Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

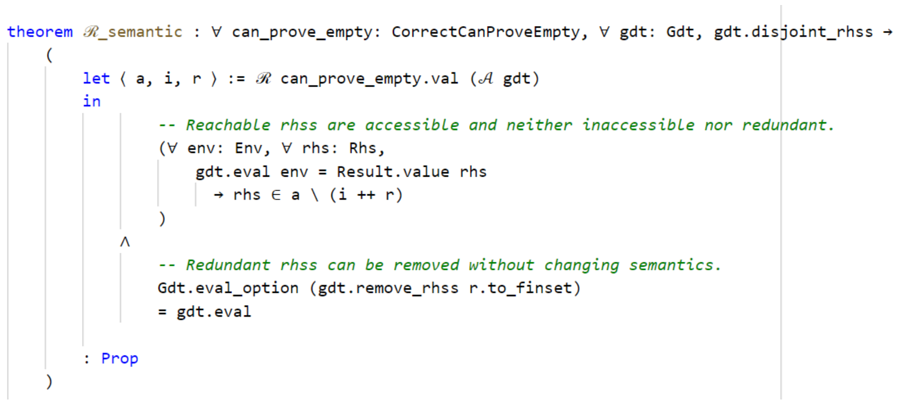
Lehrstuhl Prof. Dr.-Ing. Snelting

# Formal Verification of Pattern Matching Analyses

Masterarbeit von

**Henning Dieterichs**

an der Fakultät für Informatik



```
theorem ℛ_semantic : ∀ can_prove_empty: CorrectCanProveEmpty, ∀ gdt: Gdt, gdt.disjoint_rhss →
    (
        let ⟨ a, i, r ⟩ := ℛ can_prove_empty.val (𝒜 gdt)
        in
                -- Reachable rhss are accessible and neither inaccessible nor redundant.
                (∀ env: Env, ∀ rhs: Rhs,
                    gdt.eval env = Result.value rhs
                      → rhs ∈ a \ (i ++ r)
                )
            ∧
                -- Redundant rhss can be removed without changing semantics.
                Gdt.eval_option (gdt.remove_rhss r.to_finset)
                = gdt.eval

        : Prop
    )
```

| | |
|---|---|
| **Erstgutachter:** | Prof. Dr.-Ing. Gregor Snelting |
| **Zweitgutachter:** | Prof. Dr. rer. nat. Bernhard Beckert |
| **Betreuende Mitarbeiter:** | M. Sc. Sebastian Graf |
| | |
| **Abgabedatum:** | **?today?** |

# Zusammenfassung

Deutsche Übersetzung einfügen

The algorithms presented in Lower Your Guards [1] analyze pattern matching expressions and detect uncovered cases and inaccessible or redundant right hand sides. They already have been implemented in the Glasgow Haskell Compiler and spotted previously unknown bugs in real world code. While these algorithms have been validated empirically, their correctness has not been precisely defined nor proven yet.

This thesis establishes a precise notion of correctness and presents formal proofs that these algorithms are indeed correct. These proofs are formalized in Lean 3.

# Contents

# 1 Introduction

In functional programming, pattern matching is a very popular feature. This is particularly true for Haskell, where you can define algebraic data types and easily match on them in function definitions. With increasingly complex data types and function definitions however, pattern matching can be yet another source of mistakes.

Figure 1.1 showcases common types of mistakes that can arise with pattern matching.

Most importantly, the function `f` is not defined on all values: Evaluating `f Case4` will cause a runtime error! In other words, the pattern used to define `f` is not exhaustive, as the input `Case4` is uncovered. This is usually an oversight by the programmer and should be avoided.

Also, `f` will never evaluate to 3 or 4 - replacing these values with any other value would not change any observable behaviour of `f`. Such right hand sides (*RHS*s) are called *inaccessible*. Inaccessible RHSs indicate a code smell and should be avoided too. Sometimes, such RHSs can simply be removed from the pattern.

**Figure 1.1:** A Pattern Matching Example In Haskell

```haskell
data Case = Case1 | Case2 | Case3 | Case4

f :: Case -> Bool -> Integer
f Case1 _ = 1
f Case2 _ = 2
f x True | Case1 <- x = 3
         | Case2 <- x = 4
f Case3 _ = 5
```

Lower Your Guards (LYG) [1] provides robust algorithms that are able to detect such mistakes and also can deal with the intricacies of lazy evaluation.

However, the algorithms presented in LYG are only checked empirically so far: Their implementations in the Glasgow Haskell Compiler just *seem to work*.

Obviously, these algorithms would be incorrect if they mark a RHS as inaccessible even though it actually is accessible. This could have fatal consequences: A novice programmer acting on such misinformation might delete a RHS that is very much in use!

As LYG does not give a complete characterization of correctness, we first want to establish a precise notion of correctness and then check that these algorithms indeed comply with it. At the very least, a verifying tool should be verified itself!

The large number of case distinctions made in the algorithms motivates the use of a theorem prover; a natural proof would not be very trustworthy due to the high technical demand and risk of missing edge cases.

In chapter 2, we give a summary of the relevant algorithms defined in LYG. Then, in chapter 3, we present and discuss our formalization of LYG and of several correctness statements. Finally, in chapter 4, we present an overview of some of our formal correctness proofs.

We contribute:

- An abstract formalization of LYG, specified in Lean 3
- A notion of correctness of LYG and its formalization
- Formal proofs that LYG is correct
- A problematization of the variable binding mechanism in LYG refinement types

# 2 Background

## 2.1 Lower Your Guards

Lower Your Guards (LYG) [1] describes algorithms that analyze pattern matching expressions and report uncovered cases, but also redundant and inaccessible right hand sides.

LYG was designed for use in the Glasgow Haskell Compiler, but the algorithm and its data structures are so universal that they can be leveraged for other programming languages with pattern matching constructs too.

All definitions and some examples of this chapter are taken from LYG [1].

### 2.1.1 Inaccessible vs. Redundand RHSs

A closer look at figure 1.1 reveals that while both RHS 3 and 4 are inaccessible, the semantics of `f` changes if both are removed. This means that an automated refactoring cannot just remove all inaccessible leaves!

The reason for this is the term $t :=$ `f Case3 undefined` and the fact that Haskell uses a lazy evaluation strategy. If both RHSs 3 and 4 are removed, $t$ evaluates to 5 - the term `undefined` is never evaluated as no pattern matches against it. However, if nothing or only one of the RHSs 3 or 4 is removed, `undefined` will be matched with `True` and thus $t$ will throw a runtime error!

To communicate this difference, LYG introduces the concept of *redundant* and *inaccessible* RHSs: A redundant RHS can be removed from its pattern matching expression without any observable difference. An inaccessible RHS is never evaluated, but its removal might lead to observable changes. This definition implies that redundant RHSs are inaccessible.

As of listing 1.1, LYG will mark RHS 3 as inaccessible and RHS 4 as redundant. This choice is somewhat arbitrary, as RHS 3 could be marked as redundant and RHS 4 as inaccessible as well, and will be discussed in more detail chapter ref.

### 2.1.2 Guard Trees

For all analyses, LYG first transforms Haskell specific pattern match expressions to simpler *guard trees*. This transformation removes a lot of complexity, as many different Haskell constructs can be desugared to the same guard tree. Guard trees also simplify adapting LYG to other programming languages and they enable studying LYG mostly independent from Haskell. Their syntax is defined in figure 2.1.

