} R(\tau_j[\overline{\tau}_d/\overline{\tau}'])$$

Lemma 10. *The sized datatype unfolding step preserves consistency.*

Proof. We consider inputs $(P_{in},\theta,\gamma) \in [\![P;\Theta;\Gamma]\!]_v$ and model M_i that is consistent with the inputs at step i. If $M_i \models \neg b$, then we extend M_i to M_{i+1} by setting all introduced blocker constants to *false* by Lemma 1. Let us consider the case where $M_i \models b$. Consistency of M_i with the reducibility relation associated to τ_i ensures that extractions $v \overset{ext}{\lhd} (M_i(t), \tau)$ and $v_j \overset{ext}{\lhd} (M_i(t_j), \tau_j)$ for $1 \le j \le r$ are defined. We let $\gamma' = \{y_j \mapsto v_j \mid 1 \le j \le r\}$ and consistency further ensures that $\gamma'(\theta(d[\overline{\tau}'](e'_m)))$ is reducible and $M_i \models \lfloor\!\lfloor \tau_i \rfloor\!\rfloor_t$ iff $v \in [\![\gamma'(\theta(d[\overline{\tau}'](e'_m)))]\!]_v$.

By reducibility of $\gamma'(\theta(d[\overline{\tau}'](e'_m)))$, we have $\gamma'(\theta(e'_m)) \to^* v_m \in [\![\mathsf{Nat}]\!]_v$. Consider embedding $(b, e'_m) \rhd (t_m, \Phi_m)$. By Lemma 6, there exists $M_m \models \Phi_m \cup \{b\}$ such that $v_m \overset{ext}{\lhd} (M_m(t_m), \mathsf{Nat})$. If $v_m = \mathsf{Zero}$, then we have $M_m \models \neg(t_m > \mathsf{Zero})$ and we can again extend M_i to M_{i+1} by setting blockers to *false* by Lemma 1. If $v_m > \mathsf{Zero}$, then we have $M_m \models t_m > \mathsf{Zero}$. By definition of the denotation, we have $v \in [\![\gamma'(\theta(d[\overline{\tau}'](e'_m)))]\!]_v$ iff $v = C_j[\mathit{erase}(\gamma'(\theta(\overline{\tau})))](v'_1)$ for some $1 \le j \le n$ and $v'_1 \in [\![\tau_j[m/v_m][\overline{\tau}_d/\gamma'(\theta(\overline{\tau}'))]]\!]_v$. Consider value substitution $\gamma'_j = \gamma' \cup \{m \mapsto v_m, x_j \mapsto v'_1\}$. By distributivity of substitutions, we have $[\![\tau_j[m/v_m][\overline{\tau}_d/\gamma'(\theta(\overline{\tau}'))]]\!]_v = [\![\gamma'_j(\theta(\tau_j[\overline{\tau}_d/\overline{\tau}']))]\!]_v$. By bounded reducibility and types, as well as the syntactic constraint on datatype definitions, we know that the type $\tau_j[m/v_m][\overline{\tau}_d/\overline{\tau}']$ is reducible. Hence, by Lemma 9, there exists a model $M_{\tau_j} \models \Phi_{\tau_j} \cup \{b_j\}$ such that $M_{\tau_j} \models t_{\tau_j}$ iff $v'_1 \in [\![\gamma'_j(\theta(\tau_j[\overline{\tau}_d/\overline{\tau}']))]\!]_v$. This implies in turn that we have $M_{\tau_j} \models t_{\tau_j}$ iff $v \in [\![\gamma'(d[\overline{\tau}'](e'_m)))]\!]_v$. For $1 \le k \le n$ where $k \ne j$ we define model $M_{\neg k}$ where b_k and all blockers introduced by the embedding of the reducibility relation associated to each $\tau_k[\overline{\tau}_d/\overline{\tau}']$ are set to *false*. By Lemma 1, we have $M_{\neg k} \models \Phi_{\tau_k}$. Finally, we define the union $M_{i+1} = M_i \cup M_{\tau_j} \cup \bigcup_{1\le k\le n, k\ne j} M_{\neg k}$ which is consistent at step $i+1$. □

Intersection datatype. Let us now consider the case where τ' is an intersection datatype $d[\overline{\tau}']$. Unfortunately, the denotation of the intersection datatype does not directly translate to a denotation on datatype's constructor fields in the general case. Indeed, the definition of reducibility tells us that for reducible inputs $(P_{in},\theta,\gamma) \in [\![P;\Theta;\Gamma_i, y_1 : \tau_{y_1}, \cdots, y_r : \tau_{y_r}]\!]_v$, a value $v : d[\overline{\tau}']$ belongs to $[\![\gamma(\theta(d[\overline{\tau}']))]\!]_v$ iff for all $v_m \in [\![\mathsf{Nat}]\!]_v$, we have $v \in [\![\gamma(\theta(d[\overline{\tau}'](v_m)))]\!]_v$. This quantification over n cannot be precisely embedded into our quantifier-free fragment in the general case. However, if the datatype is strictly positive, then the unfolding procedure becomes straightforward.

Consider the datatype definition (**type** $d[\overline{\tau}_d](m) := C_1(x_1 : \tau_1) \mid \cdots \mid C_n(x_n : \tau_n)) \in P$, and consider some value $v : d[\overline{\tau}']$ where we have $v = C_j[erase(\overline{\tau}')](v_1)$ for some $1 \le j \le n$. Now, let us assume that d is strictly positive, and further consider the type $\tau'_j = \mathsf{intersect}_d(\tau_j)$. By definition of strict positivity, we have $v \in [\![\gamma(\theta(d[\overline{\tau}']))]\!]_v$ iff $v_1 \in [\![\gamma(\theta(\tau'_j[\overline{\tau}_d/\overline{\tau}']))]\!]_v$. Based on these considerations, it is clear that the reducible predicate can be unfolded by relying on the intersected constructor field types.

It is interesting to note here that instead of relying on strict positivity, we could define the unfolding through the $\mathsf{gen}_{\supseteq}$ generalization procedure when the datatype is simply positive. This variant would allow generalizing measure annotations but would disallow datatype recursion under pi-types. In practice, we have found that relying on strict positivity when unfolding covered a larger set of use cases.

We will now define the unfolding result. If the datatype d does not have strictly positive polarity, then the unfolding maintains the current under-approximation and the unfolding step $i+1$ is defined as follows.

$$\Phi_{i+1} = \Phi_i \qquad S_{i+1} = S_i \qquad D_{i+1} = D_i \qquad R_{i+1} = R_i \setminus \{\tau_i\}$$

However, if the datatype is strictly positive, then we consider the constructor field types where the intersection has been pushed down into the (mutually) recursive datatype occurrences. For each $1 \le j \le n$, we again introduce a fresh boolean constant b_j and compute the following reducibility relation embedding

$$P;\Theta;\Gamma_i, y_1 : \tau_{y_1}, \cdots, y_r : \tau_{y_r} \vdash (b_j, x_j, \mathsf{intersect}_d(\tau_j)[\overline{\tau}_d/\overline{\tau}']) \overset{red}{\triangleright} (t_{\tau_j}, \Phi_{\tau_j})$$

The reducibility relation unfolding is then given similarly to the sized datatype case.

$$\begin{aligned}
\Phi_{i+1} = \Phi_i \;&\cup\; (\textstyle\bigcup_{j \in \mathsf{gen}_d} \Phi_{\tau_j}) \cup \{b \implies y_j \simeq t_j \mid 1 \le j \le r\} \\
&\cup\; \{b \implies (reducible_{\tau'}(t, t_1, \cdots, t_r) \iff \textstyle\bigwedge_{1 \le j \le n}(b_j \implies t_{\tau_j}))\} \\
&\cup\; \{(b \wedge is\text{-}C_{erase(\overline{\tau}'),j}(t)) \iff b_j \mid 1 \le j \le n\} \\
&\cup\; \{b_j \implies x_j \simeq x_{erase(\overline{\tau}'),j}(t) \mid 1 \le j \le n\}
\end{aligned}$$

$$S_{i+1} = S_i \cup \textstyle\bigcup_{1 \le j \le n} S(\tau'_j[\overline{\tau}_d/\overline{\tau}']) \qquad D_{i+1} = D_i \qquad R_{i+1} = R_i \setminus \{\tau_i\} \cup \textstyle\bigcup_{1 \le j \le n} R(\tau'_j[\overline{\tau}_d/\overline{\tau}'])$$

Lemma 11. *The intersection datatype unfolding step preserves consistency.*

Proof. We consider inputs $(P_{in}, \theta, \gamma) \in [\![P;\Theta;\Gamma]\!]_v$ and model M_i that is consistent with the inputs at step i. We again focus on the case where $M_i \models b$ and consider extractions $v \overset{ext}{\triangleleft} (M_i(t), \tau)$ and $v_j \overset{ext}{\triangleleft} (M_i(t_j), \tau_j)$ for $1 \le j \le r$. We also let $\gamma' = \{y_j \mapsto v_j \mid 1 \le j \le r\}$ and have $\gamma'(\theta(d[\overline{\tau}']))$ reducible and $M_i \models \lfloor\!\lfloor \tau_i \rfloor\!\rfloor_t$ iff $v \in [\![\gamma'(\theta(d[\overline{\tau}']))]\!]_v$. If d is not strictly positive, the case is trivial. We therefore consider the case where d is strictly positive.

Consider the definition (**type** $d[\overline{\tau}_d](m) := C_1(x_1 : \tau_1) \mid \cdots \mid C_n(x_n : \tau_n)) \in P$. We have $v : d[\theta(\overline{\tau})]$, and therefore $v = C_j[erase(\theta(\overline{\tau}))](v'_1)$ for some $1 \le j \le n$. Consider the value substitution $\gamma'_j = \gamma' \cup \{x_j \mapsto v'_1\}$. For each constructor index $1 \leq k \leq n$ where $k \neq j$ we construct a model $M_{\neg k}$ by setting all blocker constants to *false* similarly to the sized datatype case. Then, by definition of strict positivity, we have $v \in [\![\gamma'(\theta(d[\overline{\tau}']))]\!]_v$ iff $v'_1 \in [\![\gamma'(\theta(\mathsf{intersect}_d(\tau_j)[\overline{\tau}_d/\overline{\tau}']))]\!]_v$. We can therefore apply Lemma 9 to obtain a model $M_{\tau_j} \models \Phi_{\tau_j} \cup \{b_j\}$ such that $M_{\tau_j} \models t_{\tau_j}$ iff $v \in [\![\gamma'(\theta(d[\overline{\tau}']))]\!]_v$. We then let $M_{i+1} = M_i \cup M_{\tau_j} \cup \bigcup_{k \in \mathsf{gen}_d, k \neq j} M_{\neg k}$ and apply a similar argument to the sized datatype unfolding to conclude the proof. □

Pi-type. Finally, we consider the case where $\tau' = \Pi x : \tau'_1.\ \tau'_2$. The denotation of pi-types tells us that for reducible inputs $(P_{in}, \theta, \gamma) \in [\![P; \Theta; \Gamma_i, y_1 : \tau_{y_1}, \cdots, y_r : \tau_{y_r}]\!]_v$, a value $v : \Pi x : \tau'_1.\ \tau'_2$ belongs to $[\![\gamma(\theta(\Pi x : \tau'_1.\ \tau'_2))]\!]_v$ iff for $v'_1 \in [\![\gamma(\theta(\tau'_1))]\!]_v$, we have $v\ v'_1 \in [\![\gamma(\theta(\tau'_2[x/v'_1]))]\!]_e$. Note again the offending quantification which disallows a precise unfolding of the relation.

Although we will not precisely unfold the reducible predicate for pi-types, we can still constrain satisfying inputs in a useful way. Recall the expression e and typing environment $P; \Theta; \Gamma$ which were given to the unfolding procedure. Consider inputs $(P_{in}, \theta, \gamma) \in [\![P; \Theta; \Gamma]\!]_v$ and model M_i that is consistent with the inputs at step i. Further consider the extractions $\lambda x.\ e_v \overset{ext}{\lhd} (M(t), \Pi x : \tau'_1.\ \tau'_2)$ and $v_j \overset{ext}{\lhd} (M(t_j), \tau_{y_j})$ for $1 \leq j \leq r$. Finally, consider some application which is encountered during evaluation $\gamma(\theta(e)) \rightarrow^* \mathcal{E}[(\lambda x.\ e_v)\ v'_1]$. If the encountered application is such that we have $\lambda x.\ e_v \in [\![\theta(\Pi x : \tau'_1.\ \tau'_2)[y_1/v_1, \cdots, y_r/v_r]]\!]_v$ and $v'_1 \in [\![\theta(\tau'_1)[y_1/v_1, \cdots, y_r/v_r]]\!]_v$, then we have $(\lambda x.\ e_v)\ v'_1 \in [\![\theta(\tau'_2)[y_1/v_1, \cdots, y_r/v_r]]\!]_e$ by definition of the denotation.

Our model finding procedure tracks the set of known applications during unfolding. Let us therefore consider some application $l : e_1\ e_2 \in A_i$ such that $e_1 : \Pi x : \tau'_1.\ \tau'_2$. We introduce a fresh boolean constant b_1 which holds if the callers match *and* the reducibility predicate holds. We further introduce a fresh boolean constants b_2 that encodes the fact that the embedded parameter e_2 satisfies its denotation. We then compute the embedding of the reducibility relation between the embedded parameter e_2 and parameter type τ'_1 under blocker b_1

$$P; \Theta; \Gamma_i, y_1 : \tau_{y_1}, \cdots, y_r : \tau_{y_r} \vdash (b_1, \llbracket e_2 \rrbracket_t, \tau'_1) \overset{red}{\rhd} (t'_1, \Phi'_1)$$

We then further compute the reducibility relation embedding between the embedding of the application $e_1\ e_2$ and the result type τ'_2 under blocker b_2

$$P; \Theta; \Gamma_i, y_1 : \tau_{y_1}, \cdots, y_r : \tau_{y_r}, x : \tau'_1 \vdash (b_2, \llbracket e_1\ e_2 \rrbracket_t, \tau'_2) \overset{red}{\rhd} (t'_2, \Phi'_2)$$

Given these embeddings, we can finally define the unfolding result as follows.

$$\begin{aligned}\Phi_{i+1} = \Phi_i\ &\cup\ \Phi'_1 \cup \Phi'_2 \cup \{b_1 \implies y_j \simeq t_j \mid 1 \leq j \leq r\} \\ &\cup\ \{b_1 \iff (b \wedge \llbracket e_1\ e_2 \rrbracket_b \wedge \llbracket e_1 \rrbracket_t \simeq t \wedge \mathit{reducible}_{\tau'}(t, t_1, \cdots, t_r))\} \\ &\cup\ \{(b_1 \wedge t'_1) \iff b_2, b_2 \implies x \simeq \llbracket e_2 \rrbracket_t, b_2 \implies t'_2\}\end{aligned}$$

$$S_{i+1} = S_i \cup S(\tau'_1) \cup S(\tau'_2) \qquad D_{i+1} = D_i \qquad R_{i+1} = R_i \cup R(\tau'_1) \cup R(\tau'_2)$$

One could be tempted in the above to rely on a single blocker constant instead of b_1 and b_2. However, this would imply that the embedding of the reducibility relation associated to τ_1' would occur under blocker b. This in turn could force b to be invalidly falsified, for example if the callers do not match and we have $\tau_1' = \{x : \tau_? \mid \mathsf{err[Boolean]}\}$. Relying on two blocker constants thus allows us to precisely encode the dependencies between the embeddings.

It is important to realize that the clause set given above only ensures that satisfying models are consistent with the reducible predicate for this particular argument, whereas a precise unfolding would require this for *all* values in the parameter type's denotation.

Lemma 12. *The pi-type unfolding step preserves consistency.*

Proof. We consider inputs $(P_{in}, \theta, \gamma) \in [\![P;\Theta;\Gamma]\!]_v$ and model M_i that is consistent with the inputs at step i. We again assume that $M_i \models b$ as the statement is trivially given otherwise. Similarly to the application-lambda pair unfolding case, if we have either $M_i \models \neg\lfloor\!\lfloor e_1\ e_2 \rfloor\!\rfloor_b$, $M_i \models \lfloor\!\lfloor e_1 \rfloor\!\rfloor_t \not\approx t$, or $M_i \models \neg\lfloor\!\lfloor \tau \rfloor\!\rfloor_t$, then we have $M_i \models \neg b_1$ and can extend M_i to M_{i+1} by Lemma 1. Let us consider the case where $M_i \models b_1$.

Consider extractions $v \overset{ext}{\lhd} (M_i(t), \tau)$, $v_j \overset{ext}{\lhd} (M_i(t_j), \tau_j)$ for $1 \le j \le r$, $v' \overset{ext}{\lhd} (M_i(\lfloor\!\lfloor e_1\ e_2 \rfloor\!\rfloor_t), \tau_2')$ and $v_1' \overset{ext}{\lhd} (M_i(\lfloor\!\lfloor e_2 \rfloor\!\rfloor_t), \tau_1')$. Further consider value substitution $\gamma' = \{y_j \mapsto v_j \mid 1 \le j \le r\}$. Note that we have $v \in [\![\gamma'(\theta(\tau'))]\!]_v$. By definition of the pi-type denotation, if $v_1' \in [\![\gamma'(\theta(\tau_1'))]\!]_v$, then we have $v' \in [\![\gamma'(\theta(\tau_2'[x/v_1']))]\!]_v$.

We first consider the case where $v_1' \in [\![\gamma'(\theta(\tau_1'))]\!]_v$. By Lemma 9, there exist models M_1 and M_2 such that $M_1 \models \Phi_1' \cup \{b_1, t_1'\}$ and $M_2 \models \Phi_2' \cup \{b_2, t_2'\}$. We then let $M_{i+1} = M_i \cup M_1 \cup M_2$ and M_{i+1} is consistent with the inputs at step $i+1$. We then consider the case where $v_1' \notin [\![\gamma'(\theta(\tau_1'))]\!]_v$. By Lemma 9, there exists $M_1 \models \Phi_1' \cup \{b_1, \neg t_1'\}$. Lemma 1 further gives us model $M_2 \models \Phi_2' \cup \{\neg b_2\}$ and we again have $M_{i+1} = M_i \cup M_1 \cup M_2$ consistent with the inputs at step $i+1$. □

We have discussed how the reducibility relation unfoldings for intersection datatypes and pi-types are imprecise. However, they preserve soundness for proofs which is the main property of interest in this context.

Theorem 3. *For reducible program P_1, set of mutually recursive definitions P_2 with associated well-order τ_m, values $v_n, v_n' \in [\![\tau_m]\!]_v$ such that datatype definitions in P_2 are reducible below v_n, type variables Θ, reducible context $m : \tau_m, \Gamma$ such that $\Gamma[m/v_n']$ is bounded by v_n, expression e such that $m \notin FV(erase(e))$, value v and unfolding step i, if there exists no model $M \models \Phi_i$, then there exist no reducible inputs $(P_{in}, \theta, \gamma) \in [\![P;\Theta;\Gamma[m/v_n']]\!]_v$ such that $\gamma(\theta(e)) \to^* erase(\theta(v))$.*

Proof. We first show by induction on $i \in \mathbb{N}$ that if there exist inputs $(P_{in}, \theta, \gamma) \in [\![P;\Theta; m : \tau_m, \Gamma]\!]_v$ such that $\gamma(m) = v_n'$ and $\gamma(\theta(e \approx v)) \to^*$ **true**, then there exists a model M_i that is consistent with the inputs P_{in}, θ, γ at step i.

The base case $i = 0$ is given by Lemmas 6 and 9. For the inductive case, we assume a model M_i for which the hypothesis holds. We consider the following cases depending on what kind of unfolding step was performed:

Unfolding of call $f[\overline{\tau}](e_1)_i \in F_i$ (or similarly $f(e_m)[\overline{\tau}](e_1)_i \in F_i$) : Consider associated blocker constant $b_f = \lfloor\lfloor f[\overline{\tau}](\overline{e}) \rfloor\rfloor_b$. If $M_i \models \neg b_f$, then we can simply extend M_i to M_{i+1} by setting all introduced blocker constants to *false* by Lemma 1. If $M_i \models b_f$, then consistency of M_i with the call tells us that given the embedding $\lfloor\lfloor f[\overline{\tau}](e_1) \rfloor\rfloor_t = f_{\overline{\tau}}(t_1)$, associated definition (**def** $f(m : \tau_m)[\overline{\tau}_f](x : \tau_1) : \tau_2 \ := \ e_f) \in P$ and extractions $v_f \overset{ext}{\lhd} (M_i(f_{\overline{\tau}}(t_1)), \tau_2[\overline{\tau}_f/\overline{\tau}])$ and $v_1 \overset{ext}{\lhd} (M_i(t_1), \tau_1[\overline{\tau}_f/\overline{\tau}])$, we have $f[\theta(\overline{\tau})](v_1) \to^* v_f \in \mathit{value}$. Given the embedding $(b_f, e_f[\overline{\tau}_f/\overline{\tau}]) \rhd (t_f, \Phi_f)$, Lemma 6 then gives us a model $M_f \models \Phi_f \cup \{b_f\}$ which we can unify with M_i to obtain M_{i+1} that satisfies the inductive hypothesis.

Unfolding of application-lambda pair $(e_1\ e_2, \lambda x.\ e_b) \in A_i \times \Lambda_i$: Consider blocker constant b_b introduced during unfolding. If we have either $M_i \models \neg\lfloor\lfloor e_1\ e_2 \rfloor\rfloor_b$, $M_i \models \neg\lfloor\lfloor \lambda x.\ e_b \rfloor\rfloor_b$ or $M_i \models \lfloor\lfloor e_1 \rfloor\rfloor_t \neq \lfloor\lfloor \lambda x.\ e_b \rfloor\rfloor_t$, then we have $M_i \models \neg b_b$ and can again extend M_i to M_{i+1} by Lemma 1. We therefore consider the case where $M_i \models b_b$. Further consider embeddings $\lfloor\lfloor e_1\ e_2 \rfloor\rfloor_t = \mathit{dispatch}_{\tau_2 \to \tau}(t_\lambda, t_2)$ and $\lfloor\lfloor \lambda x.\ e_b \rfloor\rfloor_t = C_{\lambda x.\ e'_b}(t'_1, \cdots, t'_r)$, as well as the typed free variables $FV(\lambda x.\ e'_b) = \{y_1 : \tau'_1, \cdots, y_r : \tau'_r\}$. Consistency of M_i with $e_1\ e_2$ ensures extractions $\lambda x.\ e_v \overset{ext}{\lhd} (M_i(t_\lambda), \tau_2 \to \tau)$ and $v_2 \overset{ext}{\lhd} (M_i(t_2), \tau_2)$ are defined. By definition of $\overset{ext}{\lhd}$, we have $\lambda x.\ e_v = \theta(\lambda x.\ e'_b)[y_1/v'_1, \cdots, y_r/v'_r]$ where $v'_j \overset{ext}{\lhd} (M_i(t'_j), \tau'_j)$ for $1 \leq j \leq r$. Given the embedding $(b_b, e'_b) \rhd (t_b, \Phi_b)$, Lemma 6 again gives us a model $M_b \models \Phi_b \cup \{b_b\}$ that we can unify with M_i to obtain M_{i+1} which is consistent at step $i + 1$.

Unfolding of reducibility relation associated to $\tau \in R_i$: Given by Lemmas 10, 11 and 12.

If there exists no model $M_i \models \Phi_i$, then by contradiction there can exist no reducible inputs $(P_{in}, \theta, \gamma) \in \llbracket P; \Theta; m : \tau_m, \Gamma \rrbracket_v$ such that $\gamma(m) = v'_n$ and $\gamma(\theta(e \approx v)) \to^*$ **true**. By definition of input and typing context reducibility, we have $(P_{in}, \theta, \gamma) \in \llbracket P; \Theta; m : \tau_m, \Gamma \rrbracket_v$ where $\gamma(m) = v_m$ iff $(P_{in}, \theta, \gamma') \in \llbracket P; \Theta; \Gamma[m/v'_n] \rrbracket_v$ where $\gamma = \gamma' \cup \{m \mapsto v'_n\}$. As evaluation is performed on erased expressions and $m \notin FV(\mathit{erase}(e))$, the expressions $\gamma(\theta(e \approx v))$ and $\gamma'(\theta(e \approx v))$ will evaluate to the same value. Finally, the operational semantics ensure that $\gamma'(\theta(e \approx v)) \to^*$ **true** iff $\gamma'(\theta(e)) \to^* \mathit{erase}(\theta(v))$, which concludes our proof. □

Most of the technical details in the above statement relate to the induction through which reducibility of P_2 is shown. When the set P_2 is empty (namely when we are in the context of a reducible program), then the following much cleaner Corollary holds.

Corollary 3. *For reducible program P type variables Θ, reducible context Γ expression e, value v and unfolding step i, if there exists no model $M \models \Phi_i$, then there exist no reducible inputs $(P_{in}, \theta, \gamma) \in \llbracket P; \Theta; \Gamma \rrbracket_v$ such that $\gamma(\theta(e)) \to^* \mathit{erase}(\theta(v))$.*

3.5 Type Checking

In this section, we describe how expression, type and program reducibility can be established. The procedures we have discussed up to this point allow us to either find a set of inputs such that evaluation reaches some expected value, or show that no such inputs exist. However, we have not considered the question of showing that evaluation reaches some expected value (or set of values) for *all* inputs. We first present a bidirectional type checking algorithm which establishes reducibility. We then show that the algorithm is sound, namely if the type checking succeeds, then the relevant expression, type or program is reducible.

Our bidirectional type checking procedure is defined through the following five mutually recursive judgements where P_1 is a program, P_2 is a set of mutually recursive definitions, Θ is a set of type variables, Γ is a typing context, e is an expression and τ is a type. The program P_1 in the environment corresponds to a reducible program whereas the set P_2 consists of definitions currently being checked for reducibility. When the distinction between the program P_1 and the set of mutually recursive definitions P_2 is irrelevant, we write P to signify $P_1; P_2$ (or $P_1 \cup P_2$ depending on the context).

Context formation $P_1; P_2; \Theta \vdash \Gamma\ \textit{context}$: This judgement ensures that the context Γ is reducible with respect to $P_1 \cup P_2$ and Θ. The context formation rules are given in Figure 3.6. Note that the context formation rules are similar to those presented in Chapter 1, with the addition of Γ in the type formation judgement environment.

Type formation $P_1; P_2; \Theta; \Gamma \vdash \tau\ \textit{type}$: This judgement ensures that the type τ is reducible in the typing environment $P_1 \cup P_2; \Theta; \Gamma$. The type formation rules are given in Figure 3.7. Note that this judgement implies the type formation judgement given in the previous chapters for program $P_1 \cup P_2$ and type variables Θ.

Type inference $P_1; P_2; \Theta; \Gamma \vdash e \Uparrow \tau$ **and type checking** $P_1; P_2; \Theta; \Gamma \vdash e \Downarrow \tau$: Both the type inference and type checking judgements ensure the expression e is reducible at type τ given $P_1 \cup P_2$, Θ and Γ. We will present the type inference and checking rules below.

Validity $P_1; P_2; \Theta; \Gamma \vdash e\ \textit{holds}$: This judgement ensures that the expression e will evaluate to **true** for all reducible inputs. We call the judgements of this form that arise during bidirectional type checking *verification conditions*. We rely here on the model finding procedure described above and the judgement is given through the following rule.

$$\textsc{Holds}\quad \frac{\begin{array}{c} P; \Theta \vdash \Gamma\ \textit{context} \qquad P; \Theta; \Gamma \vdash e \Downarrow \mathsf{Boolean} \\ \Gamma = m : \tau_m, \Gamma' \qquad m \notin FV(\textit{erase}(e)) \qquad \exists\, i \in \mathbb{N}. \nexists M.\ M \models \Phi_i \end{array}}{P; \Theta; \Gamma \vdash e\ \textit{holds}}$$

$$\textsc{Empty Context}\quad P;\Theta \vdash \textit{context} \qquad\qquad \textsc{Increase Context}\quad \frac{P;\Theta;\Gamma \vdash \tau\ \textit{type}}{P;\Theta \vdash \Gamma, x:\tau\ \textit{context}}$$

Figure 3.6 – Context formation rules.

$$\textsc{Boolean Type}\quad \frac{P;\Theta \vdash \Gamma\ \textit{context}}{P;\Theta;\Gamma \vdash \mathsf{Boolean}\ \textit{type}} \qquad \textsc{Unit Type}\quad \frac{P;\Theta \vdash \Gamma\ \textit{context}}{P;\Theta;\Gamma \vdash \mathsf{Unit}\ \textit{type}} \qquad \textsc{Type Variable}\quad \frac{P;\Theta \vdash \Gamma\ \textit{context} \qquad T \in \Theta}{P;\Theta;\Gamma \vdash T\ \textit{type}}$$

$$\textsc{Datatype}\quad \frac{\begin{array}{c}(\textbf{type}\ d[\overline{\tau}_d](m)\ :=\ \cdots) \in P \\ |\overline{\tau}_d| = |\overline{\tau}| \qquad P;\Theta \vdash \Gamma\ \textit{context} \qquad P;\Theta;\Gamma \vdash \tau\ \textit{type}\ \text{for}\ \tau \in \overline{\tau} \qquad P;\Theta;\Gamma \vdash e_m \Downarrow \mathsf{Nat}\end{array}}{P;\Theta;\Gamma \vdash d[\overline{\tau}](e_m)\ \textit{type}}$$

$$\textsc{Refinement Type}\quad \frac{P;\Theta;\Gamma \vdash \tau\ \textit{type} \qquad P;\Theta;\Gamma, x:\tau \vdash p \Downarrow \mathsf{Boolean}}{P;\Theta;\Gamma \vdash \{\, x:\tau \mid p \,\}\ \textit{type}}$$

$$\textsc{Pi-Type}\quad \frac{P;\Theta;\Gamma \vdash \tau_1\ \textit{type} \qquad P;\Theta;\Gamma, x:\tau_1 \vdash \tau_2\ \textit{type}}{P;\Theta;\Gamma \vdash \Pi x:\tau_1.\ \tau_2\ \textit{type}} \qquad \textsc{Sigma-Type}\quad \frac{P;\Theta;\Gamma \vdash \tau_1\ \textit{type} \qquad P;\Theta;\Gamma, x:\tau_1 \vdash \tau_2\ \textit{type}}{P;\Theta;\Gamma \vdash \Sigma x:\tau_1.\ \tau_2\ \textit{type}}$$

Figure 3.7 – Type formation rules. The DATATYPE rule further allows a variant for the recursive type intersection $d[\overline{\tau}]$ where the type check on e_m is dropped.

$$\begin{array}{rcl}
\textbf{if}\ (c)\ \tau\ \textbf{else}\ \tau & = & \tau \\
\textbf{if}\ (c)\ d[\overline{\tau}_1](m_1)\ \textbf{else}\ d[\overline{\tau}_2](m_2) & = & d[\textbf{if}\ (c)\ \overline{\tau}_1\ \textbf{else}\ \overline{\tau}_2](\textbf{if}\ (c)\ m_1\ \textbf{else}\ m_2) \\
\textbf{if}\ (c)\ \Pi x:\tau_{1,1}.\ \tau_{1,2}\ \textbf{else}\ \Pi x:\tau_{2,1}.\ \tau_{2,2} & = & \Pi x:\textbf{if}\ (c)\ \tau_{1,1}\ \textbf{else}\ \tau_{2,1}.\ \textbf{if}\ (c)\ \tau_{2,1}\ \textbf{else}\ \tau_{2,2} \\
\textbf{if}\ (c)\ \Sigma x:\tau_{1,1}.\ \tau_{1,2}\ \textbf{else}\ \Sigma x:\tau_{2,1}.\ \tau_{2,2} & = & \Sigma x:\textbf{if}\ (c)\ \tau_{1,1}\ \textbf{else}\ \tau_{2,1}.\ \textbf{if}\ (c)\ \tau_{2,1}\ \textbf{else}\ \tau_{2,2} \\
\textbf{if}\ (c)\ \{\, x:\tau_1 \mid p_1 \,\}\ \textbf{else}\ \{\, x:\tau_2 \mid p_2 \,\} & = & \{\, x:\textbf{if}\ (c)\ \tau_1\ \textbf{else}\ \tau_2 \mid \textbf{if}\ (c)\ p_1\ \textbf{else}\ p_2 \,\} \\
\textbf{if}\ (c)\ \{\, x:\tau_1 \mid p_1 \,\}\ \textbf{else}\ \tau_2 & = & \{\, x:\textbf{if}\ (c)\ \tau_1\ \textbf{else}\ \tau_2 \mid \textbf{if}\ (c)\ p_1\ \textbf{else}\ \textbf{true} \,\} \\
\textbf{if}\ (c)\ \tau_1\ \textbf{else}\ \{\, x:\tau_2 \mid p_2 \,\} & = & \{\, x:\textbf{if}\ (c)\ \tau_1\ \textbf{else}\ \tau_2 \mid \textbf{if}\ (c)\ \textbf{true}\ \textbf{else}\ p_2 \,\}
\end{array}$$

Figure 3.8 – Recursive definition of the type computed from **if** (c) τ_1 **else** τ_2. The shorthand **if** (c) $\overline{\tau}_1$ **else** $\overline{\tau}_2$ is defined as expected by calling the type computation on each type in the sequence. Note that the refinement type, datatype and pi/sigma-type rules assume some identifier normalization which is ommitted for readability.

3.5.1 Type Inference

We now present the type inference rules that define $P_1; P_2; \Theta; \Gamma \vdash e \Uparrow \tau$. Type inference relies on the shape of the given expression e to determine which inference rule should be applied. The rule selection is a deterministic process and can be efficiently implemented as a pattern matching on expression trees.

It is important to note that our language does not feature dependent if or match types which would seem necessary to infer a precise type for the corresponding expressions. Furthermore, our structural equality contradicts extensional equality and we therefore cannot substitute free variables in types by expressions (only values). This seems to further imply the necessity of a let type in our language. However, we can avoid introducing these extra types by introducing *type simplifications* of the shape $\textbf{let}\ x\ :=\ e\ \textbf{in}\ \tau$, $\textbf{if}\ (c)\ \tau_1\ \textbf{else}\ \tau_2$ and $e\ \textbf{match}\ \{\ C_1(y_1)\ \Rightarrow\ \tau_1\ \cdots\ C_n(y_n)\ \Rightarrow\ \tau_n\ \}$. These simplifications *push* the relevant expression down into the sub-expressions of each type (namely refinement type predicates or datatype size expressions). We have seen that our definition of reducibility imposes a certain structure on reducible expressions through the simple typing judgement. Hence, we are guaranteed that when multiple types must be unified, they share a same structure. The simplification associated to the notation $\textbf{if}\ (c)\ \tau_1\ \textbf{else}\ \tau_2$ is given in Figure 3.8. Simplifications for let-bindings and match-expressions are similarly defined.

Lemma 13. *For typing environment $P; \Theta; \Gamma$ and reducible inputs $(P_{in}, \theta, \gamma) \in [\![P; \Theta; \Gamma]\!]_v$,*

- *for expression e and type τ,*

$$(\gamma(\theta(e)) \to^* v \land v \in \mathit{value}) \implies [\![\gamma(\theta(\tau[x/v]))]\!]_v = [\![\gamma(\theta(\textbf{let}\ x\ :=\ e\ \textbf{in}\ \tau))]\!]_v$$

- *for expression c and types τ_1, τ_2 such that $\mathit{erase}(\tau_1) = \mathit{erase}(\tau_2)$,*

$$\begin{aligned}\gamma(\theta(c)) \to^* \textbf{true} \implies [\![\gamma(\theta(\tau_1))]\!]_v = [\![\gamma(\theta(\textbf{if}\ (c)\ \tau_1\ \textbf{else}\ \tau_2))]\!]_v\ \land \\ \gamma(\theta(c)) \to^* \textbf{false} \implies [\![\gamma(\theta(\tau_2))]\!]_v = [\![\gamma(\theta(\textbf{if}\ (c)\ \tau_1\ \textbf{else}\ \tau_2))]\!]_v\end{aligned}$$

- *for expression e and types $\tau_1, \cdots, \tau_n$ such that $\mathit{erase}(\tau_i) = \mathit{erase}(\tau_j)$ for $1 \le i, j \le n$,*

$$\begin{aligned}\forall\, 1 \le i \le n.\ \big(\gamma(\theta(e)) \to^* C_i[\overline{\tau}](v_1) \land v_1 \in \mathit{value}\big) \implies \\ [\![\gamma(\theta(\tau_i[y_i/v_1]))]\!]_v = [\![\gamma(\theta(e\ \textbf{match}\ \{\ C_1(y_1)\ \Rightarrow\ \tau_1\ \cdots\ C_n(y_n)\ \Rightarrow\ \tau_n\ \}))]\!]_v\end{aligned}$$

Proof. The proof follows by induction on the set of given types. □

Function calls. We start by presenting the inference rules for function calls. The relevant rule is selected based on whether the associated function definition belongs to P_1 (*i.e.* it is known to be reducible) or to P_2 (*i.e.* it is currently being checked). The following CALL P_1 rule corresponds to a call to some function in the *reducible* program P_1 where the size expression was provided.

$$\textsc{Call } P_1 \quad \frac{\begin{array}{c}(\textbf{def } f(m:\tau_m)[\overline{\tau}_f](x:\tau_1):\tau_2 := e_f) \in P_1 \\ |\overline{\tau}_f| = |\overline{\tau}| \qquad P_1;P_2;\Theta;\Gamma \vdash e_m \Downarrow \tau_m \qquad P_1;P_2;\Theta;\Gamma, m:\tau_m, m \approx e_m \vdash e_1 \Downarrow \tau_1[\overline{\tau}_f/\overline{\tau}]\end{array}}{P_1;P_2;\Theta;\Gamma \vdash f(e_m)[\overline{\tau}](e_1) \Uparrow \textbf{let } m := e_m \textbf{ in let } x := e_1 \textbf{ in } \tau_2[\overline{\tau}_f/\overline{\tau}]}$$

It is clear that the CALL P_1 rule does not rely on the size expression e_m to establish reducibility of the call. However, as τ_1 and τ_2 may depend on the size binding m, we cannot simply forget about the size expression in either the parameter type checks or inferred result type. However, if the generalization procedures discussed previously can be applied to the parameter and result types, then we can type check a call without providing a size expression. This observation leads to the following rule.

$$\textsc{Call Generalization} \quad \frac{\begin{array}{c}(\textbf{def } f(m:\tau_m)[\overline{\tau}_f](x:\tau_1):\tau_2 := e_f) \in P_1 \\ |\overline{\tau}_f| = |\overline{\tau}| \qquad \vdash (\tau_1, m) \text{ gen}_{\supseteq} \tau_1' \qquad \vdash (\tau_2, m) \text{ gen}_{\subseteq} \tau_2' \qquad P_1;P_2;\Theta;\Gamma \vdash e_1 \Downarrow \tau_1'[\overline{\tau}_f/\overline{\tau}]\end{array}}{P_1;P_2;\Theta;\Gamma \vdash f[\overline{\tau}](e_1) \Uparrow \textbf{let } x := e_1 \textbf{ in } \tau_2'[\overline{\tau}_f/\overline{\tau}]}$$

Finally, the CALL P_2 rule given below corresponds to a call to some function currently being checked for reducibility. As we will see shortly, in order to ensure program reducibility, we type check the body of each function in P_2 under a context including the function's size binding ($m' : \tau_m$ in the rule given below). In order to ensure well-founded induction and co-induction, this rule checks that the provided size expression decreases in the (mutually) recursive calls.

$$\textsc{Call } P_2 \quad \frac{\begin{array}{c}(\textbf{def } f(m:\tau_m)[\overline{\tau}_f](x:\tau_1):\tau_2 := e_f) \in P_2 \qquad |\overline{\tau}_f| = |\overline{\tau}| \\ P_1;P_2;\Theta;m':\tau_m,\Gamma \vdash e_m \Downarrow \tau_m \qquad P_1;P_2;\Theta;m':\tau_m,\Gamma \vdash e_m < m' \textit{ holds} \\ P_1;P_2;\Theta;m':\tau_m,\Gamma, m:\tau_m, m \approx e_m \vdash e_1 \Downarrow \tau_1[\overline{\tau}_f/\overline{\tau}]\end{array}}{P_1;P_2;\Theta;m':\tau_m,\Gamma \vdash f(e_m)[\overline{\tau}](e_1) \Uparrow \textbf{let } m := e_m \textbf{ in let } x := e_1 \textbf{ in } \tau_2[\overline{\tau}_f/\overline{\tau}]}$$

Recursive datatypes. We now present the type inference rules which are relevant to recursive datatypes, namely the rules for match and constructor expressions. In order to destruct a datatype in a match expression, we must show that its fields are in some denotation, *i.e.* the datatype must be in the denotation of $d[\overline{\tau}](e_m)$ for some $e_m >$ Zero. The rule selection depends on the shape of the inferred type of the scrutinee. If the inferred type was a sized datatype, we can directly check that the size expression was indeed greater than Zero, leading to the following rule.

MATCH

$$\frac{\begin{array}{c} P;\Theta;\Gamma \vdash e \Uparrow d[\overline{\tau}](e_m) \\ P;\Theta;\Gamma \vdash e_m > \mathsf{Zero}\ \mathit{holds} \qquad (\textbf{type}\ d[\overline{\tau}_d](m) := C_1(x_1 : \tau_1) \mid \cdots \mid C_n(x_n : \tau_n)) \in P \\ P;\Theta;\Gamma, y_i : \textbf{let}\ m := e_m\ \textbf{in}\ \tau_i[\overline{\tau}_d/\overline{\tau}], e \approx C_i[\overline{\tau}](y_i) \vdash e_i \Uparrow \tau_{r,i}\ \text{ for }\ 1 \le i \le n \\ \mathit{erase}(\tau_{r,i}) = \mathit{erase}(\tau_{r,j})\ \text{ for }\ 1 \le i, j \le n \end{array}}{\begin{array}{l} P;\Theta;\Gamma \vdash e\ \textbf{match}\ \{\ C_1(y_1) \Rightarrow e_1 \cdots C_n(y_n) \Rightarrow e_n\ \} \Uparrow \\ \qquad\qquad e\ \textbf{match}\ \{\ C_1(y_1) \Rightarrow \tau_{r,1} \cdots C_n(y_n) \Rightarrow \tau_{r,n}\ \} \end{array}}$$

We also want to allow datatype deconstruction when a datatype intersection is inferred for the match scrutinee. Similarly to the reducibility relation unfolding of intersection datatypes, this is possible when the datatype has strictly positive polarity, giving us the following rule.

