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
                (∀ 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
    )
```

Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M. Sc. Sebastian Graf
Abgabedatum:	06. 04. 2021

Zusammenfassung

Die in Lower Your Guards vorgestellten Algorithmen analysieren Pattern Matching Definitionen und erkennen nicht abgedeckte Fälle, aber auch unzugängliche und redundante rechte Seiten.

Ihre Implementierung in GHC entdeckte erfolgreich bisher unbekannte Fehler in Haskell Quellcode. Während die empirische Validierung über eine große Menge von Haskell-Code die Behauptung der Korrektheit untermauert, fehlt den Autoren eine präzise Formalisierung sowie ein Beweis für diese Behauptung.

Diese Arbeit etabliert einen präzisen Begriff von Korrektheit und präsentiert formale Beweise, dass diese Algorithmen tatsächlich korrekt sind. Diese Beweise sind in Lean 3 formalisiert.

The algorithms presented in Lower Your Guards analyze pattern matching definitions and detect uncovered cases, but also inaccessible and redundant right hand sides.

Their implementation in the GHC spotted previously unknown bugs in real world code. While empirical validation over a large corpus of Haskell code corroborates the claim of correctness, the authors lack a precise formalization as well as a proof of that claim.

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 match used to define `f` is not exhaustive, as the input `Case4` is uncovered. This is usually an oversight by the programmer and should be brought to their attention with an appropriate warning.

Also, `f` will never evaluate to 3 or 4 - replacing these values with any other value would not change any observable behavior 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] is a compiler analysis that is able to detect such mistakes and also can deal with the intricacies of lazy evaluation.

However, LYG is only checked empirically so far: Its implementation in the Glasgow Haskell Compiler just *seems to work*.

Obviously, LYG would be incorrect if it marks a RHS as inaccessible even though it actually is accessible. This could have fatal consequences: A 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 and complete 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.

The main contributions of this thesis are as follows:

- We formalized the uncovered and redundant/inaccessible analysis of LYG in Lean 3. This formalization is discussed in detail in chapter 3. We noticed an inaccuracy in how variable scopes are handled in refinement types and present a counterexample to $\mathcal{U}$'s correctness in chapter 3.3 by exploiting shadowing variable bindings. We suggest a more explicit variable scoping mechanism of refinement types.

- We establish a notion of correctness of LYG. Its formalization in Lean is discussed in chapter 4. This notion of correctness is more precise and more complete than the notion of correctness presented in LYG.

- We present formal proofs that the redundant/inaccessible analysis of LYG satisfies this notion of correctness if our suggestion of a more explicit scoping operator is applied. Details of this proof are discussed in chapter 5.

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. Redundant 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 for 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 in chapter 5.2.2. However, for the reasons mentioned above, not both RHSs can be marked as redundant.

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. *Con* refers to data constructors, *TyCt* to type constraints.

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 (like `undefined`).

Figure 2.1: Definition of Guard Trees

Guard Syntax

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

Guard Tree Syntax

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

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 semantics for guard trees will be defined in chapter 3.2.

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.

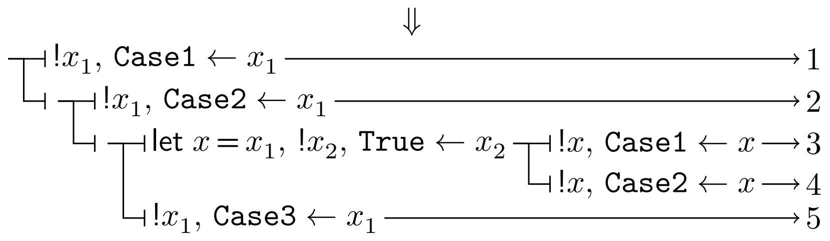
Figure 2.2 presents the transformation of figure 1.1 into a guard tree.

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.

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
```



2.1.3 Refinement Types

Refinement types [2] describe vectors of values $x_1, ..., x_n$ that satisfy a given predicate Φ. Their syntax is defined in figure 2.3.