Guard trees (*Gdt*s) are made of three elements: Uniquely numbered right hand sides, *branches* and *guarded trees*. Guarded trees refer to Haskell specific guards (*Grd*) that control the execution. *Let guards* can bind a term to a variable in a new lexical scope. *pattern match guards* can destructure a value into variables if the pattern matches or otherwise prevent the execution from entering the tree behind the guard. Finally, *bang guards* can stop the entire execution when the value of a variable does not reduce to a head normal form.

**Figure 2.1:** Definition of Guard Trees

**Guard Syntax**

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

**Guard Tree Syntax**

$$
t \in \mathsf{Gdt} \qquad \longrightarrow k \mid \left\lfloor \begin{array}{l} t_1 \\ t_2 \end{array}\right. \mid -\!\!\dashv g \longrightarrow t
$$

The evaluation of a guard tree selects the first right hand side that execution reaches. If the execution stops at a bang guard, the evaluation is said to *diverge*, otherwise, if execution falls through, the evaluation ends with a *no-match*. A formal semantic for guard trees will be defined in chapter 3.1.2.

The transformation from Haskell pattern matches to guard trees is not of much interest for this thesis and can be found in LYG [1]. To preserve semantics, it is important that the transformation inserts bang guards whenever a variable is matched against a data constructor.

Figure 2.2 presents the transformation of figure 1.1 into a guard tree.

It is usually straightforward to define a transformation from pattern matching expressions to guard trees that also preserves uncovered cases and inaccessible and redundant RHSs. This makes guard trees an ideal abstraction for the following analysis steps.

### 2.1.3 Refinement Types

*Refinement types* describe vectors of values $x_1, ..., x_n$ that satisfy a given predicate $\Phi$. Their syntax is defined in figure 2.3.

Figure 2.2: Desugaring Example

```
data Case = Case1 | Case2 | Case3 | Case4

f :: Case -> Bool -> Integer
f Case1 _ = 1
f Case2 _ = 2
f x True | Case1 <- x = 3
         | Case2 <- x = 4
f Case3 _ = 5
```

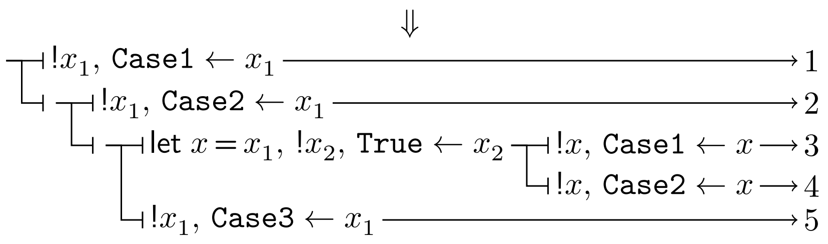

$\Downarrow$
$!x_1$, Case1 $\leftarrow x_1$ $\longrightarrow$ 1
$!x_1$, Case2 $\leftarrow x_1$ $\longrightarrow$ 2
let $x = x_1$, $!x_2$, True $\leftarrow x_2$
$!x$, Case1 $\leftarrow x$ $\longrightarrow$ 3
$!x$, Case2 $\leftarrow x$ $\longrightarrow$ 4
$!x_1$, Case3 $\leftarrow x_1$ $\longrightarrow$ 5

Figure 2.3: Definition of Refinement Types

| | | |
|---|---|---|
| $\Gamma$ | $\emptyset \mid \Gamma, x : \tau \mid \Gamma, a$ | Context |
| $\varphi$ | $\checkmark \mid \times \mid K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x \mid x \not\approx K$ | |
| | $\mid x \approx \bot \mid x \not\approx \bot \mid \mathsf{let}\ x = e$ | Literals |
| $\Phi$ | $\varphi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi$ | Formula |
| $\Theta$ | $\langle \Gamma \mid \Phi \rangle$ | Refinement type |

Refinement types are built from literals $\phi$ and closed under conjunction and disjunction. The literal $\checkmark$ refers to "true", while $\times$ refers to "false". For example:

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

Unconventionally, a literal can bind one or more variables in a way that such a binding is in scope in all literals on its right. Thus, $(\mathsf{let}\ x = y \wedge z \not\approx \bot) \wedge x \not\approx \bot$ is semantically equivalent to $z \not\approx \bot \wedge y \not\approx \bot$. While this conjunction operator is still associative, it is clearly not commutative!

LYG also describes a partial function $\mathcal{G}$ with $\mathcal{G}(\Theta) = \emptyset \Rightarrow (\Theta \text{ denotes } \emptyset)$ for all refinement types $\Theta$. $\mathcal{G}$ is used to $\mathcal{G}$enerate inhabitants of a refinement type to build elaborate error messages and to get a guarantee that a refinement type is empty. A total correct function $\mathcal{G}$ is uncomputable, since refinement types can make use of recursively defined functions! This thesis just assumes that "interesting" computable and correct functions $\mathcal{G}$ exist, so the details of $\mathcal{G}$ as proposed by LYG do not matter. In general, all proposed correctness statements should allow for an empty function $G$.

### 2.1.4 Uncovered Analysis

The goal of the uncovered analysis is to detect all cases that are not handled by a given guard tree. Refinement types are used to capture the result of this analysis.

The function $\mathcal{U}(\langle\, \Gamma \mid \checkmark \,\rangle, \cdot)$ in figure 2.4 computes a refinement type that captures all uncovered values for a given guard tree. This refinement type is empty if and only if there are not any uncovered cases. If $\mathcal{G}$ is used to test for emptiness, this already yields an algorithm to test for uncovered cases. It can be verified that the uncovered refinement type of the guard tree in figure 2.2 "semantically" equals $\langle\, x_1{:}\mathit{Case},\ x_2{:}\mathit{Bool} \mid x_1 \not\approx \bot \mathbin{\dot\wedge} x_1 \not\approx \texttt{Case1} \mathbin{\dot\wedge} x_1 \not\approx \texttt{Case2} \mathbin{\dot\wedge} x_1 \not\approx \texttt{Case3} \,\rangle$ and denotes $x_1 = \texttt{Case4}$.

**Figure 2.4:** Definition of $\mathcal{U}$

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

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

### 2.1.5 Annotated Guard Trees

*Annotated guard trees* represent simplified guard trees that have been annotated with refinement types $\Theta$. They are made of RHSs, branches and bang nodes. Their syntax is defined in figure 2.5.

**Figure 2.5:** Definition of Annotated Guard Trees

$$u \in \mathsf{Ant} \quad \longrightarrow \Theta\, k \mid \begin{smallmatrix} \ulcorner u_1 \\ \llcorner u_2 \end{smallmatrix} \mid -\!\!-\Theta\,\lightning\!-\!\!-\, u$$

### 2.1.6 Redundant/Inaccessible Analysis

The goal of the redundant/inaccessible analysis is to report as much RHSs as possible that are redundant or inaccessible. This is done by annotating a guard tree with refinement types and then checking these refinement types for emptiness. If a RHS is associated with an empty refinement type, the RHS is inaccessible and in some circumstances even redundant. The refinement type of a bang node describes all values under which an evaluation will diverge. Figure 2.6 defines a function $\mathcal{A}$ that computes such an annotation for a given guard tree. Figure 2.7 shows the annotated tree of the introductory example in figure 1.1 with abbreviated refinement types.

Such an annotated guard tree is then passed to a function $\mathcal{R}$ as defined in figure 2.8. $\mathcal{R}$ uses $\mathcal{G}$ to compute redundant and inaccessible RHSs. All other RHSs are assumed to be accessible, even though, due to $\mathcal{G}$ being a partial function, not all of them actually are accessible.