MATCH INTERSECTION

$$\frac{\begin{array}{c} P;\Theta;\Gamma \vdash e \Uparrow d[\overline{\tau}] \\ (\textbf{type}\ d[\overline{\tau}_d](m) := C_1(x_1 : \tau_1) \mid \cdots \mid C_n(x_n : \tau_n)) \in P \qquad d \text{ strictly positive} \\ P;\Theta;\Gamma, y_i : \mathsf{intersect}_d(\tau_i)[\overline{\tau}_d/\overline{\tau}], e \approx C_i[\overline{\tau}](y_i) \vdash e_i \Uparrow \tau_{r,i}\ \text{ for }\ 1 \le i \le n \\ \mathit{erase}(\tau_{r,i}) = \mathit{erase}(\tau_{r,j})\ \text{ for }\ 1 \le i, j \le n \end{array}}{\begin{array}{l} P;\Theta;\Gamma \vdash e\ \textbf{match}\ \{\ C_1(y_1) \Rightarrow e_1 \cdots C_n(y_n) \Rightarrow e_n\ \} \Uparrow \\ \qquad\qquad e\ \textbf{match}\ \{\ C_1(y_1) \Rightarrow \tau_{r,1} \cdots C_n(y_n) \Rightarrow \tau_{r,n}\ \} \end{array}}$$

Let us now consider datatype constructor expressions. When inferring the type of a constructor with annotated size expression e_m, it suffices to show that the constructor arguments have size $e_m - 1$. However, as e_m can be Zero, and $e_m - 1$ is therefore not defined, we case split the inference rule into a case where $e_m \approx \mathsf{Zero}$ and one where $e_m > \mathsf{Zero}$. In the first case, it then suffices to show that the constructor arguments will reduce to values according to the denotation. We leverage our special type $\mathsf{val}[\tau]$ here as $P;\Theta;\Gamma \vdash e \Downarrow \mathsf{val}[\tau]$ implies that e will evaluate to some value for all inputs in $[\![P;\Theta;\Gamma]\!]_v$. Note that only the erasure of the type τ in $\mathsf{val}[\tau]$ needs to be well-formed since the denotation relies on the simple typing judgement which operates on erased types. Hence, the size binding m does not need to be in the typing context for the case where $e_m \approx \mathsf{Zero}$.

CONSTRUCTOR

$$\frac{\begin{array}{c} (\textbf{type}\ d[\overline{\tau}_d](m) := \cdots \mid C_i(x_i : \tau_i) \mid \cdots) \in P \qquad |\overline{\tau}_d| = |\overline{\tau}| \\ P;\Theta;\Gamma \vdash \tau\ \mathit{type}\ \text{ for }\ \tau \in \overline{\tau} \qquad P;\Theta;\Gamma \vdash e_m \Downarrow \mathsf{Nat} \qquad P;\Theta;\Gamma, e_m \approx \mathsf{Zero} \vdash e_1 \Downarrow \mathsf{val}[\tau_i[\overline{\tau}_d/\overline{\tau}]] \\ P;\Theta;\Gamma, e_m > \mathsf{Zero}, m : \mathsf{Nat}, m \approx e_m \vdash e_1 \Downarrow \tau_i[\overline{\tau}_d/\overline{\tau}] \end{array}}{P;\Theta;\Gamma \vdash C_i(e_m)[\overline{\tau}](e_1) \Uparrow d[\overline{\tau}](e_m)}$$

Conversely to match expressions, constructor expressions can omit the size expression when the datatype is strictly positive and an intersection datatype will be inferred.

CONSTRUCTOR INTERSECTION

$$\frac{\begin{array}{c} (\textbf{type}\ d[\overline{\tau}_d](m) := \cdots \mid C_i(x_i : \tau_i) \mid \cdots) \in P \qquad d \text{ strictly positive} \\ |\overline{\tau}_d| = |\overline{\tau}| \qquad P;\Theta;\Gamma \vdash \tau\ \mathit{type}\ \text{ for }\ \tau \in \overline{\tau} \qquad P;\Theta;\Gamma \vdash e_1 \Downarrow \mathsf{intersect}_d(\tau_i)[\overline{\tau}_d/\overline{\tau}] \end{array}}{P;\Theta;\Gamma \vdash C_i[\overline{\tau}](e_1) \Uparrow d[\overline{\tau}]}$$

One should note here that similarly to the reducibility relation unfolding, one could also define the MATCH INTERSECTION and CONSTRUCTOR INTERSECTION rules in terms of the type generalization procedures. However, as previously mentioned, we have found that strict positivity was a better condition in practice.

Remaining rules. The remaining type inference rules are given in Figure 3.9. The DROP REFINEMENT rule is applied with low priority and ensures that the expected shape of types can be obtained in rules such as MATCH, PROJECTION and APPLICATION. Note that we only apply the rule (and thus widen the refinement type) under these specific conditions.

3.5.2 Type Checking

We present here the type checking rules that give the relation $P_1; P_2; \Theta; \Gamma \vdash e \Downarrow \tau$. Type checking relies both on the shape of the given expression e and the shape of the expected type τ for rule selection. The selection procedure is again deterministic and can be efficiently implemented.

Pushing the check down. A first set of type checking rules rely on the expression syntax to produce sub-expression type checks. These rules allow the syntax to help the type checking procedure by producing simpler and more controlled verification conditions for if-expressions and let-bindings. Note that the applicable rule can be selected solely based on the given expression shape.

CHECK LET
$$\frac{P;\Theta;\Gamma \vdash e_1 \Uparrow \tau_1 \qquad P;\Theta;\Gamma, x : \tau_1, x \approx e_1 \vdash e_2 \Downarrow \tau}{P;\Theta;\Gamma \vdash \textbf{let}\ x\ :=\ e_1\ \textbf{in}\ e_2 \Downarrow \tau}$$

CHECK IF
$$\frac{P;\Theta;\Gamma \vdash c \Downarrow \mathsf{Boolean} \qquad P;\Theta;\Gamma, c \vdash e_1 \Downarrow \tau \qquad P;\Theta;\Gamma, \neg c \vdash e_2 \Downarrow \tau}{P;\Theta;\Gamma \vdash \textbf{if}\ (c)\ e_1\ \textbf{else}\ e_2 \Downarrow \tau}$$

We also want to push checks down in match-expressions. However, as seen in the inference case, the type checking condition depends on the type of the scrutinee. We therefore rely on two match type checking rules, depending on what type the scrutinee is inferred to be.

CHECK MATCH
$$\frac{\begin{array}{c} P;\Theta;\Gamma \vdash e \Uparrow d[\overline{\tau}](e_m) \\ P;\Theta;\Gamma \vdash e_m > \mathsf{Zero}\ \textit{holds} \qquad (\textbf{type}\ d[\overline{\tau}_d](m)\ :=\ C_1(x_1 : \tau_1)\ \mid \cdots \mid\ C_n(x_n : \tau_n)) \in P \\ P;\Theta;\Gamma, \textbf{let}\ m\ :=\ e_m\ \textbf{in}\ \tau_i[\overline{\tau}_d/\overline{\tau}], e \approx C_i[\overline{\tau}](y_i) \vdash e_i \Downarrow \tau\ \text{ for }\ 1 \leq i \leq n \end{array}}{P;\Theta;\Gamma \vdash e\ \textbf{match}\ \{\ C_1(y_1)\ \Rightarrow\ e_1 \cdots C_n(y_n)\ \Rightarrow\ e_n\ \} \Downarrow \tau}$$

CHECK MATCH INTERSECTION
$$\frac{\begin{array}{c} P;\Theta;\Gamma \vdash e \Uparrow d[\overline{\tau}] \qquad (\textbf{type}\ d[\overline{\tau}_d](m)\ :=\ C_1(x_1 : \tau_1)\ \mid \cdots \mid\ C_n(x_n : \tau_n)) \in P \\ d\ \text{strictly positive} \qquad P;\Theta;\Gamma, y_i : \mathsf{intersect}_d(\tau_i)[\overline{\tau}_d/\overline{\tau}], e \approx C_i[\overline{\tau}](y_i) \vdash e_i \Downarrow \tau\ \text{ for }\ 1 \leq i \leq n \end{array}}{P;\Theta;\Gamma \vdash e\ \textbf{match}\ \{\ C_1(y_1)\ \Rightarrow\ e_1 \cdots C_n(y_n)\ \Rightarrow\ e_n\ \} \Downarrow \tau}$$

$$\textsc{True}\ \frac{P;\Theta \vdash \Gamma\ \mathit{context}}{P;\Theta;\Gamma \vdash \mathbf{true} \Uparrow \mathsf{Boolean}} \qquad \textsc{False}\ \frac{P;\Theta \vdash \Gamma\ \mathit{context}}{P;\Theta;\Gamma \vdash \mathbf{false} \Uparrow \mathsf{Boolean}} \qquad \textsc{Unit}\ \frac{P;\Theta \vdash \Gamma\ \mathit{context}}{P;\Theta;\Gamma \vdash () \Uparrow \mathsf{Unit}}$$

$$\textsc{Var}\ \frac{P;\Theta \vdash \Gamma\ \mathit{context} \qquad (x,\tau) \in \Gamma}{P;\Theta;\Gamma \vdash x \Uparrow \tau} \qquad \textsc{Let}\ \frac{P;\Theta;\Gamma \vdash e_1 \Uparrow \tau_1 \qquad P;\Theta;\Gamma, x : \tau_1, x \approx e_1 \vdash e_2 \Uparrow \tau_2}{P;\Theta;\Gamma \vdash \mathbf{let}\ x\ :=\ e_1\ \mathbf{in}\ e_2 \Uparrow \mathbf{let}\ x\ :=\ e_1\ \mathbf{in}\ \tau_2}$$

$$\textsc{If}\ \frac{P;\Theta;\Gamma \vdash c \Downarrow \mathsf{Boolean} \qquad P;\Theta;\Gamma, c \vdash e_1 \Uparrow \tau_1 \qquad P;\Theta;\Gamma, \neg c \vdash e_2 \Uparrow \tau_2 \qquad \mathit{erase}(\tau_1) = \mathit{erase}(\tau_2)}{P;\Theta;\Gamma \vdash \mathbf{if}\ (c)\ e_1\ \mathbf{else}\ e_2 \Uparrow \mathbf{if}\ (c)\ \tau_1\ \mathbf{else}\ \tau_2}$$

$$\textsc{Equals}\ \frac{P;\Theta;\Gamma \vdash e_1 \Uparrow \tau_1 \qquad P;\Theta;\Gamma \vdash e_2 \Uparrow \tau_2 \qquad \mathit{erase}(\tau_1) = \mathit{erase}(\tau_2)}{P;\Theta;\Gamma \vdash e_1 \approx e_2 \Uparrow \mathsf{Boolean}}$$

$$\textsc{Err}\ \frac{P;\Theta;\Gamma \vdash \tau\ \mathit{type} \qquad P;\Theta;\Gamma \vdash \mathbf{false}\ \mathit{holds}}{P;\Theta;\Gamma \vdash \mathsf{err}[\tau] \Uparrow \tau} \qquad \textsc{Pair}\ \frac{P;\Theta;\Gamma \vdash e_1 \Uparrow \tau_1 \qquad P;\Theta;\Gamma \vdash e_2 \Uparrow \tau_2}{P;\Theta;\Gamma \vdash (e_1, e_2) \Uparrow \Sigma x : \tau_1.\ \tau_2}$$

$$\textsc{Projection 1}\ \frac{P;\Theta;\Gamma \vdash e \Uparrow \Sigma x : \tau_1.\ \tau_2}{P;\Theta;\Gamma \vdash \pi_1(e) \Uparrow \tau_1} \qquad \textsc{Projection 2}\ \frac{P;\Theta;\Gamma \vdash e \Uparrow \Sigma x : \tau_1.\ \tau_2}{P;\Theta;\Gamma \vdash \pi_2(e) \Uparrow \mathbf{let}\ x\ :=\ \pi_1(e)\ \mathbf{in}\ \tau_2}$$

$$\textsc{Lambda}\ \frac{P;\Theta;\Gamma, x : \tau_1 \vdash e \Uparrow \tau_2}{P;\Theta;\Gamma \vdash \lambda x : \tau_1.\ e \Uparrow \Pi x : \tau_1.\ \tau_2} \qquad \textsc{Application}\ \frac{P;\Theta;\Gamma \vdash e_1 \Uparrow \Pi x : \tau_2.\ \tau \qquad P;\Theta;\Gamma \vdash e_2 \Downarrow \tau_2}{P;\Theta;\Gamma \vdash e_1\ e_2 \Uparrow \mathbf{let}\ x\ :=\ e_2\ \mathbf{in}\ \tau}$$

$$\textsc{Drop Refinement}\ \frac{P;\Theta;\Gamma \vdash e \Uparrow \{x : \tau \mid p\}}{P;\Theta;\Gamma \vdash e \Uparrow \tau}$$

Figure 3.9 – Remaining type inference rules.

Checking the denotation. These type checking rules are selected based on the shape of the expected type and are directly derived from the denotation of the type. The type checking rules for boolean types, refinement types, pi- and sigma-types are straightforward and defined as follows. Note that the DROP REFINEMENT inference rule defined above may be leveraged when inferring the expected Boolean type in the CHECK BOOLEAN rule.

CHECK BOOLEAN
$$\frac{P;\Theta;\Gamma \vdash e \Uparrow \mathsf{Boolean}}{P;\Theta;\Gamma \vdash e \Downarrow \mathsf{Boolean}}$$

CHECK UNIT
$$\frac{P;\Theta;\Gamma \vdash e \Uparrow \mathsf{Unit}}{P;\Theta;\Gamma \vdash e \Downarrow \mathsf{Unit}}$$

CHECK REFINEMENT
$$\frac{P;\Theta;\Gamma \vdash e \Downarrow \tau \qquad P;\Theta;\Gamma, x:\tau, x \approx e \vdash p \textit{ holds}}{P;\Theta;\Gamma \vdash e \Downarrow \{x:\tau \mid p\}}$$

CHECK PI
$$\frac{P;\Theta;\Gamma, x:\tau_1 \vdash e\ x \Downarrow \tau_2}{P;\Theta;\Gamma \vdash e \Downarrow \Pi x:\tau_1.\,\tau_2}$$

CHECK SIGMA
$$\frac{P;\Theta;\Gamma \vdash \pi_1(e) \Downarrow \tau_1 \qquad P;\Theta;\Gamma, x:\tau_1, x \approx \pi_1(e) \vdash \pi_2(e) \Downarrow \tau_2}{P;\Theta;\Gamma \vdash e \Downarrow \Sigma x:\tau_1.\,\tau_2}$$

Datatype subtyping rules. Given our definitions of valid type parameter and datatype polarity assignments, we can derive a subtyping relation between datatypes based on their type parameter instantiations and size expressions. We rely on the shape of the inferred and expected types to determine which datatype subtyping rule to apply. Note that the CHECK DATATYPE 2 and 3 both reduce to some instance of the CHECK DATATYPE rule. We again rely on the DROP REFINEMENT rule in CHECK DATATYPE and CHeck DATATYPE 2.

CHECK DATATYPE
$$\frac{\begin{gathered}(\textbf{type}\ \ d[T_1,\cdots,T_n](m)\ :=\ \cdots) \in P\\ P;\Theta;\Gamma \vdash \tau_i\ \textit{type}\ \text{ for } 1 \le i \le n \qquad P;\Theta;\Gamma \vdash e_m \Downarrow \mathsf{Nat} \qquad P;\Theta;\Gamma \vdash e \Uparrow d[\tau'_1,\cdots,\tau'_n](e'_m)\\ + \le pol_T(T_i) \implies P;\Theta;\Gamma, x:\tau'_i \vdash x \Downarrow \tau_i \;\wedge\; - \le pol_T(T_i) \implies P;\Theta;\Gamma, x:\tau_i \vdash x \Downarrow \tau'_i \text{ for } 1 \le i \le n\\ d \text{ positive} \implies P;\Theta;\Gamma \vdash e_m \le e'_m \textit{ holds} \qquad \neg(d \text{ positive}) \implies P;\Theta;\Gamma \vdash e_m \approx e'_m \textit{ holds}\end{gathered}}{P;\Theta;\Gamma \vdash e \Downarrow d[\tau_1,\cdots,\tau_n](e_m)}$$

CHECK DATATYPE 2
$$\frac{\begin{gathered}P;\Theta;\Gamma \vdash e_m \Downarrow \mathsf{Nat}\\ P;\Theta;\Gamma \vdash e \Uparrow d[\overline{\tau}'] \qquad x \text{ fresh variable} \qquad P;\Theta;\Gamma, x: d[\overline{\tau}'](e_m) \vdash x \Downarrow d[\overline{\tau}](e_m)\end{gathered}}{P;\Theta;\Gamma \vdash e \Downarrow d[\overline{\tau}](e_m)}$$

CHECK DATATYPE 3
$$\frac{n \text{ fresh variable} \qquad P;\Theta;\Gamma, n:\mathsf{Nat} \vdash e \Downarrow d[\overline{\tau}](n)}{P;\Theta;\Gamma \vdash e \Downarrow d[\overline{\tau}]}$$

Value-type rules. Finally, we present a pair of rules for checking against the $\mathsf{val}[\tau]$ types that are introduced by the CONSTRUCTOR inference rule. To be of type $\mathsf{val}[\tau]$, it suffices to reduce to some value which satisfies the erased typing judgement. This observation leads to the following typing rules.

$$\textsc{Check Value}\quad \frac{v \in \mathit{pvalue} \qquad P;\Theta;\Gamma \vdash v : \tau}{P;\Theta;\Gamma \vdash v \Downarrow \mathsf{val}[\tau]} \qquad\qquad \textsc{Check Typed}\quad \frac{P;\Theta;\Gamma \vdash e \Uparrow \tau_1 \qquad \mathit{erase}(\tau_1) = \mathit{erase}(\tau)}{P;\Theta;\Gamma \vdash e \Downarrow \mathsf{val}[\tau]}$$

Note that the conditions under which the CHECK VALUE rule holds are efficiently decidable and the rule therefore has higher priority than the CHECK TYPED rule in the algorithm.

3.5.3 Soundness

The aim of the bidirectional type checking algorithm presented above is to establish reducibility. For each of the five judgements we introduced, if the existence of a derivation implies that the corresponding reducibility relation holds, then the procedure is *sound*. These observations lead to the following statement of soundness for the different judgements.

Lemma 14. *For reducible program P_1, set of mutually recursive definitions P_2, well-order τ_m, values $v_n, v'_n \in [\![\tau_m]\!]_v$, type variables Θ, typing context Γ, expression e and type τ, if all datatype definitions in P_2 are reducible below v_n, all function definitions in P_2 are reducible below v'_n, Γ is of the form $\Gamma = m : \tau_m, \Gamma'$ and both $\Gamma'[m/v'_n]$ and $\tau[m/v'_n]$ are bounded by v_n, then*

- *if $P_1;P_2;\Theta \vdash \Gamma$ context, then $P_1 \cup P_2;\Theta \models \Gamma'[m/v'_n]$ context,*
- *if $P_1;P_2;\Theta;\Gamma \vdash \tau$ type, then $P_1 \cup P_2;\Theta;\Gamma'[m/v'_n] \models \tau[m/v'_n]$ type,*
- *if $P_1;P_2;\Theta;\Gamma \vdash e \Uparrow / \Downarrow \tau$ then $P_1 \cup P_2;\Theta;\Gamma'[m/v'_n] \models e[m/v'_n] \in [\![\tau[m/v'_n]]\!]$, and*
- *if $P_1;P_2;\Theta;\Gamma \vdash e$ holds, then for reducible inputs $(P_{in},\gamma,\theta) \in [\![P_1 \cup P_2;\Theta;\Gamma'[m/v'_n]]\!]_v$, we have $\gamma(\theta(e)) \to^* \mathbf{true}$.*

Proof. We show this by induction on the judgement derivation. In the following, when we consider value substitutions for a typing context with unnamed evidence bindings, we assume there exists a mapping to the unit literal for each such binding in the substitution.

Rules EMPTY CONTEXT and INCREASE CONTEXT : These follow immediately by induction.

Rules BOOLEAN TYPE, UNIT TYPE, TYPE VARIABLE, DATATYPE, REFINEMENT TYPE, PI-TYPE and SIGMA-TYPE : The type formation rules also follow immediately by induction.

Rules CALL P_1 and P_2 : Consider the size binding $m' : \tau_{m'}$ associated to the call. By induction, for inputs $(P_{in},\theta,\gamma) \in [\![P_1;P_2;\overline{\tau}_f;\Gamma'[m/v'_n]]\!]_v$, we have $\gamma(\theta(e_m)) \to^* v_m \in [\![\tau_{m'}]\!]_v$ and $\gamma(\theta(e_1)) \to^* v_1 \in [\![\gamma(\theta(\tau_1[m'/v_m][\overline{\tau}_f/\overline{\tau}]))]\!]_v$. Let $\theta' = \{\overline{\tau}_f \mapsto \gamma(\theta(\overline{\tau}))\}$ and $\gamma' = \{x \mapsto v_1\}$. If $f \in P_1$, then reducibility of P_1 tells us that $\gamma'(\theta'(e_f[m'/v_m])) \in [\![\gamma'(\theta'(\tau_2[m'/v_m]))]\!]_e$. If $f \in P_2$, we have $v_m < v'_n$ by induction and reducibility below v'_n of definitions in P_2 tells us the same. As evaluation of e_f does not depend on the size binding v_m, we have $\gamma'(\theta'(e_f)) \in [\![\gamma'(\theta'(\tau[m'/v_m]))]\!]_e$. As $\gamma(\theta(e_m))$ and $\gamma(\theta(e_1))$ terminate to values and θ distributes over types and expressions, we have the following equivalence by Lemma 13

$$[\![\gamma'(\theta'(\tau_2[m'/v_m]))]\!]_e = [\![\gamma(\theta(\textbf{let } m' := e_m \textbf{ in let } x := e_1 \textbf{ in } \tau_2[\overline{\tau}_f/\overline{\tau}]))]\!]_e$$

The operational semantics finally tell us that $\gamma(\theta(f(e_m)[\overline{\tau}](e_1))) \to^* \gamma'(\theta'(e_f))$ and as $[\![\cdot]\!]_e$ is closed under evaluation by definition, this concludes the case.

Rule Call Generalization : By induction, for inputs $(P_{in}, \theta, \gamma) \in [\![P; \overline{\tau}_f; \Gamma'[m/v'_n]]\!]_v$, we have $\gamma(\theta(e_1)) \to^* v_1$ where $v_1 \in [\![\gamma(\theta(\tau'_1[\overline{\tau}_f/\overline{\tau}]))]\!]_v$. By Lemma 8, there exists some $n_1 \in [\![\tau_m]\!]_v$ such that for $n_2 \in [\![\tau_m]\!]_v$ where $n_1 \leq n_2$, we have $v_1 \in [\![\gamma(\theta(\tau_1[m/n_2][\overline{\tau}_f/\overline{\tau}]))]\!]_v$. As $f \in P_1$, for $n_2 \in [\![\tau_m]\!]_v$ where $n_1 \leq n_2$, we have $\gamma(\theta(f[\overline{\tau}](e_1))) \in [\![\gamma(\theta(\tau_2[m/n_2, x/v_1][\overline{\tau}_f/\overline{\tau}]))]\!]_e$ by reducibility of P_1. We therefore have $\gamma(\theta(f[\overline{\tau}](e_1))) \in [\![\gamma(\theta(\tau'_2[x/v_1][\overline{\tau}_f/\overline{\tau}]))]\!]_e$, again by Lemma 8. We can then conclude by applying a similar argument to the previous case.

Rule Match : For inputs $(P_{in}, \theta, \gamma) \in [\![P; \Theta; \Gamma'[m/v'_n]]\!]_v$, we have $\gamma(\theta(e)) \in [\![\gamma(\theta(d[\overline{\tau}](e_m)))]\!]_e$ and $\gamma(\theta(e_m)) \to^* v_m$ where $v_m \in [\![\mathsf{Nat}]\!]_v$ and $v_m \neq \mathsf{Zero}$ by induction. By definition of the denotation, we have $\gamma(\theta(e)) \to^* C_i[\overline{\tau}'](v_1)$ where $v_1 \in [\![\gamma(\theta(\tau_i[m/v_m][\overline{\tau}_d/\overline{\tau}]))]\!]_v$. Let us now consider the value substitution $\gamma' = \gamma \cup \{y_i \mapsto v_1\}$. By induction, we have $\gamma'(\theta(e_i)) \in [\![\gamma'(\theta(\tau_i))]\!]_e$ and we can apply Lemma 13 to conclude the case.

Rule Match Intersection : For reducible inputs $(P_{in}, \theta, \gamma) \in [\![P; \Theta; \Gamma'[m/v'_n]]\!]_v$, we have $\gamma(\theta(e)) \in [\![\gamma(\theta(d[\overline{\tau}]))]\!]_e$ by induction. By definition of reducibility for open expressions, we therefore have $\gamma(\theta(e)) \to^* C_i[\overline{\tau}'](v_1)$ where $C_i[\overline{\tau}'](v_1) \in [\![\gamma(\theta(d[\overline{\tau}]))]\!]_v$. By definition of strict positivity for datatypes, we have $v_1 \in [\![\gamma(\theta(\mathsf{intersect}_d(\tau_i)[\overline{\tau}_d/\overline{\tau}]))]\!]_v$ and we can conclude this case by applying a similar argument to the previous one.

Rule Constructor : For inputs $(P_{in}, \theta, \gamma) \in [\![P; \Theta; \Gamma'[m/v'_n]]\!]_v$, we have $\gamma(\theta(e_m)) \in [\![\mathsf{Nat}]\!]_e$ by induction. We first consider the case where $\gamma(\theta(e_m)) \to^* \mathsf{Zero}$. We need to show that $\gamma(\theta(C_i(e_m)[\overline{\tau}](e_1))) \in [\![\gamma(\theta(d[\overline{\tau}](\mathsf{Zero})))]\!]_e$, and by definition of $[\![\cdot]\!]_e$, this requires that $\gamma(\theta(e_1)) \to^* v_1$ where $v_1 \in \mathit{value}$ and $v_1 : \gamma(\theta(\tau_i[\overline{\tau}_d/\overline{\tau}]))$. This is given by induction and definition of $[\![\gamma(\theta(\mathsf{val}[\tau_i[\overline{\tau}_d/\overline{\tau}]]))]\!]_v$. The case where $\gamma(\theta(e_m)) \to^* \mathsf{Succ}(v'_m)$ is then simply given by induction.

Rule Constructor Intersection : For inputs $(P_{in}, \theta, \gamma) \in [\![P; \Theta; \Gamma'[m/v'_n]]\!]_v$, by induction we have $\gamma(\theta(e_1)) \to^* v_1$ where $v_1 \in [\![\gamma(\theta(\mathsf{intersect}_d(\tau_i)[\overline{\tau}_d/\overline{\tau}]))]\!]_v$. By strict positivity and distributivity of substitutions, we have $C_i[\mathit{erase}(\gamma(\theta(\overline{\tau})))](v_1) \in [\![\gamma(\theta(d[\overline{\tau}]))]\!]_v$. Finally, by the operational semantics, we have $\gamma(\theta(C_i[\overline{\tau}](e_1))) \to^* C_i[\mathit{erase}(\gamma(\theta(\overline{\tau})))](v_1)$ which concludes the case.

Rules True and False : The rules follow by definition of $[\![\mathsf{Boolean}]\!]_v$.

Rule Unit : The rule follow by definition of $[\![\mathsf{Unit}]\!]_v$.

Rules Var, Equals, Pair, Projection 1, Lambda and Drop Refinement : These rules follow by application of the inductive hypothesis.

Rules IF, LET, PROJECTION 2 and APPLICATION : The rules follow by induction and Lemma 13.

Rule ERR : For inputs $(P_{in}, \theta, \gamma) \in [\![P; \Theta; \Gamma'[m/v'_n]]\!]_v$, we have $\gamma(\theta(\mathbf{false})) \to^* \mathbf{true}$ by induction. Hence, by contradiction, no reducible inputs exist and the case holds.

Rules CHECK LET and CHECK IF : The rules follow by induction.

Rules CHECK MATCH and CHECK MATCH INTERSECTION : The rules follow by similar arguments to those presented for rules MATCH and MATCH INTERSECTION.

Rules CHECK BOOLEAN, CHECK UNIT, CHECK REFINEMENT, CHECK SIGMA and CHECK PI : The rules follow immediately by induction.

Rule CHECK DATATYPE : For inputs $(P_{in}, \theta, \gamma) \in [\![P; \Theta; \Gamma'[m/v'_n]]\!]_v$, the inductive hypothesis tells us that $\gamma(\theta(e)) \in [\![\gamma(\theta(d[\tau'_1, \cdots, \tau'_n](e'_m)))]\!]_e$. By definition of the denotation, we therefore have $\gamma(\theta(e'_m)) \to^* v'_m \in [\![\mathsf{Nat}]\!]_v$. Let us consider some type parameter index $1 \le i \le n$. If $+ \preceq pol_T(T_i)$, then for $v \in [\![\tau'_i]\!]_v$, we have $v \in [\![\tau_i]\!]_v$ by induction. In other words, we have $[\![\tau'_i]\!]_v \subseteq [\![\tau_i]\!]_v$. Similarly, if $- \preceq pol_T(T_i)$, then we have $[\![\tau'_i]\!]_v \supseteq [\![\tau_i]\!]_v$. By definition of validity for the type parameter polarity assignment, we therefore have $[\![\gamma(\theta(d[\tau'_1, \cdots, \tau'_n](v_m)))]\!]_v \subseteq [\![\gamma(\theta(d[\tau_1, \cdots, \tau_n](v_m)))]\!]_v$.

By induction, we have $\gamma(\theta(e_m)) \to^* v_m \in [\![\mathsf{Nat}]\!]_v$, if d is positive, then again by induction we have $v_m \le v'_m$. By definition of the datatype positivity property, we thus have $[\![\gamma(\theta(d[\tau'_1, \cdots, \tau'_n](v'_m)))]\!]_v \subseteq [\![\gamma(\theta(d[\tau'_1, \cdots, \tau'_n](v_m)))]\!]_v$. If d is not positive, then we know that $v_m = v'_m$ and thus $[\![\gamma(\theta(d[\tau'_1, \cdots, \tau'_n](v'_m)))]\!]_v = [\![\gamma(\theta(d[\tau'_1, \cdots, \tau'_n](v_m)))]\!]_v$. Therefore, we have $\gamma(\theta(e)) \in [\![\gamma(\theta(d[\tau_1, \cdots, \tau_n](e_m)))]\!]_v$.

Rule CHECK DATATYPE 2 : For reducible inputs $(P_{in}, \theta, \gamma) \in [\![P; \Theta; \Gamma'[m/v'_n]]\!]_v$, the inductive hypothesis gives us $\gamma(\theta(e)) \to^* v \in [\![\gamma(\theta(d[\overline{\tau}]))]\!]_e$ and $\gamma(\theta(e_m)) \to^* v_m \in [\![\mathsf{Nat}]\!]_v$. By definition of the denotation, we have $[\![\gamma(\theta(d[\overline{\tau}]))]\!]_v \subseteq [\![\gamma(\theta(d[\overline{\tau}](v_m)))]\!]_v = [\![\gamma(\theta(d[\overline{\tau}](e_m)))]\!]_v$, hence $v \in [\![\gamma(\theta(d[\overline{\tau}](e_m)))]\!]_v$. Finally, we have $v \in [\![\gamma(\theta(d[\overline{\tau}'](e_m)))]\!]_v$ by induction and freshness of the variable x.

Rule CHECK DATATYPE 3 : For inputs $(P_{in}, \theta, \gamma) \in [\![P; \Theta; \Gamma'[m/v'_n]]\!]_v$ and for $v_m \in [\![\mathsf{Nat}]\!]_v$, we have $\gamma(\theta(e)) \in [\![(\gamma \cup \{n \mapsto v_m\})(\theta(d[\overline{\tau}](n)))]\!]_e$ by induction and freshness of n. Hence, by definition of the denotation, we have $\gamma(\theta(e)) \in [\![\gamma(\theta(d[\overline{\tau}]))]\!]_e$.

Rules CHECK VALUE and CHECK TYPED : Follow from the definition of the denotation.

Rule HOLDS : By induction, we have $P; \Theta \models \Gamma'[m/v'_n]$ *context*, and for reducible inputs $(P_{in}, \theta, \gamma) \in [\![P; \Theta; \Gamma'[m/v'_n]]\!]_v$, we have $\gamma(\theta(e)) \to^* v$ for some $v \in [\![\mathsf{Boolean}]\!]_v$. By Theorem 3, if there exists no model $M \models \Phi_i$, then we know that $v \neq \mathbf{false}$. By definition of $[\![\mathsf{Boolean}]\!]_v$, we therefore have $v = \mathbf{true}$. $\square$

In the previous soundness statement, we assumed the given program P_1 was already known to be reducible. It therefore remains to consider how to show program reducibility. As mentioned earlier, reducibility for programs is established by considering strongly connected sets of type and function definitions in the order given by the $\prec$ relation on definitions. We therefore consider a reducible program P_1 and show reducibility of some set of mutually recursive definitions P_2. These considerations lead to the following program formation judgement.

$$\textsc{Empty Program}\quad \vdash \emptyset\ \textit{program} \qquad\qquad \textsc{Increase Program}\quad \frac{\vdash P_1\ \textit{program} \qquad \forall\, d \in P_2.\ P_1; P_2 \vdash d\ \textit{well-formed}}{\vdash P_1 \cup P_2\ \textit{program}}$$

$$\textsc{Type}\quad \frac{\forall\, 1 \le i \le n.\ P_1; P_2; \overline{\tau}_d; m : \mathsf{Nat} \vdash \tau_i\ \textit{type} \qquad \textit{erase}(P_1 \cup P_2; \overline{\tau}_d) \vdash d[\overline{\tau}_d]\ \textit{well-defined}}{P_1; P_2 \vdash \textbf{type}\ \ d[\overline{\tau}_d](m)\ :=\ C_1(x_1 : \tau_1)\ \mid \cdots \mid\ C_n(x_n : \tau_n)\ \textit{well-formed}}$$

$$\textsc{Function}\quad \frac{\begin{array}{c} P_1; P_2; \overline{\tau}_f; m : \tau_m \vdash \tau_1\ \textit{type} \\ P_1; P_2; \overline{\tau}_f; m : \tau_m, x : \tau_1 \vdash \tau_2\ \textit{type} \qquad P_1; P_2; \overline{\tau}_f; m : \tau_m, x : \tau_1 \vdash e_f \Downarrow \tau_2 \end{array}}{P_1; P_2 \vdash \textbf{def}\ \ f(m : \tau_m)[\overline{\tau}_f](x : \tau_1) : \tau_2\ :=\ e_f\ \textit{well-formed}}$$

Given the program formation judgement, we can now state the main property of our bidirectional type checking algorithm, namely soundness with respect to the denotation.

Theorem 4. *For program P if $\vdash P$ program, then $\models P$ program.*

Proof. We show this by induction on the derivation.

Rule Empty Program : The statement trivially holds.

Rule Increase Program : By induction, we have $\models P_1$ *program*. Consider the well-order τ_m associated to P_2. We show by strong induction on $[\![\tau_m]\!]_v$ that for $n \in [\![\tau_m]\!]_v$, the datatype definitions in P_2 are reducible below $\mathsf{Succ}(n)$ and the function definitions in P_2 are reducible below n. Note that this property then implies reducibility of $P_1 \cup P_2$.

First, we show that all type definitions in P_2 are reducible below $\mathsf{Succ}(n)$ (Type rule) by invoking Lemma 14 with $v'_n = n$ and $v_n = n$. Note that the syntactic restriction on datatype definitions ensures that for each constructor field type τ_i, we have $\tau_i[m/v'_n]$ bounded by v_n. Next, we show that all function definitions in P_2 are reducible below n (Function rule) by invoking Lemma 14 with $v'_n = n$ and $v_n = \mathsf{Succ}(n)$. Again, the syntactic restriction on function definitions which are mutually recursive with datatype definitions ensures that both $\tau_1[m/v'_n]$ and $\tau_2[m/v'_n]$ are bounded by v_n. □

Similarly to Theorem 3 about the model finding procedure, when the set P_2 is empty, we can state the following cleaner variant of Lemma 14.

Corollary 4. *For reducible program P, set of type variables Θ, reducible typing context Γ, expression e and type τ, we have*

- *if* $P;\Theta \vdash \Gamma$ *context, then* $P;\Theta \models \Gamma$ *context,*
- *if* $P;\Theta;\Gamma \vdash \tau$ *type, then* $P;\Theta;\Gamma \models \tau$ *type,*
- *if* $P;\Theta;\Gamma \vdash e \Uparrow / \Downarrow \tau$ *then* $P;\Theta;\Gamma \models e \in [\![\tau]\!]$*, and*
- *if* $P;\Theta;\Gamma \vdash e$ *holds, then for* $(P_{in}, \gamma, \theta) \in [\![P;\Theta;\Gamma]\!]_v$*, we have* $\gamma(\theta(e)) \to^* \mathbf{true}$.

3.6 Typing Derivations and the Unfolding Procedure

We have presented a sound unfolding procedure that leverages the dependent types that appear within the original typing context. However, those are generally not the only reducibility relations that are known when the unfolding procedure is invoked in the HOLDS rule. Indeed, the type checking algorithm will derive reducibility relations for most sub-expressions of the expression e given to the unfolding procedure. Furthermore, reducibility relations are also derived for expressions in function bodies when the program reducibility is checked.

Consider a reducible program P_1, set of mutually recursive function definition P_2, set of type variables Θ, context Γ, expression e and value v such that the expression $e \approx v$ has been type checked to Boolean. Further consider some embedded sub-expression $l : e' \sqsubseteq e$ such that the type checking included derivations $P_1;P_2;\Theta;\Gamma,\Gamma' \vdash \tau'$ *type* and $P_1;P_2;\Theta;\Gamma,\Gamma' \vdash e' \Uparrow / \Downarrow \tau'$. We have seen that such derivations imply the corresponding reducibility relations. Let us now consider a set of reducible inputs $(P_{in}, \theta, \gamma) \in [\![P_1 \cup P_2;\Theta;\Gamma]\!]_v$ such that $\gamma(\theta(e)) \to^* \mathcal{E}[l : e'_2]$. Finally, consider the value substitution $\gamma' \supseteq \gamma$ that contains all substitutions that occur during the evaluation $\gamma(\theta(e)) \to^* \mathcal{E}[l : e'_2]$. We therefore have $\gamma(\theta(e)) \to^* \mathcal{E}[l : \gamma'(\theta(e'))]$.

One could expect given our type checking procedure that we have $\gamma'(\theta(e')) \in [\![\gamma'(\theta(\tau'))]\!]_e$. However, this is not the case. Let us consider the following instantiation of program P_1, set of definitions P_2, type variables Θ, context Γ and expression e.

$$P_1 = \{\mathbf{def}\ \mathsf{f}(\mathsf{x} : \{ y : \mathsf{Nat} \mid y \approx \mathsf{Zero} \}) : \{ y : \mathsf{Nat} \mid y \approx \mathsf{Zero} \} := \mathsf{x}\} \qquad P_2 = \emptyset \qquad \Theta = \emptyset$$

$$\Gamma = \mathsf{g} : \mathsf{Nat} \to \mathsf{Nat}, \mathsf{eq} : \{ u : \mathsf{Unit} \mid \mathsf{g} \approx (\lambda \mathsf{n} : \{ y : \mathsf{Nat} \mid y \approx \mathsf{Zero} \}.\ \mathsf{f}(\mathsf{n})) \}$$

$$e = \mathsf{g}(\mathsf{Succ}(\mathsf{Zero})) \approx \mathsf{Zero}$$

We have $\models P_1 \cup P_2$ *program*, $P_1 \cup P_2;\Theta \models \Gamma$ *context* and $P_1 \cup P_2;\Theta;\Gamma \models e \in [\![\mathsf{Boolean}]\!]$. We further have $P_1 \cup P_2;\Theta;\mathsf{g} : \mathsf{Nat} \to \mathsf{Nat}, \mathsf{n} : \{ y : \mathsf{Nat} \mid y \approx \mathsf{Zero} \} \vdash \mathsf{f}(\mathsf{n}) \Uparrow \{ y : \mathsf{Nat} \mid y \approx \mathsf{Zero} \}$ derived during type checking of e. Now let us consider the reducible inputs $P_{in} = \emptyset$, $\theta = \emptyset$ and $\gamma = \{\mathsf{g} \mapsto \lambda \mathsf{n} : \mathsf{Nat}.\ \mathsf{f}(\mathsf{n}), \mathsf{eq} \mapsto \mathsf{Unit}\}$. It is clear that we have $\gamma(\theta(e)) \to^* \mathsf{f}(\mathsf{Succ}(\mathsf{Zero}))$. If we consider the value substitution $\gamma' = \gamma \cup \{\mathsf{n} \mapsto \mathsf{Succ}(\mathsf{Zero})\}$ under which the sub-expression $\mathsf{f}(\mathsf{n})$ is evaluated, we have $\gamma'(\theta(\mathsf{f}(\mathsf{n}))) \notin [\![\gamma'(\theta(\{ y : \mathsf{Nat} \mid y \approx \mathsf{Zero} \}))]\!]_e$ as $\gamma'(\theta(\mathsf{f}(\mathsf{n}))) \to^* \mathsf{Succ}(\mathsf{Zero})$. We therefore cannot directly assume that e' is reducible at type τ'.

The above finding is a consequence of our erased operational semantics. Let us consider the following slightly modified program P_1.

$$P_1 \ = \ \{\textbf{def}\ \mathsf{f}\,(\mathsf{x} : \{\, y : \mathsf{Nat} \ \mid\ y \approx \mathsf{Zero}\,\}) \ : \ \mathsf{Nat} \ := \ \textbf{if}\,(x \approx \mathsf{Zero})\ \mathsf{x}\ \textbf{else}\ \mathsf{err}[\mathsf{Nat}]\,\}$$

This version of P_1 remains reducible, however the lambda $\lambda \mathsf{n} : \mathsf{Nat}.\ \mathsf{f}(\mathsf{n})$ is no longer reducible at type $\mathsf{Nat} \to \mathsf{Nat}$ as it only takes Zero to Zero. In this case, it is therefore sound to assume that the output of $\mathsf{f}(\mathsf{n})$ will be Zero regardless of the value substituted for n. We see here that although programs can have specifications that are semantically similar, their impact or not on the operational semantics can make a significant difference during unfolding.

Although it is unsound to simply assume that sub-expressions present in the derivations are reducible at their annotated types, these types can still be leveraged during unfolding. Indeed, while $\gamma'(\theta(e')) \in [\![\gamma'(\theta(\tau'))]\!]_e$ may not hold in general, by definition of reducibility we know that it holds as long as $(P_{in}, \theta, \gamma') \in [\![P_1 \cup P_2; \Theta; \Gamma, \Gamma']\!]_v$. In other words, if for each binding $x : \tau \in \Gamma, \Gamma'$ we have $\gamma'(x) \in [\![\gamma'(\theta(\tau))]\!]_v$, then e' is reducible at type τ'. It is clear by definition of input reducibility that this property holds for all bindings in Γ. By considering the type checking procedure, we can further observe that most bindings in the Γ' typing context are also guaranteed to satisfy this property. Indeed, only the bindings which stem from lambda parameters may lead to a mismatch between the value substitution γ' and the context Γ'.