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

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

2.1.4 Binding Mechanism Of Refinement Types

Refinement type literals, such as the let-literal or the pattern-match-literal, can bind one or more variables. Unconventionally however, a binding is in scope of a literal if and only if the binding literal is the left operand of a parent conjunction.

Thus, $(\mathsf{let}\ x = y \wedge x \not\approx \bot) \wedge x \not\approx \bot$ is semantically equivalent to $y \not\approx \bot \wedge x \not\approx \bot$. Clearly, $\wedge$ is not associative!

To utilize this behavior, the operator $\dot{\wedge}$ replaces the rightmost operand of the top conjunction tree of the left argument (figure 2.3).

2.1.5 $\mathcal{G}$enerating Inhabitants

LYG also describes a partial function $\mathcal{G}$ with $\mathcal{G}(\Theta) = \emptyset \Rightarrow (\Theta \text{ denotes } \emptyset)$ for all refinement types Θ. $\mathcal{G}$ is used to $\mathcal{G}$enerate inhabitants of a refinement type to

Figure 2.3: Definition of Refinement Types

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

Operations on Θ

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

build elaborate error messages, or, more importantly, to get a guarantee that a refinement type is empty. A total correct function $\mathcal{G}$ is uncomputable, as there are expressions (making use of recursively defined functions) that match a certain data constructor if and only if a given turing machine halts! 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 $\mathcal{G}$.

2.1.6 $\mathcal{U}$ncovered 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 no 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 $x_1 = \texttt{Case4}$.

2.1.7 $\mathcal{A}$nnotated Guard Trees

Annotated guard trees represent simplified guard trees that have been annotated with refinement types Θ. They are made of RHSs, branches and bang nodes. Their syntax is defined in figure 2.5. Annotated guard trees are the result of the function $\mathcal{A}$, which is discussed in the next chapter.

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

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

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

Figure 2.5: Definition of Annotated Guard Trees

$$u \in \mathsf{Ant} \qquad \longrightarrow \Theta\, k \ \mid\ \left[\begin{array}{l} u_1 \\ u_2 \end{array}\right. \mid\ \relbar\!\!\relbar\, \Theta \lightning \relbar\!\!\relbar\, u$$

2.1.8 $\mathcal{R}$edundant/Inaccessible Analysis

The goal of the redundant/inaccessible analysis is to report as many 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.

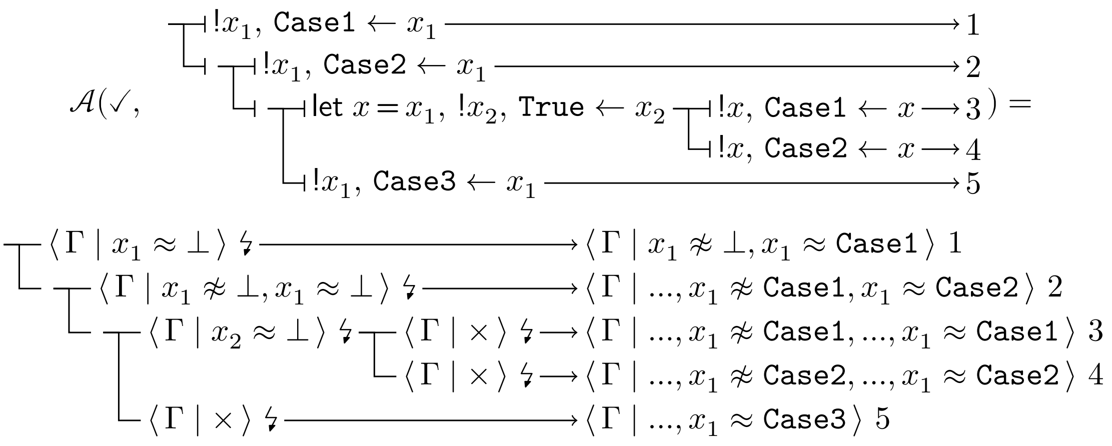
Figure 2.9 computes inaccessible and redundant leaves for an annotated tree that is ($\mathcal{G} = \emptyset$)-equivalent to the annotated tree from figure 2.7 for sensible functions $\mathcal{G}$. It states that RHS 4 in 1.1 is redundant and can be removed, while RHS 3 is just inaccessible.