Figure 2.9 computes inaccessible and redundant leaves for an annotated tree that is ($\mathcal{G} = \emptyset$)-equivalent to the annotated tree from figure 2.7 for sensible functions $\mathcal{G}$. It states that RHS 4 in 1.1 is redundant and can be removed, while RHS 3 is just inaccessible.

**Figure 2.6:** Definition of $\mathcal{A}$

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

$$\begin{array}{lcl}
\mathcal{A}(\Theta,\ \longrightarrow n\,) & = & \longrightarrow \Theta\, n \\
\mathcal{A}(\Theta,\ \left[\begin{array}{l} t_1 \\ t_2 \end{array}\right.) & = & \left[\begin{array}{l} \mathcal{A}(\Theta, t_1) \\ \mathcal{A}(\mathcal{U}(\Theta, t_1), t_2) \end{array}\right. \\
\mathcal{A}(\Theta,\ \dashv !x \longrightarrow t\,) & = & \longrightarrow \Theta \mathbin{\dot\wedge} (x \approx \bot)\,\lightning \longrightarrow \mathcal{A}(\Theta \mathbin{\dot\wedge} (x \not\approx \bot), t) \\
\mathcal{A}(\Theta,\ \dashv \mathsf{let}\ x = e \longrightarrow t\,) & = & \mathcal{A}(\Theta \mathbin{\dot\wedge} (\mathsf{let}\ x = e), t) \\
\mathcal{A}(\Theta,\ \dashv K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x \longrightarrow t\,) & = & \mathcal{A}(\Theta \mathbin{\dot\wedge} (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x), t)
\end{array}$$

**Figure 2.7:** Examplary Evaluation of $\mathcal{A}$

$$\mathcal{A}\left(\left[\begin{array}{l}
\dashv !x_1,\ \texttt{Case1} \leftarrow x_1 \longrightarrow 1 \\
\left[\begin{array}{l}
\dashv !x_1,\ \texttt{Case2} \leftarrow x_1 \longrightarrow 2 \\
\left[\begin{array}{l}
\dashv \mathsf{let}\ x = x_1,\ !x_2,\ \texttt{True} \leftarrow x_2 \left[\begin{array}{l} \dashv !x,\ \texttt{Case1} \leftarrow x \longrightarrow 3 \\ \dashv !x,\ \texttt{Case2} \leftarrow x \longrightarrow 4 \end{array}\right. \\
\dashv !x_1,\ \texttt{Case3} \leftarrow x_1 \longrightarrow 5
\end{array}\right.
\end{array}\right.
\end{array}\right.\right) =$$

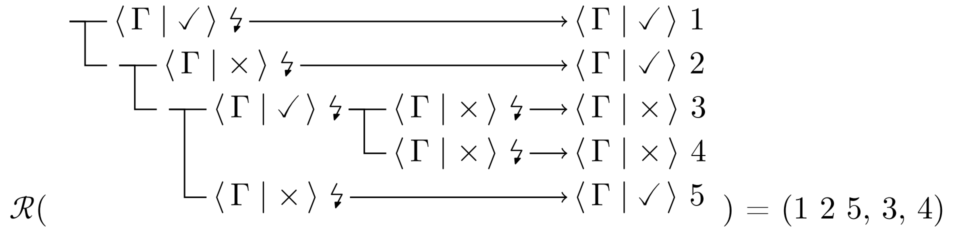
$$\left[\begin{array}{l}
\langle \Gamma \mid x_1 \approx \bot \rangle\,\lightning \longrightarrow \langle \Gamma \mid x_1 \not\approx \bot, x_1 \approx \texttt{Case1} \rangle\ 1 \\
\left[\begin{array}{l}
\langle \Gamma \mid x_1 \not\approx \bot, x_1 \approx \bot \rangle\,\lightning \longrightarrow \langle \Gamma \mid ..., x_1 \not\approx \texttt{Case1}, x_1 \approx \texttt{Case2} \rangle\ 2 \\
\left[\begin{array}{l}
\langle \Gamma \mid x_2 \approx \bot \rangle\,\lightning \left[\begin{array}{l} \langle \Gamma \mid \times \rangle\,\lightning \longrightarrow \langle \Gamma \mid ..., x_1 \not\approx \texttt{Case1}, ..., x_1 \approx \texttt{Case1} \rangle\ 3 \\ \langle \Gamma \mid \times \rangle\,\lightning \longrightarrow \langle \Gamma \mid ..., x_1 \not\approx \texttt{Case2}, ..., x_1 \approx \texttt{Case2} \rangle\ 4 \end{array}\right. \\
\langle \Gamma \mid \times \rangle\,\lightning \longrightarrow \langle \Gamma \mid ..., x_1 \approx \texttt{Case3} \rangle\ 5
\end{array}\right.
\end{array}\right.
\end{array}\right.$$

**Figure 2.8:** Definition of $\mathcal{R}$. $\mathcal{R}$ partitions all RHSs into probably-accessible ($\overline{k}$), inaccessible ($\overline{n}$) and $\mathcal{R}$edundant ($\overline{m}$) RHSs.

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

$$\begin{array}{lcl}
\mathcal{R}(\longrightarrow \Theta\, n\,) & = & \begin{cases} (\epsilon, \epsilon, n), & \text{if } \mathcal{G}(\Theta) = \emptyset \\ (n, \epsilon, \epsilon), & \text{otherwise} \end{cases} \\
\mathcal{R}(\left[\begin{array}{l} t \\ u \end{array}\right.) & = & (\overline{k}\,\overline{k'}, \overline{n}\,\overline{n'}, \overline{m}\,\overline{m'}) \text{ where } \begin{array}{ll} (\overline{k}, \overline{n}, \overline{m}) & = \mathcal{R}(t) \\ (\overline{k'}, \overline{n'}, \overline{m'}) & = \mathcal{R}(u) \end{array} \\
\mathcal{R}(\longrightarrow \Theta\,\lightning \longrightarrow t\,) & = & \begin{cases} (\epsilon, m, \overline{m'}), & \text{if } \mathcal{G}(\Theta) \neq \emptyset \text{ and } \mathcal{R}(t) = (\epsilon, \epsilon, m\,\overline{m'}) \\ \mathcal{R}(t), & \text{otherwise} \end{cases}
\end{array}$$

**Figure 2.9:** Examplary Evaluation of $\mathcal{R}$


$\langle \Gamma \mid \checkmark \rangle \lightning \longrightarrow \langle \Gamma \mid \checkmark \rangle$ 1
$\langle \Gamma \mid \times \rangle \lightning \longrightarrow \langle \Gamma \mid \checkmark \rangle$ 2
$\langle \Gamma \mid \checkmark \rangle \lightning$ $\langle \Gamma \mid \times \rangle \lightning \longrightarrow \langle \Gamma \mid \times \rangle$ 3
$\langle \Gamma \mid \times \rangle \lightning \longrightarrow \langle \Gamma \mid \times \rangle$ 4
$\mathcal{R}($
$\langle \Gamma \mid \times \rangle \lightning \longrightarrow \langle \Gamma \mid \checkmark \rangle$ 5
$) = (1\ 2\ 5,\ 3,\ 4)$

## 2.2 Lean

### 2.2.1 The Lean Theorem Prover

Lean is an interactive theorem prover that is based on the calculus of inductive constructions [2] [3] and is developed by Microsoft Research. It features dependent types, offers a high degree of automation through tactics and can also be used as a programming language. Due to the Curry-Howard isomorphism, writing functional definitions intended to be used in proofs, writing proofs and writing proof-generating custom tactics is very similar.

We use Lean 3 for this thesis and want to give a brief overview of its syntax. See [4] for a detailed documentation.