We now present a procedure to soundly leverage the type formation and type inferece/checking judgements. Consider the bindings $\Gamma' = x_1 : \tau_1, \cdots, x_n : \tau_n$ and indices $0 \le l_1 < \cdots < l_m \le n$ in the context Γ' which correspond to lambda parameters. We introduce fresh boolean constants $b_1, \cdots, b_{m+1}$ which will serve a similar purpose to the constant b_1 introduced in the pi-type unfolding. We then compute the following reducibility relation embeddings

$$P_1 \cup P_2; \Theta; \Gamma, x_1 : \tau_1, \cdots, x_{l_i - 1} : \tau_{l_i - 1} \vdash (b_{l_i}, x_{l_i}, \tau_{l_i}) \ \overset{red}{\triangleright} (t_i, \Phi_i) \ \text{ for } \ 1 \le i \le m$$

$$P_1 \cup P_2; \Theta; \Gamma, \Gamma' \vdash (b_{m+1}, \llfloor e' \rrfloor_t, \tau') \ \overset{red}{\triangleright} (t_\tau, \Phi_\tau)$$

It is important to realize that when $m > 0$, the embedded constants x_{l_i} are not fresh but correspond to the constants introduced during some unfolding of the lambda containing e'. We finally introduce the following general clause set which corresponds to the reducibility relation $P_1 \cup P_2; \Theta; \Gamma, \Gamma' \models e' \in [\![\tau']\!]$.

$$\Phi_{red} = (\textstyle\bigcup_{1 \le i \le m} \Phi_i) \cup \Phi_\tau \cup \{ b_i \wedge t_i \iff b_{i+1} \ \mid\ 1 \le i \le m \} \cup \{ \llfloor e' \rrfloor_b \implies b_1, b_{m+1} \implies t_\tau \}$$

We can then extend the unfolding procedure we presented above by including unfoldings of reducibility relations from the reducibility assignment. Lemmas 9 and 6 will ensure that these steps preserve consistency.

As our type checking procedure traverses the whole program, we also have derivations associated to expressions within function bodies. Even for function definitions in P_2, by splitting the HOLDS rule into verification condition generation and deferred verification condition checking, we obtain derivations that are valid for all calls that decrease the size expression. We can leverage these derivations by unfolding them for expression embeddings that occur after function call unfolding. Note that the extra condition on the size decrease is also guaranteed by the type checking algorithm and can be safely assumed when unfolding the derivations for expressions within the function's body.

In practice, to improve scalability of the unfolding procedure, we have found it useful to unfold only derivations associated to function call expressions. Furthermore, by introducing let-expressions with (checked) type annotations, we allow the user to (somewhat) control the reducibility relations on which the unfolding procedure will depend. Given the function definition **def** $\mathsf{check}[T](x : T) : T = x$, consider the following expressions:

$$\textbf{let } x \ := \ \mathsf{check}[\tau](e_1) \textbf{ in } e_2 \qquad\qquad \textbf{let } x : \tau \ := \ e_1 \textbf{ in } e_2$$

Although both expressions are semantically equivalent, the unfolding procedure will only assume that e_1 is reducible at type τ in the first case. One can therefore syntactically control which reducibility relations will be assumed by the procedure. Note that the e_2 expression will be type checked under (practically) identical typing environments in both case.

3.7 Finding Reducible Counterexamples

We focused in this chapter on proofs and put aside the counterexample finding capabilities of our unfolding procedure. We also saw that synthesizing reducible inputs is a hard task in the general case. We discuss in this section how sound counterexamples can be reported even in the presence of dependent types.

Firstly, our unfolding procedure is *precise* in the absence of pi-types with dependent result types and intersection datatypes which cannot be generalized. Our model finding procedure can therefore be extended to produce reducible inputs within this limited fragment. Similarly to how function calls are *blocked* through the $\mathsf{block}(F_i)$ clause set, we define the clause set $\mathsf{block}(R_i)$ which imposes input reducibility as follows:

$$\mathsf{block}(R_i) = \{\neg\llbracket\tau\rrbracket_b \mid \tau \in R_i,\ \tau = d[\overline{\tau}] \vee \tau = d[\overline{\tau}](e_m)\}$$

The unfolding procedure can then extract reducible inputs from models satisfying the clause set $\Phi_i \cup \mathsf{block}(F_i) \cup \mathsf{block}(A_i, \Lambda_i, D_i) \cup \mathsf{block}(R_i)$. As the set of values has not changed, the extraction procedure presented in the previous chapter remains applicable. Note however that the final **else** branch in the synthetic functions extracted for lambdas now *must* be some default value as a recursive call would contradict reducibility.

Secondly, counterexamples that are not correct by construction can be type checked against the context. Given the typing environment $P;\Theta;\Gamma$ under which unfolding was performed and a model $M_i \models \Phi_i \cup \mathsf{block}(F_i) \cup \mathsf{block}(A_i, \Lambda_i, D_i) \cup \mathsf{block}(R_i)$, consider the extracted inputs

$(P_{in}, \theta, \gamma) \lhd M_i$. We can leverage our algorithmic type checking procedure to verify that 1) $\vdash P \cup P_{in}$ *program*, and 2) for each $(x, \tau) \in \Gamma$, we have $P \cup P_{in}; \emptyset; \emptyset \vdash \gamma(x) \Downarrow \gamma(\theta(\tau))$. Hence, we have a sound (but incomplete) procedure that can produce reducible counterexamples.

The above considerations imply a syntactically defined fragment for which counterexample finding remains sound and complete. Furthermore, even when we fall outside this fragment, we can still report sound counterexamples. We have found that these alternative procedures are relevant in practice and many verification conditions generated during (idiomatic) program verification will satisfy one of the soundness conditions. Finally, even when our type checking procedure fails to show that some extracted set of inputs are reducible, the (potentially spurious) counterexample has shown to be valuable feedback to the user.

4 Proofs and Counterexamples with Impredicative Quantifiers

Many properties of interest can be expressed (and proved) in the dependently-typed language presented in the previous chapter. However, it is sometimes practical to specify certain properties through universal or existential quantifiers. Furthermore, quantifier support can prove to be crucial when considering encodings of complex features into a language for which verification can be performed, as we will see in Chapter 5. In this chapter, we will discuss how our language can be extended with impredicative quantifiers through the use of a Prop type.

The focus of this chapter is on quantifier support in our model finding procedure. In order to give some context, we will start by outlining how our language can be extended with quantified propositions. We introduce a propositional type whose inhabitants consist of (ground) propositional formulas. Note that we employ the term *propositional* here to denote structured logical propositions (*i.e.* boolean connectives and quantified propositions) as opposed to propositional logic. We extend the denotation to include this type and present a new logical relation that corresponds to *true* propositions. We then adapt our refinement types to expect propositional refinements instead of the previously given boolean predicates and let their denotation refer to the new logical relation. Type checking in the presence of propositional refinements then simply relies on the model finding procedure.

In order to provide automation in the presence of quantified propositions, we extend our model finding procedure to embed the new logical relation associated to propositional formulas. The embedding follows a similar approach to the higher-order function embedding presented in Chapter 2: propositions are embedded into synthetic datatype constructors based on the proposition's structure, and the logical relation is embedded into special predicates which are then incrementally unfolded. We finally describe how reducible inputs can sometimes be extracted even in the presence of universal quantification.

In this chapter, we will only outline the relevant reducibility relations and give some intuition about the properties of the extended model finding procedure. A full formalization is outside the scope of this thesis and left for future work.

4.1 Language and Semantics

We extend the language from Chapter 3 with a propositional type Prop. We further introduce a set of propositional expressions (or propositional formulas) given by the *prop* grammar below. These expressions consist of universally and existentially quantified propositions, conjunction, disjunction and negation propositions, as well as a *truth* proposition which specifies that a given boolean expression evaluates to **true**. Similarly to lambdas, propositional expressions do not reduce during evaluation and ground propositional expressions therefore correspond to values. These considerations then lead to the following grammar extensions.

$$\begin{array}{rcl} \mathit{type} & ::= & \cdots \mid \mathsf{Prop} \\ \mathit{expr} & ::= & \cdots \mid \mathit{prop} \\ \mathit{value} & ::= & \cdots \mid p \in \mathit{prop} \text{ if } FV(p) = \emptyset \\ \mathit{prop} & ::= & \forall\, \mathit{id} : \mathit{type}.\ \mathit{expr} \mid \exists\, \mathit{id} : \mathit{type}.\ \mathit{expr} \\ & & \mid \mathit{expr} \wedge \mathit{expr} \mid \mathit{expr} \vee \mathit{expr} \mid \neg\, \mathit{expr} \mid \mathsf{True}(\, \mathit{expr}\,) \end{array}$$

Note that propositional expressions are constructed from the *expr* non-terminal and may therefore involve function calls or if-expressions returning Prop-typed values. The erased type formation and typing judgements are extended as expected.

An important difference between propositional expressions and lambdas is that dependent types must not be erased within the types of quantified variables during evaluation. Indeed, the propositions $\exists x : \mathsf{Nat}.\ x > \mathsf{Zero}$ and $\exists x : \{\, y : \mathsf{Nat} \mid y \approx \mathsf{Zero}\,\}.\ x > \mathsf{Zero}$ are not equivalent even though the types Nat and $\{\, y : \mathsf{Nat} \mid y \approx \mathsf{Zero}\,\}$ share the same erasure. In order to correctly evaluate Prop-typed expressions, our operational semantics would therefore have to maintain the full runtime type information and only perform specific erasures when necessary (such as during equality evaluation). In practice, however, our system will ensure that Prop-typed expressions only appear in erased positions (such as type refinements) and can therefore never occur during evaluation. Hence, only the logical semantics of propositions are relevant.

Let us now consider the denotation of the Prop type. In order to ensure that propositional values contain well-formed types, we would like to define the denotation by induction on the propositional values and rely on the reducibility relation for types. However, as the Prop type is impredicative, such a definition is not necessarily well-formed. We must therefore rely on a notion of *reducibility candidates* as described in [Gir90] in order to handle propositions that quantify over the Prop type. The technical details of such a definition are outside the scope of this thesis and we simply illustrate the denotation of Prop through the rules given in Figure 4.1. The reducibility relation for types is then extended with the rule $P;\Theta;\Gamma \models \mathsf{Prop}\ \mathit{type}$.

Let us now discuss the semantics of propositional truth. We want to define a logical relation $[\![\cdot]\!]_p$ such that given a propositionally-typed value v, we have $[\![v]\!]_p$ iff the proposition v is *true*. The truth of a propositional value corresponds to the mathematical property specified by the

$$\textsc{Forall}\quad \frac{\models \tau\ \textit{type} \qquad \forall\, v \in [\![\tau]\!]_v.\ e[x/v] \in [\![\mathsf{Prop}]\!]_e}{(\forall x : \tau.\ e) \in [\![\mathsf{Prop}]\!]_v} \qquad \textsc{Exists}\quad \frac{\models \tau\ \textit{type} \qquad \forall\, v \in [\![\tau]\!]_v.\ e[x/v] \in [\![\mathsf{Prop}]\!]_e}{(\exists x : \tau.\ e) \in [\![\mathsf{Prop}]\!]_v}$$

$$\textsc{Conjunction}\quad \frac{e_1 \in [\![\mathsf{Prop}]\!]_e \qquad e_2 \in [\![\mathsf{Prop}]\!]_e}{e_1 \wedge e_2 \in [\![\mathsf{Prop}]\!]_v} \qquad \textsc{Disjunction}\quad \frac{e_1 \in [\![\mathsf{Prop}]\!]_e \qquad e_2 \in [\![\mathsf{Prop}]\!]_e}{e_1 \vee e_2 \in [\![\mathsf{Prop}]\!]_v} \qquad \textsc{Negation}\quad \frac{e \in [\![\mathsf{Prop}]\!]_e}{\neg\, e \in [\![\mathsf{Prop}]\!]_v}$$

$$\textsc{Truth}\quad \frac{e \in [\![\mathsf{Boolean}]\!]_e}{\mathsf{True}(e) \in [\![\mathsf{Prop}]\!]_v}$$

Figure 4.1 – Denotation of the Prop type given with respect to some program P. As the denotation is defined for ground expressions without type variables, we rely on type reducibility judgements of the shape $P; \emptyset; \emptyset \models \tau$ *type*. For readability, we denote these by $\models \tau$ *type*. These rules are given for illustrative purposes as a well-formed definition would require the introduction of reducibility candidates.

proposition. For example, a conjunction proposition $e_1 \wedge e_2$ is *true* iff both sub-expressions e_1 and e_2 correspond to *true* propositions. The special leaf proposition $\mathsf{True}(e)$ then allows the logical propositions to interact with the operational semantics by providing an evaluation constraint on the boolean expression e, namely that $e \to^* \textbf{true}$.

A natural (yet naive) approach to assigning truth semantics would be to define a logical relation on propositional values which relies on evaluation to produce values for sub-propositions. However, such a definition would not be well-formed as the size of an expression may increase during evaluation. It is therefore necessary to rely on a more complex logical relation $[\![\cdot]\!]_p$ for ground expressions which is well-founded on the lexicographic ordering of 1) the multiset ordering of size bindings present in the expression, and 2) the structure of the expression. The full well-formed definition of this relation is outside the scope of this thesis but should follow similar principles to those exposed in [Wer94, p. 71] (but for truth instead of proofs).

In order to nonetheless provide some intuition into the propositional truth relation, we present the following (incomplete) definition of $[\![\cdot]\!]_p$ for propositional expressions. It is important to realize that these rules are only illustrative and do not feature all the technical details on which a well-formed definition would depend. Note that we assume that the propositional expression is in the denotation of Prop in the following.

$$\begin{array}{ll}
[\![\forall x : \tau.\ e]\!]_p \iff \forall\, v \in [\![\tau]\!]_v.\ [\![e[x/v]]\!]_p & \qquad [\![e_1 \wedge e_2]\!]_p \iff [\![e_1]\!]_p \wedge [\![e_2]\!]_p \\
[\![\exists x : \tau.\ e]\!]_p \iff \exists\, v \in [\![\tau]\!]_v.\ [\![e[x/v]]\!]_p & \qquad [\![e_1 \vee e_2]\!]_p \iff [\![e_1]\!]_p \vee [\![e_2]\!]_p \\
[\![\mathsf{True}(e)]\!]_p \iff e \to^* \textbf{true} & \qquad [\![\neg\, e]\!]_p \iff \neg [\![e]\!]_p
\end{array}$$

Finally, we adapt our denotation of refinement types to take propositional refinements (instead of booleans). This allows our system to refer to propositional truth through refinements. These considerations lead to a refinement type denotation of $[\![\{x:\tau \mid p\}]\!]_v = \{v \in [\![\tau]\!]_v \mid [\![p[x/v]]\!]_p\}$ and the REFINEMENT TYPE rule in the reducibility relation for types is given as follows.

$$\textsc{Refinement Type}\quad \frac{P;\Theta;\Gamma \models \tau\ \mathit{type} \qquad P,\Theta;\Gamma, x:\tau \models p \in [\![\mathsf{Prop}]\!]}{P;\Theta;\Gamma \models \{x:\tau \mid p\}\ \mathit{type}}$$

Note that boolean refinements can be specified through the $\mathsf{True}(\cdot)$ proposition, namely the boolean-refined type $\{x:\tau \mid p\}$ becomes $\{x:\tau \mid \mathsf{True}(p)\}$.

4.2 Embedding Propositions

In this section, we describe how the propositional expressions introduced above can be embedded into SMT terms. Similarly to the function type and lambda embeddings presented in Chapter 2, we rely on a synthetic *Prop* datatype to embed the Prop type and propositional expressions. As in the lambda case, we track the set Q of known embedded propositional expressions. We then rely on the normalization procedure for lambdas introduced in Chapter 2 to extract the structure and structural closures of propositions. Note however that, unlike lambdas, proposition will not be erased before computing their structure and it may therefore contain (un-erased) dependent types. Based on the known propositional structures, we generate a set of constructors which are used to define the *Prop* datatype and perform propositional expression embedding.

We start by presenting the *Prop* algebraic datatype generation procedure. Consider the normalized proposition structures associated to propositions in Q. Given the (unlabelled) normalized structures $\cdots, e'_i, \cdots$ of the propositions in Q, the typed free variables $\cdots, y_{i,j} : \tau_{i,j}, \cdots$ of each e'_i and the type embedding $\tau_{i,j} \rhd \sigma_{i,j}$ of each free variable type, we define the *Prop* SMT algebraic datatype as follows. Note that we (again) rely on an Else constructor to allow an unbounded number of unknown propositions.

$$\textbf{datatype}\ \ \mathit{Prop} := \cdots \mid C_{e'_i}(\cdots, y_{i,j} : \sigma_{i,j}, \cdots) \mid \cdots \mid \mathsf{Else}(n : \mathbb{N})$$

This datatype definition naturally leads to the type embedding rule $P;\Theta \vdash \mathsf{Prop} \rhd \mathit{Prop}$ and the following expression embedding rule for propositional expressions.

$$\frac{\begin{array}{c} e \in \mathit{prop} \\ \mathit{normalize}(e) = (e', \{y_1 \mapsto e_1, \cdots, y_n \mapsto e_n\}) \qquad P;\Theta;\Gamma \vdash (b, e_i) \rhd (t_i, \Phi_i)\ \text{ for }\ 1 \le i \le n \end{array}}{P;\Theta;\Gamma \vdash (b, e) \rhd (C_{e'}(t_1, \cdots, t_n), \Phi_1 \cup \cdots \cup \Phi_n)}$$

In order to embed the propositional truth relation, we rely on a special *true-prop* predicate which takes a single parameter with sort *Prop*. Note that our language only relies on the propositional truth relation in refinement types. Hence, only the reducibility relation embedding rule associated to refinement types needs to be adapted to rely on the *true-prop* predicate.

$$\frac{P;\Theta;\Gamma \vdash (b,\, t,\, \tau) \overset{red}{\triangleright} (t_\tau, \Phi_\tau) \qquad P;\Theta;\Gamma, x : \tau \vdash (b,\, p) \triangleright (t_p, \Phi_p)}{P;\Theta;\Gamma \vdash (b,\, t,\, \{x : \tau \mid p\}) \overset{red}{\triangleright} (t_\tau \wedge \mathit{true\text{-}prop}(t_p), \Phi_\tau \cup \Phi_p \cup \{b \implies x \simeq t\})}$$

Similarly to how $\mathit{reducible}_{\tau'}$ predicates are used in the reducibility relation embedding, the propositional truth relation is under-approximated by the *true-prop* predicate. We will rely on a notion of consistency of *true-prop* interpretations with the associated propositional truth relation embeddings which is given as expected based on previous consistency definitions.

4.3 Unfolding Propositions

We will now discuss how the precision of the approximation given by the propositional truth relation embedding can be incrementally increased by unfolding the relation associated to the embeddings. In this section, we will therefore extend the unfolding procedure presented in the previous chapters with support for the propositional truth relation embedding.

Our approach to truth relation unfolding features two distinct procedures: 1) a propositional expression unfolding procedure, and 2) a quantifier instantiation procedure. Propositional expression unfolding proceeds by considering propositional expression and truth relation embedding pairs similarly to the application - lambda pair unfolding procedure. The conjunction, disjunction, negation and $\mathsf{True}(\cdot)$ propositions are unfolded in a straightforward manner by relying on the truth relation embedding and expression embeddings. When unfolding a universally or existentially quantified proposition, we rely on the instantiation procedure. The quantifier instantiation procedure will track the set of quantified propositions for which the propositional expression unfolding has occurred. The procedure will then instantiate the quantified variables with likely candidates based on the quantifier polarity.

4.3.1 Unfolding Propositional Expressions

We will first present the propositional expression unfolding procedure. Recall from Chapter 3 that the unfolding procedure takes a typing environment $P;\Theta;\Gamma$, an expression e and a value v and then produces a sequence of state tuples $(\Phi_i, F_i, A_i, \Lambda_i, D_i, R_i)$. In order to unfold the truth relation, we extend the state tuple with the sets T_i of known (embedded) expressions for which the propositional truth relation was embedded and Q_i of known (embedded) propositional expressions. In other words, for $e \in T_i$, the term $\mathit{true\text{-}prop}(\llbracket e \rrbracket_t)$ will belong to the set of generated SMT terms, and for $p \in Q_i$, we will have $p \in \mathit{prop}$. We further track a set $E_i \subseteq T_i \times Q_i$ of expression - proposition pairs for which the truth relation has been unfolded.

In order to compute the sets Q_i and T_i during unfolding, we introduce the following helper functions $Q(\cdot)$ to collect propositional expressions and $T(\cdot)$ to collect truth relations.

$$Q(e) = \{ e_p \in \mathit{prop} \mid e_p \sqsubseteq e,\ \llbracket e_p \rrbracket_t \text{ defined} \} \qquad T(\tau) = \{ p \mid \{ x : \tau_p \mid p \} \sqsubseteq \tau,\ \llbracket p \rrbracket_t \text{ defined} \}$$

The $Q(\cdot)$ function is defined for both expressions and types whereas the $T(\cdot)$ function is defined exclusively for types as the truth relation only affects refinements. The initial set T_0 and the set T_{i+1} obtained by unfolding a call, application or reducibility relation are defined similarly to the sets of embedded reducibility relations R_0 and R_{i+1}. (It is however clear that the reducibility relation embeddings which are removed from R_i when computing R_{i+1} do not belong to T_i and the removal step is ignored.) The sets Q_0 and Q_{i+1} are similarly obtained by analogy with the lambda sets Λ_0 and Λ_{i+1}. Finally, we have $E_0 = \emptyset$ and $E_{i+1} = E_i$ for the call, application and reducibility relation unfolding steps.

Let us now consider the propositional truth relation unfolding step for propositional expressions. Consider some expression $e_t \in T_i$ and propositional formula $e_p \in Q_i$. Consider a fresh boolean blocker b_t which serves a similar purpose to the b_b introduced in application unfolding. We therefore introduce the clause set $\Phi_t = \{ b_t \iff (\llbracket e_p \rrbracket_b \wedge \llbracket e_t \rrbracket_b \wedge \llbracket e_p \rrbracket_t \simeq \llbracket e_t \rrbracket_t) \}$ on which the extended clause set Φ_{i+1} will rely. The unfolding procedure will extend the set E_i with the unfolded pair e_t, e_p, which gives us $E_{i+1} = E_i \cup \{(e_t,\ e_p)\}$. The set T_i will be extended with each expression e'_t for which the embedded truth relation predicate $\mathit{true\text{-}prop}(\llbracket e'_t \rrbracket_t)$ appears in $\Phi_{i+1} \setminus \Phi_i$. Finally, the remaining sets are extended by leveraging the relevant associated helper functions. We now present the clause set Φ_{i+1} which results from unfolding. The unfolding strategy will depend on the shape of the proposition e_p.

Conjunction proposition. We start by considering the case where $e_p = e_1 \wedge e_2$. Consider the embeddings $(b_t,\ e_1) \rhd (t_1, \Phi_1)$ and $(b_t,\ e_2) \rhd (t_2, \Phi_2)$. We then define Φ_{i+1} as follows.

$$\Phi_{i+1} = \Phi_i \cup \Phi_t \cup \Phi_1 \cup \Phi_2 \cup \{ b_t \implies \mathit{true\text{-}prop}(\llbracket e_t \rrbracket_t) \iff (\mathit{true\text{-}prop}(t_1) \wedge \mathit{true\text{-}prop}(t_2)) \}$$

Disjunction proposition. The unfolding when $e_p = e_1 \vee e_2$ is given similarly to the conjunction case. We again rely on the embeddings $(b_t,\ e_1) \rhd (t_1, \Phi_1)$ and $(b_t,\ e_2) \rhd (t_2, \Phi_2)$, and extend the clause set as follows.

$$\Phi_{i+1} = \Phi_i \cup \Phi_t \cup \Phi_1 \cup \Phi_2 \cup \{ b_t \implies \mathit{true\text{-}prop}(\llbracket e_t \rrbracket_t) \iff (\mathit{true\text{-}prop}(t_1) \vee \mathit{true\text{-}prop}(t_2)) \}$$

Negation proposition. Let us now consider the case where $e_p = \neg\, e_1$. As in the previous cases, the unfolding is given simply by translating the logical connective into its corresponding SMT term. Given the embedding $(b_t,\ e_1) \rhd (t_1, \Phi_1)$, we perform the following set extensions.

$$\Phi_{i+1} = \Phi_i \cup \Phi_t \cup \Phi_1 \cup \{ b_t \implies \mathit{true\text{-}prop}(\llbracket e_t \rrbracket_t) \iff \neg\, \mathit{true\text{-}prop}(t_1) \}$$

It is important to note here that this unfolding of the propositional truth relation associated to the negation proposition implies that our system admits double negation (and therefore the law of excluded middle as well). It is clear that this property is consistent with the partial definition of the propositional truth relation $[\![\cdot]\!]_p$ given above. Our system thus allows impredicative polymorphism as well as the law of excluded middle. However, as we do not support values of type Type, our system does not admit large elimination and therefore avoids the corresponding well-known paradox.

Truth proposition. Next, let us consider the unfolding when $e_p = \mathsf{True}(e_1)$. The unfolding here simply relies on the underlying boolean expression embedding. Given the embedding $(b_t, e_1) \rhd (t_1, \Phi_1)$, we therefore extend the clause set as follows.

$$\Phi_{i+1} = \Phi_i \cup \Phi_t \cup \Phi_1 \cup \{ b_t \implies \textit{true-prop}(\lfloor\!\lfloor e_t \rfloor\!\rfloor_t) \iff t_1 \}$$

Note that in this case, no new propositional truth relation embedding will appear in the resulting clause set and we have $T_{i+1} = T_i$.

Existentially quantified proposition. We now consider the case where $e_p = \exists x : \tau.\ p$. As our system admits double negation elimination, we have the following correspondance $[\![\exists x : \tau.\ p]\!]_p \iff [\![\neg\, \forall x : \tau.\ \neg\, p]\!]_p$. In order to simplify the handling of quantified propositions, we unfold existentially quantified propositions into equivalent universals. Consider the embedding $(b_t, \forall x : \tau.\ \neg\, p) \rhd (t_\forall, \Phi_\forall)$. The unfolding then proceeds as follows.

$$\Phi_{i+1} = \Phi_i \cup \Phi_t \cup \Phi_\forall \cup \{ b_t \implies \textit{true-prop}(\lfloor\!\lfloor e_t \rfloor\!\rfloor_t) \iff \neg\, \textit{true-prop}(t_\forall) \}$$

Universally quantified proposition. Finally, we consider universally quantified propositions, namely the case where $e_p = \forall x : \tau.\ p$. Unlike the unfolding procedures given above for the remaining propositional expressions, the truth relation associated to universally quantified propositions is not precisely unfolded. Instead, we rely on a quantifier instantiation procedure to incrementally increase the precision of the under-approximation given by $\textit{true-prop}(\lfloor\!\lfloor e_t \rfloor\!\rfloor_t)$. As this procedure will rely on the b_t blocker constant, we let $\Phi_{i+1} = \Phi_i \cup \Phi_t$.

4.3.2 Instantiating Quantified Propositions

In this sub-section, we will discuss how given a universally quantified proposition unfolding $(e_t, \forall x : \tau.\ p) \in E_i$ and the associated boolean constant b_t, the precision of the propositional truth relation embedding $\textit{true-prop}(\lfloor\!\lfloor e_t \rfloor\!\rfloor_t)$ can be improved by extending the Φ_i clause set through quantifier instantiation.

Although we are only considering universally quantified propositions, their *polarity* determines whether they correspond to universal or existential quantifications. Indeed, due to the correspondance between universal and existential quantifications, if $\neg\, \textit{true-prop}(\lfloor\!\lfloor \forall x : \tau.\ p \rfloor\!\rfloor_t)$

holds, then the quantification is in fact existential. Our instantiation procedure depends on whether the proposition corresponds to a universal or existential quantification.

Similarly to the propositional truth relation unfolding procedure, our quantifier instantiation procedure corresponds to a step in the global unfolding procedure. We are therefore given an unfolding state at step i and will generate the state at step $i+1$. We again focus on the generation of the clause set Φ_{i+1} and the remaining sets are obtained by applying the relevant helper functions to embedded expressions and reducibility relations.

Let us start by considering the existential case. We introduce a fresh boolean constant b_p and then compute the reducibility relation embedding $(b_p, x, \tau) \overset{red}{\triangleright} (t_\tau, \Phi_\tau)$ and expression embedding $(b_p, p) \triangleright (t_p, \Phi_p)$. The clause set is then extended as follows.

$$\Phi_{i+1} = \Phi_i \cup \Phi_\tau \cup \Phi_p \cup \{(b_t \wedge \neg \mathit{true\text{-}prop}(\llbracket e_t \rrbracket_t)) \iff b_p, b_p \implies (t_\tau \wedge t_p)\}$$

Note that in the above, the blocker constant b_p holds iff the expected quantified proposition is being unfolded (*i.e.* the blocker constant b_t introduced in the previous sub-section holds) *and* the quantification corresponds to an existential (*i.e.* $\neg \mathit{true\text{-}prop}(\llbracket e_t \rrbracket_t)$ holds).

We now consider the universal case. We rely on function calls and applications in order to find likely instantiations for the quantified variable. For each call, respectively application in p that takes a quantified argument (namely x), we consider each corresponding call in $F_i \cup U_i$, respectively application in A_i and instantiate the quantifier with the associated argument. More concretely, given $f[\overline{\tau}](x) \sqsubseteq p$ and $f[\overline{\tau}'](e_1) \in F_i \cup U_i$ where $\mathit{erase}(\overline{\tau}) = \mathit{erase}(\overline{\tau}')$, we will instantiate x with e_1. The matching will also be performed when either one or both calls feature a size expression. For function applications, given $e_1\ x \sqsubseteq p$ and $e_1'\ e_2 \in A_i$, if we have $e_1 : \tau_1$ and $e_1' : \tau_1$, then we will instantiate x with e_2.

Given an expression e with which we want to instantiate x, let us now discuss the resulting clause set. We again introduce a fresh boolean constant b_p and compute the reducibility relation embedding $(b_p, x, \tau) \overset{red}{\triangleright} (t_\tau, \Phi_\tau)$. The constant b_p will serve a similar purpose to the constant introduced in the existential case, with the additional constraint that b_p only holds if $\llbracket e \rrbracket_b$ holds as well. In this case, the reducibility relation will imply the propositional truth relation and we therefore introduce a second fresh boolean constant b_τ that holds when both b_p and the reducibility relation embedding hold. Finally, we compute the expression embedding $(b_\tau, p) \triangleright (t_p, \Phi_p)$ and define the clause set extension as follows.

$$\begin{aligned} \Phi_{i+1} = \Phi_i \cup \Phi_\tau \cup \Phi_p \cup \{\, & (b_t \wedge \mathit{true\text{-}prop}(\llbracket e_t \rrbracket_t) \wedge \llbracket e \rrbracket_b) \iff b_p, \\ & b_p \implies x \simeq \llbracket e \rrbracket_t, (b_p \wedge t_\tau) \iff b_\tau, b_\tau \implies t_p\} \end{aligned}$$

Note that the use of two blocker constants b_p and b_τ is required for similar reasons to those exposed when defining the reducibility relation unfolding associated to pi-types in Chapter 3.

It is important to note here that quantifier instantiations will introduce new calls and applications in $F_{i+1} \cup U_{i+1}$ and A_{i+1} for which instantiation can be performed. This may lead to an exponential blowup of instantiation targets. We have found that a useful heuristic which avoids (or delays) the exponential blowup is to defer instantiation of introduced calls and applications where the size of the argument is greater than the size of the given expression. For example, given a proposition $\forall x : \tau.\ \mathsf{True}(f\ x \approx g\ x) \land \mathsf{True}(f\ x \approx f\ (f\ x))$ instantiated with some application $f\ e_1 \in A_i$, further instantiations involving the introduced application $f\ (f\ e_1)$ will be deferred (whereas instantiations based on $g\ e_1$ will proceed as normal).

In order to ensure that all universally quantified propositions are instantiated at least once, we introduce an instantiation of x with a fresh constant. We have found that in practice, this instantiation is only rarely necessary for the model finding procedure to terminate and we therefore only perform this instantiation with low priority.

In our implementation, we allow quantification over multiple variables, as well as multi-argument calls and applications. Our procedure therefore follows a similar approach to the one outlined in [GdM09] generalized to our setting. We maintain sets of potential instantiations associated to each quantified variable determined by the arguments which appear in compatible concrete calls and applications. Instantiation is then performed by selecting an assignment for each quantified variable among the associated instantiation candidates.

In the above, we saw that the instantiation of a quantified proposition depends on its *polarity*. We saw that polarity of the proposition is determined by the polarity of the $\textit{true-prop}(\llbracket e_t \rrbracket_t)$ propositional truth embedding. This would seem to imply that quantifiers must be instantiated for both polarities as the property is dynamic. However, this is not the case. Indeed, the polarity of propositions can be statically tracked during the propositional expression unfolding if the initial polarity is known. The initial propositional truth embedding will always occur in a reducibility relation embedding associated to a refinement type. These either occur with positive polarity when embedding the reducibility relation associated to the typing context, or with negative polarity when they occur on the left-hand side of an implication (either during pi-type unfolding or universal quantifier instantiation). The polarity of quantified propositions is therefore statically known and only one of the existential or universal variants of instantiation need be performed.

4.4 Finding Counterexamples with Universal Quantifiers

In the previous section, we described how the under-approximation given by the propositional truth relation embedding can be refined incrementally refined through unfolding and instantiation. This allows our model finding procedure to show that no reducible inputs exist, even in the presence of impredicative quantifiers. In this section, we discuss how reducible counterexamples can also be generated by the unfolding procedure.

Let us start by considering how propositional unfoldings that have not yet occurred can be blocked. We follow a similar approach to application blocking presented in Chapter 2. Based on the sets T_i, Q_i and E_i, we define the following clause set

$$\begin{aligned}\mathsf{block}(T_i, Q_i, E_i) = \textstyle\bigcup_{e_t \in T_i} \{ \bigvee_{e_p \in Q_i} & \llbracket e_p \rrbracket_b \ \wedge \ \llbracket e_t \rrbracket_t \simeq \llbracket e_p \rrbracket_t \ \wedge \\ & \textstyle\bigwedge_{(e_t, e'_p) \in E_i} \neg(\llbracket e'_p \rrbracket_b \wedge \llbracket e_p \rrbracket_t \simeq \llbracket e'_p \rrbracket_t) \\ \implies \quad & \neg \llbracket e_t \rrbracket_b \}\end{aligned}$$

We are interested in finding models such that *all* relevant unfoldings have occurred. We therefore define the set block_i of all call, application, reducibility relation and propositional truth relation blocking clauses as follows.

$$\mathsf{block}_i = \mathsf{block}(F_i) \cup \mathsf{block}(A_i, \Lambda_i, D_i) \cup \mathsf{block}(R_i) \cup \mathsf{block}(T_i, Q_i, E_i)$$

Consider a model $M_i \models \Phi_i \cup \mathsf{block}_i$. We will assume in the following that the set R_i contains no unblocked reducibility relation for which the unfolding is imprecise (namely pi-types and non-generalizable datatypes). The unfolding of conjunction, disjunction, negation and truth propositions, as well as the instantiation of quantifiers with negative polarity (*i.e.* existentials), are all *precise*. Furthermore, the clause set $\mathsf{block}(T_i, Q_i, E_i)$ is such that all relevant proposition unfoldings have occurred. In other words, as long as the set E_i contains no unblocked universal quantifier unfolding, then the extracted inputs $(P_{in}, \theta, \gamma) \lhd M_i$ will be reducible.

The condition on unblocked universal quantifications given above is very strict. It will typically be violated as soon as a universally quantified proposition appears in the typing context Γ under which model finding is performed. For certain propositional fragments, it is however possible to extend our model finding procedure to report reducible inputs.

We define a fragment of *essentially uninterpreted propositions* which is an adaptation of the *essentially uninterpreted formulas* defined in [GdM09]. This fragment is selected such that for essentially uninterpreted universally quantified proposition $\forall x : \tau.\ p$, the propositional truth relation $\llbracket \forall x : \tau.\ p \rrbracket_p$ is consistent with the First-Order Logic semantics of the universally quantified truth relation embedding $\forall x : \sigma_\tau.\ \textit{true-prop}(\llbracket p \rrbracket_t)$ (modulo some unfoldings). A proposition is considered essentially uninterpreted iff 1) universally quantified variables appear only in function application argument positions, 2) the caller functions (which take a quantified argument) correspond to variables bound in the typing context, and 3) the reducibility relation embedding and unfolding procedures are precise for the quantified variable types. It has been shown in [GdM09] that by compactness, our universal quantifier instantiation procedure is a semi-decision procedure for essentially uninterpreted formulas (with a restriction on intended structures, which we adapt to our setting as well). These considerations imply that our quantifier instantiation procedure is a semi-decision procedure for essentially uninterpreted propositions (as long as the instantiation selection process is fair). In the following, we will focus on universal quantifier instantiations and assume existential instantiations are part of the propositional truth relation unfolding procedure.

Consider some model $M_i \models \Phi_i \cup \mathsf{block}_i$ such that all unblocked reducibility relation embeddings are precise, as well as the extracted inputs $(P_{in}, \theta, \gamma) \lhd M_i$. We know that these inputs are reducible as long as the relevant propositional truth relations are consistent with their embeddings. Let us now further consider that all universally quantified propositions for which the embedded propositional truth relation is unblocked are essentially uninterpreted. Finally, we assume that all unfolding steps after i correspond to quantifier instantiations. As the instantiation procedure is a semi-decision procedure, if we can show that for all $j > i$, there exists some $M_j \models \Phi_j$ such that $(P_{in}, \theta, \gamma) \lhd M_j$, then the inputs are reducible.

Let us now define a clause set such that satisfying models correspond to reducible inputs. First, let $EUQ_i \subseteq E_i$ the set of unfolded essentially uninterpreted universally quantified propositions with positive polarity. Then, for each $(e_t,\ \forall x : \tau.\ p) \in EUQ_i$, let $I_i(e_t,\ \forall x : \tau.\ p)$ the set of expressions with which the quantified variable x has been instantiated at unfolding step i, and $K_i(e_t,\ \forall x : \tau.\ p) = \{e_2 \mid f\ x \sqsubseteq p,\ e_1\ e_2 \in A_i,\ f : \tau_1,\ e_1 : \tau_1\}$ the set of known potential instantiations for x. Note that past instantiations are included in the set of potential instantiations, namely $I_i(e_t,\ \forall x : \tau.\ p) \subseteq K_i(e_t,\ \forall x : \tau.\ p)$. For each unfolded proposition $(e_t,\ e_p) \in E_i$, we construct a clause set $\mathsf{model}(e_t,\ e_p)$ which ensures that satisfying models are consistent with the associated propositional truth relation. Recall the blocker constant b_p that was introduced when unfolding the proposition. The $\mathsf{model}(e_t,\ e_p)$ clause set is then defined as follows.

- If e_p does not correspond to a universally quantified proposition with positive polarity, then we let $\mathsf{model}(e_t,\ e_p) = \emptyset$.
- If $e_p = \forall x : \tau.\ p$, has positive polarity and is *not* an essentially uninterpreted proposition, then we let $\mathsf{model}(e_t,\ e_p) = \{\neg b_p\}$.
- If $(e_t, e_p) \in EUQ_i$, then set define the $\mathsf{model}(e_t,\ e_p)$ clause set as follows.

  $$\mathsf{model}(e_t,\ e_p) = \{(b_p \land \llbracket e_1 \rrbracket_b) \implies \textstyle\bigvee_{e_2 \in I_i(e_t, e_p)} \llbracket e_2 \rrbracket_b \land \llbracket e_1 \rrbracket_t \simeq \llbracket e_2 \rrbracket_t \mid e_1 \in K_i(e_t,\ e_p)\}$$

  Note that we ensure that the set $I_i(e_t,\ e_p)$ is non-empty by instantiating the quantified variable with a fresh constant, as previously mentioned.

We then define the clause set union $\mathsf{model}_i = \bigcup_{(e_t, e_p) \in E_i} \mathsf{model}(e_t,\ e_p)$. Now consider a model $M_i \models \Phi_i \cup \mathsf{block}_i \cup \mathsf{model}_i$. By definition of model_i, we know that all (unblocked) truth relation embeddings associated to universally quantified propositions are such that the proposition must be essentially uninterpreted. Given a set of embedded expressions S, we define its unblocked interpretation as $M_i(S) = \{M_i(\llbracket e \rrbracket_t) \mid e \in S,\ M_i \models \llbracket e \rrbracket_b\}$. The clause set model_i then further ensures that for $(e_t,\ e_p) \in EUQ_i$, we have $M_i(K_i(e_t,\ e_p)) = M_i(I_i(e_t,\ e_p))$.

4.4.1 Soundness

We now discuss why models that satisfy the model_i clause set correspond to reducible inputs. We argue by induction on $j \geq i$ that given $M_i \models \Phi_i \cup \mathsf{block}_i \cup \mathsf{model}_i$ and extracted inputs $(P_{in}, \theta, \gamma) \lhd M_i$, there exists a model $M_j \models \Phi_j$ such that $(P_{in}, \theta, \gamma) \lhd M_j$ and for each unfolded essentially uninterpreted universally quantified proposition $(e_t, e_p) \in EUQ_i$, we have $M_j(K_j(e_t, e_p)) = M_j(I_j(e_t, e_p))$. Recall that this implies that the inputs P_{in}, θ, γ are reducible.

We start by considering the cases where unfolding is performed behind a negated blocker constant. For such unfoldings, we can extend M_j to M_{j+1} by setting all introduced blocker constants to *false* and we have $M_{j+1} \models \Phi_{j+1} \cup \mathsf{block}_{j+1} \cup \mathsf{model}_{j+1}$ with $(P_{in}, \theta, \gamma) \lhd M_{j+1}$. It is clear that call, application, (precise) reducibility relation and propositional truth relation unfoldings fall into this category by construction of block_j. This is also the case for quantifier instantiations associated to propositional truth relation unfoldings which do not belong to EUQ_j by construction of model_j. Finally, an instantiation of some $(e_t, e_p) \in EUQ_j$ by expression e also occurs behind a negated blocker constant when either $M_j \models \neg \llbracket e_t \rrbracket_b$, $M_j \models \neg \llbracket e_p \rrbracket_b$, $M_j \models \llbracket e_t \rrbracket_t \not\simeq \llbracket e_p \rrbracket_t$, $M_j \models \neg \mathit{true\text{-}prop}(\llbracket e_t \rrbracket_t)$ or $M_j \models \neg \llbracket e \rrbracket_b$.

It remains to consider the unblocked instantiation of an essentially uninterpreted universally quantified proposition $(e_t, e_p) \in EUQ_j$ where $e_p = \forall x : \tau.\ p$ with some expression e. First, recall that the reducibility relation embedding and unfolding associated to type τ must be precise, hence no quantified propositional truth relation may be introduced by instantiation and we have $EUQ_{j+1} = EUQ_j$. By definition of the quantifier instantiation procedure, we have $e \in K_j(e_t, e_p)$, and the inductive hypothesis ensures that there exists $e' \in I_j(e_t, e_p)$ such that $M_j \models \llbracket e' \rrbracket_b$ and $M_j \models \llbracket e \rrbracket_t \simeq \llbracket e' \rrbracket_t$. In other words, this instantiation is equivalent to some previous instantiation of x and for each term t introduced by the instantiation with e, there exists a corresponding t' introduced by the instantiation with e'. We can therefore extend M_j to M_{j+1} by letting $M_{j+1}(t) = M_j(t')$ for each term pair t, t'. This extension ensures that we have $M_{j+1} \models \Phi_{j+1}$. It is clear that we further have $M_{j+1}(I_{j+1}(e_t, e_p)) = M_{j+1}(I_j(e_t, e_p))$. Let us now consider the set of potential instantiations $K_{j+1}(e'_t, e'_p)$ for some $(e'_t, e'_p) \in EUQ_{j+1}$ where $e'_p = \forall y : \tau'.\ p'$. By definition of K_{j+1}, there must exist applications $g\ y \sqsubseteq p'$ and $e_1\ e_2 \sqsubseteq p$ such that $g : \tau_1$ and $e_1 : \tau_1$. If we consider the embeddings t_2, respectively t'_2 of e_2 obtained when instantiating x with e, respectively e', we have $M_{j+1}(t_2) = M_{j+1}(t'_2)$ by construction of M_{j+1}. Hence, we have $M_{j+1}(K_{j+1}(e'_t, e'_p)) = M_{j+1}(K_j(e'_t, e'_p))$ and therefore $M_{j+1}(K_{j+1}(e'_t, e'_p)) = M_{j+1}(I_{j+1}(e'_t, e'_p))$, which concludes the induction.