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

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

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

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



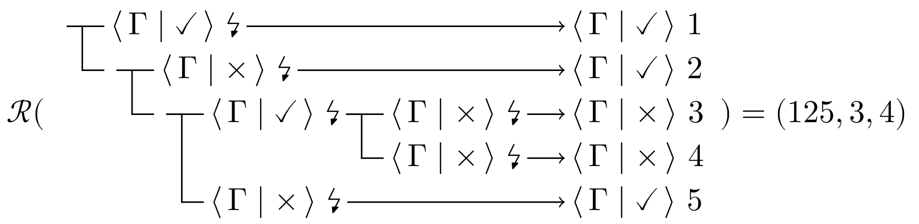
A(✓,
!x₁, Case1 ← x₁ → 1
!x₁, Case2 ← x₁ → 2
let x = x₁, !x₂, True ← x₂
!x, Case1 ← x → 3
!x, Case2 ← x → 4
!x₁, Case3 ← x₁ → 5
) =
⟨Γ | x₁ ≈ ⊥⟩ ↯ → ⟨Γ | x₁ ≉ ⊥, x₁ ≈ Case1⟩ 1
⟨Γ | x₁ ≉ ⊥, x₁ ≈ ⊥⟩ ↯ → ⟨Γ | ..., x₁ ≉ Case1, x₁ ≈ Case2⟩ 2
⟨Γ | x₂ ≈ ⊥⟩ ↯
⟨Γ | ×⟩ ↯ → ⟨Γ | ..., x₁ ≉ Case1, ..., x₁ ≈ Case1⟩ 3
⟨Γ | ×⟩ ↯ → ⟨Γ | ..., x₁ ≉ Case2, ..., x₁ ≈ Case2⟩ 4
⟨Γ | ×⟩ ↯ → ⟨Γ | ..., x₁ ≈ Case3⟩ 5

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

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

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

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


$\mathcal{R}($
$\langle \Gamma \mid \checkmark \rangle$ ↯ $\longrightarrow \langle \Gamma \mid \checkmark \rangle$ 1
$\langle \Gamma \mid \times \rangle$ ↯ $\longrightarrow \langle \Gamma \mid \checkmark \rangle$ 2
$\langle \Gamma \mid \checkmark \rangle$ ↯ $\langle \Gamma \mid \times \rangle$ ↯ $\longrightarrow \langle \Gamma \mid \times \rangle$ 3
$\langle \Gamma \mid \times \rangle$ ↯ $\longrightarrow \langle \Gamma \mid \times \rangle$ 4
$\langle \Gamma \mid \times \rangle$ ↯ $\longrightarrow \langle \Gamma \mid \checkmark \rangle$ 5
$) = (125, 3, 4)$

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 [3] [4] 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 function 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 [5] 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 [6] 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 Formal Definitions

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 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.

The Result Monad

First, we defined a generic `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
```

Denotational Semantics for Guards

For some abstract environment type `Env`, we would like to have a denotational semantics `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 semantics 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 distinguish between non-diverging (*total*) `tgrd`s and possibly diverging `bang` guards:

```
inductive Grd
| tgrd (tgrd: TGrd)
| bang (var: Var)
```

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

In order to define a denotational semantics on `Grd`, we postulated the functions `tgrd_eval : TGrd → Env → option Env` and `is_bottom : Var → Env → bool` as well as a type `Var` that represents variables. While `tgrds` can change the environment, `bang` guards cannot:

```
def Grd.eval : Grd → Env → Result Env
| (Grd.tgrd grd) env :=
    match tgrd_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, we could have set `Var := Env → bool` and `TGrd := Env → option Env` and replaced `is_bottom` and `tgrd_eval` with `id`, yielding the following definition:

```
inductive Grd'
| tgrd (grd: Env → option Env)
| bang (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.

The Guard Module

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)
    (TGrd : Type)
    (tgrd_eval : TGrd → 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. We also require most types to be inhabited so that we can construct module-independent examples.

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

3.2 Guard Trees

Syntax of Guard Trees

With the definition of `Grd`, guard trees are defined as inductive data type:

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

Semantics of Guard Trees

`Gdt.eval` defines a denotational semantics on guard trees, using the semantics of guards. It returns the first RHS that matches a given environment. If a guard diverges, the entire evaluation diverges. Otherwise, if no RHSs matches, *no-match* is returned.

```
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
```

RHSs in Guard Trees

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
```

Removing RHSs in Guard Trees

`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 the semantics of 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
```

3.3 Refinement Types

Refinement types presented another challenge. Defining refinement types through a proper type system would have required to model 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 Φ denotes a predicate on environments:

```
def Φ.eval: Φ → Env → bool
```

With a proper `GuardModule` instantiation, the environment can be used to not only carry runtime values, but also their type! A (well) typed environment can assist in proving a refinement type to be empty.

Variable Scoping Rules / Incorrectness of $\mathcal{U}$

Another problem that had to be solved was the formalization of the unconventional binding mechanism of refinement types through conjunctions, as described in chapter 2.1.4. In particular, this causes $\mathcal{U}$ to be incorrect (for some intuitive notion of correctness) with regards to the guard tree semantics we defined in chapter 3.2. While the following guard tree *gdt* does not match for $x =$ `False`, its uncovered refinement type Θ computed by $\mathcal{U}$ is empty due to the constraints $x \not\approx \mathit{False}$ and $x \not\approx \mathit{True}$ that refer to the same variable and thus represent a contradiction!

$$gdt := \left[\begin{array}{l} \mathsf{let}\ x = \texttt{True},\ \texttt{False} \leftarrow x \longrightarrow 1 \\ \texttt{True} \leftarrow x \longrightarrow 2 \end{array}\right.$$

$$\begin{aligned} \Theta := \mathcal{U}(\checkmark, gdt) &= \langle\, x{:}\mathit{Bool} \mid ((\mathsf{let}\ x = \mathit{True} \mathbin{\dot\wedge} x \not\approx \mathit{False}) \mathbin{\dot\wedge} x \not\approx \mathit{True} \,\rangle \\ &= \langle\, x{:}\mathit{Bool} \mid \mathsf{let}\ x = \mathit{True} \wedge (x \not\approx \mathit{False} \wedge x \not\approx \mathit{True}) \,\rangle \end{aligned}$$

In the example, the let binding brings a variable x into scope that shadows an outer variable. Due to the definition of $\mathcal{U}$ and the scoping rules of refinement types, this shadowing binding of x overrides the outer variable x in contexts where it is incorrect to do so.

In particular, the term $\mathcal{U}(\mathcal{U}(\Theta, t_1), t_2)$ in the branch case of $\mathcal{U}$ is problematic: All bindings introduced in Θ should still be exposed by $\mathcal{U}(\Theta, t_1)$ so that variables in t_2 are resolved correctly. However, variable definitions introduced in t_1 must not be visible to t_2 and thus must not be exposed by $\mathcal{U}(\Theta, t_1)$! This is clearly violated by the term $\mathcal{U}(\Theta \dot{\wedge} (\mathsf{let}\ x = e), t)$ that defines the let-case in $\mathcal{U}$ - it exposes both all bindings from Θ and the new binding $\mathsf{let}\ x = e$.

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

There are several ways of how this problem can be addressed.

- Replace the term $\mathcal{U}(\mathcal{U}(\Theta, t_1), t_2)$ in the definition of $\mathcal{U}$ with $\mathcal{U}((\mathcal{U}(\Theta, t_1) \cup \times) \dot{\wedge} \Theta, t_2)$.

  "$\dot{\wedge}$" stops at "$\cup$", so the $\cup$-operator acts as scope boundary. To bring the variables defined by Θ into scope again, Θ is joined a second time, potentially causing a refinement type of exponential size. While we believe that $\mathcal{U}$ is correct with this updated definition, we decided against this solution as the construction to limit the scope feels like a band aid and is unnecessarily complex.

- Adjust the semantics of guard trees so that variables defined in a branch override shadowed variables in all later branches.

  We managed to prove correctness of $\mathcal{U}$ as stated in LYG with this updated semantics of guard trees. However, this semantics is not only very unconventional, but also dramatically increases the complexity when reasoning about the effect of removing an inaccessible RHS.

  Since variable bindings introduced by guards that guard only inaccessible RHSs stay visible until the evaluation ends (and are not only relevant for the inaccessible RHSs), removing such guards almost always causes a different final environment. In this sense, almost no inaccessible RHSs are redundant - which is not the intention of the analysis and clearly not the case for the GHC implementation of LYG. To make the analysis more meaningful, we could require each variable name to be unique. With this assumption, such environment modification should have no impact on the evaluation result. Due to the high complexity of this approach, we decided against it too.

- Limit the scope of variables in refinement types.

A data constructor `Φ.tgrd_in` : `TGrd` → Φ → Φ is introduced that limits the scope of the guard to the nested refinement type and any scoping behavior of the ∧-operator is removed. This simplifies the scoping mechanism, but requires to adapt $\mathcal{U}$, as done in chapter 3.4. We chose this approach due to its clear modeling, in hope to reduce the complexity of the formal proofs.

This problem does not arise in the GHC implementation of LYG as it uses a different encoding for refinement types.

Syntax of Refinement Types

Finally, this is our formalized syntax of refinement types:

```
inductive Φ
| false
| true
| tgrd_in (tgrd: TGrd) (ty: Φ)
| not_tgrd (tgrd: TGrd)
| var_is_bottom (var: Var)
| var_is_not_bottom (var: Var)
| or (ty1: Φ) (ty2: Φ)
| and (ty1: Φ) (ty2: Φ)
```

Since the negation of a guard cannot bind variables, it does not need to have a nested refinement type that would see bound variables. The same applies to `var_is_bottom` and its negation.

Semantics of Refinement Types

The semantics 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.tgrd_in grd ty) env := match tgrd_eval grd env with
    | some env := ty.eval env
    | none := ff
    end
| (Φ.not_tgrd grd) env :=
    match tgrd_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
```

With this definition the evaluation of the second operand of a conjunction is obviously independent of any environment effects applied in the evaluation of the first operand!

Definition of `is_empty`

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)
```

Definition of `can_prove_empty`

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 generation of inhabitants which are superfluous in this context. It also avoids dealing with partial functions, which are not directly 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 `CorrectCanProveEmpty` bundles a correct `can_prove_empty` function:

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

3.4 Uncovered Analysis

As discussed in chapter 3.3, LYG's definition of $\mathcal{U}$ has problems with guard trees that define shadowing bindings. LYG defined $\mathcal{U}$ as follows (see chapter 2.1.6 for the discussion of this definition):

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

Equipped with the data constructor `Φ.tgrd_in`, we can fix the shadowing problem and formalize $\mathcal{U}$ now. Instead of using Φ as accumulator type, our formalization uses a function $\Phi \to \Phi$: The new accumulator explicitly applies a context to a refinement type. This happens implicitly in LYG's definition through the use of $\Theta \mathbin{\dot\wedge} \cdot$.

Note that all occuring accumulator functions are homomorphisms modulo the semantics of refinement types, i.e. `Gdt.eval` $(f\ (a.and\, b)) =$ `Gdt.eval` $((f\ a).and\ (f\ b))$. We carefully make use of this to get formalized definitions of $\mathcal{U}$ and $\mathcal{A}$ that can be interleaved, as done in LYG.