Inductive data types can be defined with the keyword `inductive`. The #`check` instruction can be used to type-check terms:

```
inductive my_nat : Type
| zero : my_nat
| succ : my_nat → my_nat

#check my_nat.succ my_nat.zero
#check my_nat.zero.succ -- equivalent term, using dot notation
```

The keyword `def` can be used to bind terms and define recursive functions:

```
def my_nat.add : my_nat → my_nat → my_nat
-- Patterns can be used in definitions
| my_nat.zero b := b
| (my_nat.succ a) b := (a.add b).succ
```

Likewise, `def` can be used to bind proof terms to propositions. Propositions are stated as type and proved by constructing a term of that type. Π-types are used to introduce generalized type variables:

```
-- This type states that for all a, a + zero = a
def my_nat.add_zero_eq : Π a: my_nat, a.add my_nat.zero = a :=
    -- Proof by induction
    @my_nat.rec
        -- Induction Hypothesis
        (λ a, a.add my_nat.zero = a)
        -- Case Zero
        (my_nat.add.equations._eqn_1 my_nat.zero)
        -- Case Succ
        (λ a h,
            @eq.subst my_nat
                (λ x, (my_nat.succ a).add my_nat.zero = x.succ)
                (a.add my_nat.zero)
                a
                h
                (my_nat.add.equations._eqn_2 a my_nat.zero)
        )
```

Proofs are usually much shorter when using Leans tactic mode. Also, definitions can be parametrized (which generalizes the parameter) and the keywords `lemma` and `theorem` can be used instead of `def`:

```
lemma my_nat.add_zero' (a: my_nat): a.add my_nat.zero = a :=
begin
    induction a,
    { refl, },
    { simp [my_nat.add, *], },
end
```

### 2.2.2 The Lean Mathematical Library

*Mathlib* [5] is a community project that offers a rich mathematical foundation for many theories in Lean 3. Its theories of finite sets, lists, boolean logic and permutations have been very useful for this thesis.

Mathlib also offers many advanced tactics like `finish`, `tauto` or `linarith`. These tactics help significantly in proving trivial lemmas.

# 3 Formalization

Before any property of LYG can be proven or even stated in Lean, all relevant definitions must be formalized. Since nothing can be left vague in Lean, a lof of decisions had to be made to back up LYG by a fully defined model. This chapter discusses these decisions.

## 3.1 Definitions

### 3.1.1 Abstracting LYG: The Guard Module

LYG does not specify an exact guard or expression syntax. Instead, the notation "..." is often used to indicate a sensible continuation to make guards powerful enough to model all Haskell constructs. This is rather problematic for a precise formalization and presented the first big challenge of this thesis. As we wanted to avoid formalizing Haskell and its semantics, we had to carefully design an abstraction that is as close as possible to LYG while pinning down guards to a closed but extendable theory.

First, we defined an abstract `Result` monad to capture the result of an evaluation. Due to laziness, evaluation of guard trees can either end with a specific right hand side, not match any guard or diverge:

```
inductive Result (α: Type)
| value: α → Result
| diverged: Result
| no_match: Result
```

A `bind` operation can be easily defined on `Result` to make it a proper monad with `Result.value` as unit function:

```
def Result.bind { α β: Type } (f: α → Result β): Result α → Result β
| (Result.value val) := f val
| Result.diverged := Result.diverged
| Result.no_match := Result.no_match
```

For some abstract environment type `Env`, we would like to have a denotational semantic `Grd.eval` for guards `Grd`:

```
Grd.eval : Grd → Env → Result Env
```

Abstracting Grd.eval would unify all guard constructs available in Haskell and those used by LYG. However, LYG needs to recognize all guards that can lead to a diverged evaluation: Removing all RHSs behind such a guard would inevitably remove the guard itself. As this might change the semantic of the guard tree, LYG cannot mark all such RHSs as redundant unless there is a proof that the guard will never diverge. As a consequence, Grd.eval cannot be abstracted away.

Instead, we explicitly distinguished between non-diverging xgrds and non-no-matching bang guards:

```
inductive Grd
| xgrd (xgrd: XGrd)
| bang (var: Var)
```

While xgrds classically represent guards and Grds represent guards with side effects in this context, we decided to follow the naming conventions of LYG and chose the name xgrd for side-effect free (non-diverging) guards rather than renaming Grd.

In order to define a denotational semantic on Grd, we postulated the functions xgrd_eval : XGrd → Env → option Env and is_bottom : Var → Env → bool as well as a type Var that represents variables. Note that bang guards cannot change the environment while xgrds can:

```
def Grd.eval : Grd → Env → Result Env
| (Grd.xgrd grd) env :=
    match xgrd_eval grd env with
    | none := Result.no_match
    | some env' := Result.value env'
    end
| (Grd.bang var) env :=
    if is_bottom var env
    then Result.diverged
    else Result.value env
```

Alternatively, all postulated functions can be inlined, yielding the following definition:

```
inductive Grd'
| guarded_by (grd: Env → option Env)
| diverge_if (test: Env → bool)
```

However, this could make the set of guard trees and refinement types uncountable. While this is not problematic for aspects explored by this thesis, it could make implementing a correct function $\mathcal{G}$ impossible, as it cannot reason anymore about guards in a computable way if Env is instantiated with a non-finite type.

In Lean, type classes provide an ideal mechanism to define such ambient abstractions. They can be opened, so that all members of the type class become implicitly available in all definitions and theorems. Every implicit usage pulls the type class into its signature so that consumers can provide a concrete implementation of the type class.

We defined and opened a type class *GuardModule* that describes the presented abstraction:

```
class GuardModule :=
    (Rhs : Type)
    [rhs_decidable: decidable_eq Rhs]
    (Env : Type)
    (XGrd : Type)
    (xgrd_eval : XGrd → Env → option Env)
    (Var : Type)
    (is_bottom : Var → Env → bool)

variable [GuardModule]
open GuardModule
```

We also postulated a type `Rhs` to refer to right hand sides. For technical reasons, equality on this type must be decidable. This abstracts from the numbers that are used in LYG to distinguish right hand sides.

All following definitions and theorems implicitly make use of this abstraction.

## 3.1.2 Guard trees

With the definition of `Grd`, guard trees are defined as follows:

```
inductive Gdt
| rhs (rhs: Rhs)
| branch (tr1: Gdt) (tr2: Gdt)
| grd (grd: Grd) (tr: Gdt)
```

`Gdt.eval` defines a semantic on guard trees, using the semantic of guards:

```
def Gdt.eval : Gdt → Env → Result Rhs
| (Gdt.rhs rhs) env := Result.value rhs
| (Gdt.branch tr1 tr2) env :=
    match tr1.eval env with
    | Result.no_match := tr2.eval env
    | r := r
    end
| (Gdt.grd grd tr) env := (grd.eval env).bind tr.eval
```

Every guard tree contains a (non-empty) finite set of right hand sides:

```
def Gdt.rhss: Gdt → finset Rhs
| (Gdt.rhs rhs) := { rhs }
| (Gdt.branch tr1 tr2) := tr1.rhss ∪ tr2.rhss
| (Gdt.grd grd tr) := tr.rhss
```

In LYG, it is implicitly assumed that the right hand sides of a guard tree are numbered unambiguously. This has to be stated explicitly in Lean with the following recursive predicate:

```
def Gdt.disjoint_rhss: Gdt → Prop
| (Gdt.rhs rhs) := true
| (Gdt.branch tr1 tr2) :=
        disjoint tr1.rhss tr2.rhss
        ∧ tr1.disjoint_rhss ∧ tr2.disjoint_rhss
| (Gdt.grd grd tr) := tr.disjoint_rhss
```