4.4.2 Completeness

We now discuss a (somewhat weak) completeness guarantee of our model finding procedure in the presence of universally quantified propositions. In [GdM09], the authors present a heuristic to select likely candidate instantiations based on satisfying models. We adapt the approach to our setting, improve the candidate selection process when aiming for satisfiability, and discuss completeness guarantees for a semantically defined class of inputs.

We start by introducing the notion of *finite range*. Given some lambda value $\lambda x : \tau.\ e_b$, we say $\lambda x : \tau.\ e_b$ has finite range iff the set $\{ v_b \mid v \in \llbracket \tau \rrbracket_v,\ e_b[x / v_b] \rightarrow^* v_b,\ v_b \in \mathit{value} \}$ is finite. Let us now consider reducible inputs P_{in}, θ, γ and an essentially uninterpreted universally quantified proposition $\forall x : \tau.\ p$. We say the proposition is *finitely reducible* iff for each application

$f\ x \sqsubseteq p$, the lambda $\gamma(f)$ has finite range. It is important to realize that finite reducibility allows the propositional truth relation to be checked by only considering finitely many values. Indeed, consider the set of function-typed variables $f_1, \cdots, f_n$ such that $f_i\ x \sqsubseteq p$ for $1 \leq i \leq n$. Further consider the associated (finite) ranges $r_i = \mathit{range}(\gamma(f_i))$ for $1 \leq i \leq n$. We then have the correspondance $[\![\gamma(\theta(\forall x : \tau.\ p))]\!]_p \iff \bigwedge_{v_1 \in r_1} \cdots \bigwedge_{v_n \in r_n} [\![\gamma(\theta(p[f_1\ x / v_1, \cdots, f_n\ x / v_n]))]\!]_p$.

Recall that our unfolding procedure takes as input a typing environment $P;\Theta;\Gamma$, expression e and value v. Consider reducible inputs $(P_{in}, \theta, \gamma) \in [\![P;\Theta;\Gamma]\!]_v$ such that $\gamma(\theta(e)) \rightarrow^* \mathit{erase}(\theta(v))$. Let us assume that all relevant reducibility relations are precisely handled by the model finding procedure, and all universally quantified propositions for which the truth relation is relevant are essentially uninterpreted and finitely reducible.

We now present an adaptation of our unfolding procedure that is guaranteed to find reducible inputs in this restricted setting. The adaptation simply imposes a restriction on the ordering of unfolding steps, and is given as follows. We start by performing a sequence of unfolding steps with no quantifier instantiations until some unfolding step i is reached where there exists $M_i \models \Phi_i \cup \mathsf{block}_i$. Let us assume that there exists no $M_i' \models \Phi_i \cup \mathsf{block}_i \cup \mathsf{model}_i$ (as we can simply return the extracted reducible inputs otherwise). There must therefore exist some set of blocker clauses in model_i that disallows satisfiability of the full clause set. Let us now consider an unblocked quantified proposition unfolding $(e_t, e_p) \in \mathit{EUQ}_i$ such that $M_i(K_i(e_t, e_p)) \neq M_i(I_i(e_t, e_p))$. As I_i is always a subset of K_i by construction, there must therefore exist some $e_1 \in K_i(e_t, e_p)$ such that $M_i \models \llbracket e_1 \rrbracket_b$ and $M_i(\llbracket e_1 \rrbracket_t) \not\in M_i(I_i(e_t, e_p))$. Our procedure then instantiates the proposition e_p with the expression e_1 and resumes the non-instantiation unfolding until a model $M_j \models \Phi_j \cup \mathsf{block}_j$ is found for some $j > i$, and so forth.

The intuition behind the completeness of the given procedure is that the set $M_i(I_i(e_t, e_p))$ increases with each instantiation, yet the set $S = \bigcup_{j \geq i} M_j(I_j(e_t, e_p))$ is finite. Indeed, since we have $M_i \models \mathsf{block}_i$ and M_i is consistent with the inputs, all call, application and reducibility relation unfoldings which do not stem from quantifier instantiations and occur after step i will be blocked in subsequent models. Hence, all candidate instantiations which appear after these unfoldings will not belong to S. It remains to consider the candidate instantiations that stem from instantiation unfolding steps. Consider $(e_t', \forall y : \tau'.\ p') \in \mathit{EUQ}_i$ and $e_1 \in S$ such that e_1 appeared in S after instantiating $\forall y : \tau'.\ p'$. Given the restrictions imposed on essentially uninterpreted propositions, and given $e_p = \forall x : \tau.\ p$, there must exist some $f\ x \sqsubseteq p$ and $e_1\ \mathcal{C}[g\ y] \sqsubseteq p'$ where $f : \tau_1$ and $e_1 : \tau_1$. As $\gamma(g)$ has finite range, only finitely many candidate instantiations which stem from instantiation unfoldings can belong to S.

5 Encoding Scala Programs

The verification procedure presented in Chapter 3 operates on a simple functional language with dependent types (which we hereafter call the *verification language*). Our main goal is to allow verification of programs written in a useful subset of the Scala language. Hence, many features supported in the input Scala fragment are not present in our verification language. These features are handled through encodings that eliminate the unsupported language constructs from the program. These encodings are given as a transformation pipeline that takes a Scala program and produces a program in the verification language that under-approximates the original program. Verification of the encoded program should then imply correctness of the original Scala program.

The transformation pipeline consists in a sequence of transformations defined over a hierarchy of intermediate representations rooted in the verification language. Our system relies on a variant of the verification language which allows multi-parameter and multi-field definitions, however, for clarity, we will consider (unless specified otherwise) that definitions only allow a single parameter or field. Each intermediate representation extends the one below with some set of language features. The transformation(s) associated to the intermediate representation will then encode these features into constructs of the lower intermediate representations (or fail if some feature cannot be encoded).

The hierarchy of intermediate representations (referred to as *trees* in our system) and the transformations that eliminate the constructs introduced by each tree definition are given in Figure 5.1. In the remainder of this chapter, we will discuss the tree definitions and associated encoding transformations in the pipeline. We present the trees and transformations in reverse order in order to clarify which language constructs are present in the encoding target. The target therefore always corresponds to the previously discussed transformation's input.

Certain transformations presented in this chapter were inspired by corresponding transformation phases in the Leon verification system [SKK11, BKKS13]. This is in particular the case of the method and inner function lifting transformations, as well as all transformations defined on imperative trees. Dr. Jad Hamza contributed to the development of the trait sealing

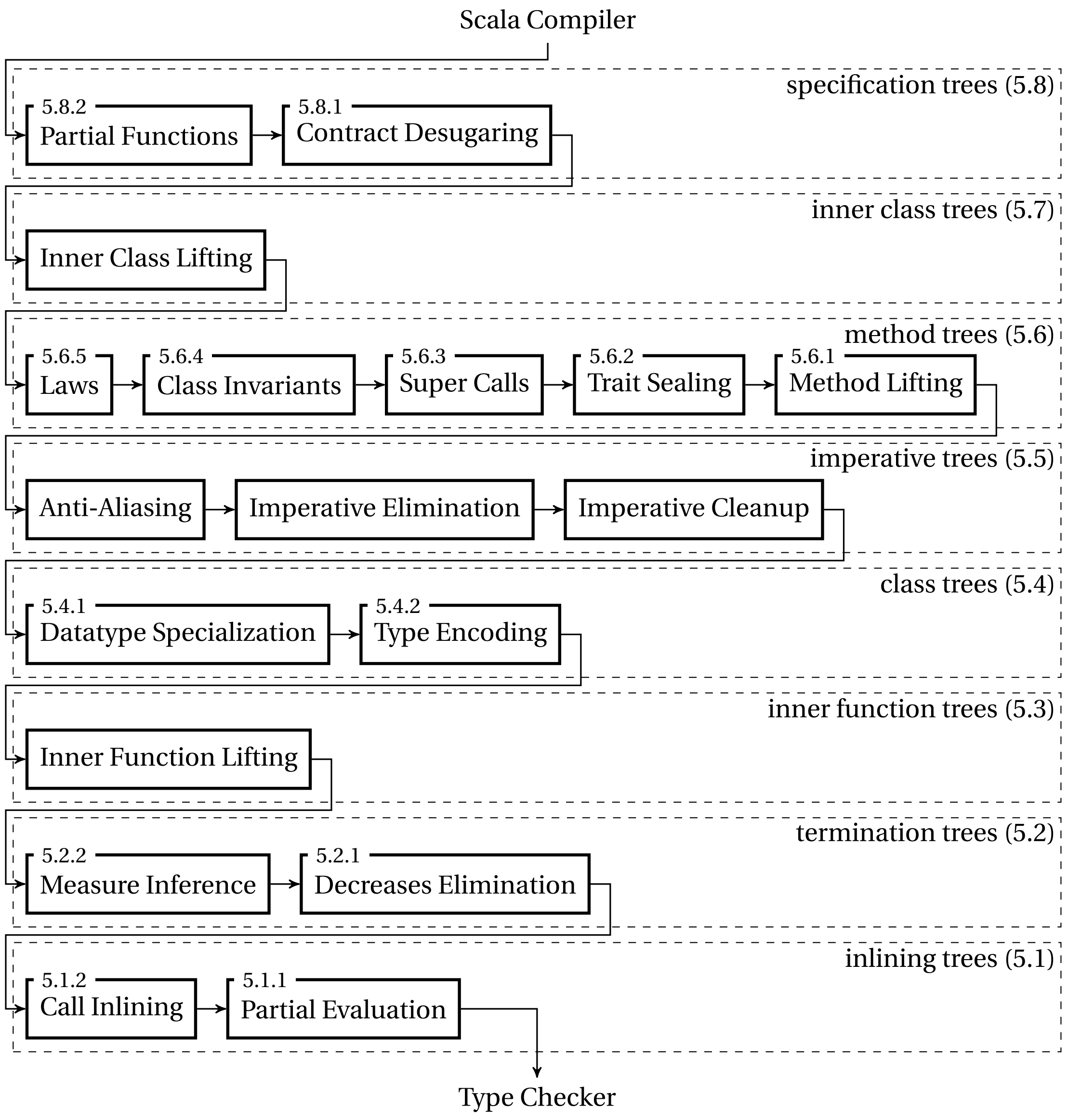

Figure 5.1 – Transformation pipeline from the Scala compiler's internal AST into the verification language presented in Chapters 1 through 4. The pipeline is based on a hierarchy of (abstract syntax) trees where each AST definition extends the one below it with new types, expressions and definitions. The encoding transformations are then grouped according to the trees on which they operate. Once all transformations have been completed for a given tree, the resulting program will belong to the tree definition below.
For each AST definition, respectively encoding transformation, the relevant section, respectively sub-section is indicated when applicable.

transformation, and Romain Ruetschi was heavily involved in (or led) the development of the inner class lifting, laws, super calls, call inlining and partial evaluation transformations.

5.1 Inlining Trees

As mentioned above, our transformation pipeline's final target is the verification language. The inlining trees extend the verification language with function annotations that allow the user to require call inlining (@inline and @inlineOnce) and partial evaluation (@partialEval). The grammar of the verification language is therefore extended as follows.

$$\begin{aligned} \mathit{fdef} &::= \langle\,\mathit{fannot}\,\rangle^* \ \mathbf{def}\ \ \mathit{id}(\,\mathit{id}:\mathit{type}\,)[\,\mathit{tdecls}\,](\,\mathit{id}:\mathit{type}\,):\mathit{type} := \ \mathit{expr} \\ \mathit{fannot} &::= \ \mathsf{@partialEval} \mid \mathsf{@inline} \mid \mathsf{@inlineOnce} \end{aligned}$$

5.1.1 Partial Evaluation

This transformation partially evaluates calls to functions marked with the @partialEval annotation. Our partial evaluator performs semantic-preserving simplifications such as dead branch pruning, function call unfolding, lambda application inlining, constant folding, etc. These simplifications allow the model finder to scale in the presence of functions with a large branching factor that lead to an exponential blowup of the clause set.

Statically unfolding recursive function calls is a potentially endless task. Furthermore, even when unfolding terminates, the resulting expression may be harder for the model finder to handle if no simplifications could be applied. Our partial evaluator therefore only unfolds a function call under the following conditions.

1. The size expression (see Chapter 3) associated with the call decreases. The partial evaluator tracks the current size binding and only unfolds recursive calls if it can show that the size expression will evaluate to some value which is smaller than the binding.

   For example, consider the size binding $m : \mathsf{Nat}$ and size expression $m-1$ (which is given as m **match** $\{\, \mathsf{Succ}(m') \Rightarrow m' \ \ \mathsf{Zero} \Rightarrow \mathsf{err}[\mathsf{Nat}]\,\}$ in the verification language). As $m-1$ will get stuck when m is Zero, the size expression does not decrease. However, if we have $m > \mathsf{Zero}$ in our context, then the size expression is decreasing.

2. *Progress* is made by unfolding the call. In order to avoid unfolding calls that simply inline the whole body, unfoldings are only performed if some branch containing a (mutually) recursive function call could be eliminated in the unfolded body.

It is interesting to note here that an alternative approach to the dead branch elimination performed by the partial evaluator is to prioritize unfolding of function calls whose blockers belong to the unsatisfiable core reported by the SMT solver during model finding. We have found that in practice, both approaches are complementary. The pruning performed during partial evaluation is more precise but slower, while the prioritized unfolding approach may ignore important unfoldings when multiple unsatisfiable cores exist, as well as delay proof finding by focusing on irrelevant branches.

5.1.2 Call Inlining

Our system allows the user to mark certain functions with an @inline annotation which indicates that calls to this function must be inlined. Inlining therefore allows the user to somewhat control the order in which call unfoldings occur in the model finder. Inlining proceeds recursively and this annotation is only allowed on non-recursive functions to ensure termination of the transformation. As our system will type check the marked function and type checking is modular, we can avoid type checking the inlined body and simply check the argument types.

We further support an @inlineOnce annotation for which inlining is *not* recursively performed. This annotation can therefore be attached to recursive functions as well and provides control over whether inlining should be transitive or not.

5.2 Termination Trees

The termination trees introduce a decreases(·) construct which enables specification of the ranking function by which termination is given. We also perform measure inference which allows the system to automatically show program termination. These considerations result in the following *fdef* grammar extension.

$$\begin{array}{rcl} \mathit{fdef} & ::= & \cdots \mid \langle\, \mathit{fannot} \,\rangle^* \ \textbf{def}\ \ \mathit{id}[\ \mathit{tdecls}\ ](\ \mathit{id} : \mathit{type}\ ) : \mathit{type} := \mathit{fbody} \\ \mathit{fbody} & ::= & \{\, \mathsf{decreases}(\ \mathit{expr}\ );\ \mathit{expr}\,\} \mid \mathit{expr} \end{array}$$

Note that we admit function definitions which feature neither size binding nor decreases(·) construct. Such definitions will trigger the automated measure inference procedure.

5.2.1 Decreases Elimination

It is clear that the size binding and expressions syntax presented in Chapter 3 is not compatible with the source Scala syntax. In order to allow users to manually specify termination arguments, we provide the decreases language construct which corresponds to a traditionnal ranking function specification. The translation into the verification language is straightforwardly given as follows.

$$\begin{array}{l} \textbf{def}\ \ f[\overline{\tau}_f](x:\tau_1):\tau_2 := \{\, \mathsf{decreases}(e_m);\ \mathcal{C}[\,f[\overline{\tau}](e)\,]\,\} \ \rightsquigarrow \\ \quad \textbf{def}\ \ f(m:\tau_m)[\overline{\tau}_f](x:\{x:\tau_1 \mid e_m \le m\}):\tau_2 := \mathcal{C}[\,f(\textbf{let}\ x := e\ \textbf{in}\ e_m)[\overline{\tau}](e)\,] \end{array}$$

The well-order τ_m is obtained by computing the type of the expression e_m. Note that the expression e_m is then used both as a refinement in the function parameter type and to compute the size expressions for recursive calls.

All calls to mutually recursive functions in the body of the annotated function definition will be translated into sized calls. The corresponding size expression is computed based on the callee's associated decreases construct, hence all mutually recursive functions that do not have a size binding must feature a decreases statement. We saw in Chapter 3 that the type $\{ x : \tau_1 \mid e_m \leq m \}$ generalizes to τ_1 and the CALL P_1 GENERALIZATION type inference rule allows calls outside the strongly connected component to feature no size expression, hence such calls are preserved during the translation.

Given our encoding of the decreases construct and the type checking algorithm presented in Chapter 3, this translation phase enables the user to provide inductive termination arguments. However, although our verification procedure can handle coinductive definitions, we have not found any satisfactory syntax to specify sized datatypes in the Scala source. Hence, users cannot provide termination arguments for function definitions that rely on coinduction. We have recently begun exploring an extension to the Scala compiler which will allow source-level specification of sized types, so this limitation should disappear in the near future.

5.2.2 Measure Inference

In order to avoid having to annotate each (inductive) Scala function with the required decreases construct, our system attempts to infer ranking functions when no annotation is given. Furthermore, we also infer certain likely refinements on function result types and first-class function parameter types to improve the automation featured by our inference procedure.

In order to uniformly consider ranking functions over arbitrary types, we define an embedding of values into the natural numbers. We will refer to this embedding as the size of a value. This embedding is defined by matching on the erased type of the given expression.

```
def size($e$: $expr$, $\tau$: $type$): $expr$ = $\tau$ match {
  case Nat ⇒ $e$
  case Unit | Boolean | $T$ | $\tau_1 \to \tau_2$ ⇒ 0
  case $(\tau_1, \tau_2)$ ⇒ size($\pi_1(e)$, $\tau_1$) + size($\pi_2(e)$, $\tau_2$)
  case $d[\overline{\tau}]$ ⇒ size-$d[\overline{\tau}](e)$ + 1
}
```

Note that in the above embedding, we relied on a size-d function when computing the size of a datatype expression. Given the definition **type** $d[\overline{\tau}_d](m) := C_1(x_1 : \tau_1) \mid \cdots \mid C_n(x_n : \tau_n)$, we synthesize a (potentially recursive) function size-d as follows.

def size-$d[\overline{\tau}_d](x : d[\tau_1])$: Nat := x **match** $\{ C_1(y_1) \Rightarrow$ size$(y_1, \tau_1) \cdots C_n(y_n) \Rightarrow$ size$(y_n, \tau_n) \}$

The termination of these functions is given through a meta-theoretic argument based on the datatype structure and will not be checked by our system.

Ranking function inference. We synthesize decreases statements by selecting likely candidate ranking functions and performing partial type checking to validate them. Partial type checking follows the same principles as our bidirectional type checking algorithm but only checks verification conditions that are related to size expression decrease. Considered ranking functions consist of the total parameter size, the size of a single projection (for parameters with nested pair type), as well as the lexicographic orderings of projection sizes. We have found these to cover a wide range of termination arguments in our experiments while keeping the ranking function candidate exploration tractable.

In order to handle ranking functions that only transitively decrease, we further consider inlinings of recursive function calls. The resulting ranking function can then be constructed as a lexicographic ordering of the candidate function and an index which is computed based on the considered inlinings. For example, consider the following mutually recursive definitions.

```
def isEven(n: Nat): Boolean = if (n ≈ 0) true else if (n ≈ 1) false else isOdd(n − 1)
def isOdd(n: Nat): Boolean = !isEven(n)
```

After establishing that n transitively decreases in isOdd after inlining the call to isEven, our system will synthesize a ranking function of (n, 0) for isEven and (n, 1) for isOdd.

Refinement inference. Our system further improves the likelihood of successful ranking function inference by automatically strengthening refinement types that will participate in the relevant partial type checks. This is again performed by considering candidate strengthenings and performing (a different set of) partial type checks to confirm them.

Candidate refinements are introduced in function result types in order to relate input and output sizes. Given a function definition with signature **def** $f[\overline{\tau}_f](x:\tau_1)\ :\ \tau_2$, we will attempt to replace τ_2 by a refinement type $\{\, x:\tau_2 \mid \mathsf{size}(x) \le \mathsf{size}(x)\,\}$. Such refinements are particularly useful in cases where some recursive function relies on a function outside the strongly connected component which is itself recursive.

We further consider candidate refinements in the input type of pi-types that occur within a sigma-typed parameter. Here, we want to relate the size of the pi-type input with the size of the previous sigma-type projection. In other words, given the function signature **def** $f[\overline{\tau}_f](x:\Sigma\, y:\tau_1.\ \Pi\, z:\tau_2.\ \tau_3)\ :\ \tau_4$, we will attempt to replace the type τ_2 by a refinement type $\{\, z:\tau_2 \mid \mathsf{size}(x) \le \mathsf{size}(y)\,\}$. These refinements are useful when dealing with recursion within lambda arguments to higher-order functions such as map or fold.

Let us now consider the following program which defines a substitution function over a (very simple) abstract syntax tree given by the Expr datatype.

```
type Option[A] := Some(value: A) | None
type List[A] := Cons(head: A, tail: List[A]) | Nil
type Expr := Var(id: Nat) | App(caller: Expr, args: List[Expr])

def map[A,B](l: List[A], f: A → B): List[B] = l match {
  case Cons(x, xs) ⇒ Cons[B](f(x), map[A,B](xs, f))
  case Nil ⇒ Nil[B]
}

def subst(e: Expr, f: Expr → Option[Expr]): Expr = f(e) match {
  case Some(v) ⇒ v
  case None ⇒ e match {
    case Var(_) ⇒ e
    case App(c, args) ⇒ App(subst(c, f), map[Expr,Expr](args, λx. subst(x, f)))
  }
}
```

The two refinement inference procedures described above will then establish the following signature for the given map function by inferring both result type and pi-type refinements.

$$\textbf{def}\ \mathsf{map}[A,B](l:\mathsf{List}[A], f:\{x:A \mid \mathsf{size}(x)\le\mathsf{size}(l)\}\to B):\{r:\mathsf{List}[B] \mid \mathsf{size}(r)\le\mathsf{size}((l,f))\}$$

It is important to realize here that without the inferred pi-type refinement, our type checking procedure would not be able to show that the size expression decreases in the recursive call to subst within the lambda λx. subst(x, f).

5.3 Inner Function Trees

These trees extend the verification trees with support for inner function definitions that may close over local variables. The inner function lifting transformation lifts all inner functions into top-level definitions. In order to preserve the context under which the inner function was defined, we extend the (type) parameters of the inner function with all (type) variables that appear in the typing context under which the inner function was defined.

5.4 Class Trees

The class trees and associated transformations bridge the gap between algebraic datatypes and Scala trait and class definitions. The datatype syntax on which we have relied up to this point is incompatible with Scala syntax. However, there exist type hierarchies in Scala that are semantically equivalent to our datatypes.

In this section, we present a datatype specialization transformation which translates certain scala type hierarchies into datatypes, and a type encoding transformation which eliminates the remaining Scala definitions. We will present these transformations in this order (instead of the usual reverse pipeline order) so as to clarify which type definitions will be handled by the more general type encoding. When discussing each encoding, we will present the relevant grammar extensions which are given by the class trees.

5.4.1 Datatype Specialization

We start by introducing the new Scala type definitions with which the class trees extend our supported language. The fragment we consider features sealed traits and final case class definitions with multiple inheritance. Case classes feature a single (public) constructor field. Trait and class bodies must be empty, and we will discuss further in the pipeline how methods and trait fields can be handled. The fragment further admits case class constructors and trait and class type deconstruction through match-expressions. These considerations lead to the following extension of the type definition grammar *tdef*. Note that the syntax for class and trait types, case class constructors and match-expressions coincides with the corresponding syntax for datatypes, hence no further grammar extensions are required.

$$\begin{array}{rcl} \textit{tdef} & ::= & \cdots \mid \textbf{sealed trait } \textit{id}[\ \textit{tdecls}\ ]\ \textit{extensions} \\ & & \mid \textbf{final case class } \textit{id}[\ \textit{tdecls}\ ]\ (\ \textit{id} : \textit{type}\ )\ \textit{extensions} \\ \textit{extensions} & ::= & \epsilon \mid \textbf{extends } \textit{id}[\ \textit{tparams}\ ]\ \langle\ \textbf{with } \textit{id}[\ \textit{tparams}\ ]\ \rangle^* \end{array}$$

We will rely hereafter on the general form **trait** $t[\overline{\tau}_t]$ **extends** $\overline{\tau}_p$ to denote sealed trait definitions, and **class** $c[\overline{\tau}_c](x : \tau_1)$ **extends** $\overline{\tau}_p$ to denote final case class definitions. We denote the generalization over trait and class definitions by $n[\overline{\tau}_n]$ **extends** $\overline{\tau}_p$.

The datatype specialization transformation will encode certain Scala trait and class definitions into datatype definitions. For example, consider the following Scala List type hierarchy.

```
sealed trait List[T]
final case class Cons[T](head: T, tail: List[T]) extends List[T]
final case class Nil[T]() extends List[T]
```

These three Scala type definitions can be encoded into the single datatype

```
type List[T](m) := Cons(head: T, tail: List[T](m − 1)) | Nil
```

Let us now consider the conditions under which encoding Scala type definitions into datatypes is possible. We say the trait or class definition $n[\overline{\tau}_n]$ **extends** $\overline{\tau}_p$ is a *datatype candidate* iff it has at least one child, each child definition $c[\overline{\tau}_c]$ **extends** $\cdots$ **with** $n[\overline{\tau}'_n]$ **with** $\cdots$ is a datatype candidate and we have $\overline{\tau}'_n = \overline{\tau}_c$. In other words, for each descendant definition d of n, the type $d[\overline{\tau}]$ is a subtype of $n[\overline{\tau}]$ according to Scala static semantics. We say that $n[\overline{\tau}_n]$ **extends** $\overline{\tau}_p$ is a *root datatype candidate* iff it is a datatype candidate, it has no parents (*i.e.* $\overline{\tau}_p = \emptyset$), and for each descendant $d[\overline{\tau}_d]$ **extends** $\overline{\tau}'_p$, for each parent type $\tau \in \overline{\tau}_p$, τ is a subtype of $n[\overline{\tau}_d]$. Intuitively, the Scala type hierarchy given by a root datatype candidate corresponds to a datatype with some additional internal subtyping relations.

Each root datatype candidate, along with all its descendant definitions, will be encoded into a single datatype definition. Given a root datatype candidate $n[\overline{\tau}_n]$, we will construct a datatype definition where each descendant case class of n will correspond to a datatype constructor. As our encoding of the Scala type hierarchy flattens it into a single datatype definition, trait and class types which are (strict) subtype of the root datatype candidate will have no corresponding datatype in the target language. However, we will see that these types can be encoded through adequately chosen root datatype refinements.

We introduce a type encoding procedure encode : *type* → *type* which encodes subtypes of root datatype candidates into refinement types. Consider a root datatype candidate $n[\overline{\tau}_n]$ and trait or class type $d[\overline{\tau}]$ such that $d[\overline{\tau}]$ is a strict subtype of $n[\overline{\tau}]$. As the type hierarchy is sealed, we can determine whether some value has type $d[\overline{\tau}]$ by considering the set of constructors which are below d in the hierarchy. Given the set $c_1, \cdots, c_r$ of class definitions for which $c_i[\overline{\tau}]$ is a subtype of $d[\overline{\tau}]$, these considerations lead to the following encoding rule.

$$\mathsf{encode}(d[\overline{\tau}]) \;=\; \{x : n[\mathsf{encode}(\overline{\tau})] \mid x\,\textbf{match}\,\{\,c_1(y_1) \;\Rightarrow\; \textbf{true} \cdots c_r(y_r) \;\Rightarrow\; \textbf{true} \;\_\; \Rightarrow\; \textbf{false}\,\}\}$$

The remaining type encodings are performed recursively on the type structure. Based on this type encoding, and given the case class definitions $\cdots$, **class** $c_j[\overline{\tau}_j](x_j : \tau_j)$ **extends** $\overline{\tau}_p, \cdots$ which are descendants of the root datatype candidate $n[\overline{\tau}_n]$, we can construct the encoded datatype definitions as follows.

$$\textbf{type}\;\; n[\overline{\tau}_n](m) \;:=\; \cdots \;\mid\; c_j(x_j : \mathsf{encode}(\tau_j[\overline{\tau}_j/\overline{\tau}_n])) \;\mid\; \cdots$$

Note that the size expressions based on the binding m can be automatically assigned to the recursive datatype occurrences in each $\mathsf{encode}(\tau_j[\overline{\tau}_j/\overline{\tau}_n])$. Finally, the transformation will need to adapt match-expressions by introducing (potential) extra error branches when the scrutinee has a non-root trait or class type.

5.4.2 Type Encoding

We now discuss how more complex Scala type hierarchies which were not handled by the datatype specialization transformation described above can be encoded into the target language. Consider the following simplified definition of the List type in the Scala standard library. We use the identifier Cons here instead of :: for readability.

```
sealed abstract class List[+A]
final case class Cons[+B](head: B, tail: List[B]) extends List[B]
final case object Nil extends List[Nothing]
```

We want an encoding that allows us to represent constructor types (*e.g.* Cons[B] and Nil.**type**), the top type Any and the bottom type Nothing. We further want the encoding to be consistent with the subtyping relations given by Scala's type system (*e.g.* List[Nat] <: List[Any] and List[Any] <: Any). Finally, we want the encoding to be compatible with features from the verification language such as refinement types and datatypes in order to allow verification of annotated Scala programs. In this sub-section, we will present a procedure that encodes (sealed non-datatype) type hierarchies with multiple inheritance, the top and bottom types, as well as union and intersection types. We will then discuss how the procedure can be extended to handle other features of the Scala type system such as declaration-site variance.

The fragment we consider here features trait and class definitions as given above, as well as the top type Any, the bottom type Nothing, union types and intersection types. These considerations lead to the following *type* grammar extension.

$$\mathit{type} \quad ::= \quad \cdots \mid \mathsf{Any} \mid \mathsf{Nothing} \mid \mathit{type} \cup \mathit{type} \mid \mathit{type} \cap \mathit{type}$$

Note that the current version of Scala only features a limited form of intersection types (*i.e.* through the **with** keyword) and no union types. However, Scala 3 will introduce support for general union and intersection types, hence we include them in our considered fragment. In the following, we will rely on a helper function $\mathsf{tpe} : \mathit{expr} \to \mathit{type}$ that computes the most precise type of a given Scala expression. We will further rely on the least-upper-bound $\sqcap$ and greatest-lower-bound $\sqcup$ operations on the type lattice given by the Scala subtyping relation.

The main insight in our encoding procedure is that the Scala type hierarchy can be flattened into a single datatype in the verification language, and refinement types can then be leveraged to ensure that type encodings are sufficiently precise. Given the known Scala type definitions, we will synthesize a (monomorphic) datatype definition based on the final case classes in the hierarchy. The type parameters which appear in the Scala trait or class definitions will be erased similarly to how type erasure is performed during compilation to bytecode. We will then generate (mutually) recursive predicate functions that encode the constraints imposed by the Scala type definitions on the erased datatype definition.

Let us consider a value v which has Scala type $\mathsf{List}[\tau]$ for some concrete type τ. Scala's type system ensures that v must be of the shape $\mathsf{Cons}(v_1, \cdots \mathsf{Cons}(v_n, \mathsf{Nil}) \cdots)$ where v_i has type τ for $1 \leq i \leq n$. Given some predicate p_τ which determines whether a value has type τ, we can therefore recursively traverse the value v in order to determine whether it is indeed a value of type $\mathsf{List}[\tau]$. Given a Scala program which contains the single List type hierarchy (and some number of function definitions), consider the following datatype definition.

$$\textbf{type}\ \mathsf{Object} := \mathsf{Cons}(\mathsf{head} : \mathsf{Object}, \mathsf{tail} : \mathsf{Object}) \mid \mathsf{Nil} \mid \mathsf{Other}(n : \mathsf{Nat})$$

This datatype ensures that the Scala definition of case class equality is preserved in the encoding. The Other constructor allows open programs analogously to how the Else constructor was used in the lambda datatype. Let us now consider the following mutually recursive propositional function definitions. Note that we rely here and in the following on the shorthands True and False to denote the propositions True(**true**) and True(**false**).

```
def isList(obj: Object, A: Object ⇒ Prop): Prop = isCons(obj, A) ∨ isNil(obj)

def isCons(obj: Object, B: Object ⇒ Prop): Prop = x match {
  case Cons(x, xs) ⇒ B(x) ∧ isList(xs, B)
  case _ ⇒ False
}

def isNil(obj: Object): Prop = obj match {
  case Nil ⇒ True
  case _ ⇒ False
}
```

These definitions correspond to the constraints on the Object datatype given by the Scala List type hierarchy. In other words, the propositional function call isList(v, p_τ) will hold whenever Scala's type system assigns the type List[τ] to the value v. We can therefore leverage refinement types to precisely encode the constraints given by Scala's type system. Based on these functions, it is clear that the type encoding of List[τ] is $\{x : \text{Object} \mid \text{isList}(x, p_\tau)\}$.

Recall that we want our encoding to be consistent with the subtyping relations that hold in Scala's type system. Let us consider the following type encodings.

$$\begin{array}{rcl}
\text{Any} & \rightsquigarrow & \text{Object} \\
\text{List[Any]} & \rightsquigarrow & \{x : \text{Object} \mid \text{isList}(x, \lambda y.\ \text{True})\} \\
\text{List[Nil.}\mathbf{type}\text{]} & \rightsquigarrow & \{x : \text{Object} \mid \text{isList}(x, \lambda y.\ \text{isNil}(y))\} \\
\text{Cons[Nil.}\mathbf{type}\text{]} & \rightsquigarrow & \{x : \text{Object} \mid \text{isCons}(x, \lambda y.\ \text{isNil}(y))\}
\end{array}$$

According to Scala's type system, these types are connected by the following chain of subtyping relations Cons[Nil.**type**] <: List[Nil.**type**] <: List[Any] <: Any. Given the denotations defined in Chapter 3, we see that the subset relation between the denotations of the encodings is consistent with the given relations.

In the remainder of this sub-section, we will define our type encoding procedure and discuss some potential extensions to handle type parameter bounds, declaration-site variance, higher-kinded types, as well as instance checks, cast expressions and field accesses.

Encoding Scala Programs

We will now present our procedure that encodes programs with Scala types into the target language. At a high level, this encoding is enabled by the mutually recursive type encoding, expression encoding, typing relation encoding and conversion procedures. We will start by describing how (non-datatype) trait and class definitions are encoded into datatypes, then present the type and expression encoding procedures, and finally define the more involved typing relation encoding and conversion procedures.

We encode Scala types through refinements of a unique monomorphic Object datatype. This datatype will feature multiple constructors that are generated based on case class definitions and types which exist in the target language (called *native types* hereafter). In order to model open programs, we introduce a special Open(x : Nat) constructor in the Object datatype that does not correspond to any type definition in the Scala program. The refinements on which our type encoding relies are obtained through an encoding of the *typing relation*, namely an encoding of what it means for value v to have type τ.

As the considered Scala fragment allows subtyping, our encoding will rely on a conversion procedure that can convert an expression between different compatible Scala types. Conversion is necessary, for example, when passing a boolean-typed expression into a position that expects a value of type Any. Indeed, as Any is encoded into the Object datatype, our type checking procedure will fail when trying to prove that the boolean expression has type Object. Similarly to how boxing is leveraged by the Java and Scala compilers, we rely on boxing and unboxing when converting between native types and compatible Scala types.

Since the Object datatype is monomorphic, type parameters that appear in Scala types will be erased to the Object datatype during encoding. For function definitions, we replace the erased type parameters by typing predicate parameters with type Object → Prop which are leveraged to precisely encode the typing relation. In datatype and case class definitions, the type parameters are completely erased in order to preserve Scala equality semantics. We will again rely on the conversion procedure in order to box and unbox expressions that flow in and out of type parameter positions.

Our aim is to embed code with Scala types into our target language. Hence, if a datatype or function definition already belongs to the target language, no encoding should be necessary. We go even further by having our encoding procedure operate at the granularity of type parameters. Type parameters are partitioned into a subset that should be maintained and another that will be erased. As an example of this partitioning, let us consider a foldLeft implementation over the List type previously defined.

```
def foldLeft[A,B](list: List[A], zero: B, op: (B, A) ⇒ B): B = list match {
  case Cons(x, xs) ⇒ foldLeft[A,B](xs, op(zero, x), op)
  case Nil ⇒ zero
}
```

Our encoding procedure will replace the A type parameter by a typing predicate parameter with type Object → Prop. However, the second type parameter does not appear in a position that requires a typing predicate and can therefore be maintained, resulting in the following function definition encoding.

```
def foldLeft[B](A: Object ⇒ Prop, list: {x : Object | isList(x, A) },
                  zero: B, op: (B, {x : Object | A(x) }) ⇒ B): B = list match {
  case Cons(x, xs) ⇒ foldLeft[B](A, xs, op(zero, x), op)
  case Nil ⇒ zero
}
```

We have found that this granularity of the encoding procedure allows for Scala code and the verification language to co-exist in a same program without sacrificing predictability and scalability of verification. In our presentation of the encoding procedure, we will restrict our discussion to complete program encodings where all type parameters are erased.

In order to improve readability of the procedures, we will consider that all function and type definitions feature a single type parameter. One can then extrapolate the procedures to handle an arbitrary number of type parameters. Furthermore, we allow multi-parameter variants of function definitions and calls in the target language resulting from encoding. It is clear that these can be transformed into single-parameter versions through adequate use of nested sigma-types, pair constructions and pair projections. Finally, we will ignore language constructs that were introduced in the verification and inner function trees as their encodings can be straightforwardly derived from the presented procedures.

The various encoding and conversion procedures are given in the context of some program P in the fragment described above. We define the following four mutually recursive procedures:

1. The type encoding encode : *type* → *type* takes a Scala type τ and encodes it into a corresponding type encode(τ) in the target language.
2. The expression encoding encode : (*expr*, *type*, *type*) → *expr* takes a Scala expression e, its Scala type τ, an expected Scala type τ' and encodes it into a corresponding expression encode(e, τ, τ') which has type encode(τ'). In the following, to improve readability we denote the encoding encode(e, tpe(e), tpe(e)) by encode(e).
3. The typing relation encoding typed : (*expr*, *type*, *type*) → *expr* takes an encoded expression e (with erased type Object), its Scala type τ, and a Scala type τ' against which the expression is being checked, and produces a Prop-typed expression typed(e, τ, τ') in the target language that holds when e evaluates to some value with type τ'.
4. The conversion procedure convert : (*expr*, *type*, *type*) → *expr* takes an encoded expression e, its Scala type τ, and a Scala type τ' to which e should be converted, and produces an expression convert(e, τ, τ') whose type corresponds to the encoding of τ'. The conversion procedure is inverse to itself, namely convert(convert(e, τ, τ'), τ', τ) should produce an expression that is equivalent to e.

Before defining the encoding and conversion procedures described above, we discuss how function, datatype, trait and class definitions are encoded. As previously mentioned, the function definition type parameter is replaced by a *typing predicate* parameter. Given the function type parameter T in the Scala program, the corresponding function definition in the encoded program will take a typing predicate parameter $T : \mathsf{Object} \to \mathsf{Prop}$. The type T which appears within the function signature and body is then precisely encoded into the refinement type $\{x : \mathsf{Object} \mid T\ x\}$. Given the type and expression encoding procedures described above, these considerations lead to the following function definition encoding.

$$\begin{array}{l}\mathsf{encode}(\mathbf{def}\ f(m:\tau_m)[T](x:\tau_1):\tau_2 \ := \ e_f) \ = \\ \quad \mathbf{def}\ f(m:\mathsf{encode}(\tau_m))(T:\mathsf{Object}\to\mathsf{Prop},\, x:\mathsf{encode}(\tau_1)):\mathsf{encode}(\tau_2) \ := \ \mathsf{encode}(e_f)\end{array}$$

It is important to note that the Scala operational semantics are defined for programs where type parameters have been erased. Hence, the introduced typing predicate parameter will not affect the result of evaluation. We integrate this insight in our model finding procedure by omitting the typing predicate parameter in our embedding of function calls. This allows the underlying SMT solver to ignore the typing predicate when applying the congruence rule for uninterpreted function symbols.

We avoid introducing typing predicates in the encodings of datatype, trait and class definitions in order to ensure that equality does not depend on type parameter instantiations and is consistent with the Scala semantics. We therefore introduce a special type ? which corresponds to a type erasure. This type will be encoded into the (unrefined) Object type. The encoding of datatype definitions is then given as follows.

$$\begin{array}{l}\mathsf{encode}(\mathbf{type}\ d[T](m) \ := \ C_1(x_1:\tau_1) \mid \cdots \mid C_n(x_n:\tau_n)) \ = \\ \quad \mathbf{type}\ d(m) \ := \ C_1(x_1:\mathsf{encode}(\tau_1[T/?])) \mid \cdots \mid C_n(x_n:\mathsf{encode}(\tau_n[T/?]))\end{array}$$

Trait and class definitions do not lead to corresponding type definitions in the encoded program. Instead, for each $(\mathbf{class}\ c[T](x:\tau_1)\ \mathbf{extends}\ \overline{\tau}_p) \in P$, we generate a constructor $c(x:\mathsf{encode}(\tau_1[T/?]))$ for the Object datatype. We will see when discussing the typing relation encoding and conversion procedures that in addition to the encodings given here, these procedures will rely on synthetic function definitions that are generated based on trait, class and datatype definitions in the original program.

Type and Expression Encodings

We now present the type encoding procedure. The type encoding distinguishes "native" types which exist in the target language and Scala types which were introduced in the extended fragment. The encoding of native types proceeds by recursion while the encoding of Scala types (as well as type parameters) takes place through a refinement of the Object type.

Native types, namely the unit and boolean types, datatypes, pi- and sigma-types, are handled by recursively encoding their component types, leading to the following encoding rules.

$$\mathsf{encode}(\mathsf{Unit}) = \mathsf{Unit} \qquad \mathsf{encode}(\mathsf{Boolean}) = \mathsf{Boolean} \qquad \mathsf{encode}(d[\tau]) = d$$

$$\mathsf{encode}(d[\tau](e_m)) = d(\mathsf{encode}(e_m)) \qquad \mathsf{encode}(\Sigma x : \tau_1 .\, \tau_2) = \Sigma x : \mathsf{encode}(\tau_1).\, \mathsf{encode}(\tau_2)$$

$$\mathsf{encode}(\Pi x : \tau_1 .\, \tau_2) = \Pi x : \mathsf{encode}(\tau_1).\, \mathsf{encode}(\tau_2)$$

The newly introduced Scala types are encoded into the Object type refined by the typing relation encoding, leading to the following encoding rule.

$$\mathsf{encode}(\tau) = \{ x : \mathsf{Object} \mid \mathsf{typed}(x, ?, \tau) \}$$

This rule is applied when τ is either a trait type, a class type, the top type, the bottom type, an intersection type, a union type or a type parameter. The rule further applies to the special erasure type ? introduced above (we will see that $\mathsf{typed}(x, ?, ?)$ always holds).