```
def 𝒰_acc : (Φ → Φ) → Gdt → Φ
| acc (Gdt.rhs _) := 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.tgrd grd) tr) :=
            (acc (Φ.not_tgrd grd))
        .or (𝒰_acc (acc ∘ (Φ.tgrd_in grd)) tr)

def 𝒰 : Gdt → Φ := 𝒰_acc id
```

3.5 Redundant / Inaccessible Analysis

Formalization of Annotated Trees

The formalization of annotated trees is straightforward. However, we allow arbitrary annotations rather than only accepting refinement types. This will become useful in formal proofs when we no longer care about the specific refinement types, but only whether 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
```

Formalization of $\mathcal{A}$

Similar to the formalization of $\mathcal{U}$ in chapter 3.4, we also need to address the shadowing problem when formalizing $\mathcal{A}$. This is LYG's definition of $\mathcal{A}$ as stated in chapter 2.1.8:

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

Our formalization in Lean follows. Analogous to our formalization of $\mathcal{U}$, instead of contextualizing refinement types by combining them with the accumulator through $\dot{\wedge}$, we model the accumulator as an explicit function that contextualizes its argument:

```
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.tgrd grd) tr) :=
    (𝒜_acc (acc ∘ (Φ.tgrd_in grd)) tr)

def 𝒜 : Gdt → Ant Φ := 𝒜_acc id
```

Note that in the branch case, $\mathcal{A}$`_acc` and $\mathcal{U}$`_acc` are called with the same arguments. Even more, both functions have the same recursion structure, which makes it possible to interleave both functions. This is done in chapter 3.6.

Formalization of $\mathcal{R}$

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 `can_prove_empty`.

This is $\mathcal{R}$ as presented in LYG and chapter 2.1.8:

$$
\begin{array}{lll}
\mathcal{R}(\;\longrightarrow \Theta\, n\;) & = & \begin{cases} (\epsilon, \epsilon, n), & \text{if } \mathcal{G}(\Theta) = \emptyset \\ (n, \epsilon, \epsilon), & \text{otherwise} \end{cases} \\
\mathcal{R}(\;\sqsubset^{t}_{u}\;) & = & (\overline{k}\,\overline{k'}, \overline{n}\,\overline{n'}, \overline{m}\,\overline{m'}) \;\text{ where } \begin{array}{ll} (\overline{k}, \overline{n}, \overline{m}) & = \mathcal{R}(t) \\ (\overline{k'}, \overline{n'}, \overline{m'}) & = \mathcal{R}(u) \end{array} \\
\mathcal{R}(\;-\!\!-\Theta\,\lightning\!-\!\!-\, t\;) & = & \begin{cases} (\epsilon, m, \overline{m'}), & \text{if } \mathcal{G}(\Theta) \neq \emptyset \text{ and } \mathcal{R}(t) = (\epsilon, \epsilon, m\,\overline{m'}) \\ \mathcal{R}(t), & \text{otherwise} \end{cases}
\end{array}
$$

This definition has a surprisingly direct representation in Lean:

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

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

Since $\mathcal{A}$`_acc` and $\mathcal{U}$`_acc` have the same recursion structure, they can be combined into a single function that shares the recursive invocations. The following function $\mathcal{U}\mathcal{A}$`_acc` computes the uncovered refinement type and the annotated guard tree for a given guard tree at the same time. This improves performance if a lazy evaluation strategy is used in combination with sharing as the accumulator can be fully shared.

```
def 𝒰𝒜_acc : (Φ → Φ) → Gdt → Φ × Ant Φ
| acc (Gdt.rhs rhs) := (Φ.false, Ant.rhs (acc Φ.true) rhs)
| acc (Gdt.branch tr1 tr2) :=
    let (U1, A1) := 𝒰𝒜_acc acc tr1,
        (U2, A2) := 𝒰𝒜_acc (U1.and ∘ acc) tr2
    in  (U2, Ant.branch A1 A2)
| acc (Gdt.grd (Grd.bang var) tr) :=
    let (U, A) := 𝒰𝒜_acc (acc ∘ (Φ.var_is_not_bottom var).and) tr
    in (U, Ant.diverge (acc (Φ.var_is_bottom var)) A)
| acc (Gdt.grd (Grd.tgrd grd) tr) :=
    let (U, A) := 𝒰𝒜_acc (acc ∘ (Φ.tgrd_in grd)) tr
    in ((acc (Φ.not_tgrd grd)).or U, A)
```