`Gdt.remove_rhss` defines how a set of RHSs can be removed from a guard tree. This definition is required to state that all redundant RHSs can be removed without changing semantics. Note that the resulting guard tree might be empty when all RHSs are removed!

```
def Gdt.branch_option : option Gdt → option Gdt → option Gdt
| (some tr1) (some tr2) := some (Gdt.branch tr1 tr2)
| (some tr1) none := some tr1
| none (some tr2) := some tr2
| none none := none

def Gdt.grd_option : Grd → option Gdt → option Gdt
| grd (some tr) := some (Gdt.grd grd tr)
| _ none := none

def Gdt.remove_rhss : finset Rhs → Gdt → option Gdt
| rhss (Gdt.rhs rhs) := if rhs ∈ rhss then none else some (Gdt.rhs rhs)
| rhss (Gdt.branch tr1 tr2) :=
    Gdt.branch_option
        (tr1.remove_rhss rhss)
        (tr2.remove_rhss rhss)
| rhss (Gdt.grd grd tr) := Gdt.grd_option grd (tr.remove_rhss rhss)
```

Finally, to deal with such empty guard trees, `Gdt.eval_option` lifts `Gdt.eval` to `option Gdt`:

```
def Gdt.eval_option : option Gdt → Env → Result
| (some gdt) env := gdt.eval env
| none env := Result.no_match
```

### 3.1.3 Refinement Types

Refinement types presented another challenge. Defining refinement types through a proper type system would have required to model some Haskell types. Instead, we tried to rely on the same abstractions used to define guard trees in hope that guard trees and refinement types can be related. In this formalization, a refinement type $\Phi$ denotes a predicate on environments:

```
def Φ.eval: Φ → Env → bool
```

Another problem that had to be solved was the formalization of the unconventional binding mechanism of refinement types through conjunctions. If shadowing is not excluded, determining the uncovered refinement type of the following guard tree is problematic:

$$\begin{array}{lll} \dashv \mathsf{let}\ x = \texttt{False},\ \mathsf{let}\ y = \texttt{False} & \dashv & \mathsf{let}\ x = \texttt{True},\ \texttt{True} \leftarrow y \longrightarrow 1 \\ & \llcorner\!\dashv & \mathsf{let}\ y = \texttt{True},\ \texttt{True} \leftarrow x \longrightarrow 2 \end{array}$$

Since the uncovered refinement type is built syntactically from the guard tree, it must contain all its guards. $\mathsf{let}\ x = \texttt{False}$ and $\mathsf{let}\ y = \texttt{False}$ have to be at the very left of it. Also, since we do not want to lose the recursive definition of $\mathcal{U}$, the encoding of $\mathsf{let}\ x = \texttt{True},\ \texttt{True} \leftarrow y$ must occur in the uncovered refinement type either left or right to the encoding of $\mathsf{let}\ y = \texttt{True},\ \texttt{True} \leftarrow x$. In both cases, the inner let guards shadow an outer let guard for the guards to its right.

Shadowing is unproblematic for the semantic of guard trees though: If the first guard tree of a branch fails to match, its environment just before the failing guard and with it possible shadowing bindings are discarded. The second branch is always evaluated with the same environment that the first guard tree has been evaluated with.

This imbalance between refinement types and the semantic of guard trees could be fixed by either adjusting the semantics of guard trees or by introducing a guard operator for refinement types with a restricted binding scope. However, if the semantic of guard trees would not undo the effect of no-matching branches to the environment, an inner let binding would prevent an inaccessible right hand side from being redundant. Even though it never matches, the let binding would always have an effect on the final environment and removing it would be observable.

This problem does not arise in the GHC implementation of LYG, as every binding uses a fresh variable name and only bound variables are read.

We decided to change the scoping mechanism by introducing a data constructor `Φ.xgrd_in` : `XGrd` $\to$ $\Phi$ $\to$ $\Phi$ that limits the scope of the guard to the nested refinement type. This change has only a small impact on the definitions of LYGs algorithms.

Finally, this is our formalized syntax of refinement types:

```
inductive Φ
| false
| true
| xgrd_in (xgrd: XGrd) (ty: Φ)
| not_xgrd (xgrd: XGrd)
| var_is_bottom (var: Var)
| var_is_not_bottom (var: Var)
| or (ty1: Φ) (ty2: Φ)
| and (ty1: Φ) (ty2: Φ)
```

The semantic of refinement types is easily defined and implicitly uses the guard module:

```
def Φ.eval: Φ → Env → bool
| Φ.false env := ff
| Φ.true env := tt
| (Φ.xgrd_in grd ty) env := match xgrd_eval grd env with
    | some env := ty.eval env
    | none := ff
    end
| (Φ.not_xgrd grd) env :=
    match xgrd_eval grd env with
    | some env := ff
    | none := tt
    end
| (Φ.var_is_bottom var) env := is_bottom var env
| (Φ.var_is_not_bottom var) env := !is_bottom var env
| (Φ.or t1 t2) env := t1.eval env || t2.eval env
| (Φ.and t1 t2) env := t1.eval env && t2.eval env
```

A refinement type Φ is called *empty*, if it does not match any environment. This is formalized by the predicate `Φ.is_empty`:

```
def Φ.is_empty (ty: Φ): Prop := ∀ env: Env, ¬(ty.eval env)
```

Instead of a partial function $\mathcal{G}$ with $\mathcal{G}(\Phi) = \emptyset$ if and only if Φ is empty, we define a total function `can_prove_empty` and a predicate `correct_can_prove_empty` that ensures its correctness. This abstracts from the in this context unneeded generation of inhabitants. It also avoids dealing with partial functions, which are not supported by Lean.

```
variable can_prove_empty: Φ → bool
def correct_can_prove_empty : Prop :=
    ∀ ty: Φ, can_prove_empty ty = tt → ty.is_empty
```

The subtype `CorrectIsEmpty` bundles a correct `can_prove_empty` function:

```
def CorrectIsEmpty := {
    can_prove_empty : Φ → bool
    // correct_can_prove_empty can_prove_empty
}
```

### 3.1.4 𝒰ncovered Analysis

Since we changed the binding behavior of conjunctions and added the `Φ.xgrd_in` data constructor, we also had to change the definition of 𝒰 slightly to formalize it. Instead of using Φ as accumulator type, this formalization uses the type $\Phi \to \Phi$.

```
def 𝒰_acc : (Φ → Φ) → Gdt → Φ
| acc (Gdt.rhs _) := acc Φ.false
| acc (Gdt.branch tr1 tr2) := (𝒰_acc ((𝒰_acc acc tr1).and ∘ acc) tr2)
| acc (Gdt.grd (Grd.bang var) tr) :=
    𝒰_acc (acc ∘ (Φ.var_is_not_bottom var).and) tr
| acc (Gdt.grd (Grd.xgrd grd) tr) :=
            (acc (Φ.not_xgrd grd))
        .or
            (𝒰_acc (acc ∘ (Φ.xgrd_in grd)) tr)

def 𝒰 : Gdt → Φ := 𝒰_acc id
```

### 3.1.5 ℛedundant / Inaccessible Analysis

The formalization of the annotated tree is straightforward. However, rather than just accepting refinement types, we allow arbitrary annotations. This will become useful in formal proofs when we no longer care about the specific refinement types but only if they are empty.

```
inductive Ant (α: Type)
| rhs (a: α) (rhs: Rhs): Ant
| branch (tr1: Ant) (tr2: Ant): Ant
| diverge (a: α) (tr: Ant): Ant
```