Let us now consider the expression encoding procedure. The main task of this encoding is to perform boxing and unboxing of expressions when necessary. It is clear that boxing must occur when an expression with native type flows into a position that expects the Any type. Furthermore, as type parameters are erased, boxing and unboxing must occur when data flows into and out of positions with parametric type.

We first present a set of expression encoding rules that rely solely on the expression shape to determine which rule should be applied. Similarly to the shape-directed type checking judgement rules given in Chapter 3, the encoding is pushed down into let-expression bodies and if- and match-expression branches.

$$\begin{aligned}
&\mathsf{encode}(\mathbf{let}\ x := e_1\ \mathbf{in}\ e_2, \tau, \tau') = \\
&\quad \mathbf{let}\ x := \mathsf{encode}(e_1)\ \mathbf{in}\ \mathsf{encode}(e_2, \tau, \tau') \\
&\mathsf{encode}(\mathbf{if}\ (c)\ e_1\ \mathbf{else}\ e_2, \tau, \tau') = \\
&\quad \mathbf{if}\ (\mathsf{encode}(c))\ \mathsf{encode}(e_1, \tau, \tau')\ \mathbf{else}\ \mathsf{encode}(e_2, \tau, \tau') \\
&\mathsf{encode}(e\ \mathbf{match}\ \{ C_1(y_1) \Rightarrow e_1 \cdots C_n(y_n) \Rightarrow e_n \}, \tau, \tau') = \\
&\quad \mathsf{encode}(e)\ \mathbf{match}\ \{ C_1(y_1) \Rightarrow \mathsf{encode}(e_1, \tau, \tau') \cdots C_n(y_n) \Rightarrow \mathsf{encode}(e_n, \tau, \tau') \}
\end{aligned}$$

We next present a set of encoding rules where the given and expected type exactly match the type of e (computed through $\mathsf{tpe}(e)$). In other words, these rules are defined for encodings of the shape $\mathsf{encode}(e, \mathsf{tpe}(e), \mathsf{tpe}(e))$ (denoted by $\mathsf{encode}(e)$). The rule is then selected by examining the shape of the given expression.

We start with the function call encoding rules. Recall that type parameters are erased and typing predicate parameters are introduced in their stead. We therefore generate a typing predicate lambda by leveraging the typing relation encoding and convert the call argument to an expression with boxed type parameters. Then the call result is converted back to an expression with unboxed type parameters. The (omitted) generalized call encoding is similarly defined by relying on the generalized version of the parameter type τ_1.

$$\frac{(\textbf{def}\ f(m:\tau_m)[T](x:\tau_1):\tau_2\ :=\ e_f)\in P}{\begin{array}{l}\mathsf{encode}(f(e_m)[\tau](e_1))\ =\ \mathsf{convert}(f(\mathsf{encode}(e_m))(\\ \quad \lambda x:\mathsf{Object}.\ \mathsf{typed}(x,?,\tau), \mathsf{encode}(e_1,\tau_1[T/\tau],\tau_1[T/?])), \tau_2[T/?], \tau_2[T/\tau])\end{array}}$$

We then present the case class and datatype constructor encoding rules. Type parameters are again erased but no typing predicate is introduced in these cases. The constructor argument is simply converted into an expression with boxed type parameters. Again, the (omitted) generalized datatype constructor encoding is defined by relying on the generalization of τ_i.

$$\frac{(\textbf{class}\ c[T](x:\tau_1)\ \textbf{extends}\ \overline{\tau}_p)\in P}{\mathsf{encode}(c[\tau](e_1))\ =\ c(\mathsf{encode}(e_1,\tau_1[T/\tau],\tau_1[T/?]))}$$

$$\frac{(\textbf{type}\ \ d[T](m)\ :=\ \cdots\ |\ C_i(x_i:\tau_i)\ |\ \cdots)\in P}{\mathsf{encode}(C_i(e_m)[\tau](e_1))\ =\ C_i(\mathsf{encode}(e_m))(\mathsf{encode}(e_1,\tau_i[T/\tau],\tau_i[T/?]))}$$

The remaining encoding rules with matching type are given below. Note that the encoding rules for function applications and equalities are slightly more involved as they must ensure that the expected type is correctly propagated.

$$\mathsf{encode}(x)\ =\ x \qquad \mathsf{encode}(())\ =\ () \qquad \mathsf{encode}(\textbf{true})\ =\ \textbf{true} \qquad \mathsf{encode}(\textbf{false})\ =\ \textbf{false}$$

$$\mathsf{encode}(\pi_i(e))\ =\ \pi_i(\mathsf{encode}(e)) \qquad \mathsf{encode}((e_1,e_2))\ =\ (\mathsf{encode}(e_1),\mathsf{encode}(e_2))$$

$$\mathsf{encode}(\mathsf{err}[\tau])\ =\ \mathsf{err}[\mathsf{encode}(\tau)] \qquad \mathsf{encode}(\lambda x:\tau_1.\ e)\ =\ \lambda x:\mathsf{encode}(\tau_1).\ \mathsf{encode}(e)$$

$$\frac{\mathsf{tpe}(e_1)=\tau_2\rightarrow\tau}{\mathsf{encode}(e_1\ e_2)\ =\ \mathsf{encode}(e_1)\ \mathsf{encode}(e_2,\mathsf{tpe}(e_2),\tau_2)}$$

$$\frac{\tau_{lub}=\mathsf{tpe}(e_1)\sqcap\mathsf{tpe}(e_2)}{\mathsf{encode}(e_1\approx e_2)\ =\ \mathsf{encode}(e_1,\mathsf{tpe}(e_1),\tau_{lub})\approx\mathsf{encode}(e_2,\mathsf{tpe}(e_2),\tau_{lub})}$$

Finally, we give the encoding rule that performs conversion between the given and expected types. This rule only applies if no other expression encoding rule does. In particular, the expression is neither a let-, if- nor match-expression and the types τ and τ' do not both match the computed type $\mathsf{tpe}(e)$.

$$\mathsf{encode}(e,\tau,\tau')\ =\ \mathsf{convert}(\mathsf{encode}(e),\tau,\tau')$$

Typing Relation and Conversions

We now present the typing relation encoding and conversion procedures. For each type, we discuss the potential synthetic function definitions on which the typing relation encoding and conversion rules rely, as well as the Object datatype constructors that are necessary for boxing.

We will start by presenting the typing relation encoding and conversion rules for native types. As previously mentioned, expressions with native type allow both boxed and unboxed variants. Given a native type τ, we will generally define an erased version $\tau_?$ of τ and add a constructor $\mathsf{Box}(x : \tau_?)$ to the Object datatype. Based on this constructor and the erasure $\tau_?$ of τ, we can define the following typing relation encoding and conversion rules.

$$\mathsf{typed}(e, ?, \tau) \;=\; e\;\mathbf{match}\;\{\,\mathsf{Box}(y) \;\Rightarrow\; \mathsf{typed}(y, \tau_?, \tau) \;\; \_ \;\Rightarrow\; \mathsf{False}\,\}$$

$$\mathsf{convert}(e, ?, \tau) \;=\; e\;\mathbf{match}\;\{\,\mathsf{Box}(y) \;\Rightarrow\; \mathsf{convert}(y, \tau_?, \tau) \;\; \_ \;\Rightarrow\; \mathsf{err}[\mathsf{encode}(\tau)]\,\}$$

$$\mathsf{typed}(e, \tau, ?) \;=\; \mathsf{True} \qquad\qquad \mathsf{convert}(e, \tau, ?) \;=\; \mathsf{Box}(\mathsf{convert}(e, \tau, \tau_?))$$

For most native types, we will therefore simply present the corresponding boxing constructor and rely on the rules given above. Note that the rule for $\mathsf{typed}(e, \tau, ?)$ does not require a boxing constructor and applies to any type τ (including Scala types).

For Scala types and type parameters, no unboxed variant exists. Hence, the conversion procedure between given and erased types is simply the identity function. We therefore have the following conversion rules when τ is either the erased type ?, a trait type, a class type, the top type, the bottom type, a union type, an intersection type or a type parameter.

$$\mathsf{convert}(e, \tau, ?) \;=\; e \qquad\qquad \mathsf{convert}(e, ?, \tau) \;=\; e$$

Boolean and unit types. The typing relation encoding and conversion rules for unboxed boolean expressions are given as follows.

$$\mathsf{typed}(e, \mathsf{Boolean}, \mathsf{Boolean}) \;=\; \mathsf{True} \qquad\qquad \mathsf{convert}(e, \mathsf{Boolean}, \mathsf{Boolean}) \;=\; e$$

The boxed variant of boolean expressions then relies on a $\mathsf{Bool}(x : \mathsf{Boolean})$ constructor of the Object datatype where the erased type corresponding to Boolean remains Boolean.

For the unit type, the typing and conversion rules for unboxed expressions are given similarly to the boolean case. For boxing, we introduce a $\mathsf{Unit}(x : \mathsf{Unit})$ constructor in the Object datatype. As there exists only a single unit value, there is no need to box it and we can directly convert between the Unit constructor and $()$ value. The typing relation encoding and conversion rules for unit expressions are then given as follows.

$$\mathsf{typed}(e, \mathsf{Unit}, \mathsf{Unit}) \;=\; \mathsf{True} \qquad\qquad \mathsf{typed}(e, ?, \mathsf{Unit}) \;=\; e \approx \mathsf{Unit}(())$$

$$\mathsf{convert}(e, \mathsf{Unit}, \mathsf{Unit}) \;=\; e \qquad \mathsf{convert}(e, ?, \mathsf{Unit}) \;=\; () \qquad \mathsf{convert}(e, \mathsf{Unit}, ?) \;=\; \mathsf{Unit}(())$$

Sigma-types. The encoding and conversion rules for unboxed expressions are given by recursing down into the pair. Note that special care must be taken to correctly propagate the bindings that occur within dependent types.

$$
\begin{aligned}
&\mathsf{typed}(e, \Sigma x : \tau_1.\, \tau_2, \Sigma x : \tau_1'.\, \tau_2') \;=\\
&\quad \textbf{let } x := \pi_1(e) \textbf{ in } \mathsf{typed}(x, \tau_1, \tau_1') \wedge\\
&\qquad \textbf{let } x' := \mathsf{convert}(x, \tau_1, \tau_1') \textbf{ in } \mathsf{typed}(\pi_2(e), \tau_2, \tau_2'[x/x'])\\
&\mathsf{convert}(e, \Sigma x : \tau_1.\, \tau_2, \Sigma x : \tau_1'.\, \tau_2') \;=\\
&\quad \textbf{let } x := \pi_1(e) \textbf{ in let } x' := \mathsf{convert}(x, \tau_1, \tau_1') \textbf{ in } (x', \mathsf{convert}(\pi_2(e), \tau_2, \tau_2'[x/x']))
\end{aligned}
$$

Boxing of sigma-types relies on a $\mathsf{Pair}(x : (\mathsf{Object}, \mathsf{Object}))$ constructor and the erased version of sigma-type $\Sigma x : \tau_1.\, \tau_2$ is therefore $(?, ?)$ (recall that ? is encoded into the Object type).

Pi-types. The typing relation encoding for unboxed pi-types relies on universal quantifiers to precisely encode the relation. A first quantified proposition ensures that the expected parameter type is a subtype of the given one (by contravariance). A second proposition then ensures that the result of the function application has the expected result type (by covariance). These considerations lead to the following encoding rule.

$$
\begin{aligned}
&\mathsf{typed}(e, \Pi x : \tau_1.\, \tau_2, \Pi x : \tau_1'.\, \tau_2') \;=\\
&\quad \forall x : \mathsf{encode}(\tau_1').\, \mathsf{typed}(x, \tau_1', \tau_1) \wedge\\
&\quad \forall x : \mathsf{encode}(\tau_1).\, \mathsf{typed}(x, \tau_1, \tau_1') \implies\\
&\qquad \textbf{let } x' := \mathsf{convert}(x, \tau_1, \tau_1') \textbf{ in } \mathsf{typed}(e\ x, \tau_2, \tau_2'[x/x'])
\end{aligned}
$$

One should note that these quantifiers significantly limit the scalability of verification. However, we have found in our experiments that this typing relation encoding rule is only rarely necessary during verification of Scala programs.

The conversion between unboxed pi-types wraps the expression into a lambda with the expected parameter type, applies the expression to the converted parameter, and converts the application back to the expected result type.

$$
\begin{aligned}
&\mathsf{convert}(e, \Pi x : \tau_1.\, \tau_2, \Pi x : \tau_1'.\, \tau_2') \;=\\
&\quad \lambda x : \mathsf{encode}(\tau_1').\, \textbf{let } x' := \mathsf{convert}(x, \tau_1', \tau_1) \textbf{ in } \mathsf{convert}(e\ x', \tau_2[x/x'], \tau_2')
\end{aligned}
$$

Note that this conversion rule breaks the function equality semantics presented in Chapter 2. Indeed, the wrapping operations modify the structure of the encoded lambda. However, the reference equality semantics which Scala employs for function-typed values are already inconsistent with our structural equality. We must therefore ensure that no equality checks occur between function-typed expressions during verification, regardless of the boxing and unboxing strategy used during type encoding.

Similarly to how sigma-types were boxed, we rely on a $\mathsf{Function}(x : \mathsf{Object} \to \mathsf{Object})$ constructor here and the erased version of $\Pi x : \tau_1.\, \tau_2$ is therefore $? \to ?$.

Refinement types. The typing relation encoding ignores refinements in the expression type τ, and when given a refined expected type, the encoding takes the conjunction of the underlying typing relation and the refinement's encoding.

$$\mathsf{typed}(e, \{x : \tau \mid p\}, \tau') = \mathsf{typed}(e, \tau, \tau')$$

$$\mathsf{typed}(e, \tau, \{x : \tau' \mid p\}) = \mathsf{typed}(e, \tau, \tau') \land \textbf{let } x := \mathsf{convert}(e, \tau, \tau') \textbf{ in } \mathsf{encode}(p)$$

Refinement types disapear completely in the simple type system, so conversions can be simply deferred to the underlying type. Hence, no boxing constructor is required here and conversion is given by the following rules.

$$\mathsf{convert}(e, \{x : \tau \mid p\}, \tau') = \mathsf{convert}(e, \tau, \tau') \qquad \mathsf{convert}(e, \tau, \{x : \tau' \mid p\}) = \mathsf{convert}(e, \tau, \tau')$$

Datatypes. Consider the definition $(\textbf{type } d[T](m) := C_1(x_1 : \tau_1) \mid \cdots \mid C_n(x_n : \tau_n)) \in P$. In order to encode the typing relations associated to datatypes, we generate the following (potentially recursive) function based on the datatype definition. This definition follows from the denotation of sized datatypes presented in Chapter 3.

$$\begin{aligned}&\textbf{def } \mathsf{is}\text{-}d(m : \mathsf{Nat}, x : d(m), T : \mathsf{Object} \to \mathsf{Prop}) : \mathsf{Prop} := \\ &\quad m > \mathsf{Zero} \implies x \textbf{ match } \{ C_1(y_1) \Rightarrow \mathsf{typed}(y_1, \tau_1, \tau_1) \cdots C_n(y_n) \Rightarrow \mathsf{typed}(y_n, \tau_n, \tau_n) \}\end{aligned}$$

The typing relation encoding for the unboxed sized datatypes that will appear within the field types τ_1 to τ_n is then given as follows.

$$\mathsf{typed}(e, d[\tau](e_m), d[\tau'](e'_m)) = \mathsf{is}\text{-}d(\mathsf{encode}(e'_m), e, \lambda x : \mathsf{Object}.\, \mathsf{typed}(x, ?, \tau'))$$

Recall that the syntactic restriction we imposed on datatype definitions in Chapter 3 ensures that the size expression will decrease in recursive datatypes. Hence, the size parameter m will also decrease in recursive calls of the is-d function and the function is therefore terminating.

For the intersection datatype, we rely on a quantified proposition, again in accordance with the denotation presented in Chapter 3, giving us the following encoding rule.

$$\mathsf{typed}(e, d[\tau], d[\tau']) = \forall m : \mathsf{Nat}.\, \mathsf{is}\text{-}d(m, e, \lambda x : \mathsf{Object}.\, \mathsf{typed}(x, ?, \tau'))$$

If the datatype definition is inductive (and generalizes), we can instead generate a is-d function for the intersection datatype which uses the datatype structure as its measure. This allows us to avoid relying on the expensive quantifier to encode the typing relation.

Let us now discuss the conversion procedures for datatypes. Similarly to boolean expressions, conversion between unboxed datatypes simply returns the same expression.

$$\mathsf{convert}(e, d[\tau], d[\tau']) = e \qquad \mathsf{convert}(e, d[\tau](e_m), d[\tau'](e'_m)) = e$$

Now consider the following boxing constructors $\mathsf{Box}\text{-}d_m(x : \Sigma\, m : \mathsf{Nat}.\ d(m))$ for sized datatypes and $\mathsf{Box}\text{-}d(x : d)$ for intersection datatypes. It is clear that one can define boxing and unboxing procedures for both sized and intersection datatypes through these constructors. However, this implies that equality of boxed values will depend on whether the datatype was sized or not, which is not consistent with the operational semantics. We avoid the issue of sized versus intersection datatype boxing by providing no boxing and unboxing conversion procedures for sized datatypes. If the program encoding encounters rules of the shape $\mathsf{convert}(e, ?, d[\tau'](e'_m))$ or $\mathsf{convert}(e, d[\tau](e_m), ?)$, the encoding simply fails. Recall that definitions which fall within the verification language require no encoding, so one can write useful programs with sized datatypes that satisfy this restriction. Boxing of intersection datatypes then relies on the $\mathsf{Box}\text{-}d$ constructor given above and the erased version of $d[\tau]$ is $d[?]$.

Type parameters. Recall that type parameters are erased in favor of typing predicates. Hence, for each expected type parameter T that appears in the typing relation encoding, the function definition encoding will ensure that we have a corresponding variable T in scope with type $\mathsf{Object} \to \mathsf{Prop}$. We also define an identity conversion rule when the expected and given types are both T. These considerations lead to the following encoding and conversion rules.

$$\mathsf{typed}(e, \tau, T) \ = \ T\ e \qquad\qquad \mathsf{convert}(e, T, T) \ = \ e$$

Trait and class types. Similarly to how the typing relation encoding associated to datatypes relied on a synthetic function definition, we generate function definitions to encode the typing relation associated to both trait and class types. Consider some nominal type $n[\tau']$ where n corresponds to either a trait or class definition. We will generate a (potentially recursive) function definition with the following signature.

$$\textbf{def}\ \ \mathsf{is}\text{-}n(x : \mathsf{Object}, T : \mathsf{Object} \to \mathsf{Prop}) \ : \ \mathsf{Prop}$$

The typing relation associated to the nominal type is then encoded as follows.

$$\mathsf{typed}(e, \tau, n[\tau']) \ = \ \mathsf{is}\text{-}n(e, \lambda x : \mathsf{Object}.\ \mathsf{typed}(x, ?, \tau'))$$

Note that we rely here on the structure of the Object parameter as the function's measure. If the trait or class definition is not inductive, then the $\mathsf{is}\text{-}n$ function may be non-terminating. In such cases, we introduce an extra Nat parameter which we make sure decreases in the recursive calls. We then encode the typing relation through a universal quantifier as in the case of non-inductive datatype intersections.

Let us now define the body of the $\mathsf{is}\text{-}n$ function. First consider the case where n corresponds to a trait definition $(\textbf{trait}\ n[T]\ \textbf{extends}\ \overline{\tau}_p) \in P$. Further consider each child definition $c_i[T_i]\ \textbf{extends}\ \cdots\ \textbf{with}\ n[\tau_i]\ \textbf{with}\ \cdots$ in P. Note that c_i may correspond to either a trait or class definition. For each child, we compute the type $c_i[\tau'_i]$ where $\tau'_i = T$ if $\tau_i = T_i$ else $?$. The is-n function is then defined as follows.

$$\textbf{def}\ \textsf{is-}n(x : \textsf{Object},\ T : \textsf{Object} \to \textsf{Prop})\ :\ \textsf{Prop}\ :=\ \cdots \lor \textsf{typed}(x, ?, c_i[\tau'_i]) \lor \cdots$$

Note that although we "forget" certain typing constraints here by replacing them with unkowns, this definition of is-n actually precisely encodes the typing relation. Indeed, since our traits have no members, only type parameters that are shared with descendant case classes may constrain values of nominal trait types.

Now consider the case where we have (**class** $n[T](x : \tau)$ **extends** $\overline{\tau}_p) \in P$. Recall that the Object datatype features a constructor $n(x : \textsf{encode}(\tau[T/?]))$. The body of the generated is-n function is then given as follows.

$$\begin{aligned}&\textbf{def}\ \textsf{is-}n(x : \textsf{Object},\ T : \textsf{Object} \to \textsf{Prop})\ :\ \textsf{Prop}\ := \\ &\quad x\ \textbf{match}\ \{\ n(y)\ \Rightarrow\ \textsf{typed}(y, \tau[T/?], \tau)\ \ \_\ \Rightarrow\ \textsf{False}\ \}\end{aligned}$$

Finally, we introduce a conversion rule for trait and class types. As expressions with trait or class type are always boxed, conversion is given by the identity function.

$$\textsf{convert}(e, n[\tau], n[\tau'])\ =\ e$$

Top and bottom types. Encoding the typing relation associated to the top and bottom types is straightforward. Indeed, all values have type Any whereas no value has type Nothing. We further provide a conversion rule that allows any expression to be widened to the Any type. This widening of expressions is equivalent to the boxing procedure described above and uses the conversion rules associated to the erased ? type.

$$\textsf{typed}(e, \tau, \textsf{Any})\ =\ \textsf{True} \qquad \textsf{typed}(e, \tau, \textsf{Nothing})\ =\ \textsf{False}$$

$$\textsf{convert}(e, \tau, \textsf{Any})\ =\ \textsf{convert}(e, \tau, ?)$$

Union and intersection types. The typing relation associated to an expected union or intersection type is handled through the relevant propositional connective between the underlying typing relations, leading to the following encoding rules.

$$\textsf{typed}(e, \tau, \tau_1 \cup \tau_2)\ =\ \textsf{typed}(e, \tau, \tau_1)\ \lor\ \textsf{typed}(e, \tau, \tau_2)$$

$$\textsf{typed}(e, \tau, \tau_1 \cap \tau_2)\ =\ \textsf{typed}(e, \tau, \tau_1)\ \land\ \textsf{typed}(e, \tau, \tau_2)$$

If the given expression type τ is a union type $\tau_1 \cup \tau_2$ (respectively intersection type $\tau_1 \cap \tau_2$), then we know by definition of the type encoding that e is a boxed expression, even when both τ_1 and τ_2 are native. In order to ensure that the typing relation encoding produces well-formed expressions, we therefore rely on the erased type ? to signify that we were given some boxed expression of unknown type. These considerations lead to the following encoding rules.

$$\mathsf{typed}(e, \tau_1 \cup \tau_2, \tau') = \mathsf{typed}(e, ?, \tau') \qquad \mathsf{typed}(e, \tau_1 \cap \tau_2, \tau') = \mathsf{typed}(e, ?, \tau')$$

We further introduce conversion rules that box and unbox the given expression when converting to and from a union or intersection type.

$$\mathsf{convert}(e, \tau_1 \cup \tau_2, \tau') = \mathsf{convert}(e, ?, \tau') \qquad \mathsf{convert}(e, \tau_1 \cap \tau_2, \tau') = \mathsf{convert}(e, ?, \tau')$$

$$\mathsf{convert}(e, \tau, \tau_1 \cup \tau_2) = \mathsf{convert}(e, \tau, ?) \qquad \mathsf{convert}(e, \tau, \tau_1 \cap \tau_2) = \mathsf{convert}(e, \tau, ?)$$

Extending the Fragment

Let us now discuss certain extensions to the supported Scala fragment and how the encoding procedures described above can be adapted to handle them. We describe here the addition of type parameter bounds, declaration-site variance, higher-kinded types, instance checks, casts and field accesses. We have not yet found a satisfactory encoding strategy for type members and therefore omit them from the discussion.

Type parameter bounds. Let us assume that each type parameter T has a lower bound τ_1 and an upper bound τ_2, denoted by $T >: \tau_1 <: \tau_2$. We can assume that all type parameters are annotated with such bounds as omitted lower, respectively upper bounds can be filled with the Nothing, respectively Any type. We extend our erased type ? to include bounds as well, denoted by $? >: \tau_1 <: \tau_2$. All type parameter erasures (replacing T by ?) that occur in our encoding and conversion rules will propagate the bounds into the erased type. If a bound was recursive (namely $T \sqsubseteq \tau_1$ or $T \sqsubseteq \tau_2$), we substitute the type parameter by the widened erased type $? >: \mathsf{Nothing} <: \mathsf{Any}$ within the bound to ensure the encoding terminates.

The conversion procedure is unaffected by the type parameter bounds. However, the typing relation encoding will need to check that the given expression satisfies the upper bound of an expected erased type, leading to the following rule.

$$\mathsf{typed}(e, \tau, ? :> \tau_1 <: \tau_2) = \mathsf{typed}(e, \tau, \tau_2)$$

The typing predicate parameters introduced during function definition encoding are also affected by the bounds as they no longer correspond to unconstrained propositional functions. Instead, we introduce a refinement on the predicate result type that encodes the type bounds constraint. The typing predicate parameter then has the following type

$$\{\, T : \mathsf{Object} \rightarrow \mathsf{Prop} \mid \forall x : \mathsf{Object}.\ \mathsf{typed}(x, ?, \tau_1) \implies (T\ x) \land (T\ x) \implies \mathsf{typed}(x, ?, \tau_2) \,\}$$

Note that this encoding allows for recursive type bounds where T occurs within τ_1 or τ_2. If the type bounds are not recursive, then we can avoid the quantifier by refining the Prop result type instead of the full function type.

Recall that our encoding procedure can preserve certain datatype and function type parameters. We have seen how type bounds can be handled for erased type parameters, and it is clear that we could simply erase all type parameters with non-trivial bounds in our encoding. However, the existence of non-trivial bounds alone does not require type parameter erasure. Indeed, bounds can be encoded into extra parameters $T_{:>} : \tau_1 \to T$ and $T_{<:} : T \to \tau_2$ which are then used in the conversion procedure. Similarly to the typing predicate parameters, these parameters can later be ignored in the model finding procedure.

Declaration-site variance. It turns out that our encoding procedure is (almost) entirely agnostic to declaration-site variance. Indeed, the relation between denotations in the verification language encoding is consistent with the Scala subtyping relation between traits and classes with well-formed type parameter variance annotations. In order to extend our procedure with support for declaration-site variance, it therefore suffices to take into account the variance annotations in the precise type $\mathsf{tpe}(\cdot)$ and typing lattice meet and join computations.

Although the denotation of encoded types is consistent with the Scala subtyping relation, our type checking procedure is not always able to show this. Consider for example the following encodings of the List[Any] and List[Boolean] types.

$$\begin{array}{rcl}\mathsf{List[Any]} & \to & \{\, x : \mathsf{Object} \mid \mathsf{isList}(x, \lambda x.\, \mathsf{True}) \,\} \\ \mathsf{List[Boolean]} & \to & \{\, x : \mathsf{Object} \mid \mathsf{isList}(x, \lambda x.\, x\ \textbf{match}\ \{\, \mathsf{Bool}(y) \Rightarrow \mathsf{True}\ \_ \Rightarrow \mathsf{False} \,\}) \,\}\end{array}$$

A type check of the form $x : \mathsf{encode}(\mathsf{List[Boolean]}) \vdash x \Downarrow \mathsf{encode}(\mathsf{List[Any]})$, which is simply given by the subtyping rule between nominal types in Scala, will need the model finding procedure to show that the following implication holds

$$\mathsf{isList}(x, \lambda x.\, x\ \textbf{match}\ \{\, \mathsf{Bool}(y) \Rightarrow \mathsf{True}\ \_ \Rightarrow \mathsf{False} \,\}) \implies \mathsf{isList}(x, \lambda x.\, \mathsf{True})$$

Such a proof cannot be automatically discharged by our system as it lacks an inductive invariant. However, the relevant inductive principle can be established based on the definition of the List type. We can therefore synthesize and encode the following recursive Scala function.

```
def asList[T₁, T₂ >: T₁](list: List[T₁]): List[T₂] = list match {
  case Cons(x, xs) ⇒ Cons[T₂](x, asList[T₁, T₂](xs))
  case Nil ⇒ Nil
}
```

This definition provides the necessary inductive principle to show subtyping between list types. The encoding of this definition can be verified by our algorithmic type checker. Let us now consider the derivation of the following type checking judgement

$$x : \mathsf{encode}(\mathsf{List[Boolean]}) \vdash \mathsf{encode}(\mathsf{asList[Boolean, Any]}(x)) \Downarrow \mathsf{encode}(\mathsf{List[Any]})$$

The derivation will only rely on the model finding procedure to automatically show that $x\ \textbf{match}\ \{\, \mathsf{Bool}(y) \Rightarrow \mathsf{True}\ \_ \Rightarrow \mathsf{False} \,\} \implies \mathsf{True}$ holds, namely the typing relation associated to Boolean implies the relation associated to Any. The type check is thus *lifted* into the type parameters similarly to how the Scala subtyping relation handles variance.

Conversion function can be synthesized based on the Scala trait and class definitions that feature declaration-site variance. During expression encoding, if the given type is subtype of the expected type and the subtyping relation relies on variance, we can then insert a call to the corresponding synthesized conversion function. This allows our type checking procedure to verify the encodings of programs that feature subtyping with variance.

Higher-kinded types. We saw that type parameters (with kind $*$) are encoded into typing predicates, namely expressions with type $\mathsf{Object} \to \mathsf{Prop}$. Higher-kinded type parameters naturally translate to higher-order function types. Hence, a type parameter with kind $* \to *$ will correspond to a typing predicate with type $(\mathsf{Object} \to \mathsf{Prop}) \to \mathsf{Object} \to \mathsf{Prop}$. More generally, each $*$ kind translates to the type $\mathsf{Object} \to \mathsf{Prop}$ in the kind's structure.

It is clear that the approach to higher-kinded types given above will rely on boxed higher-kinded type applications. Let us consider the higher-kinded type application $F[\tau']$. Our encoding of function definitions will ensure that we have a higher-order typing predicate F in our scope. The typing relation encoding is then given by the following rule.

$$\mathsf{typed}(e, \tau, F[\tau']) \;=\; (F\ \lambda x : \mathsf{Object}.\ \mathsf{typed}(x, ?, \tau'))\ e$$

Conversion between higher-kinded type applications is then given by the identity function.

Instance checks. Based on the typing relation encoding, we can further encode instance checks, namely $e.\mathsf{isInstanceOf}[\tau]$ expressions. The encoding depends on whether we are in a propositional context or not. Propositional instance checks, on which specifications can rely, are encoded through the typing relation encoding. Outside the propositional context, we only allow instance checks on erased (or compatible) types which are encoded into boolean-typed expressions based on the set of relevant datatype constructors. Note that this restriction in the non-propositional context is coherent with Scala's erased runtime semantics.

Type casts. Let us now consider type casts, namely $e.\mathsf{asInstanceOf}[\tau]$ expressions. Casts are encoded through the conversion procedure. In order to ensure that casts are *safe*, we introduce a type checking constraint which requires the typing relation encoding to hold between the expression (before conversion) and the expected target type.

Field accesses. Finally, field accesses can be encoded into match-expressions where the error term is inserted into all branches which do not correspond to the constructor associated to the accessed field. Note that this approach can also be applied to fields of specialized datatypes.

5.5 Imperative Trees

The next level in the tree hierarchy introduces imperative features such as mutable fields, mutable local variables, as well as while loops. In order to eliminate imperative features, the transformations rely on an effect system for variable and field mutation. The introduced imperative constructs and transformations into functional code are described extensively in [BKKS13, Bla17] and will be omitted here.

In addition to imperative features, these trees introduce a ghost effect system which allows users to specify portions of the program that will be erased at runtime. The effect system will ensure that definitions marked with @ghost can only influence the evaluation of other @ghost definitions. Our system provides a Scala compiler plugin which erases these ghost definitions in the generated bytecode.

5.6 Method Trees

In this section, we extend our trees with method definitions and calls, as well as open type hierarchies. We will present transformations which respectively eliminate method definitions and calls, unsealed traits, super calls to overridden methods, and finally two special specification constructs which we respectively call *class invariants* and *laws*. As each transformation eliminates a clearly defined set of trees, we will present the relevant language extensions when describing the encoding procedure.

5.6.1 Method Lifting

We start by describing the method lifting transformation which transforms methods within (sealed) trait and class definitions into top-level function definitions. We therefore extend the language with abstract and concrete method definitions, method calls, immutable and mutable abstract fields, as well as **this** expressions. These considerations lead to the following grammar extensions.

$$\begin{array}{rcl}
\textit{expr} & ::= & \cdots \mid \textit{expr}.\textit{id}[\ \textit{tparams}\ ](\ \textit{expr}\ ) \mid \textbf{this} \\
\textit{mdef} & ::= & \textit{fdef} \mid \langle\ \textit{fannot}\ \rangle^* \ \textbf{def}\ \ \textit{id}[\ \textit{tdecls}\ ](\ \textit{id} : \textit{type}\ ) : \textit{type} \\
 & & \mid \textbf{val}\ \textit{id} : \textit{type} \mid \textbf{var}\ \textit{id} : \textit{type} \\
\textit{tdef} & ::= & \cdots \mid \textbf{sealed trait}\ \textit{id}[\ \textit{tdecls}\ ]\ \textit{extensions tbody} \\
 & & \mid \textbf{final case class}\ \textit{id}[\ \textit{tdecls}\ ](\ \langle\ \textbf{var}\ \rangle?\ \textit{id} : \textit{type}\ )\ \textit{extensions tbody} \\
\textit{tbody} & ::= & \epsilon \mid \{\ \langle\ \textit{mdef}\ \rangle^*\ \}
\end{array}$$

Abstract (immutable and mutable) fields are treated analogously to abstract methods which must then be overridden by case class fields. In other words, they correspond to required state.

This behavior is a restriction on the Scala semantics of abstract fields, however we have found it useful to enable modular reasoning about expected state of abstract types.

As the trait definitions considered in this transformation are sealed, we assume here that all abstract methods and fields are correctly overridden by concrete definitions. We will see in the following sub-section how this invariant will be guaranteed by our system when faced with open type hierarchies with unsealed traits.

The method lifting transformation encodes methods into top-level functions by introducing an extra parameter for the method receiver and dispatching overrides based on the receiver type. For dispatch, we rely on instance checks and cast expressions as discussed in the context of type encoding. For example, consider the following List type definition with an abstract size method which is defined in the children case classes

```
sealed trait List[T]                              { def size: Int }
case class Cons[T](h: T, t: List[T]) extends List[T] { def size = 1 + t.size }
case class Nil[T]() extends List[T]               { def size = 0 }
```

Method lifting would remove the method definitions from the trait and class bodies and lift them into the following top-level definitions

```
def List-size[T](thiss: List[T]): Int =
  if (thiss.isInstanceOf[Cons[T]]) Cons-size[T](thiss.asInstanceOf[Cons[T]])
  else Nil-size[T](this.asInstanceOf[Nil[T]])

def Cons-size[T](thiss: Cons[T]): Int = 1 + List-size[T](thiss.t)

def Nil-size[T](thiss: Nil[T]): Int = 0
```

Note that we preserve the function definitions which were given in the source (as opposed to inlining the bodies of Cons-size and Nil-size in List-size). This allows us to both 1) leverage more precise static dispatches given by the Scala compiler, and 2) increase the modularity of reasoning in the model finding procedure by "hiding" the precise body of overriding function definitions behind further unfoldings.

When applying method lifting to a concrete (non-abstract) overridden method, we preserve the current function's body as an **else** branch in the dispatch. Note that if all descendant types define an overriding method, then the branch may become dead. In such cases, we safely prune the **else** branch in order to simplify the resulting program.

5.6.2 Trait Sealing

We now extend our language to support open type hierarchies. We introduce unsealed traits, final methods, as well as an @mutable annotation which indicates that some (open) a type hierarchy allows mutable state. These considerations lead to the following grammar extension.

$$\begin{aligned} \mathit{tdef} &::= \cdots \mid \langle\, \texttt{@mutable}\,\rangle?\ \textbf{trait}\ \mathit{id}[\ \mathit{tdecls}\ ]\ \mathit{extensions}\ \mathit{tbody} \\ \mathit{mdef} &::= \cdots \mid \langle\, \mathit{fannot}\,\rangle^*\ \textbf{final def}\ \ \mathit{id}[\ \mathit{tdecls}\ ](\ \mathit{id} : \mathit{type}\ ) : \mathit{type} := \mathit{fbody} \end{aligned}$$

Let us start by considering the meaning of the @mutable annotation. This annotation determines which open type hierarchies can allow mutable state. In other words, only extensions of unsealed trait definitions marked as @mutable are allowed to introduce mutable state. This implies that while unmarked unsealed traits may feature mutable state either through locally defined mutable fields or due to mutable fields inherited from ancestor traits, their descendants will not be allowed to introduce more mutable state. This allows the user to precisely specify the scope of mutation which may occur within potential method overrides. Consider for example the following (unmarked and unsealed) trait definition.

```
trait MutableBool {
  var b: Boolean
  def access(): Int
}
```

We know here that each MutableBool value features exactly two distinct states and calls to access (for a same instance of MutableBool) can only return at most two different values.

The trait sealing transformation seals open traits by generating a synthetic case class child for each unsealed trait definition. Consider some (unsealed) trait definition **trait** $t[\overline{\tau}_t]$ **extends** $\overline{\tau}_p$. Further consider the set of abstract (immutable or mutable) field definitions in t and its ancestors **val**/**var** $x_1 : \tau_1, \cdots,$ **val**/**var** $x_n : \tau_n$. We assume here that each type τ_i is adequately substituted with the relevant type parameter instantiations; hence, given value v with type $t[\overline{\tau}_t]$, the field access $v.x_i$ has type τ_i. Finally, consider a field **val**/**var** r : Nat that is mutable iff the trait was marked with the @mutable annotation. We then generate a case class definition with the following signature that represents unknown trait extensions.

final case class Open-$t[\overline{\tau}_t]$($\langle$**var**$\rangle$? r : Nat, $\langle$**var**$\rangle$? $x_1 : \tau_1, \cdots,$ $\langle$**var**$\rangle$? $x_n : \tau_n$) **extends** $t[\overline{\tau}_t]$

This case class definition represents unknown trait extensions. The fields $x_1, \cdots, x_n$ concretize the state which was required by parent traits. The field r : Nat ensures that the open type $t[\overline{\tau}]$ allows infinitely many distinct values. Note that if the trait was marked @mutable, then its children can introduce arbitrary state extensions. We therefore rely on the unbounded domain of the mutable field **var** r : Nat to encode this property.

The final step in the sealing procedure consists in introducing suitable method overrides. Unsealed trait extensions may introduce arbitrary overrides of non-final methods. Hence, we introduce method overrides in the Open-t class definitions for all non-final methods in ancestor traits. In order to allow arbitrary implementations, we treat these methods as uninterpreted and rely on a special non-deterministic construct for their body. It is important to note here that these special non-deterministic functions require dedicated support at the model finding level (calls are unblocked without introducing new clauses during unfolding).

5.6.3 Super Calls

We further extend our expression grammar with the **super** keyword which allows calling (overridden) methods of parent classes. As the method lifting transformation replaces the body of the overridden method with the dispatching expression, this transformation duplicates (relevant) concrete overridden methods. Super calls are then dispatched to the function's copy which is no longer considered overridden. In order to avoid having the trait sealing transformation introduce overrides for the synthetic super methods, we mark these as final.

For example, let us consider the following Scala type hierarchy involving a super call:

```
trait Parent { def method(x: Int): Int = 2 * x }
trait Child extends Parent { override def method(x: Int): Int = 2 * super.method(x) }
```

The following three function definitions will be generated by applying the super call and method lifting transformations. (Note that the trait sealing transformation would in addition introduce synthetic overrides for both Parent.method and Child.method.)

```
def Parent-method(thiss: Parent, x: Int): Int =
  if (thiss.isInstanceOf[Child]) Child-method(thiss.asInstanceOf[Child], x) else 2 * x

def Parent-super-method(thiss: Parent, x: Int): Int = 2 * x

def Child-method(thiss: Child, x: Int): Int = 2 * Parent-super-method(thiss, x)
```

5.6.4 Class Invariants

The method trees further introduce a special specification construct which we call *class invariants*. This construct allows users to specify invariants that will hold for all instances of a given trait or class type. Syntactically, a class invariant is specified in a trait or class body through a require statement with a Prop-typed argument.

$$tbody \ ::= \ \cdots \mid \{\, \mathsf{require}(\, expr \,)\ \langle\, mdef \,\rangle^{*}\, \}$$

The first encoding step consists in lifting the proposition into a parameterless method definition with signature **def** invariant() : Prop. This allows the method lifting transformation to dispatch invariants of descendant types. However, class invariants in descendant type definitions do not replace the ancestor invariant but are taken as a conjunction. Hence, given an invariant require(p), we generate the following invariant method.

$$\textbf{def}\ \mathsf{invariant}() : \mathsf{Prop} := \textbf{super}.\mathsf{invariant}()\ \wedge\ p$$

If the invariant was given in a root type definition, we omit the super call. In order to ensure that method lifting correctly dispatches the invariant methods, we generate invariant methods that simply return the True proposition in root type definitions if necessary (namely if the type definition does not feature a class invariant but has a descendant which does).

The second encoding step then consists in refining all occurrences of types which feature a class invariant (or for which an ancestor or descendant does) by the relevant invariant method call. In other words, the type $t[\overline{\tau}]$ will be replaced by $\{\, x : t[\overline{\tau}] \mid x.\mathsf{invariant}() \,\}$. As the method lifting transformation introduces new receiver parameters, relevant type occurrences may be added to the program at this stage. This second step is therefore partially performed by the method lifting transformation which will introduce a refined receiver parameter.

It is important to note here that since class invariants are lifted into method definitions, straightforwardly applying the above rule would result into lifted invariant methods with the following signature: **def** $\mathsf{invariant}(\mathsf{thiss} : \{\, x : A \mid \mathsf{invariant}(x) \,\}) : \mathsf{Prop}$. It is clear that our type checking procedure will fail to validate this definition due to the recursion in the parameter type. A similar issue appears for function or method definitions which are (transitively) called from the class invariant as they may introduce invalid mutual recursion between the invariant definition and injected refinements.

In order to avoid unsupported mutual recursion as a result of refinement injection, we skip definitions which are (transitively) called from the invariant (as well as the invariant method itself). It is important to note that this approach introduces a somewhat unexpected dependency between the specifications and the call-graph. However, we have found that this dependency is only rarely relevant in practice.

5.6.5 Laws

Before discussing *laws*, the final language construct introduced by the method trees, let us discuss the implications of refinements in abstract methods result types. Consider, for example, the following Monoid definition.

```
trait Monoid[T] {
  def unit: T
  def combine(x: T, y: T): T
}
```

In addition to providing concrete unit and combine definitions, a valid monoid implementation should satisfy the three monoid laws. In other words, given a monoid instance m and values a,b,c, the following properties must hold.

associativity : m.combine(a, m.combine(b, c)) ≈ m.combine(m.combine(a, b), c),
left identity : m.combine(m.unit, a) ≈ a,
right identity : m.combine(a, m.unit) ≈ a.

These properties can be specified in the Monoid type definition through the use of abstract methods with refined result types, leading to the following definition.

```
trait Monoid[T] {
  def unit: T
  def combine(x: T, y: T): T
  def associativity(x: T, y: T, z: T):
    {r : Unit | combine(x,combine(y,z)) ≈ combine(combine(x,y),z) }
  def left_identity(x: T): {r : Unit | combine(unit,x) ≈ x }
  def right_identity(x: T): {r : Unit | combine(x,unit) ≈ x }
}
```

One can then view the concrete implementation of these abstract methods as the *proof* that the property indeed holds for the given concrete monoid.

Laws allow users to specify such properties in a natural manner. A law is a (concrete) method definition in some trait marked with an @law annotation. Such methods correspond to syntactic sugar for defining abstract methods with refined result types. We allow laws to be either boolean- or Prop-typed and insert the necessary conversions during encoding. For example, the associativity method defined in the previous example can be given as the following law.

```
@law def associativity(x: T, y: T, z: T): Boolean =
  combine(x, combine(y, z)) ≈ combine(combine(x, y), z)
```

Law overrides correspond to *proofs* of the corresponding abstract law methods. The encoding of these proof methods ensures that the type checking procedure will check the implication between the given proof and the specified property. For example, consider the following proof definition of associativity in the context of a List monoid implementation with append.

```
final case class ListMonoid[T]() extends Monoid[List[T]] {
  ...
  override def associativity(x: List[T], y: List[T], z: List[T]): Boolean = x match {
    case Cons(l, ls) ⇒ associativity(ls, y, z)
    case Nil() ⇒ true
  }
}
```

The encoding procedure will transform this definition into the following method which will be successfully verified by the type checking procedure.

```
override def associativity(x: List[T], y: List[T], z: List[T]):
  {r : Unit | combine(x,combine(y,z)) ≈ combine(combine(x,y),z) } = x match {
  case Cons(l, ls) ⇒ let u = associativity(ls, y, z) in ()
  case Nil() ⇒ ()
}
```

Empirically, we have found that many law proofs turn out to be trivial in class implementations. In order to reduce the burden on the user and improve code readability, we allow proofs (*i.e.* law overrides) to be omitted and generate a default (trivial) proof in such cases.

5.7 Inner Class Trees

These trees introduce support for local type definitions and anonymous classes. Analogously to how inner functions are lifted into top-level definitions, inner classes (and traits) are lifted into top-level class (and trait) definitions. In order to preserve the context under which the type definition occurs, we introduce extra type parameters and (concrete or abstract) fields in the lifted definition. Type definitions may close over local immutable variables, however we disallow closure over local function definitions and var-bindings. If the local class (or trait) definition refers to methods or fields of an enclosing class, then we introduce an outer reference field to the lifted definition through which the method or field can be accessed.

It is important to note here that, as with first-class functions, the equality semantics given by this encoding do not line up with those of the Scala language. Indeed, Scala equality for inner classes ignores potential outer references. It is therefore important to ensure that the program does not rely on equality between values of such types.

5.8 Specification Trees

Although the type system presented in Chapter 3 relies on refinement types for specification, the Scala language features no such construct. The specification trees introduce Scala-compatible constructs which allow specifying properties of interest. These constructs are then desugared into the dependent types supported by our type checking procedure.

5.8.1 Contract Desugaring

We introduce contract-style specifications through require, ensuring and assert statements. The require statement corresponds to a precondition, ensuring to a postcondition, and assert to an assertion. These considerations lead to the following grammar extensions.

$$\begin{aligned} \mathit{fbody} &::= \cdots \mid \{\, \langle\, \mathsf{require}(\, \mathit{expr}\,);\ \rangle?\ \langle\, \mathsf{decreases}(\, \mathit{expr}\,);\ \rangle?\ \mathit{expr}\,\}\ \mathsf{ensuring}(\, \lambda \mathit{id}.\ \mathit{expr}\,) \\ \mathit{expr} &::= \cdots \mid \mathsf{assert}(\, \mathit{expr}\,)\ \textbf{in}\ \mathit{expr} \end{aligned}$$

We further provide a shorthand $\{\, e\,\}$.holds which corresponds to $\{\, e\,\}\ \mathsf{ensuring}(\lambda b.\ b)$. These specification statements constitute syntactic sugar over refinement types. Let us first discuss the transformation of require and ensuring statements. The require statement introduces a refinement on the parameter type, and the $\mathsf{ensuring}(\lambda r.\ p_2)$ defines a refinement on the result type resulting in the following definition encoding.

$$\begin{aligned} &\textbf{def}\ f[\overline{\tau}_f](x:\tau_1):\tau_2\ :=\ \{\,\mathsf{require}(p_1);\ e_f\,\}\ \mathsf{ensuring}(\lambda r.\ p_2)\ \rightsquigarrow \\ &\qquad\qquad\qquad \textbf{def}\ f[\overline{\tau}_f](x:\{x:\tau_1 \mid p_1\}):\{r:\tau_2 \mid p_2\}\ :=\ e_f \end{aligned}$$

Assertions allow us to check a proposition and then introduce it into the typing context. Given the type annotated let-expression discussed in Chapter 3, the assert statement is therefore encoded as follows.

$$\mathsf{assert}(p)\ \textbf{in}\ e\ \rightsquigarrow\ \textbf{let}\ u:\{u':\mathsf{Unit} \mid p\}\ :=\ ()\ \textbf{in}\ e$$

5.8.2 Partial Functions

Thanks to propositional quantifiers, the contracts described above allow specifying certain properties about first-class functions. However, these constructs do not allow us to specify partial functions, namely pi-types where the input domain is restricted. We introduce a special type $\tau_1 \mapsto \tau_2$ which behaves similarly to a function type of the form $\{x:\tau_1 \mid p\} \to \tau_2$.

The $\tau_1 \mapsto \tau_2$ type is syntactic sugar for a $\mathsf{PartialFunction}[\tau_1,\tau_2]$ class type. The corresponding case class definition features two fields, namely a *pre* field which specifies the domain over which the partial function is defined, and an f field which defines the partial function.

$$\textbf{final case class}\ \mathsf{PartialFunction}[A,B](\mathit{pre}: A \to \mathsf{Prop},\ f:\{x:A \mid \mathit{pre}\ x\} \to B)$$

This class definition allows us to specify the domain over which a first-class function must be defined. For example, the parameter $g:\{x:\tau_1 \mid p\} \to \tau_2$ can be specified by combining the declaration $g:\tau_1 \mapsto \tau_2$ and precondition $\mathsf{require}(\forall x:\tau_1.\ (p\ x) \implies (g.\mathit{pre}\ x))$.

In order to construct instances of type $\tau_1 \mapsto \tau_2$ in a natural manner, we introduce the following lambda syntax $\lambda x:\tau_1.\ \{\,\mathsf{require}(p);\ e\,\}$ which is then desugared into the partial function $\mathsf{PartialFunction}[\tau_1,\tau_2](\lambda x:\tau_1.\ p, \lambda x:\{x:\tau_1 \mid p\ x\}.\ e)$. We similarly desugar Scala partial functions and lambdas whose body is a call to some named function with a precondition. (Note that we rely on a Scala implicit conversion in order to convert lambdas which have type $\tau_1 \to \tau_2$ into expressions with type $\tau_1 \mapsto \tau_2$ in the Scala type checker.)

6 The Stainless Verification System

In this chapter, we present our verification system based on the model finding, type checking and encoding procedures presented in Chapters 1 to 5. Our system is split into two main components, namely 1) **Inox**, a verification condition solver based on the model finding procedure, and 2) **Stainless**, a program verifier which encodes and type checks Scala programs by relying on Inox to handle verification conditions. Both projects are implemented in Scala. Inox and Stainless were developed as a redesign and extension of the Leon verification system. We will describe the functionality and interface of each component and present certain important design decisions.

In order to showcase the practicality of our system, we will then present a series of verification benchmarks. These benchmarks range over functional datastructure definitions, higher-order collection APIs, algebraic properties and mathematical truths. The benchmark set makes use of both object-oriented and functional features of the Scala language and explores different approaches to verification supported by our tools. Our total benchmark set, including regression tests, comprises over 15K lines of Scala code and verifies in under 10 minutes.

6.1 Inox Solver

The model finding procedures presented in Chapters 1 through 4 are implemented in the Inox solver. The solver is developed as a library whose main task is to discharge verification conditions. Inox features a programmatic API (which Stainless uses), a TIP [CJRS15] frontend for the fragment without dependent types, and a frontend which performs Hindley-Milner type inference on the verification language. Inox features multiple SMT solver backends such as Z3 [dMB08] (also through a native interface [KKS11]), CVC4 [BCD^{+}11] and Princess [Rüm08] (which gives us an option to have a JVM-only software stack). Solving can also be performed in portfolio mode in order to fully leverage the strengths of each SMT solver.

The Scala file and line counts for the Inox tool broken down by directory are given in Figure 6.1. The embeddings and unfolding procedure implementations (Section 6.1.1) reside in the

Directory	Files	LoC
`ast`	19	6.4K
`evaluators`	6	1K
`parser`	42	4.9K
`solvers`	51	11K
`tip`	6	1K
`transformers`	10	1.3K
`utils`	27	2.7K
Tests	37	5.1K
Total	198	33.4K

Figure 6.1 – Scala file and line count statistics for Inox broken down by (main) directory. These statistics omit documentation and regression tests given in the TIP format [CJRS15].

`solvers` directory, and the abstract syntax trees (Section 6.1.3) are defined in `ast`. It is clear given the size of the directories that these features form the main complexity of the Inox tool. The third large component defined in the `parser` directory is the Hindley-Milner based frontend which includes an elaboration procedure and typing constraint solver.

6.1.1 Proofs and Counterexamples

Inox relies on the unfolding procedure to perform both counterexample and proof search. At each unfolding step i, we check satisfiability of Φ_i. If the clause set is unsatisfiable, then we have found a proof. Otherwise, we check satisfiability of $\Phi_i \cup \mathsf{block}_i \cup \mathsf{model}_i$ (recall block_i and model_i from Chapter 4). If a model is found, then the procedure terminates. Otherwise, we proceed with the next unfolding step. A visual representation of this process can be found in Figure 6.2. The procedure can be viewed as alternatively under- and over-approximating the propositional truth relation and each unfolding step incrementally refines the approximations.

$$\begin{tikzcd}
\Phi_0 \arrow[rr] \arrow[dd] \arrow[dr] & & \Phi_1 \arrow[rr] \arrow[dd] \arrow[dr] & & \Phi_2 \arrow[rr] \arrow[dd] \arrow[dr] & & \cdots \\
& \Phi_0 \cup \mathsf{b}_0 \arrow[d] & & \Phi_1 \cup \mathsf{b}_1 \arrow[d] & & \Phi_2 \cup \mathsf{b}_2 \arrow[d] & \\
\text{Unsat?} & \text{Sat?} & \text{Unsat?} & \text{Sat?} & \text{Unsat?} & \text{Sat?} &
\end{tikzcd}$$

Figure 6.2 – A sequence of alternating satisfiability checks in the unfolding procedure where b_i corresponds to the clause set $\mathsf{block}_i \cup \mathsf{model}_i$.

The unfolding procedure described in the first half of the thesis computes embeddings (with freshened variables) at each unfolding step (Sections 1.4, 2.3 and 3.4). Function and lambda bodies will therefore often be embedded multiple times during unfolding. Our implementation performs staged embedding by computing a unique embedding for each (typed) function and lambda definition and then freshening constants in the resulting SMT term and clause set. Inox further implements the incremental lambda embedding strategy described in 2.6.1 and the lambda tracking optimization discussed in 2.6.2.

Inox features an interpreter which implements the operational semantics of our language. It further supports an (incomplete) checker for ground propositional truth relations. This checker relies on the interpreter for True(·) propositions and invokes the model finding procedure when dealing with quantifiers. Inox finally features a denotation checker that verifies whether a value belongs to some (ground) type. The denotation checker encodes the reducibility relation into some (ground) proposition and then relies on the propositional truth checker. While the propositional truth and denotation checkers are incomplete, they are both sound.

The propositional truth and denotation checkers allows the solver to validate potential counterexamples and mark spurious invalid (or unverified) counterexamples when given verification conditions in fragments where counterexample finding is unsound. Furthermore, given the alternating procedure described above, in the case where Φ_i is satisfiable (and therefore the proof check failed), then the SMT solver can report a satisfying model. Although this model is not guaranteed to constitute a valid counterexample, we can apply the validation procedure to potentially terminate early. However, as validation can sometimes be an expensive process, this approach is not enabled by default.

6.1.2 Additional Theories

The Inox input language extends the verification language with certain constructs which correspond to SMT theory symbols. These allow users to leverage the theory solvers in the underlying SMT solvers and improve automation in the model finding procedure (in comparison with equivalent recursive definitions).

Each theory is associated to a (possibly parametric) type in the extended verification language. For each such type, the language then features a set of native operations which are embedded into theory symbols at the SMT level. The set of types (and associated operations) for which Inox provides specific support is given as follows.

BigInt : This type corresponds to the theory of integers with linear arithmetic. We further allow non-linear arithmetic when relying on compatible SMT solver backends (*e.g.* Z3).

$\mathsf{Int}\langle 0-9\rangle+$: This family of types corresponds to the (size parametric) bitvector theory.

String : This type corresponds to the theory of character strings with length, substring and concatenation operations. The SMT solver backends support the theory of 8-bit character strings whereas we are interested in the JVM 16-bit character strings for Scala compliance. Our embedding and extraction procedures therefore encode and decode strings literals and operations such that the 16-bit semantics are guaranteed.

$\mathsf{Set}[\tau]$: This type corresponds to the parametric theory of finite sets.

$\mathsf{Bag}[\tau]$: This type corresponds to the parametric theory of finite multisets.

$\mathsf{Map}[\tau_1, \tau_2]$: This type corresponds to the parametric theory of maps with finite domain, namely the theory of SMT arrays.

Note that not all SMT solver backends support this set of theories. For example, only Z3 features support for multisets. We introduce a set of *theory encoders* which translate theory symbols into functions and datatypes annotated with useful specifications for proof derivation. These encodings significantly reduce the automation provided by the SMT solver, however counterexample finding typically remains tractable.

It is important to note here that since sets, multisets and maps can contain other values, the reducibility relation embedding and unfolding procedure presented in Chapter 3 are extended to handle these types as well.

6.1.3 Extensible Tree Definitions

A common approach when dealing with programs in the context of compilers, static analyzers or verifiers is to introduce some abstract syntax tree definition which represents the considered types, expressions and definitions. Based on these trees, various (higher-order) transformation operations are implemented such as pre- and post-traversals, folds, etc. Inox provides extensible tree definitions which are designed for type-safe transformations between different tree variants.

The abstract syntax trees are defined as inner types within a `Trees` trait, as shown in Figure 6.3. The `Expr`, `Type` and `Definition` traits are left unsealed in order to allow extensions of the `Trees` type to introduce new AST nodes. Nodes therefore have path-dependent types prefixed by some instance of the `Trees` trait. For each new tree definition, we define a corresponding tree deconstructor (see Figure 6.4) which allows decomposing and reconstructing nodes across different tree definitions. This deconstructor then enables us to define a generic transformer trait (Figure 6.5) from any tree definition to another. Programs are represented through a `Symbols` type which contains mappings from identifiers to function and type definitions. The `Symbols` type is also defined within the `Trees` trait and has access to the path-dependent AST node definitions. These APIs allow us to define the tree hierarchy described in Chapter 5 with the Inox trees as the root verification language.

The abstract syntax tree and program definitions are immutable. This allows our system to behave predictably in the presence of both encoding and decoding transformations. Furthermore, this property enables safe concurrent tree manipulations. The immutability of trees implies that transformations must perform tree copying operations which come at the price of lower performance. However, we have found that the cost associated to tree copying in the Stainless encoding pipeline is negligible in comparison to the time spent performing satisfiability checks in the underlying SMT solvers.

6.1.4 Simplifier

Inox features a simplification procedure that is aimed at improving counterexample (or proof) finding performance. The main goal of the procedure is therefore to eliminate function calls

```
trait Trees {
  trait Expr
  case class Let(id: Identifier, expr: Expr, body: Expr) extends Expr
  ...

  trait Type
  case class RefinementType(id: Identifier, tpe: Type, pred: Expr) extends Type
  ...

  protected def deconstructor(other: Trees): TreeDeconstructor {
    val s: Trees.this.type
    val t: other.type
  }
}
```

Figure 6.3 – (Partial) Inox trees definition API. In extensions of the Trees trait, the deconstructor method must generate a TreeDeconstructor which is defined for the lower bound between the current tree definition and the given one.

```
trait TreeDeconstructor {
  protected val s: Trees
  protected val t: Trees

  def deconstruct(expr: s.Expr): (s.ExprComponents, (t.ExprComponents) ⇒ t.Expr)
  def deconstruct(tpe: s.Type): (s.TypeComponents, (t.TypeComponents) ⇒ t.Type)
}
```

Figure 6.4 – Inox tree deconstruction. *ExprComponents* and *TypeComponents* correspond to the constitutent identifiers, expressions, types, and modifiers of the given expression or type prefixed with either s or t. Hence, the deconstruct methods allow us to decompose and reconstruct AST nodes while passing from one tree definition to another.

```
trait TreeTransformer {
  protected val s: Trees
  protected val t: Trees

  def transform(expr: s.Expr): t.Expr
  def transform(tpe: s.Type): t.Type

}
```

Figure 6.5 – (Partial) Inox Tree transformation API. As the input and output type prefixes are distinct, the transformation function signatures ensure that all children of the input AST node must be correctly transformed (should the transformed node depend on them).

and applications from the considered property (and program) in order to reduce the exponential blowup during unfolding. However, we also want to ensure that no relevant information is lost during simplification to avoid unexpected proof failures. The main transformations we apply are let-expression elimination and call merging.

Let us first discuss let-expression elimination in the context of some expression **let** $x := e_1$ **in** e_2. If $x \notin FV(e_2)$ and we can establish that the embedding of e_1, as well as the entailed unfoldings, will not impact satisfiability of the Φ_i clause sets, then we can replace the let-expression with e_2. Note that this applies both in the case of type annotated and unannotated let-expressions. The satisfiability preservation check is performed by determining whether evaluation of e_1 may encounter an error term, a function application, or a function call with refined result type. (Recall that the pi-type reducibility relation will be unfolded for applications and typing derivations associated to function calls are assumed by the unfolding procedure.)

Call merging is performed when (type compatible) calls to a same function appear in multiple branches of an if- or match-expression. In such cases, the call is lifted outside the branching expression and the if- or match-expression is pushed down into the call argument. For example, the if-expression **if** (c) $f[\overline{\tau}](e_1)$ **else** $f[\overline{\tau}](e_2)$ would be simplified into the single call expression $f[\overline{\tau}]($**if** (c) e_1 **else** $e_2)$. This transformation significantly reduces the exponential blowup of unfoldings when calls occurring in a recursive function definition can be merged.

6.2 Stainless Verifier

The type checking and encoding procedures presented in Chapters 3 and 5 are implemented in the Stainless verifier. Stainless relies on either the Scala 2.12 or Scala 3 compilers to parse and type check (according to Scala's type system) the input programs, then encodes them into the verification language supported by Inox, and finally algorithmically verifies the encoded programs by discharging verification conditions through Inox.

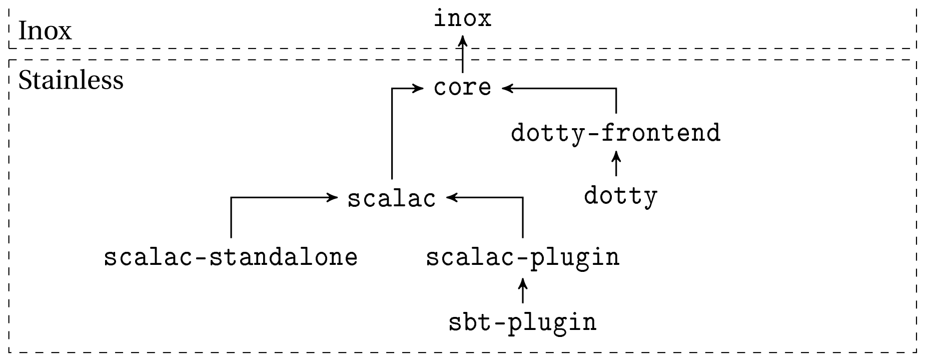

Figure 6.6 – Dependenty graph between Stainless sub-projects.

The Stainless system is split into multiple sub-projects as shown in Figure 6.6. The `core` project defines the encoding and type checking procedures. The `scalac` project uses the Scala

2.11.12 compiler as a frontend, while the `dotty-frontend` and `dotty` projects rely on the Dotty (Scala 3) compiler. Stainless is designed as a compiler phase, and the `scalac-plugin` and `sbt-plugin` projects provide an `sbt` plugin which allows easy integration of Stainless into Scala projects. Finally, the `scalac-standalone` project allows Stainless to be built as a self-contained jar for simpler installation and prototyping. The Stainless implementation features around 30K lines of Scala code, out of which over 20K reside in the `core` project.

6.2.1 Parallelism and Caching

In order to enable parallelism in our verification system, we consider each type or function definition in the original Scala program and generate a sub-program that contains its minimal set of (transitive) dependencies. We can then parallelize both the encoding and verification procedures at the level of definitions in the original Scala program by considering each sub-program independently. Note that we therefore only verify the definition(s) that correspond to the original Scala definition in the encoded sub-program (and not the full sub-program). As each definition in the original Scala program is verified (in the context of its own sub-program), all definitions in each sub-program end up being verified. It is important to note that certain transformations will introduce new definitions when encoding the original Scala definition. We consider these definitions to be *derived* and they must be verified as well.

When verifying a definition in the original Scala program, the associated sub-program must be entirely encoded. As programs are often structured in a modular manner where smaller building blocks are assembled into more complex features, many sub-programs will contain shared definitions. Most encoding transformations are compositional and rely only on the dependencies of a given definition. Hence, the encoding of shared definitions will often lead to duplicated effort. We alleviate this issue by introducing lock-free caches in key points of the transformation pipeline that allow sharing transformation results of shared definitions. We rely on precise and minimal cache key computations tailored to each transformation in order to maximize the cache hit rate. Note that these caches are not meant to be persisted and therefore do not require serialization support.

Stainless features persistent caching both at the level of extracted sub-programs and verification conditions. These caches rely on a canonization operation which normalizes the variable names in the program or verification condition, as well as a serialization procedure which allows the cache keys to be persisted. In order to reduce the cache size, we only store programs, respectively verification conditions, for which type checking, respectively solving, was successful. This allows us to reduce the size of cached results and avoid keys which will have low hit rates once the faulty snippet has been fixed.

6.2.2 Verification-Aware Libraries

Stainless provides a set of libraries containing useful type and function definitions. These include collections (see Appendix A for the List type definition), common math operations, and utilities which empower verification. The libraries are designed such that both the verification and unfolding procedures can effectively handle the provided definitions.

```
@inline implicit def any2EqProof[A](x: ⇒ A): EqProof[A] = EqProof(() ⇒ x, () ⇒ x)
case class EqEvidence[A](x: () ⇒ A, y: () ⇒ A, evidence: () ⇒ Boolean) {
  require(x() == y() && evidence())
  @inline def |(that: EqProof[A]): EqProof[A] = {
    require(evidence() =⇒ (y() == that.x()))
    EqProof(x, that.y)
  }
  @inline def |(that: EqEvidence[A]): EqEvidence[A] = {
    require(evidence() =⇒ (y() == that.x()))
    EqEvidence(x, that.y, that.evidence)
  }
}
case class EqProof[A](x: () ⇒ A, y: () ⇒ A) {
  require(x() == y())
  @inline def ==|(proof: ⇒ Boolean): EqEvidence[A] = {
    require(proof)
    EqEvidence(x, y, () ⇒ proof)
  }
  def qed: Boolean = (x() == y()).holds
}
```

Figure 6.7 – Stainless equational reasoning library designed for effective simplifications.

Consider for example the equational reasoning library given in Figure 6.7. This library allows users to write equational proofs of the following form

$(\mathsf{a} \mathrel{==|} \mathsf{proof}_{\mathsf{a==b}} \mid \mathsf{b} \mathrel{==|} \mathsf{proof}_{\mathsf{b==c}} \mid \mathsf{c}).\mathsf{qed}$

where $\mathsf{proof}_{\mathsf{a==b}}$ constitutes *evidence* that a is indeed equal to b (and likewise for $\mathsf{proof}_{\mathsf{b==c}}$). The library is constructed in such a way that each equation is verified independently, and the Inox simplifier then collapses the sequence of equations into a single end-to-end equation. This allows scalable equational reasoning by ensuring the typing context is not polluted with irrelevant evidence expressions.

6.3 Evaluation

We have evaluated the Stainless verification system on a set of verification tasks. These tasks include verifying functional datastructure definitions, sorting algorithms, interpreters and higher-order collection APIs. We have further used Stainless to prove certain categorical laws and mathematical truths. We summarize these tasks in Figure 6.8. The table displays the size of given programs and time spent during type checking. In order to showcase the effectiveness of the measure inference transformation, we further provide the number (and type) of inferred measures and the time spent during measure inference.

It is important to note here that the implementation of Stainless and Inox do not (yet) fully reflect the procedures presented in this thesis. For example, the type checking procedure is currently implemented as a simple forward pass through function bodies which generates verification conditions for key program constructs. However, we expect the procedures to be comparable on the presented benchmarks.

We now discuss certain selected benchmarks from Figure 6.8.

`GodelNumbering` : In this benchmark, we show that the pairing function $2^x(2y+1)-1$ constitutes a bijection between the natural numbers and pairs of natural numbers. We define the natural numbers through a Nat datatype and prove a series of lemmas about linear and non-linear arithmetic. This benchmark makes extensive use of the equational reasoning library described above. The full benchmark is provided in Appendix B.

`EuclidGCD` and `EuclidEGCD` : We show in `EuclidGCD` that Euclid's algorithm computes the greatest common divisor, and in `EuclidEGCD` we show that applying Euclid's extended algorithm on the natural numbers $a, b \in \mathbb{N}$ gives us a triplet $(r, x, y) \in \mathbb{N}^3$ such that $ax + by = r$ and r divides both a and b. Both benchmarks require proving some extra lemmas about non-linear division and remainders.

`ListMonad` and `OptionMonad` : In these benchmarks, we proved that the monadic laws hold for the List and Option monads. It is interesting to note that In order to prove the flatMap associativity property for the List type, we must perform two separate inductions. This leads to the property statement and proof given in Figure 6.9.

`ConcRope` and `ConcTree` : These benchmarks model operations from the Scala data parallel library [PO15] and were first presented in [MKK17] where termination was assumed.

In addition to our case studies and benchmarks, Stainless has been applied to smart contract verification [JH] and modeling of actor systems [Rue18]. These projects extend the Stainless input language with additional domain-specific constructs and introduce new transformations which encode them into the dependently-typed verification language. Inox has further been used (independently of Stainless) as a backend for experimental integration of refinement types in the Scala type checker [SK16] and as a default tactic for theorem proving [Ede].

Operation	LoC	Funs	Rec Funs	Measures	Dec.	Infer (s)	Check (s)
`Ackermann`	10	1	1	1 lex	0	1.0	0.6
`AliasPartial`	16	4	1	1 sum	0	0.4	0.0
`AmortizedQueue`	125	15	4	4 arg	0	0.2	3.5
`AnyDown`	21	1	1	1 lex	0	0.8	0.0
`AssociativeFold`	104	10	7	7 arg	0	0.1	1.3
`AssociativeList`	51	6	5	5 arg	0	0.1	0.7
`BalancedParentheses`	388	38	12	12 arg	0	9.1	14.0
`BinarySearchTrees`	79	8	5	5 arg	0	0.9	3.0
`ChurchNum`	21	5	0	-		-	0.0
`ConcRope`	468	30	13	1 lex/3 sum/9 arg	0	159.7	27.9
`ConcTree`	320	24	12	2 sum/10 arg	0	38.2	12.2
`ConstantPropagation`	268	17	10	3 sum/7 arg	0	41.5	3.9
`CountTowardsZero`	16	1	1	1 sum	0	0.3	0.5
`EuclidEGCD`	82	9	2	1 arg	1	8.9	2.6
`EuclidGCD`	203	23	2	1 arg	1	4.0	7.7
`GodelNumbering`	420	46	25	2 sum/20 arg	3	14.1	24.6
`IndirectHO`	16	2	1	1 sum	0	0.4	0.0
`IndirectIntro`	24	3	2	1 lex/1 sum	0	1.5	0.4
`Indirect`	20	3	1	1 sum	0	0.5	0.0
`InsertionSort`	69	7	6	1 sum/5 arg	0	0.7	0.9
`Knapsack`	71	5	4	1 sum/2 arg	1	1.4	0.5
`LeftPad`	90	9	4	2 sum	2	5.2	4.4
`ListMonad`	81	12	5	1 lex/4 arg	0	3.5	1.2
`ListOperations`	104	17	9	1 lex/8 arg	0	6.5	1.1
`List`	887	95	63	4 sum/59 arg	0	4.9	10.6
`McCarthy91`	24	2	1	0	1	-	0.7
`MergeSorts`	199	21	17	4 sum/12 arg	1	9.4	4.3
`OddEvenComplex`	26	3	2	0	2	-	0.8
`OddEvenMoreComplex`	20	2	2	0	2	-	0.7
`OddEven`	34	4	4	4 sum	0	0.5	0.4
`OptionMonad`	49	9	0	-		-	0.7
`QuickSorts`	219	25	15	5 sum/8 arg	2	39.5	10.8
`ReachabilityChecker`	502	29	15	1 sum/7 arg	6	38.0	25.0
`RedBlackTree`	99	11	6	6 arg	0	2.1	3.7
`StableSort`	115	11	6	1 lex/5 arg	0	20.7	2.0
`ToChurch`	26	5	1	1 sum	0	0.5	0.2
`UpDown`	47	4	2	2 sum	0	0.4	0.4
`XPlus2N`	18	4	1	1 sum	0	0.6	0.3
Total	5332	521	268	245	22	415.6	171.6
Valid Regression Tests	13015					270.0	256.0
Invalid Regression Tests	3063					-	42.0

Figure 6.8 – Summary of Stainless evaluation results. For each benchmark, we give the total lines of code (LoC), the number of function definitions (Funs) and how many are recursive (Rec Funs), the number and type of inferred measures, the number of user-provided decreases statements (Dec.), and the respective measure inference and verification times (in seconds). The inferred measures are either [lex]icographic, a [sum] of argument sizes, or a specific [arg]ument. Note that the valid regression tests line count includes certain benchmarks from the list above. Further note that the invalid regression tests ensure that verification does indeed fail when attempting to prove invalid properties.

```
def associativity[T,U,V](list: List[T], f: T ⇒ List[U], g: U ⇒ List[V]): Unit = {
  associativity_induct(list, Nil(), Nil(), f, g)
} ensuring { _ ⇒
  flatMap(flatMap(list, f), g) ≈ flatMap(list, (x: T) ⇒ flatMap(f(x), g))
}

def associativity_induct[T,U,V](
  list: List[T], flist: List[U], glist: List[V], f: T ⇒ List[U], g: U ⇒ List[V]): Unit = {
    glist match {
      case Cons(ghead, gtail) ⇒ associativity_induct(list, flist, gtail, f, g)
      case Nil() ⇒ flist match {
        case Cons(fhead, ftail) ⇒ associativity_induct(list, ftail, g(fhead), f, g)
        case Nil() ⇒ list match {
          case Cons(head, tail) ⇒ associativity_induct(tail, f(head), Nil(), f, g)
          case Nil() ⇒ ()
        }
      }
    }
  } ensuring { _ ⇒
    append(glist, flatMap(append(flist, flatMap(list, f)), g)) ≈
      append(append(glist, flatMap(flist, g)), flatMap(list, (x: T) ⇒ flatMap(f(x), g)))
  }
```

Figure 6.9 – Proof of flatMap associativity for the List monad. The associativity_induct function definition provides the inductive argument to the (generalized) associativity property.

7 Related Work

In this chapter, we will review research endeavours that are related to this thesis. This discussion will follow the general structure of the thesis. We will start by discussing counterexample finding and proof automation for first-order languages. We will then extend this discussion to include higher-order functions as well as dependent types. In this context, we will present related dependent type systems and discuss relevant approaches to supporting recursive and sized types. We will then compare our quantifier instantiation and model finding technique to alternative approaches, as well as discuss quantifier support in other verification systems. Finally, we will consider the approach of verification through encoding into a verifiable language and present related automated termination checking techniques.

Before discussing competing approaches, it is important to position our work in Stainless with respect to its precursor Leon [SKK11, BKKS13]. A monomorphic version of the first-order unfolding procedure presented in Chapter 1 was implemented in the Leon verification system which further featured support for inner functions, method lifting, imperative code elimination, as well as simple termination checking. This thesis extends the Leon approach with support for generic polymorphism, higher-order functions, dependent types, powerful measure inference, as well as various advanced Scala language features. Moreover, we provide proofs of long-standing theoretical claims associated to the approach.

First-order languages. Several industrial-grade verification systems for first-order languages, such as ACL2 [CDMV11] and Spec# [LM08], have been developed over the years and successfully applied to real-world case studies [KMM13]. ACL2 features some support for counterexample finding [CM11, Man13], however the approach is incomplete and ACL2's Common Lisp input language renders the question of polymorphism unapplicable. Spec# relies on the Boogie verification condition generator [BCD$^+$05] which supports generic polymorphism as well as an incomplete approach to counterexample generation and exploration [LLM11]. VeriFun [WS03] is another first-order verifier which supports polymorphic recursive functions [WAS06] and features fast but incomplete disproving capabilities [AWSS06].

Another important first-order language is the language of SMT itself. The SMTLIB standard has recently introduced support for recursive function definitions [BFT17], and a variant of the named function unfolding procedure based on [SKK11] has been added to the Z3 solver [Bjø]. Moreover, terminating recursive functions can be encoded by relying on first-order universal quantifiers in cases where the solver does not (yet) support recursive function definitions. Specific work in this direction has been implemented in the CVC4 solver [RBCT16], and model finding in the presence of universal quantification has further gained some traction recently [GdM09, RTB17]. SMT solvers generally don't provide support for parametric polymorphism, however techniques based on monomorphisation of polymorphic problems have been successfully applied in this context [BDT14, BBPS16]. Furthermore, there exist theoretical foundations for SMT solvers that support parametric polymorphism [KGGT07]. Nevertheless, none of the techniques and approaches described in this paragraph provide completeness of model finding in the presence of recursive function encodings through universal quantifiers (aside from the approach in Z3 which is analogous to our own).

Our approach to recursive function unfolding is similar in essence to loop unfolding, a technique that has been explored in the context of model checking for imperative programs in tools such as CBMC [CKL04] or F-SOFT [IYG+08]. The CORALL solver [LQ13] applies these techniques to show the unreachability of errors similarly to how our unfolding procedure can show counterexample inexistence. Model finding is also performed in the context of specification refinement [TJ07, Tag08] where non-spurious counterexamples can be immediately reported as disproving the property of interest. Due to the imperative nature of the supported input languages, these procedures are often more tricky and complex than our rather direct embedding into SMT.

Finally, it is worth mentioning the Haskell Bounded Model Checker [CR] which features a likewise complete counterexample finding procedure for monomorphic first-order haskell programs which combines symbolic execution with sat-based solving. This approach has proved effective in finding counterexamples for programs with large branching factors.

Higher-order functions. The procedure discussed in Chapter 2 extends our work presented in [VKK15] with support for first-class functions which appear within datatypes, as well as extraction of inputs with recursive functions. We will focus here on approaches that target higher-order functions in languages which do not feature dependent types as these will be treated separately below.

The Dafny [HLQ11] programming language provides some support for higher-order functions through translation into first-order verification conditions. These are then discharged by Boogie [Lei08], which provides support for (incomplete) counterexample generation and exploration [LLM11]. It is however unclear whether the Boogie counterexample corresponds to valid higher-order inputs at the Dafny level. Shape analysis of symbolic execution traces has also proved effective in verifying higher-order functional programs, and even feature relatively complete counterexample finding [NH15, NTH17]. It is important to note that the relative completeness guarantee here is given with respect to a first-order solver and is weaker than our completeness for counterexamples.

Another important class of higher-order languages consists of logics that have been augmented with some form of higher-order functions, for example (Classical) Higher-Order Logic. The main automated theorem provers for HOL are Satallax [Bro12], LEO-II [BS13] and the more recent Leo-III [SB18]. Satallax relies on a tableau calculus with an underlying SAT solver and first-class function enumeration, whereas the Leo provers leverage a variety of techniques in portfolio or collaborative mode. Note that these theorem provers can also be used for model finding by negating the input formula and extracting the assignments for existentially quantified variables. Recent work has also shown promising results in (partially) bridging the gap between First-Order and Higher-Order Logic in both automated theorem provers based on the superposition calculus [BBCW18, BR18] and SMT solvers [BRO$^+$19].

The difficulty of detecting invalid properties in both interactive and automated HOL theorem provers has led to the development of model finders for HOL. These model finders are based on logical semantics (as opposed to our operational semantics), yet these semantics will sometimes coincide, for example in the fragment of total functions over inductive datatypes. Two well known examples of such model finders are Nitpick [BN10] and its successor Nunchaku [CB16] which rely on symbolic model enumeration and encoding into First-Order Logic to construct valid models for HOL formulas.

Finally, other logics featuring polymorphism and higher-order functions have been handled through encodings into first-order SMT clauses [BDT14, CJRS15]. However, these approaches often rely on a program monomorphisation (which does not always exist), and are aimed at theorem proving, thus relying on an under-approximation which can lead to spurious counterexamples.

Automation for dependent types. Proof automation in the presence of dependent types has been explored in the past through encodings into First-Order Logic [TS95]. This topic is making a comeback in recent years with the rise of automated theorem provers [Cza16], and a hammer for Coq based on an unsound yet practical encoding is showing promising results [CK16, CK18]. The significant improvement of quantifier support in SMT solvers has further led both the F* and Lean theorem provers to rely on dependent type encodings during proof search [SWS$^+$13, Agu16, dMKA$^+$15].

Further noteworthy approaches supporting automated verification in the presence of dependent types are those based on Liquid Types [RKJ08, VSJ$^+$14b, VTC$^+$18] and those based on higher-order recursion schemas [Kob09, KTU10, KSU11, OR11, Ter10, SAK15]. Both approaches typically rely on a CEGAR loop during verification and feature a high level of automation. However, the Liquid Types approach restricts the expressivity of refinements in order to ensure decidability of type checking. Moreover, the approach based on recursion schemas operates on programs featuring only integer and function types, and has not been shown to scale to more complex (and useful) types of data.

Counterexample (or model) finding in the presence of dependent types is an emerging research topic. The importance of counterexample finders in interactive theorem provers based on Higher-Order Logic has been demonstrated by tools such as Nitpick [BN10] and Nunchaku [CB]. However, counterexample finders for systems based on dependent type theory are still lacking [Gru]. Nunchaku has recently introduced support for dependent datatypes and record types through encodings into First-Order Logic [CB16], however support for the more complex pi-types is still missing.

As mentioned in Chapter 3, producing reducible inputs in the presence of (dependent) pi-types largely intersects with the topic of synthesis from specifications [MW71, KMPS12, KKKS13, RKT$^+$17]. We will however not discuss this relation any further as our approach does not handle model finding in this case.

Verification through (dependent) type checking. Dependent types constitute a natural foundation for verification systems and many well-known frameworks such as Coq [BC04], Idris [Bra13], Lean [dMKA$^+$15], F* [SWS$^+$13], Agda [BDN09] and LiquidHaskell [VTC$^+$18] are based on dependently-typed languages. Certain systems also feature refinement types as a specification mechanism and rely on SMT solvers to improve automation. However, the verification techniques which are then applied are quite diverse. Approaches based on liquid typing [RKJ08, VRJ13, VSJ$^+$14b, VTC$^+$18] rely on some form of abstract interpretation, F* [SWS$^+$13, SHK$^+$16] is based on a weakest precondition calculus, whereas others [Dun07, GF10] define algorithmic type checking procedures similar to our own.

Most systems given above are based on expressive calculi, such as the Calculus of Inductive Constructions [CH88, CP88] for Coq, which allow formalization of complex mathematical properties while maintaining decidability of type checking. These systems are based on Martin-Löf type theory [MLS84] and rely on equality judgements or types instead of our boolean equality expression. We saw that decidable equality is a prerequisite to complete counterexample finding in the absence of dependent types and quantifiers, however our system could possibly benefit from relying on extensional propositional equality instead when verifying Scala programs. These calculi often feature decidable type checking, however this property generally comes at the cost of automation.

Inductive (and, when supported, coinductive) types are favored over recursive types in most dependent type systems [BC04, Bra13, dMKA$^+$15, SWS$^+$13, BDN09], as well as systems with logical foundations [NPW02]. This distinction allows these systems to introduce an induction (or coinduction) principle given a type definition. Logical relations associated to sized inductive and coinductive types are generally given similarly to our own. However, systems which support the more general recursive types are less common. The TORES system [JPT18] introduces index-stratified types which are similar to our recursive types but does not support our intersection types over index-stratified types.

Our typing rules for function definitions and calls follow the same principles as type-based termination [HPS96, Par98, Xi01, BFG$^+$04, Abe04, Abe07]. Our recursive functions (fixpoint operations) rely on strong induction over arbitrary well-orders which is similar to the approach described in [Abe12]. Our type generalization procedure then serves a similar purpose to the limit ordinal in coinductive definitions. Note however that our approach eliminates the size expression entirely and thus sacrifices generality in order to avoid having to deal with potential paradoxes due to reasoning with a limit ordinal [HPS96, Abe08, Abe10]. In practice, the conditions on which [Abe10] relies in order to eliminate these paradoxes are fairly restrictive as well.

Quantifier support. Many interactive theorem provers [NPW02, dMKA$^+$15, BC04] and verification frameworks [Lei10, SWS$^+$13] support some form of reasoning or specification involving existential and universal quantifiers. Certain systems such as Isabelle/HOL [NPW02] rely on logic as their foundations. Others, as in our case, rely on dependent type systems [BC04, dMKA$^+$15, SWS$^+$13, BDN09] based on intuitionistic Martin-Löf type theory [MLS84]. Our definition of propositional values and the propositional truth relation can be viewed as a restricted version of the impredicative Prop type in the Calculus of Inductive Constructions [CH88, CP88] on which Coq is based. In comparison with Coq, our system sacrifices expressivity of propositional types and expressions in favor of better support for automation.