Similar to the formalization of 𝒰`_acc`, the accumulator of 𝒜`_acc` needs to be extended to functions as well.

```
def 𝒜_acc : (Φ → Φ) → Gdt → Ant Φ
| acc (Gdt.rhs rhs) := Ant.rhs (acc Φ.true) rhs
| acc (Gdt.branch tr1 tr2) :=
    Ant.branch
        (𝒜_acc acc tr1)
        (𝒜_acc ((𝒰_acc acc tr1).and ∘ acc) tr2)
| acc (Gdt.grd (Grd.bang var) tr) :=
    Ant.diverge
        (acc (Φ.var_is_bottom var))
        (𝒜_acc (acc ∘ ((Φ.var_is_not_bottom var).and)) tr)
| acc (Gdt.grd (Grd.xgrd grd) tr) :=
    (𝒜_acc (acc ∘ (Φ.xgrd_in grd)) tr)
```

```
def 𝒜 : Gdt → Ant Φ := 𝒜_acc id
```

It remains to formalize the function $\mathcal{R}$ that partitions all right hand sides of an annotated guard tree into accessible, inaccessible and redundant right hand sides, by using the function $\texttt{can}_\texttt{p}\texttt{rove}_\texttt{e}\texttt{mpty}$:

```
def ℛ : Ant Φ → list Rhs × list Rhs × list Rhs
| (Ant.rhs ty n) :=
    if can_prove_empty ty
    then ([], [], [n])
    else ([n], [], [])
| (Ant.diverge ty tr) :=
    match ℛ tr, can_prove_empty ty with
    | ([], [], m :: ms), ff := ([], [m], ms)
    | r, _ := r
    end
| (Ant.branch tr1 tr2) :=
    match (ℛ tr1, ℛ tr2) with
    | ((k, n, m), (k', n', m')) := (k ++ k', n ++ n', m ++ m')
    end
```

## 3.2 Correctness Statements

As we have all the required definitions at this point, we can state and formalize several correctness properties. We provide proofs for all these properties on GitHub [6]. Chapter 4 will discuss parts of these proofs in more detail.

### 3.2.1 Semantic of $\mathcal{U}$

$\mathcal{U}$ should compute a refinement type that denotes all values that are not covered by a given guard tree. This does not include values under which the execution diverges. Also, note that this theorem carries over to all semantically equivalent definitions of $\mathcal{U}$.

```
theorem 𝒰_semantic: ∀ gdt: Gdt, ∀ env: Env,
        (𝒰 gdt).eval env ↔ (gdt.eval env = Result.no_match)
```

### 3.2.2 Semantic of $\mathcal{R}$ and $\mathcal{A}$

For a given guard tree and a given correct function `can_prove_empty` (which corresponds to $\mathcal{G}$ in LYG), $\mathcal{R}$ should compute a triple $(a, i, r)$ of accessible, inaccessible and redundant right hand sides. Whenever the given guard tree evaluates to a RHS, this RHS must be accessible and neither inaccessible nor redundant. Also, RHSs that are redundant can be removed without changing semantics. Note that redundant RHSs could be marked as inaccessible or even accessible instead without violating this theorem. The opposite is not true: Not all accessible RHSs can be marked as inaccessible and not all inaccessible RHSs can be marked as redundant - see chapters 1 and 2 for counterexamples.

```
theorem ℛ_semantic:
    ∀ can_prove_empty: CorrectCanProveEmpty,
    ∀ gdt: Gdt, gdt.disjoint_rhss →
    (
        let ⟨ a, i, r ⟩ := ℛ can_prove_empty.val (𝒜 gdt)
        in
                (∀ env: Env, ∀ rhs: Rhs,
                    gdt.eval env = Result.value rhs
                      → rhs ∈ a \ (i ++ r)
                )
            ∧
                Gdt.eval_option (gdt.remove_rhss r.to_finset)
                = gdt.eval

        : Prop
    )
```

# 4 Formalized Proofs

This chapter gives an overview of the formal proofs of the correctness statements from the previous chapter. The full Lean proofs can be found on GitHub [6].

To reduce the complexity of the definitions from chapter 3, we came up with several internal definitions. They include accumulator-free alternatives `U` and `A` for the functions $\mathcal{U}$ and $\mathcal{A}$.

Correctness of `U` can be shown directly and this result can be transferred easily to $\mathcal{U}$ too. It is much more difficult to show correctness of $\mathcal{R}/\mathcal{A}$ though, so we will discuss this in more detail. In chapter 4.2, we show that redundant RHSs can be removed without changing semantics. Then, in chapter 4.3, we show that if a guard tree evaluates to a RHS, this RHS must be marked as accessible. Together, these properties form the correctness statement as presented in chapter 3.2.2.

## 4.1 Simplification $A$ of $\mathcal{A}$

Let us have another look at the definition of $\mathcal{A}$:

```
def 𝒜_acc : (Φ → Φ) → Gdt → Ant Φ
| acc (Gdt.rhs rhs) := Ant.rhs (acc Φ.true) rhs
| acc (Gdt.branch tr1 tr2) := Ant.branch
        (𝒜_acc acc tr1)
        (𝒜_acc ((𝒰_acc acc tr1).and ∘ acc) tr2)
| acc (Gdt.grd (Grd.bang var) tr) := Ant.diverge
        (acc (Φ.var_is_bottom var))
        (𝒜_acc (acc ∘ ((Φ.var_is_not_bottom var).and)) tr)
| acc (Gdt.grd (Grd.xgrd grd) tr) :=
    (𝒜_acc (acc ∘ (Φ.xgrd_in grd)) tr)

def 𝒜 : Gdt → Ant Φ := 𝒜_acc id
```

It is difficult to reason about $\mathcal{A}\_acc$, as we are only interested in certain well behaving accumulator values (in particular homomorphisms) and not arbitrary functions. Since this definition is central to many proofs, we defined a much easier function $A$ that does not need an accumulator:

```
def A : Gdt → Ant Φ
| (Gdt.rhs rhs) := Ant.rhs Φ.true rhs
| (Gdt.branch tr1 tr2) := Ant.branch (A tr1) $ (A tr2).map ((U tr1).and)
| (Gdt.grd (Grd.bang var) tr) := Ant.diverge (Φ.var_is_bottom var)
                    $ (A tr).map ((Φ.var_is_not_bottom var).and)
| (Gdt.grd (Grd.xgrd grd) tr) := (A tr).map (Φ.xgrd_in grd)
```

We then showed that $A$ and $\mathcal{A}$ are semantically equivalent:

```
-- Captures the semantic of an Ant Φ
def Ant.eval_rhss (ant: Ant Φ) (env: Env) :=
    ant.map (λ ty, ty.eval env)

theorem A_sem_eq_𝒜 (gdt: Gdt):
    (A gdt).eval_rhss = (𝒜 gdt).eval_rhss
```

However, we wanted flexibility regarding syntactical equality. In fact, $\mathcal{A}$(gdt) as stated is not syntactical equal to $A$(gdt) for all gdt due to ($\mathcal{A}$`_acc acc tr1).and` ∘ `acc` not being equal to acc ∘ ($\mathcal{A}$`_acc id tr1).and`. While we only noticed later that we could rewrite $\mathcal{A}$ so that it indeed produces refinement types identical to those computed by $A$, we liked the insight that we got by not assuming or forcing syntactic equality. Also, this would allow for future semantic-preserving modifications of $\mathcal{A}$ that could alter the syntactical structure of the computed refinement types without losing any results: After all, the only fact that the following proofs use to reason about $\mathcal{A}$ is its semantic equivalence to $A$. They explicitly do not unfold $\mathcal{A}$s definition!