It is important to note that certain verification systems do not support specifications involving existential or universal quantification. This is for example the case of Liquid Haskell [VSJ14a, VTC$^+$18] which focuses on decidability and automation of type checking. Note however that some restricted form of quantifiers can be simulated through dependent sigma- and pi-types.

Most systems provide automation by embedding proof goals or verification conditions into some variant of First-Order Logic and relying on off-the-shelf automated theorem provers or SMT solvers to discharge them. Quantifiers at the source level are thus embedded into (possibly instrumented) quantifiers in the target prover. Such techniques have been successfully applied in state-of-the-art interactive theorem provers [MP08, CK18, EMT$^+$17, dMKA$^+$15] as well as verification frameworks [Lei10, SHK$^+$16]. While this approach enables these systems to take full advantage of the improvements in the underlying automated prover, the encoding is usually all-or-nothing and disallows the integration of more specialized reasoning techniques during solving. Certain SMT solvers are however begining to bridge this gap by providing more high-level constructs as encoding targets [RBCT16, Bjø].

We saw that our quantifier instantiation procedure follows a similar approach to the one presented in [GdM09]. Most SMT solvers perform instantiation through *E-matching* [DNS05, dMB07], however this approach requires having access to the congruence closure (or *E-graph*) which is not tracked by our system. An alternative approach which has shown effective in practice performs instantiations by selecting likely value terms as instantiation candidates [RTB17]. However, both approaches only constitute decision (or semi-decision) procedures on severly limited fragments [BRK$^+$15].

As discussed, our approach extends the work in [GdM09] with broader model finding capabilities. Although the technique introduced in [GdM09] also performs model-directed instantiations, valid models are only guaranteed to be found when the set of instantiations becomes saturated. Furthermore, although our instantiation selection requires considering a larger clause set, the resulting candidates are more likely to be relevant to the final result. A different approach to model finding relies on domain finiteness to construct valid models [RTGK13, RTG$^+$13], and has shown effective as a backend for model finding in the presence of more complex features [CB, CB16]. These techniques do not rely on a symbolic treatment of candidate instantiations but instead directly perform instantiations with likely values.

Verification through encoding. Many verification systems support a rich and expressive input language which is then encoded into an intermediate language on which verification is performed. For example, Boogie [BCD$^+$05, LLM11] and Why3 [FP13] are two such intermediate languages which are used in verifiers such as Dafny [Lei10], Spec# [LM08], VCC [CMST10] and Krakatoa [FM07]. Viper [MSS16, MSS17], another intermediate verification language, introduces support for permission logics such as separation logic and relies internally on Boogie (as one of multiple possible backends). This design principle allows verification frameworks to support high-level language features simply by defining an appropriate encoding.

A similar approach consists in encoding programs into powerful logics (such as Higher-Order Logic or the Calculus of Inductive Constructions) which can then be handled by existing theorem provers. The expressivity of these encoding targets allows both shallow embeddings and powerful specification constructs. However, these features generally come at the cost of reduced automation during theorem proving. This approach has, for example, been successfully applied to object-oriented program verification [BW08].

Interestingly, intermediate verification languages based on dependent type systems are rare, possibly due to the difficulty in automating verification in the presence of dependent types. A notable outlier is the F* language which has been used as an encoding target for verifying Javascript [SWS$^+$13] and F#. As our approach to automation is fairly different, we believe our verification language will be an interesting alternative encoding target for verification systems.

Most Scala language constructs supported by our encoding procedures have a corresponding counterpart in other verification systems such as Dafny [Lei10, HLQ11, ALN15], Spec# [LM08], Krakatoa [FM07], etc. However, our verification language is quite rich in comparison with Boogie and Why3. This allows our system to often avoid introducing under-approximations during encoding and can lead to significant differences in the encoding procedures. This precision of encoding ensures that counterexamples generally remain valid with respect to the original Scala source code. Furthermore, as our verification language has dedicated support for high-level language constructs, we avoid common issues such as trigger selection in quantified encodings [LP16].

Automated termination checking. Most verification systems feature some level of automation for termination checking. Some systems rely on simple heuristics such as the lexicographic ordering of parameters [Lei10, SHK$^+$16] or use syntactic criteria [Abe, Bou12], whereas others perform more complex analysis but are restricted to first-order languages [MT09, CDMV11, FZG18]. Our system allows semantic termination criteria and considers a variety of candidate measures in order to increase coverage.

First-order techniques such as Size-Change Termination [LJB01] and termination analysis of Term Rewriting Systems [GTSF04] have been successfully extended to the higher-order setting [GRS$^+$11] by reducing the question to a first-order termination problem on an approximate call-graph. It is however not clear how a proof in the approximate call-graph can be extracted into measures and refinements in order to produce a program which can be validated by the type checking procedure.

A different line of work reduces the problem of termination in the presence of higher-order functions to a question of binary reachability [KTUK14, LR12]. These techniques rely on predicate abstraction and CEGAR [KSU11], as well as ranking function inference in order to feature a high level of automation. However, the approach is defined on a simply-typed lambda calculus without datatypes and has not been shown to apply to practical programs.

Conclusion

In this thesis, we have presented a verification system for an expressive subset of the Scala language based on a dependent type system with refinements. We have discussed how to provide automation in the system, perform verification through type checking and support complex Scala features by encoding them into an intermediate verification language. We have demonstrated that the resulting system is effective through a series of verification tasks. We have found that thanks to the expressivity of the supported Scala fragment, our system can verify idiomatic functional Scala programs which contain advanced language features and appear natural to Scala developers.

We have presented a powerful proof automation technique which is complete for counterexamples for a higher-order functional language with recursive functions and datatypes. We have further extended this technique to support dependent types and propositional quantifiers, as well as discussed both syntactic and semantic fragments for which counterexample finding remains complete. Our verification system relies on this procedure to automatically prove (or disprove) verification conditions. In our experience, the counterexample finding procedure enables practical and predictable proof development, and counterexamples have proved critical in avoiding wasted efforts on invalid statements.

Based on our proof automation procedure, we have presented a bidirectional type checking algorithm that ensures both normalization and functional correctness (which further implies the absence of stuck terms). In addition to the usual constructs featured in dependently-typed languages, our type system supports sized (or indexed) recursive types, mutual recursion between type and function definitions, as well as corecursive (or productive) functions. The language and type system have proved sufficiently expressive to encode an important subset of the Scala language while preserving predictability of verification.

We have shown how to encode different features of the Scala language into our verification language. We have presented a shallow encoding of open type hierarchies with subtyping, variance, multiple inheritance and methods which allows us to handle object-oriented Scala programs and leverage the resulting modularity during verification. Our encoding further supports Scala-style contracts [Ode10] which will be statically checked by our system. In order to reduce the annotation burden, we have discussed a measure inference technique which can synthesize ranking functions and refinements necessary to termination checking. We have shown that these encodings are effective by verifying a set of benchmarks which feature advanced Scala language constructs.

Lessons Learned

From the outset, the goal of our work was to enable verification of an expressive subset of the Scala language. The original Leon system upon which Stainless is based already featured elegant counterexample finding and verification procedures for monomorphic first-order programs with executable specifications. However, the supported language did not allow idiomatic Scala programs containing higher-order functions, parametric polymorphism, or subtyping. We will describe here certain important challenges and the lessons we learned while introducing support for these language features in our verification system.

We started by considering support for higher-order functions and parametric polymorphism. Extending the counterexample finding procedure proved surprisingly natural once suitable embedding and extraction strategies were selected. However, designing useful specification mechanisms and a sound verification procedure turned out to be more challenging. Our first attempt was to introduce executable quantifiers into the language, as well as a few other ad-hoc specification constructs such as datatype invariants. Unfortunately, we found that defining reasonable (namely both sound and useful) operational semantics for these extensions was a non-obvious task. Hence, the soundness of our verification system became fairly nebulous.

In a bid to build a formal metatheory for our system and resolve our soundness questions, we considered Tait's reducibility method [Tai67] and its various extensions to dependent type theory. Dependent types (and especially refinement types) would allow us to represent the pre- and postconditions featured by our original first-order language. Furthermore, by introducing an error term, we could even capture the influence of executable contracts on the program traces. Dependent types with refinements were therefore a promising candidate around which to build a metatheory. This adoption of type theory as our system foundations allowed a major shift in both our vision of and approach to verification soundness.

By relying on dependent type theory, soundness of verification became straightforward. Furthermore, our long-standing questions of both defining and proving termination in the presence of higher-order functions were naturally resolved as well. All that remained was for us to extend our counterexample finding procedure to take the newly introduced dependent types into account. We discovered that the unfolding techniques explored in the context of function calls and applications could be further adapted to the reducibility relation. We found that building an end-to-end soundness proof for our dependently-typed language, including the counterexample finding procedure, was feasible and significantly increased our confidence in the overall verification system. Indeed, exploring the metatheory allowed us to identify a rather frightening number of soundness issues in our previous ad-hoc approach.

A further unexpected benefit of primitive support for dependent types in our verification procedure was a significant increase in the expressivity of our verification language. This allowed us to encode complex high-level language features without introducing native support for new constructs in the verification language (our previous strategy). In addition to improving our verification procedure, dependent types therefore enabled simpler and clearer encodings of complex language constructs.

Future Work

Although our verification system already enables verification of useful properties about expressive Scala programs, there remain many important and interesting questions to tackle.

Undecidable equality. The syntactic notion of equality for first-class functions defined in Chapter 2 presents certain disadvantages. Indeed, syntactic equality is inconsistent with the Scala operational semantics, and our system may report invalid (according to Scala) counterexamples involving higher-order first-class functions. Syntactic equality is further inconsistent with common mathematical notions of equality, which makes it difficult to treat first-class functions as mathmatical functions in our system. In a bid to resolve these limitations, it could be useful to consider alternative (undecidable) notions of function equality. In order to allow type checking and reasoning about equality, our system would then require the introduction of an equality type and/or an equality proposition. We have already started preliminary exploration of such constructs in the context of our Coq formalization which features both an equality type and and equality judgement.

The counterexample finding procedure presented in Chapter 2 relies on the syntactic equality notion, both in the embedding of function types into algebraic datatypes (which imposes syntactic equality), and in the extraction of lambda values. The incremental embedding strategy presented in 2.6.1 can be adapted to admit non-syntactic notions of equality simply by relaxing the generated disequality clauses. However, extracting valid lambda values without relying on an equality operation remains challenging. A promising direction consists in generating clauses from which distinguishing inputs can be extracted. These inputs can then be used to differentiate between function values by comparing application results.

Formalizing propositions. In Chapter 4, we discussed how our counterexample (or proof) finding procedure can be extended to handle impredicative propositional quantifiers. However, we did not give a formal defininition of the reducibility and truth relations associated to propositions. As previously mentioned, these definitions are involved as we need to introduce both an arity and an environment in the reducibility definition.

Our truth relation embedding and unfolding procedures enable verification of programs which feature propositional refinements by relying on the counterexample finder. In practice, however, although our quantifier instantiation procedure provides considerable automation, it is useful to introduce type checking rules specialized for proving propositional truth relations. Some propositional expressions imply natural type checking rules such as those given below for the truth relation associated to conjunction and disjunction propositions.

$$\textsc{Conjunction Truth}\quad \frac{P;\Theta;\Gamma \vdash e_1 \; \mathit{holds} \qquad P;\Theta;\Gamma \vdash e_2 \; \mathit{holds}}{P;\Theta;\Gamma \vdash e_1 \wedge e_2 \; \mathit{holds}} \qquad \textsc{Disjunction Truth}\quad \frac{P;\Theta;\Gamma \vdash e_1 \; \mathit{holds} \vee P;\Theta;\Gamma \vdash e_2 \; \mathit{holds}}{P;\Theta;\Gamma \vdash e_1 \vee e_2 \; \mathit{holds}}$$

However, for certain other propositional expressions, there is no clear algorithmic procedure for (directly) verifying the truth relation. For example, a sound but non-algorithmic approach to checking the truth relation associated to an existentially quantified proposition would be to synthesize a witness expression for which the proposition holds. An alternative (and algorithmic) approach would be to leverage the Curry-Howard isomorphism and indirectly prove the truth relation by type checking an expression against the corresponding type. For example, we would have the following rules for universally and existentially quantified propositions.

$$\textsc{Universal Truth}\quad \frac{P;\Theta;\Gamma \vdash e \Downarrow \Pi x:\tau.\,\{u : \mathsf{Unit} \mid p\}}{P;\Theta;\Gamma \vdash e \Downarrow \{u : \mathsf{Unit} \mid \forall x:\tau.\,p\}} \qquad \textsc{Existential Truth}\quad \frac{P;\Theta;\Gamma \vdash e \Downarrow \Sigma x:\tau.\,\{u : \mathsf{Unit} \mid p\}}{P;\Theta;\Gamma \vdash e \Downarrow \{u : \mathsf{Unit} \mid \exists x:\tau.\,p\}}$$

Selecting and implementing a type checking strategy will enable structured reasoning about propositional truth relations in our system beyond what is provided by the truth relation embedding and unfolding procedures.

Enabling metaprogramming. While our proof automation technique is often effective in practice, relying on a single approach to automation can prove limiting. Indeed, certain problems can be handled more efficiently and effectively by different techniques and there exists no single best approach to automation. One can partially address the issue by introducing dedicated support for alternative automation techniques within the tool, however this strategy does not scale. Instead, the approach taken by many verification systems, and especially interactive theorem provers, is to introduce support for metaprogramming [MAD$^+$19] or tactics [NPW02, W$^+$04, BC04, KZK$^+$18, dMKA$^+$15, GT16]. These features allow users to programmatically define and distribute their own automation techniques without requiring a deep knowledge or understanding of the tool implementation.

A first step towards user-defined automation is to reflect the program trees within the language. We can define a datatype in the verification language which corresponds to the abstract syntax trees of the language itself. We can then implement operations such as evaluation or tree transformations on the datatype through (potentially recursive) function definitions. The second and more important step consists in linking the datatype (and possibly certain functions as well) with the system trees through reification and splicing. The challenge here will be in determining which axioms relating operations on reified and spliced trees are necessary in order for our system to allow proof splicing.

Supporting more Scala features. In Chapter 5, we presented an encoding pipeline for many Scala language constructs. Although the resulting fragment is expressive and supports a large class of idiomatic Scala programs, there remain important Scala features which are not handled by our system. Some noteworthy examples are exceptions, mutation with aliasing, non-constructor parameter fields, type members and existential types. In order to allow verification of industrial Scala code, it is important that the language coverage of our tool be close to complete. A first step in this direction would be to introduce support for a special unknown construct to which all unsupported features are mapped. The challenge then consists in ensuring that the handling of the unknown construct in our system is consistent with the original unsupported Scala features.

Let us consider support for exceptions in particular. Our system can verify that no exception is ever thrown, however it can be useful to provide some more fine-grained support for verification in the presence of exceptions, *e.g.* when exceptions are used for control flow. By relying on an effect system for exceptions, these can be eliminated by either performing a CPS (continuation-passing style) transformation or replacing effectful function result types by $\mathsf{Either}[\mathsf{Exception}, \tau]$ and introducing match expressions at relevant call sites. We have already begun exploratory work in both directions.

Final Words

This work makes a case for a verification system which supports a large (and growing!) subset of the Scala language. Targetting such a high-level and feature-rich programming language has proved an opportunity to explore a multi-paradigm verification approach combining object-oriented, functional and type theoretic elements in a single system. I believe that by integrating the advances in industrial programming language design and formal verification, we can both improve the state of the art in formal verification and increase industry adoption by lowering the entry barrier. I hope that someday, in the not-too-distant future, support for formal verification will become just another programming paradigm standing on an equal footing with the object-oriented and functional paradigms.

A Stainless List Library

The Stainless system provides a library which includes various common type and function definitions. In particular, the library contains an (invariant) parametric List type definition with common operations. In order to facilitate verification tasks involving lists, many operations feature useful (verified) contracts. We provide the List definition here for reference.

```
import stainless._
import stainless.lang._
import stainless.annotation._
import stainless.math._
import stainless.proof._

sealed abstract class List[T] {

  def size: BigInt = (this match {
    case Nil() ⇒ BigInt(0)
    case Cons(h, t) ⇒ 1 + t.size
  }) ensuring (_ ≥ 0)

  def length = size

  def content: Set[T] = this match {
    case Nil() ⇒ Set()
    case Cons(h, t) ⇒ Set(h) ++ t.content
  }

  def contains(v: T): Boolean = (this match {
    case Cons(h, t) ⇒ h == v || t.contains(v)
    case Nil() ⇒ false
  }) ensuring { _ == (content contains v) }

  def ++(that: List[T]): List[T] = (this match {
    case Nil() ⇒ that
    case Cons(x, xs) ⇒ Cons(x, xs ++ that)
```

```
}) ensuring { res ⇒
  (res.content == this.content ++ that.content) &&
  (res.size == this.size + that.size) &&
  (that != Nil[T]() || res == this)
}

def head: T = {
  require(this != Nil[T]())
  val Cons(h, _) = this
  h
}

def tail: List[T] = {
  require(this != Nil[T]())
  val Cons(_, t) = this
  t
}

def apply(index: BigInt): T = {
  require(0 ≤ index && index < size)
  if (index == BigInt(0)) {
    head
  } else {
    tail(index−1)
  }
}

def ::(t:T): List[T] = Cons(t, this)

def :+(t:T): List[T] = {
  this match {
    case Nil() ⇒ Cons(t, this)
    case Cons(x, xs) ⇒ Cons(x, xs :+ (t))
  }
} ensuring(res ⇒ (res.size == size + 1) && (res.content == content ++ Set(t)))

def reverse: List[T] = {
  this match {
    case Nil() ⇒ this
    case Cons(x,xs) ⇒ xs.reverse :+ x
  }
} ensuring (res ⇒ (res.size == size) && (res.content == content))

def take(i: BigInt): List[T] = { (this, i) match {
  case (Nil(), _) ⇒ Nil[T]()
  case (Cons(h, t), i) ⇒
    if (i ≤ BigInt(0)) {
        Nil[T]()
      } else {
        Cons(h, t.take(i−1))
      }
}} ensuring { res ⇒
  res.content.subsetOf(this.content) && (res.size == (
    if (i ≤ 0) BigInt(0)
    else if (i ≥ this.size) this.size
    else i
  ))
}

def drop(i: BigInt): List[T] = { (this, i) match {
  case (Nil(), _) ⇒ Nil[T]()
  case (Cons(h, t), i) ⇒
    if (i ≤ BigInt(0)) {
      Cons[T](h, t)
    } else {
      t.drop(i−1)
    }
}} ensuring { res ⇒
  res.content.subsetOf(this.content) && (res.size == (
    if (i ≤ 0) this.size
    else if (i ≥ this.size) BigInt(0)
    else this.size − i
  ))
}

def slice(from: BigInt, to: BigInt): List[T] = {
  require(0 ≤ from && from ≤ to && to ≤ size)
  drop(from).take(to−from)
}

def replace(from: T, to: T): List[T] = { this match {
  case Nil() ⇒ Nil[T]()
  case Cons(h, t) ⇒
    val r = t.replace(from, to)
    if (h == from) {
      Cons(to, r)
    } else {
      Cons(h, r)
    }
}} ensuring { (res: List[T]) ⇒
  res.size == this.size &&
  res.content == (
    (this.content ++ Set(from)) ++
    (if (this.content contains from) Set(to) else Set[T]())
    )
  }

  private def chunk0(s: BigInt, l: List[T], acc: List[T], res: List[List[T]], s0: BigInt): List[List[T]] =
     {
    require(s > 0 && s0 ≥ 0)
    l match {
      case Nil() ⇒
        if (acc.size > 0) {
          res :+ acc
        } else {
          res
        }
      case Cons(h, t) ⇒
        if (s0 == BigInt(0)) {
          chunk0(s, t, Cons(h, Nil()), res :+ acc, s−1)
        } else {
          chunk0(s, t, acc :+ h, res, s0−1)
        }
    }
  }

  def chunks(s: BigInt): List[List[T]] = {
    require(s > 0)

    chunk0(s, this, Nil(), Nil(), s)
  }

  def zip[B](that: List[B]): List[(T, B)] = { (this, that) match {
    case (Cons(h1, t1), Cons(h2, t2)) ⇒
      Cons((h1, h2), t1.zip(t2))
    case _ ⇒
      Nil[(T, B)]()
  }} ensuring { _.size == (
    if (this.size ≤ that.size) this.size else that.size
  )}

  def −(e: T): List[T] = { this match {
    case Cons(h, t) ⇒
      if (e == h) {
        t − e
      } else {
        Cons(h, t − e)
      }
    case Nil() ⇒
      Nil[T]()
  }} ensuring { res ⇒
  res.size ≤ this.size &&
  res.content == this.content ++ Set(e)
}

def ++(that: List[T]): List[T] = { this match {
  case Cons(h, t) ⇒
    if (that.contains(h)) {
      t ++ that
    } else {
      Cons(h, t ++ that)
    }
  case Nil() ⇒
    Nil[T]()
}} ensuring { res ⇒
  res.size ≤ this.size &&
  res.content == this.content ++ that.content
}

def &(that: List[T]): List[T] = { this match {
  case Cons(h, t) ⇒
    if (that.contains(h)) {
      Cons(h, t & that)
    } else {
      t & that
    }
  case Nil() ⇒
    Nil[T]()
}} ensuring { res ⇒
  res.size ≤ this.size &&
  res.content == (this.content & that.content)
}

def padTo(s: BigInt, e: T): List[T] = { (this, s) match {
  case (_, s) if s ≤ 0 ⇒
    this
  case (Nil(), s) ⇒
    Cons(e, Nil().padTo(s−1, e))
  case (Cons(h, t), s) ⇒
    Cons(h, t.padTo(s−1, e))
}} ensuring { res ⇒
  if (s ≤ this.size)
    res == this
  else
    res.size == s &&
    res.content == this.content ++ Set(e)
}
```

```
def indexOf(elem: T): BigInt = { this match {
  case Nil() ⇒ BigInt(−1)
  case Cons(h, t) if h == elem ⇒ BigInt(0)
  case Cons(h, t) ⇒
    val rec = t.indexOf(elem)
    if (rec == BigInt(−1)) BigInt(−1)
    else rec + 1
}} ensuring { res ⇒
  (res ≥ 0) == content.contains(elem)
      }

def init: List[T] = {
  require(!isEmpty)
  (this : @unchecked) match {
    case Cons(h, Nil()) ⇒
      Nil[T]()
    case Cons(h, t) ⇒
      Cons[T](h, t.init)
  }
} ensuring ( (r: List[T]) ⇒
  r.size == this.size − 1 &&
  r.content.subsetOf(this.content)
)

def last: T = {
  require(!isEmpty)
  (this : @unchecked) match {
    case Cons(h, Nil()) ⇒ h
    case Cons(_, t) ⇒ t.last
  }
} ensuring { this.contains _ }

def lastOption: Option[T] = { this match {
  case Cons(h, t) ⇒
    t.lastOption.orElse(Some(h))
  case Nil() ⇒
    None[T]()
}} ensuring { _.isDefined != this.isEmpty }

def headOption: Option[T] = { this match {
  case Cons(h, t) ⇒
    Some(h)
  case Nil() ⇒
    None[T]()
}} ensuring { _.isDefined != this.isEmpty }

def tailOption: Option[List[T]] = { this match {
  case Cons(h, t) ⇒
    Some(t)
  case Nil() ⇒
    None[List[T]]()
}} ensuring { _.isDefined != this.isEmpty }

def unique: List[T] = this match {
  case Nil() ⇒ Nil()
  case Cons(h, t) ⇒
    Cons(h, t.unique − h)
}

def splitAt(e: T): List[List[T]] = split(Cons(e, Nil()))

def split(seps: List[T]): List[List[T]] = this match {
  case Cons(h, t) ⇒
    if (seps.contains(h)) {
      Cons(Nil(), t.split(seps))
    } else {
      val r = t.split(seps)
      Cons(Cons(h, r.head), r.tail)
    }
  case Nil() ⇒
    Cons(Nil(), Nil())
}

def evenSplit: (List[T], List[T]) = {
  val c = size/2
  (take(c), drop(c))
}

def splitAtIndex(index: BigInt) : (List[T], List[T]) = { this match {
  case Nil() ⇒ (Nil[T](), Nil[T]())
  case Cons(h, rest) ⇒ {
    if (index ≤ BigInt(0)) {
      (Nil[T](), this)
    } else {
      val (left,right) = rest.splitAtIndex(index − 1)
      (Cons[T](h,left), right)
    }
  }
}} ensuring { (res:(List[T],List[T])) ⇒
  res._1 ++ res._2 == this &&
  res._1 == take(index) && res._2 == drop(index)
}

def updated(i: BigInt, y: T): List[T] = {
    require(0 ≤ i && i < this.size)
    this match {
      case Cons(x, tail) if i == 0 ⇒
        Cons[T](y, tail)
      case Cons(x, tail) ⇒
        Cons[T](x, tail.updated(i − 1, y))
    }
  }

  private def insertAtImpl(pos: BigInt, l: List[T]): List[T] = {
    require(0 ≤ pos && pos ≤ size)
    if(pos == BigInt(0)) {
      l ++ this
    } else {
      this match {
        case Cons(h, t) ⇒
          Cons(h, t.insertAtImpl(pos−1, l))
        case Nil() ⇒
          l
      }
    }
  } ensuring { res ⇒
    res.size == this.size + l.size &&
    res.content == this.content ++ l.content
  }

  def insertAt(pos: BigInt, l: List[T]): List[T] = {
    require(−pos ≤ size && pos ≤ size)
    if(pos < 0) {
      insertAtImpl(size + pos, l)
    } else {
      insertAtImpl(pos, l)
    }
  } ensuring { res ⇒
    res.size == this.size + l.size &&
    res.content == this.content ++ l.content
  }

  def insertAt(pos: BigInt, e: T): List[T] = {
    require(−pos ≤ size && pos ≤ size)
    insertAt(pos, Cons[T](e, Nil()))
  } ensuring { res ⇒
    res.size == this.size + 1 &&
    res.content == this.content ++ Set(e)
  }

  private def replaceAtImpl(pos: BigInt, l: List[T]): List[T] = {
```

```
  require(0 ≤ pos && pos ≤ size)
  if (pos == BigInt(0)) {
    l ++ this.drop(l.size)
  } else {
    this match {
      case Cons(h, t) ⇒
        Cons(h, t.replaceAtImpl(pos−1, l))
      case Nil() ⇒
        l
    }
  }
} ensuring { res ⇒
  res.content.subsetOf(l.content ++ this.content)
}

def replaceAt(pos: BigInt, l: List[T]): List[T] = {
  require(−pos ≤ size && pos ≤ size)
  if(pos < 0) {
    replaceAtImpl(size + pos, l)
  } else {
    replaceAtImpl(pos, l)
  }
} ensuring { res ⇒
  res.content.subsetOf(l.content ++ this.content)
}

def rotate(s: BigInt): List[T] = {
  if (isEmpty) {
    Nil[T]()
  } else {
    drop(s mod size) ++ take(s mod size)
  }
} ensuring { res ⇒
  res.size == this.size
}

def isEmpty = this match {
  case Nil() ⇒ true
  case _ ⇒ false
}

def nonEmpty = !isEmpty

def map[R](f: T ⇒ R): List[R] = { this match {
  case Nil() ⇒ Nil[R]()
  case Cons(h, t) ⇒ f(h) :: t.map(f)
}} ensuring { _.size == this.size }
```

```
def foldLeft[R](z: R)(f: (R,T) ⇒ R): R = this match {
  case Nil() ⇒ z
  case Cons(h,t) ⇒ t.foldLeft(f(z,h))(f)
}

def foldRight[R](z: R)(f: (T,R) ⇒ R): R = this match {
  case Nil() ⇒ z
  case Cons(h, t) ⇒ f(h, t.foldRight(z)(f))
}

def scanLeft[R](z: R)(f: (R,T) ⇒ R): List[R] = { this match {
  case Nil() ⇒ z :: Nil()
  case Cons(h,t) ⇒ z :: t.scanLeft(f(z,h))(f)
}} ensuring { !_.isEmpty }

def scanRight[R](z: R)(f: (T,R) ⇒ R): List[R] = { this match {
  case Nil() ⇒ z :: Nil[R]()
  case Cons(h, t) ⇒
    val rest@Cons(h1,_) = t.scanRight(z)(f)
    f(h, h1) :: rest
}} ensuring { !_.isEmpty }

def flatMap[R](f: T ⇒ List[R]): List[R] =
  ListOps.flatten(this map f)

def filter(p: T ⇒ Boolean): List[T] = { this match {
  case Nil() ⇒ Nil[T]()
  case Cons(h, t) if p(h) ⇒ Cons(h, t.filter(p))
  case Cons(_, t) ⇒ t.filter(p)
}} ensuring { res ⇒
  res.size ≤ this.size &&
  res.content.subsetOf(this.content) &&
  res.forall(p)
}

def filterNot(p: T ⇒ Boolean): List[T] =
  filter(!p(_)) ensuring { res ⇒
    res.size ≤ this.size &&
    res.content.subsetOf(this.content) &&
    res.forall(!p(_))
  }

def partition(p: T ⇒ Boolean): (List[T], List[T]) = { this match {
  case Nil() ⇒ (Nil[T](), Nil[T]())
  case Cons(h, t) ⇒
    val (l1, l2) = t.partition(p)
    if (p(h)) (h :: l1, l2)
    else (l1, h :: l2)
}} ensuring { res ⇒
  res._1 == filter(p) &&
  res._2 == filterNot(p)
}

def withFilter(p: T ⇒ Boolean) = filter(p)

def forall(p: T ⇒ Boolean): Boolean = this match {
  case Nil() ⇒ true
  case Cons(h, t) ⇒ p(h) && t.forall(p)
}

def exists(p: T ⇒ Boolean) = !forall(!p(_))

def find(p: T ⇒ Boolean): Option[T] = { this match {
  case Nil() ⇒ None[T]()
  case Cons(h, t) ⇒ if (p(h)) Some(h) else t.find(p)
}} ensuring { res ⇒ res match {
  case Some(r) ⇒ (content contains r) && p(r)
  case None() ⇒ true
}}

def groupBy[R](f: T ⇒ R): Map[R, List[T]] = this match {
  case Nil() ⇒ Map.empty[R, List[T]]
  case Cons(h, t) ⇒
    val key: R = f(h)
    val rest: Map[R, List[T]] = t.groupBy(f)
    val prev: List[T] = if (rest isDefinedAt key) rest(key) else Nil[T]()
    (rest ++ Map((key, h :: prev))) : Map[R, List[T]]
}

def takeWhile(p: T ⇒ Boolean): List[T] = { this match {
  case Cons(h,t) if p(h) ⇒ Cons(h, t.takeWhile(p))
  case _ ⇒ Nil[T]()
}} ensuring { res ⇒
  (res forall p) &&
  (res.size ≤ this.size) &&
  (res.content subsetOf this.content)
}

def dropWhile(p: T ⇒ Boolean): List[T] = { this match {
  case Cons(h,t) if p(h) ⇒ t.dropWhile(p)
  case _ ⇒ this
}} ensuring { res ⇒
  (res.size ≤ this.size) &&
    (res.content subsetOf this.content) &&
    (res.isEmpty || !p(res.head))
  }

  def count(p: T ⇒ Boolean): BigInt = { this match {
    case Nil() ⇒ BigInt(0)
    case Cons(h, t) ⇒
      (if (p(h)) BigInt(1) else BigInt(0)) + t.count(p)
  }} ensuring {
    _ == this.filter(p).size
  }

  def indexWhere(p: T ⇒ Boolean): BigInt = { this match {
    case Nil() ⇒ BigInt(−1)
    case Cons(h, _) if p(h) ⇒ BigInt(0)
    case Cons(_, t) ⇒
      val rec = t.indexWhere(p)
      if (rec ≥ 0) rec + BigInt(1)
      else BigInt(−1)
  }} ensuring {
    _ ≥ BigInt(0) == (this exists p)
  }

  def toSet: Set[T] = foldLeft(Set[T]()){
    case (current, next) ⇒ current ++ Set(next)
  }
}

case class Cons[T](h: T, t: List[T]) extends List[T]

case class Nil[T]() extends List[T]

object List {
  def fill[T](n: BigInt)(x: T) : List[T] = {
    if (n ≤ 0) Nil[T]()
    else Cons[T](x, fill[T](n−1)(x))
  } ensuring { res ⇒
    (res.content == (if (n ≤ BigInt(0)) Set.empty[T] else Set(x))) &&
    res.size == (if (n ≤ BigInt(0)) BigInt(0) else n)
  }

  def range(start: BigInt, until: BigInt): List[BigInt] = {
    require(start ≤ until)
    decreases(until − start)
    if(until ≤ start) Nil[BigInt]() else Cons(start, range(start + 1, until))
  } ensuring{(res: List[BigInt]) ⇒ res.size == until − start }
  def mkString[A](l: List[A], mid: String, f: A ⇒ String) = {
    def rec(l: List[A]): String = l match {
      case Nil() ⇒ ""
      case Cons(a, b) ⇒ mid + f(a) + rec(b)
    }
    l match {
      case Nil() ⇒ ""
      case Cons(a, b) ⇒ f(a) + rec(b)
    }
  }
}

object ListOps {
  def flatten[T](ls: List[List[T]]): List[T] = ls match {
    case Cons(h, t) ⇒ h ++ flatten(t)
    case Nil() ⇒ Nil()
  }

  def isSorted(ls: List[BigInt]): Boolean = ls match {
    case Nil() ⇒ true
    case Cons(_, Nil()) ⇒ true
    case Cons(h1, Cons(h2, _)) if(h1 > h2) ⇒ false
    case Cons(_, t) ⇒ isSorted(t)
  }

  def sorted(ls: List[BigInt]): List[BigInt] = { ls match {
    case Cons(h, t) ⇒ sortedIns(sorted(t), h)
    case Nil() ⇒ Nil[BigInt]()
  }} ensuring { isSorted _ }

  private def sortedIns(ls: List[BigInt], v: BigInt): List[BigInt] = {
    require(isSorted(ls))
    ls match {
      case Nil() ⇒ Cons(v, Nil())
      case Cons(h, t) ⇒
        if (v ≤ h) {
          Cons(v, ls)
        } else {
          Cons(h, sortedIns(t, v))
        }
    }
  } ensuring { res ⇒ isSorted(res) && res.content == ls.content + v }

  def toMap[K, V](l: List[(K, V)]): Map[K, V] = l.foldLeft(Map[K, V]()){
    case (current, (k, v)) ⇒ current ++ Map(k -> v)
  }
}
```

```
object :: {
  def unapply[A](l: List[A]): Option[(A, List[A])] = l match {
    case Nil() ⇒ None()
    case Cons(x, xs) ⇒ Some((x, xs))
  }
}


import ListOps._

object ListSpecs {
  def snocIndex[T](l: List[T], t: T, i: BigInt): Boolean = {
    require(0 ≤ i && i < l.size + 1)
    ((l :+ t).apply(i) == (if (i < l.size) l(i) else t))
  }.holds because (
    l match {
      case Nil() ⇒ true
      case Cons(x, xs) ⇒ if (i > 0) snocIndex[T](xs, t, i−1) else true
    }
  )

  @induct
  def consIndex[T](h: T, t: List[T], i: BigInt): Boolean = {
    require(0 ≤ i && i < t.size + 1)
    (h :: t).apply(i) == (if (i == 0) h else t.apply(i − 1))
  }.holds

  def reverseIndex[T](l: List[T], i: BigInt): Boolean = {
    require(0 ≤ i && i < l.size)
    l.reverse.apply(i) == l.apply(l.size − 1 − i)
  }.holds because(
    l match {
      case Nil() ⇒ true
      case Cons(x,xs) ⇒
        snocIndex(xs.reverse, x, i) &&
        (if (i < xs.size) consIndex(x, xs, l.size − 1 − i) && reverseIndex[T](xs, i) else true)
    }
  )

  def snocLast[T](l: List[T], x: T): Boolean = {
    ((l :+ x).last == x)
  }.holds because {
    l match {
      case Nil() ⇒ true
      case Cons(y, ys) ⇒ {
        ((y :: ys) :+ x).last ==| trivial |
        (y :: (ys :+ x)).last ==| trivial |
        (ys :+ x).last ==| snocLast(ys, x) |
        x
      }.qed
    }
  }

  def headReverseLast[T](l: List[T]): Boolean = {
    require (!l.isEmpty)
    (l.head == l.reverse.last)
  }.holds because {
    val Cons(x, xs) = l;
    {
      (x :: xs).head ==| trivial |
      x ==| snocLast(xs.reverse, x) |
      (xs.reverse :+ x).last ==| trivial |
      (x :: xs).reverse.last
    }.qed
  }

  def appendIndex[T](l1: List[T], l2: List[T], i: BigInt): Boolean = {
    require(0 ≤ i && i < l1.size + l2.size)
    (l1 ++ l2).apply(i) == (if (i < l1.size) l1(i) else l2(i − l1.size))
  }.holds because {
    l1 match {
      case Nil() ⇒ true
      case Cons(x,xs) ⇒
        (i != BigInt(0)) =⇒ appendIndex[T](xs, l2, i − 1)
    }
  }

  @induct
  def appendAssoc[T](l1: List[T], l2: List[T], l3: List[T]): Boolean = {
    (l1 ++ l2) ++ l3 == l1 ++ (l2 ++ l3)
  }.holds

  @induct
  def rightUnitAppend[T](l1: List[T]): Boolean = {
    l1 ++ Nil() == l1
  }.holds

  def leftUnitAppend[T](l1: List[T]): Boolean = {
    Nil() ++ l1 == l1
  }.holds

  def snocIsAppend[T](l: List[T], t: T): Boolean = {
    (l :+ t) == l ++ Cons[T](t, Nil())
```

```
}.holds because {
  l match {
    case Nil() ⇒ true
    case Cons(x,xs) ⇒ snocIsAppend(xs,t)
  }
}

def snocAfterAppend[T](l1: List[T], l2: List[T], t: T): Boolean = {
  (l1 ++ l2) :+ t == l1 ++ (l2 :+ t)
}.holds because {
  l1 match {
    case Nil() ⇒ true
    case Cons(x,xs) ⇒ snocAfterAppend(xs,l2,t)
  }
}

def snocReverse[T](l: List[T], t: T): Boolean = {
  (l :+ t).reverse == Cons(t, l.reverse)
}.holds because {
  l match {
    case Nil() ⇒ true
    case Cons(x,xs) ⇒ snocReverse(xs,t)
  }
}

def reverseReverse[T](l: List[T]): Boolean = {
  l.reverse.reverse == l
}.holds because {
  l match {
    case Nil() ⇒ trivial
    case Cons(x, xs) ⇒ {
      (xs.reverse :+ x).reverse ==| snocReverse[T](xs.reverse, x) |
      x :: xs.reverse.reverse ==| reverseReverse[T](xs) |
      (x :: xs)
    }.qed
  }
}

def reverseAppend[T](l1: List[T], l2: List[T]): Boolean = {
  (l1 ++ l2).reverse == l2.reverse ++ l1.reverse
}.holds because {
  l1 match {
    case Nil() ⇒ {
      (Nil() ++ l2).reverse ==| trivial |
      l2.reverse ==| rightUnitAppend(l2.reverse) |
      l2.reverse ++ Nil() ==| trivial |
      l2.reverse ++ Nil().reverse
      }.qed
      case Cons(x, xs) ⇒ {
        ((x :: xs) ++ l2).reverse ==| trivial |
        (x :: (xs ++ l2)).reverse ==| trivial |
        (xs ++ l2).reverse :+ x ==| reverseAppend(xs, l2) |
        (l2.reverse ++ xs.reverse) :+ x ==|
          snocAfterAppend(l2.reverse, xs.reverse, x) |
        l2.reverse ++ (xs.reverse :+ x) ==| trivial |
        l2.reverse ++ (x :: xs).reverse
      }.qed
    }
  }

  def snocFoldRight[A, B](xs: List[A], y: A, z: B, f: (A, B) ⇒ B): Boolean = {
    (xs :+ y).foldRight(z)(f) == xs.foldRight(f(y, z))(f)
  }.holds because {
    xs match {
      case Nil() ⇒ true
      case Cons(x, xs) ⇒ snocFoldRight(xs, y, z, f)
    }
  }

  def folds[A, B](xs: List[A], z: B, f: (B, A) ⇒ B): Boolean = {
    val f2 = (x: A, z: B) ⇒ f(z, x)
    ( xs.foldLeft(z)(f) == xs.reverse.foldRight(z)(f2) ) because {
      xs match {
        case Nil() ⇒ true
        case Cons(x, xs) ⇒ {
          (x :: xs).foldLeft(z)(f) ==| trivial |
          xs.foldLeft(f(z, x))(f) ==| folds(xs, f(z, x), f) |
          xs.reverse.foldRight(f(z, x))(f2) ==| trivial |
          xs.reverse.foldRight(f2(x, z))(f2) ==|
            snocFoldRight(xs.reverse, x, z, f2) |
          (xs.reverse :+ x).foldRight(z)(f2) ==| trivial |
          (x :: xs).reverse.foldRight(z)(f2)
        }.qed
      }
    }
  }.holds

  def scanVsFoldLeft[A, B](l: List[A], z: B, f: (B, A) ⇒ B): Boolean = {
    ( l.scanLeft(z)(f).last == l.foldLeft(z)(f) )
  }.holds because {
    l match {
      case Nil() ⇒ true
      case Cons(x, xs) ⇒ scanVsFoldLeft(xs, f(z, x), f)
    }
```

```
}

@induct
def scanVsFoldRight[A,B](l: List[A], z: B, f: (A,B) ⇒ B): Boolean = {
  l.scanRight(z)(f).head == l.foldRight(z)(f)
}.holds

def appendContent[A](l1: List[A], l2: List[A]) = {
  l1.content ++ l2.content == (l1 ++ l2).content
}.holds

def flattenPreservesContent[T](ls: List[List[T]]): Boolean = {
  val f: (List[T], Set[T]) ⇒ Set[T] = _.content ++ _
  ( flatten(ls).content == ls.foldRight(Set[T]())(f) ) because {
    ls match {
      case Nil() ⇒ true
      case Cons(h, t) ⇒ {
        flatten(h :: t).content ==| trivial |
        (h ++ flatten(t)).content ==| appendContent(h, flatten(t)) |
        h.content ++ flatten(t).content ==| flattenPreservesContent(t) |
        h.content ++ t.foldRight(Set[T]())(f) ==| trivial |
        f(h, Set[T]()) ++ t.foldRight(Set[T]())(f) ==| trivial |
        (h :: t).foldRight(Set[T]())(f)
      }.qed
    }
  }
}.holds

def appendUpdate[T](l1: List[T], l2: List[T], i: BigInt, y: T): Boolean = {
  require(0 ≤ i && i < l1.size + l2.size)
  ((l1 ++ l2).updated(i, y) == (
    if (i < l1.size)
      l1.updated(i, y) ++ l2
    else
      l1 ++ l2.updated(i − l1.size, y)))
}.holds because (l1 match {
  case Nil() ⇒ true
  case Cons(x, xs) ⇒ if (i == 0) true else appendUpdate[T](xs, l2, i − 1, y)
})

def appendTakeDrop[T](l1: List[T], l2: List[T], n: BigInt): Boolean = {
  ((l1 ++ l2).take(n) == (
    if (n < l1.size) l1.take(n)
    else if (n > l1.size) l1 ++ l2.take(n − l1.size)
    else l1)) &&
    ((l1 ++ l2).drop(n) == (
      if (n < l1.size) l1.drop(n) ++ l2
```

```
        else if (n > l1.size) l2.drop(n − l1.size)
        else l2))
  }.holds because (l1 match {
    case Nil() ⇒ true
    case Cons(x, xs) ⇒ if (n ≤ 0) true else appendTakeDrop[T](xs, l2, n − 1)
  })

  def appendInsert[T](l1: List[T], l2: List[T], i: BigInt, y: T): Boolean = {
    require(0 ≤ i && i ≤ l1.size + l2.size)
    (l1 ++ l2).insertAt(i, y) == (
      if (i < l1.size) l1.insertAt(i, y) ++ l2
      else l1 ++ l2.insertAt(i − l1.size, y))
  }.holds because (l1 match {
    case Nil() ⇒ true
    case Cons(x, xs) ⇒ if (i == 0) true else appendInsert[T](xs, l2, i − 1, y)
  })

  def applyForAll[T](l: List[T], i: BigInt, p: T ⇒ Boolean): Boolean = {
    require(i ≥ 0 && i < l.length && l.forall(p))
    p(l(i))
  }.holds because (l match {
    case Nil() ⇒ trivial
    case Cons(head, tail) ⇒ if(i == 0) p(head) else applyForAll(l.tail, i − 1, p)
  })
}
```