The challenge of this approach is that `can_prove_empty` does not have to be *well defined* on refinement types modulo semantic equivalence: If two refinement types are semantically equal, `can_prove_empty` could be true for the former, but false for the latter type. A function `can_prove_empty` that is correct and has this well defined property is uncomputable if it returns `true` for the refinement type × - it would need to return `true` for all refinement types that are empty!

## 4.2 Redundant RHSs Can Be Removed Without Changing Semantics

Given a guard tree *gdt* with disjoint RHSs and an annotated guard tree *Agdt* that semantically equals `A gdt`, all redundant leaves reported by $\mathcal{R}$ (on *Agdt*, using a correct function `can_prove_empty`) can be removed from *gdt* without changing its semantic. We will later instantiate `Agdt` with $\mathcal{A}$ `gdt`. The indirection introduced by `Agdt` allows to use the simpler definition of $A$ while `can_prove_empty` still computes emptiness for refinement types in `Agdt` (see chapter 4.1 for why this is important).

```
theorem R_red_removable
    (can_prove_empty: CorrectCanProveEmpty)
    { gdt: Gdt } (gdt_disjoint: gdt.disjoint_rhss)
    { Agdt: Ant Φ }
    (ant_def: Agdt.mark_inactive_rhss = (A gdt).mark_inactive_rhss):
        Gdt.eval_option (gdt.remove_rhss (
            (R $ Agdt.map can_prove_empty.val).red.to_finset
        ))
        = gdt.eval
```

Figure 4.1 sketches the proof idea. Thin arrows mark the data flow, fat arrows the flow of reasoning. We start with a guard tree gdt. $\mathrm{ant}_\Phi$ is defined as gdt annotated with refinement types by $\mathcal{A}$. In the formal proof, we actually use $\mathrm{ant}_\Phi :=$ `Agdt`, but since $\mathrm{ant}_2$ only depends on the semantic of $\mathrm{ant}_\Phi$ and Agdt and $\mathcal{A}(\mathrm{gdt})$ have the same semantic, this does not change the proof much.

The general idea is to focus on a particular but arbitrary environment env: Reasoning about which RHSs can be removed while preserving semantics is much simpler when only considering a single environment.

In fact, we can just evaluate the guard tree on env and remove all RHSs except the one the evaluation ended with. We call those RHSs *inactive*, the resulting RHS is called *active*. If the evaluation diverged however, the diverging bang guard must not be removed; thus, all RHSs except one behind the diverging bang operator can be removed. In this case, the bang guard is active and all RHSs are inactive.

To exploit this idea, the set of RHSs $r := \mathcal{R}(\mathrm{ant}_\Phi)$.red must be related to the RHSs that can be removed when focusing on a particular environment. To relate them, we defined a function `Gdt.mark_inactive` that directly computes a boolean annotated tree that marks inactive nodes for a given guard tree and environment ($\mathrm{ant}_3$ in the figure). The definition of `Gdt.mark_inactive` is very similar to the definition of the denotational semantic of guard trees - this helps proofs that bring these concepts together.

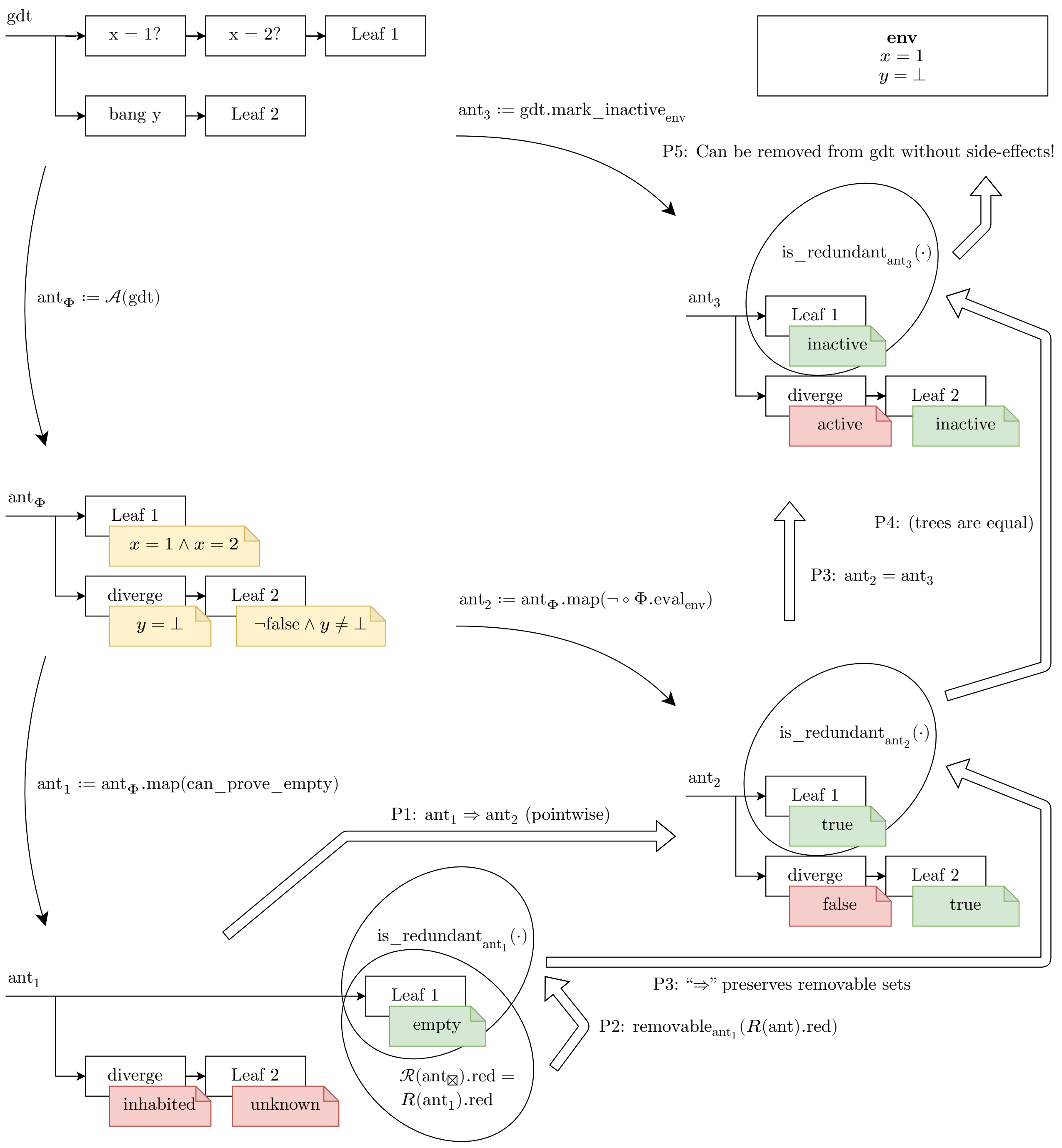
We decomposed $\mathcal{R}$, which operates on Ant $\Phi$ and needs a function `can_prove_empty`, into a function $R$ that operates on Ant bool and a function $f :=$ map(can_prove_empty) that computes an Ant bool from an Ant $\Phi$ ($\mathrm{ant}_1$ in the figure). Clearly $\mathcal{R} = R \circ f$.