B Pairing Function Benchmark

We show here that the pairing function $2^x(2y+1)-1$ gives a bijection between the natural numbers and pairs of natural numbers. The statement is established by the pair_unique function in the following program.

```
import stainless.lang._
import stainless.equations._
import stainless.annotation._

object GodelNumbering {
  sealed abstract class Nat {
    def +(that: Nat): Nat = this match {
      case Zero ⇒ that
      case Succ(n) ⇒ Succ(n + that)
    }

    def ∗(that: Nat): Nat = this match {
      case Zero ⇒ Zero
      case Succ(n) ⇒ (n ∗ that) + that
    }

    def −(that: Nat): Nat = ((this, that) match {
      case (Succ(n1), Succ(n2)) ⇒ n1 − n2
      case _ ⇒ this
    }) ensuring { res ⇒
      res.repr ≤ repr &&
      ((this > Zero && that > Zero) ==⇒ res.repr < repr)
    }

    def <(that: Nat): Boolean = ((this, that) match {
      case (Succ(n1), Succ(n2)) ⇒ n1 < n2
      case (Zero, Succ(_)) ⇒ true
      case _ ⇒ false
    }) ensuring (_ == repr < that.repr)
    def ≤(that: Nat): Boolean = (this < that) || (this == that)
    def >(that: Nat): Boolean = !(this < that) && (this != that)
    def ≥(that: Nat): Boolean = (this > that) || (this == that)

    def /(that: Nat): Nat = {
      require(that > Zero)
      decreases(repr)
      if (this < that) Zero else
      Succ((this − that) / that)
    } ensuring { res ⇒
      res.repr ≤ repr &&
      ((this > Zero && that > One) =⇒ res.repr < repr)
    }

    def %(that: Nat): Nat = {
      require(that > Zero)
      decreases(repr)
      if (this < that) this
      else (this − that) % that
    }

    def repr: BigInt = (this match {
      case Zero ⇒ BigInt(0)
      case Succ(n) ⇒ n.repr + BigInt(1)
    }) ensuring (_ ≥ BigInt(0))
  }

  case object Zero extends Nat
  case class Succ(n: Nat) extends Nat

  val One = Succ(Zero)
  val Two = Succ(One)

  @induct
  def plus_zero(n: Nat): Boolean = { n + Zero == n }.holds
  def zero_plus(n: Nat): Boolean = { Zero + n == n }.holds

  @induct
  def minus_identity(n: Nat): Boolean = (n − n == Zero).holds

  @induct
  def associative_plus(n1: Nat, n2: Nat, n3: Nat): Boolean = {
    (n1 + n2) + n3 == n1 + (n2 + n3)
  }.holds

  def commutative_plus(n1: Nat, n2: Nat): Boolean = {
```

```
  n1 + n2 == n2 + n1
}.holds because (n1 match {
  case Zero ⇒ zero_plus(n2) && plus_zero(n2)
  case Succ(p1) ⇒ n2 match {
    case Zero ⇒ zero_plus(p1) && plus_zero(p1)
    case Succ(p2) ⇒ commutative_plus(n1, p2) && commutative_plus(p1, n2)
  }
})

def distributive_times(n1: Nat, n2: Nat, n3: Nat): Boolean = {
  n1 ∗ (n2 + n3) == (n1 ∗ n2) + (n1 ∗ n3) because (n1 match {
    case Zero ⇒ true
    case Succ(p1) ⇒ {
      Succ(p1) ∗ (n2 + n3) ==| trivial |
      p1 ∗ (n2 + n3) + (n2 + n3) ==| distributive_times(p1, n2, n3) |
      (p1 ∗ n2) + (p1 ∗ n3) + (n2 + n3) ==| associative_plus((p1 ∗ n2) + (p1 ∗ n3), n2, n3) |
      (p1 ∗ n2) + (p1 ∗ n3) + n2 + n3 ==| associative_plus(p1 ∗ n2, p1 ∗ n3, n2) |
      (p1 ∗ n2) + ((p1 ∗ n3) + n2) + n3 ==| commutative_plus(p1 ∗ n3, n2) |
      (p1 ∗ n2) + (n2 + (p1 ∗ n3)) + n3 ==| associative_plus(p1 ∗ n2, n2, p1 ∗ n3) |
      ((p1 ∗ n2) + n2) + (p1 ∗ n3) + n3 ==| associative_plus((p1 ∗ n2) + n2, p1 ∗ n3, n3) |
      ((p1 ∗ n2) + n2) + ((p1 ∗ n3) + n3) ==| trivial |
      (n1 ∗ n2) + (n1 ∗ n3)
    }.qed
  })
}.holds

def commutative_times(n1: Nat, n2: Nat): Boolean = {
  (n1 ∗ n2 == n2 ∗ n1) because ((n1, n2) match {
    case (Zero, Zero) ⇒ true
    case (Zero, Succ(p2)) ⇒ commutative_times(n1, p2)
    case (Succ(p1), Zero) ⇒ commutative_times(p1, n2)
    case (Succ(p1), Succ(p2)) ⇒ {
      n1 ∗ n2 ==| trivial |
      (p1 ∗ n2) + n2 ==| commutative_times(p1, n2) |
      (n2 ∗ p1) + n2 ==| trivial |
      ((p2 ∗ p1) + p1) + n2 ==| commutative_times(p2, p1) |
      ((p1 ∗ p2) + p1) + n2 ==| associative_plus(p1 ∗ p2, p1, n2) |
      (p1 ∗ p2) + (p1 + n2) ==| commutative_plus(p1, n2) |
      (p1 ∗ p2) + (n2 + p1) ==|
        (associative_plus(p2, One, p1) && commutative_plus(p2, One)) |
      (p1 ∗ p2) + (p2 + n1) ==| associative_plus(p1 ∗ p2, p2, n1) |
      ((p1 ∗ p2) + p2) + n1 ==| trivial |
      (n1 ∗ p2) + n1 ==| commutative_times(n1, p2) |
      (p2 ∗ n1) + n1 ==| trivial |
      n2 ∗ n1
    }.qed
  })
```

```
}.holds

def distributive_times2(n1: Nat, n2: Nat, n3: Nat): Boolean = {
  (n1 + n2) ∗ n3 == (n1 ∗ n3) + (n2 ∗ n3)
}.holds because (
  commutative_times(n1 + n2, n3) &&
  distributive_times(n3, n1, n2) &&
  commutative_times(n1, n3) &&
  commutative_times(n2, n3)
)

def associative_times(n1: Nat, n2: Nat, n3: Nat): Boolean = {
  (n1 ∗ (n2 ∗ n3) == (n1 ∗ n2) ∗ n3) because (n1 match {
    case Zero ⇒ true
    case Succ(p1) ⇒ {
      n1 ∗ (n2 ∗ n3) ==| trivial |
      (p1 ∗ (n2 ∗ n3)) + (n2 ∗ n3) ==| associative_times(p1, n2, n3) |
      ((p1 ∗ n2) ∗ n3) + (n2 ∗ n3) ==| commutative_plus((p1 ∗ n2) ∗ n3, n2 ∗ n3) |
      (n2 ∗ n3) + ((p1 ∗ n2) ∗ n3) ==| distributive_times2(n2, p1 ∗ n2, n3) |
      (n2 + (p1 ∗ n2)) ∗ n3 ==| commutative_plus(n2, p1 ∗ n2) |
      ((p1 ∗ n2) + n2) ∗ n3 ==| trivial |
      (n1 ∗ n2) ∗ n3
    }.qed
  })
}.holds

def succ_<(n1: Nat, n2: Nat): Boolean = {
  require(n1 ≤ n2)
  (n1 < Succ(n2)) because (n1 match {
    case Zero ⇒ true
    case Succ(n) ⇒
      val Succ(p2) = n2
      succ_<(n, p2)
  })
}.holds

def succ_≤(n1: Nat, n2: Nat): Boolean = {
  require(n1 < n2)
  Succ(n1) ≤ n2 because (n2 match {
    case Succ(p2) if n1 != p2 ⇒ pred_<(n1, n2) && succ_≤(n1, p2)
    case _ ⇒ true
  })
}.holds

def pred_<(n1: Nat, n2: Nat): Boolean = {
  require(n1 < n2)
  val Succ(n) = n2
  ((n1 != n) ==> (n1 < n)) because (n2 match {
    case Succ(n) if n == n1 ⇒ true
    case Succ(p2) ⇒ n1 match {
      case Zero ⇒ true
      case Succ(p1) ⇒ pred_<(p1, p2)
    }
  })
}.holds

def pred_≤(n1: Nat, n2: Nat): Boolean = {
  require(n1 > Zero && n1 ≤ n2)
  val Succ(p1) = n1
  p1 < n2 because succ_<(p1, p1) && (n1 == n2 || transitive_<(p1, n1, n2))
}.holds

def transitive_<(n1: Nat, n2: Nat, n3: Nat): Boolean = {
  require(n1 < n2 && n2 < n3)
  (n1 < n3) because (n3 match {
    case Zero ⇒ true
    case Succ(n) if n == n2 ⇒ succ_<(n1, n)
    case Succ(n) ⇒ pred_<(n2, n3) && transitive_<(n1, n2, n) && succ_<(n1, n)
  })
}.holds

def antisymmetric_<(n1: Nat, n2: Nat): Boolean = {
  n1 < n2 == !(n2 ≤ n1) because ((n1, n2) match {
    case (Succ(p1), Succ(p2)) ⇒ antisymmetric_<(p1, p2)
    case _ ⇒ true
  })
}.holds

def plus_succ(n1: Nat, n2: Nat): Boolean = {
  n1 + Succ(n2) ==| associative_plus(n1, One, n2) |
  (n1 + One) + n2 ==| commutative_plus(n1, One) |
  (One + n1) + n2 ==| associative_plus(One, n1, n2) |
  Succ(n1 + n2)
}.qed

def plus_<(n1: Nat, n2: Nat, n3: Nat): Boolean = {
  require(n2 < n3)
  (n1 + n2 < n1 + n3) because (n3 match {
    case Succ(p3) if n2 == p3 ⇒
      plus_succ(n1, n2) &&
      succ_<(n1 + n2, n1 + n2)
    case Succ(p3) ⇒
      plus_succ(n1, p3) &&
      pred_<(n2, n3) &&
      plus_<(n1, n2, p3) &&
      succ_<(n1 + n2, n1 + p3)
  })
}.holds

def plus_<(n1: Nat, n2: Nat, n3: Nat, n4: Nat): Boolean = {
  require(n1 ≤ n3 && n2 ≤ n4)
  n1 + n2 ≤ n3 + n4 because (n3 match {
    case Zero ⇒ trivial
    case Succ(_) if n1 == n3 && n2 == n4 ⇒ trivial
    case Succ(_) if n1 == n3 ⇒ plus_<(n1, n2, n4)
    case Succ(p3) ⇒ pred_<(n1, n3) && plus_<(n1, n2, p3, n4) && succ_<(n1 + n2, p3 + n4)
  })
}.holds

def associative_plus_minus(n1: Nat, n2: Nat, n3: Nat): Boolean = {
  require(n2 ≥ n3)
  (n1 + n2) − n3 == n1 + (n2 − n3) because ((n2, n3) match {
    case (Succ(p2), Succ(p3)) ⇒
      {
        (n1 + Succ(p2)) − Succ(p3) ==| commutative_plus(One, p2) |
        (n1 + (p2 + One)) − Succ(p3) ==| associative_plus(n1, p2, One) |
        ((n1 + p2) + One) − Succ(p3) ==| commutative_plus(n1 + p2, One) |
        Succ(n1 + p2) − Succ(p3) ==| trivial |
        (n1 + p2) − p3 ==| associative_plus_minus(n1, p2, p3) |
        n1 + (p2 − p3) ==| trivial |
        n1 + (n2 − n3)
      }.qed
    case _ ⇒ true
  })
}.holds

def additive_inverse(n1: Nat, n2: Nat): Boolean = {
  n1 + n2 − n2 == n1
}.holds because (associative_plus_minus(n1, n2, n2) && minus_identity(n2) && plus_zero(n1))

def multiplicative_inverse(n1: Nat, n2: Nat): Boolean = {
  require(n2 > Zero)
  (n1 ∗ n2) / n2 == n1 because (n1 match {
    case Succ(p1) ⇒
      {
        (n1 ∗ n2) / n2 ==| trivial |
        (p1 ∗ n2 + n2) / n2 ==| (
          commutative_plus(p1 ∗ n2, n2) &&
          increasing_plus(n2, p1 ∗ n2) &&
          antisymmetric_<(p1 ∗ n2 + n2, n2)) |
        (Succ(((p1 ∗ n2 + n2) − n2) / n2) : Nat) ==| additive_inverse(p1 ∗ n2, n2) |
        (Succ((p1 * n2) / n2) : Nat) ==| multiplicative_inverse(p1, n2) |
        n1
      }.qed

    case _ ⇒ true
  })
}.holds

@induct
def increasing_plus(n1: Nat, n2: Nat): Boolean = {
  n1 ≤ n1 + n2
}.holds

@induct
def increasing_plus_strict(n1: Nat, n2: Nat): Boolean = {
  require(n2 > Zero)
  n1 < n1 + n2
}.holds

def increasing_times(n1: Nat, n2: Nat): Boolean = {
  require(n1 > Zero && n2 > Zero)
  n1 ≤ n1 * n2 because (n1 match {
    case Succ(Zero) ⇒ true
    case Succ(p1) ⇒
      assert(increasing_times(p1, n2))
      assert(increasing_plus_strict(p1 * n2, n2))
      assert(p1 == p1 * n2 || transitive_<(p1, p1 * n2, p1 * n2 + n2))
      assert(succ_≤(p1, p1 * n2 + n2))
      (n1 ≤ n1 * n2 ==| trivial | true).qed
  })
}.holds

def pow(n1: Nat, n2: Nat): Nat = n2 match {
  case Succ(n) ⇒ n1 * pow(n1, n)
  case Zero ⇒ One
}

def pow_positive(n1: Nat, n2: Nat): Boolean = {
  require(n1 > Zero)
  pow(n1, n2) > Zero because (n2 match {
    case Succ(p2) ⇒ pow_positive(n1, p2) && increasing_times(n1, pow(n1, p2))
    case _ ⇒ true
  })
}.holds

def isEven(n: Nat): Boolean = n match {
  case Zero ⇒ true
```

```
    case Succ(Zero) ⇒ false
    case Succ(n) ⇒ !isEven(n)
  }

  def times_two_even(n: Nat): Boolean = {
    isEven(Two * n) because (n match {
      case Zero ⇒ true
      case Succ(p) ⇒ {
        isEven(Two * n) ==| commutative_times(Two, n) |
        isEven(n * Two) ==| trivial |
        isEven(p * Two + Two) ==| commutative_plus(p * Two, Two) |
        isEven(p * Two) ==| commutative_times(Two, p) |
        isEven(Two * p) ==| times_two_even(p) |
        true
      }.qed
    })
  }.holds

  def times_two_plus_one_odd(n: Nat): Boolean = {
    !isEven(Two * n + One) because times_two_even(n) && commutative_plus(Two * n, One)
  }.holds

  def power_two_even(n: Nat): Boolean = {
    require(n > Zero)
    isEven(pow(Two, n)) because (n match {
      case Succ(p) ⇒ times_two_even(pow(Two, p))
    })
  }.holds

  def pair(n1: Nat, n2: Nat): Nat = pow(Two, n1) * (Two * n2 + One) − One

  def log2_and_remainder(n: Nat): (Nat, Nat) = {
    decreases(n.repr)
    if (isEven(n) && n > Zero) {
      val (a, b) = log2_and_remainder(n / Two)
      (Succ(a), b)
    } else {
      (Zero, n)
    }
  }

  def project(n: Nat): (Nat, Nat) = {
    val (a, b) = log2_and_remainder(Succ(n))
    (a, (b − One) / Two)
  }

  def project_1_inverse(n1: Nat, n2: Nat): Boolean = {
```

```
log2_and_remainder(Succ(pair(n1, n2))) == (n1, (Two * n2 + One)) because (n1 match {
  case Succ(p1) ⇒
    def assoc_plus_minus_one(n: Nat): Boolean = {
      (One + (pow(Two, n) * (Two * n2 + One) − One) ==
        (One + pow(Two, n) * (Two * n2 + One)) − One) because {
          pow_positive(Two, n) &&
          commutative_plus(Two * n2, One) &&
          increasing_times(pow(Two, n), (Two * n2 + One)) &&
          associative_plus_minus(One, pow(Two, n) * (Two * n2 + One), One)
        }
    }.holds

    {
      log2_and_remainder(Succ(pair(n1, n2))) ==|
        trivial |
      log2_and_remainder(Succ(pow(Two, n1) * (Two * n2 + One) − One)) ==|
        trivial |
      log2_and_remainder(One + (pow(Two, n1) * (Two * n2 + One) − One)) ==|
        assoc_plus_minus_one(n1) |
      log2_and_remainder((One + pow(Two, n1) * (Two * n2 + One)) − One) ==|
        commutative_plus(One, pow(Two, n1) * (Two * n2 + One)) |
      log2_and_remainder(pow(Two, n1) * (Two * n2 + One) + One − One) ==|
        additive_inverse(pow(Two, n1) * (Two * n2 + One), One) |
      log2_and_remainder(pow(Two, n1) * (Two * n2 + One)) ==|
        trivial |
      log2_and_remainder((Two * pow(Two, p1)) * (Two * n2 + One)) ==|
        associative_times(Two, pow(Two, p1), Two * n2 + One) |
      log2_and_remainder(Two * (pow(Two, p1) * (Two * n2 + One))) ==|
        additive_inverse(pow(Two, p1) * (Two * n2 + One), One) |
      log2_and_remainder(Two * (pow(Two, p1) * (Two * n2 + One) + One − One)) ==|
        commutative_plus(One, pow(Two, p1) * (Two * n2 + One)) |
      log2_and_remainder(Two * (One + pow(Two, p1) * (Two * n2 + One) − One)) ==|
        assoc_plus_minus_one(p1) |
      log2_and_remainder(Two * Succ(pow(Two, p1) * (Two * n2 + One) − One)) ==|
        trivial |
      log2_and_remainder(Two * Succ(pair(p1, n2))) ==| (
        times_two_even(Succ(pair(p1, n2))) &&
        project_1_inverse(p1, n2) &&
        commutative_times(Two, Succ(pair(p1, n2))) &&
        multiplicative_inverse(Succ(pair(p1, n2)), Two)) |
      (n1, Two * n2 + One)
    }.qed

  case _ ⇒
    {
      log2_and_remainder(Succ(pair(n1, n2))) ==|
        trivial |
          log2_and_remainder(Succ(pow(Two, n1) ∗ (Two ∗ n2 + One) − One)) ==|
            trivial |
          log2_and_remainder(Succ(Two ∗ n2 + One − One)) ==|
            additive_inverse(Two ∗ n2, One) |
          log2_and_remainder(Succ(Two ∗ n2)) ==|
            times_two_plus_one_odd(n2) |
          ((Zero, Succ(Two ∗ n2)): (Nat, Nat)) ==|
            commutative_plus(Two ∗ n2, One) |
          ((Zero, Two ∗ n2 + One): (Nat, Nat))
        }.qed
    })
  }.holds

  def inverse_lemma(n1: Nat, n2: Nat): Boolean = {
    project(pair(n1, n2)) == (n1, n2)
  }.holds because {
    val (p1, remainder) = log2_and_remainder(Succ(pair(n1, n2)))
    val p2 = (remainder − One) / Two

    project(pair(n1, n2)) ==| trivial |
    (p1, p2) ==| trivial |
    (p1, (remainder − One) / Two) ==| project_1_inverse(n1, n2) |
    (n1, ((Two ∗ n2 + One) − One) / Two) ==| additive_inverse(Two ∗ n2, One) |
    (n1, (Two ∗ n2) / Two) ==| commutative_times(Two, n2) |
    (n1, (n2 ∗ Two) / Two) ==| multiplicative_inverse(n2, Two) |
    (n1, n2)
  }.qed

  def pair_unique(n1: Nat, n2: Nat, n3: Nat, n4: Nat): Boolean = {
    (pair(n1, n2) == pair(n3, n4)) == ((n1, n2) == (n3, n4))
  }.holds because {
    if (pair(n1, n2) == pair(n3, n4)) {
      assert(project(pair(n1, n2)) == project(pair(n3, n4)))
      assert(inverse_lemma(n1, n2) && inverse_lemma(n3, n4))
      assert((n1, n2) == (n3, n4))
      ((n1, n2) == (n3, n4) ==| trivial | true).qed
    } else {
      assert((n1, n2) != (n3, n4))
      ((n1, n2) == (n3, n4) ==| trivial | false).qed
    }
  }
}
```

Bibliography

[Abe] Andreas Abel. *foetus* - Termination Checker for Simple Functional Programs. http://www.cse.chalmers.se/~abela/foetus. Retrieved on 23rd of February 2019.

[Abe04] Andreas Abel. Termination Checking with Types. *ITA*, 38(4):277–319, 2004.

[Abe07] Andreas Abel. *Type-Based Termination: A Polymorphic Lambda-Calculus with Sized Higher-Order Types.* PhD thesis, Ludwig Maximilians University Munich, 2007.

[Abe08] Andreas Abel. Semi-Continuous Sized Types and Termination. *Logical Methods in Computer Science*, 4(2), 2008.

[Abe10] Andreas Abel. MiniAgda: Integrating Sized and Dependent Types. In *Proceedings Workshop on Partiality and Recursion in Interactive Theorem Provers, PAR 2010, Edinburgh, UK, 15th July 2010.*, pages 14–28, 2010.

[Abe12] Andreas Abel. Type-Based Termination, Inflationary Fixed-Points, and Mixed Inductive-Coinductive Types. In *Proceedings 8th Workshop on Fixed Points in Computer Science, FICS 2012, Tallinn, Estonia, 24th March 2012.*, pages 1–11, 2012.

[Agu16] Alejandro Aguirre. Towards a provably correct encoding from F* to SMT. Technical report, INRIA, 2016.

[ALN15] Reza Ahmadi, K. Rustan M. Leino, and Jyrki Nummenmaa. Automatic Verification of Dafny Programs with Traits. In *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs, FTfJP 2015, Prague, Czech Republic, July 7, 2015*, pages 4:1–4:5, 2015.

[AWSS06] Markus Aderhold, Christoph Walther, Daniel Szallies, and Andreas Schlosser. A Fast Disprover for VeriFun. In *Proceedings of the Workshop on Non-Theorems, Non-Validity, Non-Provability (DISPROVING-06), Seattle, WA, 2006. Federated Logic Conference*, pages 59–69, 2006.

[BBCW18] Alexander Bentkamp, Jasmin Christian Blanchette, Simon Cruanes, and Uwe Waldmann. Superposition for Lambda-Free Higher-Order Logic. In *Automated*

Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, pages 28–46, 2018.

[BBPS16] Jasmin Christian Blanchette, Sascha Böhme, Andrei Popescu, and Nicholas Smallbone. Encoding Monomorphic and Polymorphic Types. *Logical Methods in Computer Science*, 12(4), 2016.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[BCD+05] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, pages 364–387, 2005.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 171–177, 2011.

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda - A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 73–78, 2009.

[BDT14] Richard Bonichon, David Déharbe, and Cláudia Tavares. Extending SMT-LIB v2 with λ-Terms and Polymorphism. In *Proceedings of the 12th International Workshop on Satisfiability Modulo Theories, SMT 2014, Vienna, Austria, July 17-18, 2014.*, pages 53–62, 2014.

[BFG+04] Gilles Barthe, Maria João Frade, Eduardo Giménez, Luís Pinto, and Tarmo Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14(1):97–141, 2004.

[BFT17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[Bjø] Nikolaj Bjørner. Z3 Recursive Function Integration. https://github.com/Z3Prover/z3/commit/c7d0d4e191f214cf60f0b75d3d64351d9a537b43. Retrieved on 30th of January 2019.

[BKKS13] Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. An Overview of the Leon Verification System: Verification by Translation to Recursive Functions.

In *Proceedings of the 4th Workshop on Scala, SCALA@ECOOP 2013, Montpellier, France, July 2, 2013*, pages 1:1–1:10, 2013.

[Bla17] Régis William Blanc. Verification by Reduction to Functional Programs. page 191, 2017.

[BN10] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, pages 131–146, 2010.

[Bou12] Pierre Boutillier. A relaxation of coq's guard condition. In *JFLA-Journées Francophones des langages applicatifs-2012*, pages 1–14, 2012.

[BR18] Ahmed Bhayat and Giles Reger. Set of Support for Higher-Order Reasoning. In *Proceedings of the 6th Workshop on Practical Aspects of Automated Reasoning co-located with Federated Logic Conference 2018 (FLoC 2018), Oxford, UK, July 19th, 2018.*, pages 2–16, 2018.

[Bra13] Edwin Brady. Idris: general purpose programming with dependent types. In *Proceedings of the 7th Workshop on Programming languages meets program verification, PLPV 2013, Rome, Italy, January 22, 2013*, pages 1–2, 2013.

[BRK$^+$15] Kshitij Bansal, Andrew Reynolds, Tim King, Clark W. Barrett, and Thomas Wies. Deciding Local Theory Extensions via E-matching. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 87–105, 2015.

[Bro12] Chad E. Brown. Satallax: An Automatic Higher-Order Prover. In *Automated Reasoning - 6th International Joint Conference, IJCAR 2012, Manchester, UK, June 26-29, 2012. Proceedings*, pages 111–117, 2012.

[BRO$^+$19] Haniel Barbosa, Andrew Reynolds, Daniel El Ouraoui, Cesare Tinelli, and Clark Barrett. Extending SMT Solvers to Higher-Order Logic. In *Computer-Aided Deduction (CADE)*, 2019.

[BS13] Christoph Benzmüller and Nik Sultana. LEO-II Version 1.5. In *Third International Workshop on Proof Exchange for Theorem Proving, PxTP 2013, Lake Placid, NY, USA, June 9-10, 2013*, pages 2–10, 2013.

[BW08] Achim D. Brucker and Burkhart Wolff. An Extensible Encoding of Object-oriented Data Models in HOL. *J. Autom. Reasoning*, 41(3-4):219–249, 2008.

[CB] Simon Cruanes and Jasmin Christian Blanchette. Nunchaku. https://github.com/nunchaku-inria/nunchaku. Retrieved on 30th of January 2019.

[CB16] Simon Cruanes and Jasmin Christian Blanchette. Extending Nunchaku to Dependent Type Theory. In *Proceedings First International Workshop on Hammers for Type Theories, HaTT@IJCAR 2016, Coimbra, Portugal, July 1, 2016.*, pages 3–12, 2016.

[CDMV11] Harsh Raju Chamarthi, Peter C. Dillinger, Panagiotis Manolios, and Daron Vroon. The ACL2 Sedan Theorem Proving System. In *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 291–295, 2011.

[CH88] Thierry Coquand and Gérard P. Huet. The Calculus of Constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

[CJRS15] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. TIP: Tons of Inductive Problems. In *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*, pages 333–337, 2015.

[CK16] Lukasz Czajka and Cezary Kaliszyk. Goal Translation for a Hammer for Coq (Extended Abstract). In *Proceedings First International Workshop on Hammers for Type Theories, HaTT@IJCAR 2016, Coimbra, Portugal, July 1, 2016.*, pages 13–20, 2016.

[CK18] Lukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for Dependent Type Theory. *J. Autom. Reasoning*, 61(1-4):423–453, 2018.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 168–176, 2004.

[CM11] Harsh Raju Chamarthi and Panagiotis Manolios. Automated specification analysis using an interactive theorem prover. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 46–53, 2011.

[CMST10] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. Local Verification of Global Invariants in Concurrent Programs. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 480–494, 2010.

[CP88] Thierry Coquand and Christine Paulin. Inductively defined types. In *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, pages 50–66, 1988.

[CR] Koen Claessen and Dan Rosén. Haskell Bounded Model Checker. https://github.com/danr/hbmc. Retrieved on 5th of November 2018.

[Cza16] Lukasz Czajka. A Shallow Embedding of Pure Type Systems into First-Order Logic. In *22nd International Conference on Types for Proofs and Programs, TYPES 2016, May 23-26, 2016, Novi Sad, Serbia*, pages 9:1–9:39, 2016.

[dMB07] Leonardo Mendonça de Moura and Nikolaj Bjørner. Efficient E-Matching for SMT Solvers. In *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, pages 183–198, 2007.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[dMKA$^+$15] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 378–388, 2015.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[Dun07] Joshua Dunfield. Refined Typechecking with Stardust. In *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*, pages 21–32, 2007.

[Ede] Romain Edelmann. Interactive Theorem Proving based on Inox. https://github.com/epfl-lara/welder. Retrieved on 27th of February 2019.

[EMT$^+$17] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, pages 126–133, 2017.

[FM07] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 173–177, 2007.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - Where Programs Meet Provers. In *Programming Languages and Systems - 22nd European Symposium* *on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 125–128, 2013.

[FZG18] Grigory Fedyukovich, Yueling Zhang, and Aarti Gupta. Syntax-guided termination analysis. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, pages 124–143, 2018.

[GdM09] Yeting Ge and Leonardo Mendonça de Moura. Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, pages 306–320, 2009.

[GF10] Andrew D. Gordon and Cédric Fournet. Principles and Applications of Refinement Types. In *Logics and Languages for Reliability and Security*, pages 73–104. 2010.

[Gir90] Jean-Yves Girard. *Proofs and Types.* Cambridge University Press, 1990.

[GRS$^+$11] Jürgen Giesl, Matthias Raffelsieper, Peter Schneider-Kamp, Stephan Swiderski, and René Thiemann. Automated Termination Proofs for Haskell by Term Rewriting. *ACM Trans. Program. Lang. Syst.*, 33(2):7:1–7:39, 2011.

[Gru] Samuel Gruetter. Counterexamples for Coq Conjectures: A survey of existing work and a call for action. https://popl19.sigplan.org/event/coqpl-2019-counterexamples-for-coq-conjectures. Retrieved on 14th of February 2019.

[GT16] Gudmund Grov and Vytautas Tumas. Tactics for the Dafny Program Verifier. In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 36–53, 2016.

[GTSF04] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Automated Termination Proofs with AProVE. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, pages 210–220, 2004.

[Har16] Robert Harper. *Practical foundations for programming languages.* Cambridge University Press, 2016.

[HLQ11] Luke Herbert, K. Rustan M. Leino, and Jose Quaresma. Using Dafny, an Automatic Program Verifier. In *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, pages 156–181, 2011.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 410–423, 1996.

[IYG$^+$08] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient SAT-based bounded model checking for software verification. *Theor. Comput. Sci.*, 404(3):256–274, 2008.

[JH] Romain Jufer and Jad Hamza. Verification and Generation of Smart Contracts using Stainless and Scala. https://github.com/epfl-lara/smart. Retrieved on 27th of February 2019.

[JPT18] Rohan Jacob-Rao, Brigitte Pientka, and David Thibodeau. Index-Stratified Types. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, pages 19:1–19:17, 2018.

[KAE$^+$10] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

[KGGT07] Sava Krstic, Amit Goel, Jim Grundy, and Cesare Tinelli. Combined satisfiability modulo parametric theories. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 602–617, 2007.

[KKKS13] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis Modulo Recursive Functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 407–426, 2013.

[KKS11] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Scala to the Power of Z3: Integrating SMT and Programming. In *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*, pages 400–406, 2011.

[KMM13] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-aided reasoning: ACL2 case studies*, volume 4. Springer Science & Business Media, 2013.

[KMPS12] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. Software Synthesis Procedures. *Commun. ACM*, 55(2):103–111, 2012.

[Kob09] Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 416–428, 2009.

[KSU11] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. Predicate Abstraction and CEGAR for Higher-Order Model Checking. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 222–233, 2011.

[KTU10] Naoki Kobayashi, Naoshi Tabuchi, and Hiroshi Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 495–508, 2010.

[KTUK14] Takuya Kuwahara, Tachio Terauchi, Hiroshi Unno, and Naoki Kobayashi. Automatic Termination Verification for Higher-Order Functional Programs. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 392–411, 2014.

[KZK$^+$18] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. Mtac2: Typed Tactics for Backward Reasoning in Coq. *PACMPL*, 2(ICFP):78:1–78:31, 2018.

[Lei08] K Rustan M Leino. This is boogie 2. *Manuscript KRML*, 178(131), 2008.

[Lei10] K. Rustan M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, pages 348–370, 2010.

[LJB01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The Size-Change Principle for Program Termination. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 81–92, 2001.

[LLM11] Claire Le Goues, K. Rustan M. Leino, and Michal Moskal. The Boogie Verification Debugger (Tool Paper). In *Software Engineering and Formal Methods - 9th International Conference, SEFM 2011, Montevideo, Uruguay, November 14-18, 2011. Proceedings*, pages 407–414, 2011.

[LM08] K. Rustan M. Leino and Peter Müller. Using the Spec# Language, Methodology, and Tools to Write Bug-Free Programs. In *Advanced Lectures on Software Engineering, LASER Summer School 2007/2008*, pages 91–139, 2008.

[LP16] K. Rustan M. Leino and Clément Pit-Claudel. Trigger Selection Strategies to Stabilize Program Verifiers. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, pages 361–381, 2016.

[LQ13] Akash Lal and Shaz Qadeer. Reachability Modulo Theories. In *Reachability Problems - 7th International Workshop, RP 2013, Uppsala, Sweden, September 24-26, 2013 Proceedings*, pages 23–44, 2013.

[LR12] Ruslán Ledesma-Garza and Andrey Rybalchenko. Binary Reachability Analysis of Higher Order Functional Programs. In *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, pages 388–404, 2012.

[MAD$^+$19] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F*: Proof Automation with SMT, Tactics, and Metaprograms. In *28th European Symposium on Programming (ESOP)*, 2019. To appear.

[Man13] Panagiotis Manolios. Counterexample Generation Meets Interactive Theorem Proving: Current Results and Future Opportunities. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, page 18, 2013.

[MKK17] Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. Contract-Based Resource Verification for Higher-Order Functions with Memoization. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 330–343, 2017.

[MLS84] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[MP08] Jia Meng and Lawrence C. Paulson. Translating Higher-Order Clauses to First-Order Clauses. *J. Autom. Reasoning*, 40(1):35–60, 2008.

[MSS16] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, pages 41–62, 2016.

[MSS17] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Dependable Software Systems Engineering*, pages 104–125. 2017.

[MT09] Panagiotis Manolios and Aaron Turon. All-Termination(T). In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference,* *TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 398–412, 2009.

[MW71] Zohar Manna and Richard J. Waldinger. Toward Automatic Program Synthesis. *Commun. ACM*, 14(3):151–165, 1971.

[NH15] Phuc C. Nguyen and David Van Horn. Relatively complete counterexamples for higher-order programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 446–456, 2015.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[NTH17] Phuc C. Nguyen, Sam Tobin-Hochstadt, and David Van Horn. Higher order symbolic execution for contract verification and refutation. *J. Funct. Program.*, 27:e3, 2017.

[Ode10] Martin Odersky. Contracts for Scala. In *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, pages 51–57, 2010.

[OR11] C.-H. Luke Ong and Steven J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 587–598, 2011.

[Par98] Lars Pareto. *Sized types*. Citeseer, 1998.

[PO15] Aleksandar Prokopec and Martin Odersky. Conc-Trees for Functional and Parallel Programming. In *Languages and Compilers for Parallel Computing - 28th International Workshop, LCPC 2015, Raleigh, NC, USA, September 9-11, 2015, Revised Selected Papers*, pages 254–268, 2015.

[RBCT16] Andrew Reynolds, Jasmin Christian Blanchette, Simon Cruanes, and Cesare Tinelli. Model Finding for Recursive Functions in SMT. In *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, pages 133–151, 2016.

[Rey98] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[RKJ08] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid Types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169, 2008.

[RKT+17] Andrew Reynolds, Viktor Kuncak, Cesare Tinelli, Clark Barrett, and Morgan Deters. Refutation-Based Synthesis in SMT. *Formal Methods in System Design (FMSD)*, 2017.

[RTB17] Andrew Reynolds, Cesare Tinelli, and Clark Barrett. Constraint solving for finite model finding in SMT solvers. *TPLP*, 17(4):516–558, 2017.

[RTG+13] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstic, Morgan Deters, and Clark Barrett. Quantifier Instantiation Techniques for Finite Model Finding in SMT. In *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, pages 377–391, 2013.

[RTGK13] Andrew Reynolds, Cesare Tinelli, Amit Goel, and Sava Krstic. Finite Model Finding in SMT. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 640–655, 2013.

[Rue18] Romain Ruetschi. Systems Modeling With Stainless, 2018.

[Rüm08] Philipp Rümmer. A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic. In *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22-27, 2008. Proceedings*, pages 274–289, 2008.

[SAK15] Ryosuke Sato, Kazuyuki Asada, and Naoki Kobayashi. Refinement Type Checking via Assertion Checking. *JIP*, 23(6):827–834, 2015.

[SB18] Alexander Steen and Christoph Benzmüller. The Higher-Order Prover Leo-III. In *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, pages 108–116, 2018.

[SHK+16] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent Types and Multi-Monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270, 2016.

[SK16] Georg Stefan Schmid and Viktor Kuncak. SMT-Based Checking of Predicate-Qualified Types for Scala. In *Proceedings of the 7th ACM SIGPLAN Symposium on Scala, SCALA@SPLASH 2016, Amsterdam, Netherlands, October 30 - November 4, 2016*, pages 31–40, 2016.

[SKK11] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *SAS*, 2011.

[SWS$^+$13] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying Higher-order Programs with the Dijkstra Monad. In *Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '13, pages 387–398, 2013.

[Tag08] Mana Taghdiri. *Automating modular program verification by refining specifications*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2008.

[Tai67] W. W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

[Ter10] Tachio Terauchi. Dependent types from counterexamples. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 119–130, 2010.

[TJ07] Mana Taghdiri and Daniel Jackson. Inferring specifications to detect errors in code. *Autom. Softw. Eng.*, 14(1):87–121, 2007.

[TS95] Tanel Tammet and Jan M. Smith. Optimized Encodings of Fragments of Type Theory in First Order Logic. In *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, pages 265–287, 1995.

[VKK15] Nicolas Voirol, Etienne Kneuss, and Viktor Kuncak. Counter-Example Complete Verification for Higher-Order Functions. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala, Scala@PLDI 2015, Portland, OR, USA, June 15-17, 2015*, pages 18–29, 2015.

[VRJ13] Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. Abstract Refinement Types. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 209–228, 2013.

[VSJ14a] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. LiquidHaskell: experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, pages 39–51, 2014.

[VSJ$^+$14b] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 269–282, 2014.

[VTC$^+$18] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: complete verification with SMT. *PACMPL*, 2(POPL):53:1–53:31, 2018.

[W$^+$04] Makarius Wenzel et al. The Isabelle/Isar Reference Manual, 2004.

[WAS06] Christoph Walther, Markus Aderhold, and Andreas Schlosser. The L 1.0 primer. Technical report, Technical Report VFR 06/01, Technische Universität Darmstadt, 2006.

[Wer94] Benjamin Werner. *Une Théorie des Constructions Inductives*. PhD thesis, Paris Diderot University, France, 1994.

[WS03] Christoph Walther and Stephan Schweitzer. About VeriFun. In *Automated Deduction - CADE-19, 19th International Conference on Automated Deduction Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, pages 322–327, 2003.

[Xi01] Hongwei Xi. Dependent Types for Program Termination Verification. In *16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, USA, June 16-19, 2001, Proceedings*, pages 231–242, 2001.

Curriculum Vitae

Nicolas Voirol

Le Carroz 26
1867, Ollon
Switzerland
✉ *voirol.nicolas@gmail.com*

Education

2014 – 2019	**PhD in CS**, *EPF*, Lausanne. Thesis title: *Verifying Functional Programs*. Advisor: Viktor Kuncak.
2011 – 2014	**Masters in CS**, *EPF*, Lausanne. Thesis title: *Termination Analysis in a Higher-Order Functional Context*. Supervisors: Etienne Kneuss and Viktor Kuncak.
2008 – 2011	**Bachelor in CS**, *EPF*, Lausanne. Exchange in 3rd year at the ETH Zurich.

Experience

2017 – 2018 8 months	**Software Engineering Intern**, *Google LLC*, Zurich. Built a news summarization system which extracts short, speakable summary sentences from news articles in realtime.
2014 4 months	**Software Engineer**, *Lightbend Inc. (formerly Typesafe)*, Lausanne. Prototyped a fast and modular linting framework for Scala (scala-abide) based on simple recurrent rule patterns.
2013 part-time 20%	**Mobile Application Developer**, *Artegis SARL*, Nyon. Combined multiple high-level technologies (GWT, SmartGWT, MGWT, Cordova) to implement a cross-platform mobile app that compiles against pre-existing GWT services to ensure compatibility with legacy web application.
2012 – 2013 2.5 months	**WUI Developer**, *Temenos SA*, Crissier. Improved the Web User Interface of a large scale international financial application by updating the chart-generating component to facilitate support and enhance performance.
2011 – 2012 part-time 60% all year long	**Research Assistant**, *EPF*, Lausanne. Contributed to the developement of an air pollution prediction model with mobile sensors ◦ Fully automated data importing with sources ranging from remote html to xsl spreadsheets. ◦ Map-matching system mapping low precision GPS coordinates to OSM roads. ◦ Region based model generation targeting prediction learning through baysian inference.
2010 – 2011 4 months over major holidays	**Web Application Developer**, *Artegis SARL*, Nyon. Application of GWT technology to update event management application to Web 2.0 ◦ SmartGWT elements (MetaData driven form generation, auto-loading lists). ◦ Social networking tools (chat-room, forum, internal messaging, profile management).

Publications

2015	N. Voirol, E. Kneuss, V. Kuncak, Counter-Example Complete Verification for Higher-Order Functions, *SCALA'15*.

Ce document a été imprimé au Centre d’impression EPFL,
imprimerie climatiquement neutre, certifiée myClimate.