Is easy to relate $\mathrm{ant}_1$ with $\mathrm{ant}_3$ if we define $\mathrm{ant}_2 := \mathrm{ant}_{\boxtimes}.\mathrm{map}(\neg \circ \Phi.\mathrm{eval}_{\mathrm{env}})$ as the direct evaluation of each refinement type under env. We can show $\mathrm{ant}_2 = \mathrm{ant}_3$ (P3), since a node is active if and only if the corresponding refinement type matches. We can also show that each boolean annotation in $\mathrm{ant}_1$ implies ("$\Rightarrow$") the corresponding boolean annotation in $\mathrm{ant}_2$ pointwise (P1): If a refinement type is empty, it must not match.

It does not require much to show that the set of RHSs $R(\mathrm{ant}_3)$.*red* can be removed from *gdt* without changing its semantic on env. We hoped that $\mathcal{R}(\mathrm{ant}_\mathrm{a})$.red would be a subset of $\mathcal{R}(\mathrm{ant}_\mathrm{b})$.red if $\mathrm{ant}_\mathrm{a} \Rightarrow \mathrm{ant}_\mathrm{b}$ to complete the proof. However, this is not the case! See chapter 4.2.1 for a counterexample.

To repair the proof idea, we defined a predicate `is_redundant_set` on sets of RHSs for a given boolean annotated tree. This predicate has the property that if $r$ is a redundant set in $\mathrm{ant}_\mathrm{a}$ and if $\mathrm{ant}_\mathrm{a} \Rightarrow \mathrm{ant}_\mathrm{b}$, then $r$ is also a redundant set in $\mathrm{ant}_\mathrm{b}$ (P3).

**Figure 4.1:** Proof Overview: Redundant RHSs can be removed without changing semantics.


gdt
x = 1?
x = 2?
Leaf 1
bang y
Leaf 2
env
$x = 1$
$y = \perp$
$\text{ant}_3 := \text{gdt.mark\_inactive}_{\text{env}}$
P5: Can be removed from gdt without side-effects!
$\text{is\_redundant}_{\text{ant}_3}(\cdot)$
$\text{ant}_3$
Leaf 1
inactive
diverge
active
Leaf 2
inactive
$\text{ant}_\Phi := \mathcal{A}(\text{gdt})$
$\text{ant}_\Phi$
Leaf 1
$x = 1 \wedge x = 2$
diverge
$y = \perp$
Leaf 2
$\neg\text{false} \wedge y \neq \perp$
P4: (trees are equal)
P3: $\text{ant}_2 = \text{ant}_3$
$\text{ant}_2 := \text{ant}_\Phi.\text{map}(\neg \circ \Phi.\text{eval}_{\text{env}})$
$\text{is\_redundant}_{\text{ant}_2}(\cdot)$
$\text{ant}_2$
Leaf 1
true
diverge
false
Leaf 2
true
$\text{ant}_1 := \text{ant}_\Phi.\text{map}(\text{can\_prove\_empty})$
P1: $\text{ant}_1 \Rightarrow \text{ant}_2$ (pointwise)
$\text{is\_redundant}_{\text{ant}_1}(\cdot)$
$\text{ant}_1$
Leaf 1
empty
diverge
inhabited
Leaf 2
unknown
$\mathcal{R}(\text{ant}_\boxtimes).\text{red} =$
$R(\text{ant}_1).\text{red}$
P3: "⇒" preserves removable sets
P2: $\text{removable}_{\text{ant}_1}(R(\text{ant}).\text{red})$

Now, we can finish the proof by showing that $R(\mathrm{ant}_1)$.red is a redundant set (P2) and that redundant sets can be removed from guard trees without changing its semantic (P5). This finishes the proof.

### 4.2.1 `is_redundant_set`

Given two boolean annotated trees $\mathrm{ant}_\mathrm{a}$ and $\mathrm{ant}_\mathrm{b}$ with $\mathrm{ant}_\mathrm{a} \Rightarrow \mathrm{ant}_\mathrm{b}$, we would like to transfer insights on $\mathrm{ant}_\mathrm{a}$ to $\mathrm{ant}_\mathrm{b}$ as stated in the previous chapter.

To reason about the removal of redundant RHSs in context of a particular environment, we defined the predicate `is_redundant_set`.

do not read further. Work in progress!

This predicate has the property that if $r$ is a redundant set in $\mathrm{ant}_\mathrm{a}$ and if $\mathrm{ant}_\mathrm{a} \Rightarrow \mathrm{ant}_\mathrm{b}$, then $r$ is also a redundant set in $\mathrm{ant}_\mathrm{b}$ (P3).

Later, it will turn out that if the boolean values of that tree indicate *inactivity*, such a redundant set of RHSs can be removed from the guard tree without changing semantics (P5). A node is inactive for a given environment if evaluation does not stop on it. The predicate is defined as following:

```
def Ant.critical_rhs_sets : Ant bool → finset (finset Rhs)
| (Ant.rhs inactive n) := ∅
| (Ant.diverge inactive tr) := tr.critical_rhs_sets ∪ if inactive
    then ∅
    else { tr.rhss }
| (Ant.branch tr1 tr2) := tr1.critical_rhs_sets ∪ tr2.critical_rhs_sets

def Ant.inactive_rhss : Ant bool → finset Rhs
| (Ant.rhs inactive n) := if inactive then { n } else ∅
| (Ant.diverge inactive tr) := tr.inactive_rhss
| (Ant.branch tr1 tr2) := tr1.inactive_rhss ∪ tr2.inactive_rhss

def Ant.is_redundant_set (a: Ant bool) (rhss: finset Rhs) :=
    rhss ∩ a.rhss ⊆ a.inactive_rhss
    ∧ ∀ c ∈ a.critical_rhs_sets, ∃ l ∈ c, l ∉ rhss
```

A redundant set consists of RHSs that are annotated with `false` and avoid critical sets. If a diverge node is annotated with `true`, all its RHSs form a critical set. Each critical set must have one RHS that is not contained in a given redundant set. This ensures that active diverge nodes do not disappear when a redundant set is removed from a guard tree.

We show that all RHSs marked as redundant by $\mathcal{R}$ indeed form a redundant set (P2). We believe that $\mathcal{R}$ actually computes a largest redundant set given a boolean annotated tree. However, a largest redundant set is not unique! In fact, $R$ arbitrarily marks the first redundant RHS as inaccessible to avoid critical sets. It could equally do the same with the last redundant RHS.

We designed redundant sets in a way that

## 4.3 Accessible RHSs Must Be Detected as Accessible

# 5 Conclusion

# Bibliography

[1] S. Graf, S. Peyton Jones, and R. G. Scott, "Lower your guards: A compositional pattern-match coverage checker," *Proc. ACM Program. Lang.*, vol. 4, Aug. 2020.

[2] "The lean theorem prover (community fork)." `https://github.com/leanprover-community/lean`. Retrieved: 20 Jan. 2021.

[3] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer, "The lean theorem prover (system description)," in *Automated Deduction - CADE-25* (A. P. Felty and A. Middeldorp, eds.), (Cham), pp. 378–388, Springer International Publishing, 2015.

[4] J. Avigad, L. de Moura, and S. Kong, "Theorem proving in lean." `https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf`. Retrieved: 20 Jan. 2021.

[5] "A mathlib overview." `https://leanprover-community.github.io/mathlib-overview.html`. Retrieved: 20 Jan. 2021.

[6] H. Dieterichs, "Lean proof." `https://github.com/hediet/masters-thesis/tree/14833277a507a4f36774d47e965ef1c72811ec41/code`. Retrieved: 20 Jan. 2021.

# Erklärung

Hiermit erkläre ich, Henning Dieterichs, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

| | |
|---|---|
| Ort, Datum | Unterschrift |

# Danke

Ich danke meinen Betreuern Sebastian Graf und Sebastian Ullrich, die mich in jeglicher Hinsicht unterstützt haben.