It is surprisingly easy to show that this function is really interleaving 𝒜_acc and 𝒰_acc:

```
theorem 𝒰𝒜_acc_eq (acc: Φ → Φ) (gdt: Gdt):
    𝒰𝒜_acc acc gdt = (𝒰_acc acc gdt, 𝒜_acc acc gdt) :=
by induction gdt generalizing acc;
    try { cases gdt_grd }; simp [𝒰𝒜_acc, 𝒰_acc, 𝒜_acc, *]
```

4 Correctness Statements

As we have all the required definitions at this point, we can state and formalize what we expect of the presented pattern match analyses to be considered correct. We provide proofs for all correctness propositions on GitHub [7]. Chapter 5 will discuss parts of these proofs in more detail.

4.1 Correctness of the Uncovered Analysis

$\mathcal{U}$ should compute a refinement type that denotes exactly all values that are not covered by a given guard tree. This does not include values under which the execution diverges!

The following theorem states correctness of $\mathcal{U}$ in Lean:

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

As an obvious consequence, a guard tree always matches (or diverges) if and only if the refinement type computed by $\mathcal{U}$ is empty. If a correct function $\mathcal{G}$ or `can_prove_empty` proves emptiness of such a computed refinement type, there are no uncovered cases by this theorem. Otherwise, a warning of potential uncovered cases should be issued!

Hence, this theorem implies correctness of the uncovered analysis: The uncovered analysis should rather report a false positive than not detect an uncovered case.

Note that this theorem carries over to all semantically equivalent definitions of $\mathcal{U}$.

4.1.1 Comparison to LYGs Notion Of Correctness

LYG states that "[...] LYG will never fail to report uncovered clauses (no false negatives), but it may report false positives" [1]. Our statement of $\mathcal{U}$s correctness is stronger: The function $\mathcal{U}$ computes a refinement type that covers exactly all environments that are not covered by the guard tree. If $\mathcal{G}$ is assumed to be correct and used to semi-decide whether the refinement type computed by $\mathcal{U}$ is empty, LYGs claim follows.

4.2 Correctness of the Redundant/Inaccessible Analysis

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. RHSs that are redundant can be removed from the guard tree without changing the semantics of the guard tree. This expresses correctness of the redundant and inaccessible analysis.

```
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
    )
```

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. However, we conjecture that a contains no inaccessible and i no redundant RHSs if `can_prove_empty` is both correct and complete (even though such a function is usually uncomputable).

4.2.1 Comparison to LYGs Notion Of Correctness

LYG states correctness of the redundant/inaccessible analysis as following: "Similarly, LYG will never report accessible clauses as redundant (no false positives), but it may fail to report clauses which are redundant when the code involved is too close to undecidable territory." [1]. Furthermore, LYG also states "A redundant equation can be removed from a function without changing its semantics, whereas an inaccessible equation cannot, [...]".

We both improved the precision of LYGs notion of correctness by formally defining every involved concept, but also made it more complete by stating that RHSs identified as redundant by LYG are indeed redundant.

While the predecessor of LYG, "GADTs Meet Their Match" [8] (in short *GMTM*), gives a formal statement about its correctness in theorem 1, it lacks a proof. Also, according to our understanding, GMTM's statement does not explicitly examine the effect of removing redundant right hand sides as we do.

5 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 [7].

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, as $\mathcal{U}$'s correctness only depends on the semantic of the computed refinement type (see chapter 4.1). It is much more difficult to show correctness of $\mathcal{R}/\mathcal{A}$ though, so we will discuss this in more detail.

In chapter 5.2, we show that redundant RHSs can be removed without changing semantics. Then, in chapter 5.3, 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 of the uncovered/redundant analysis as presented in chapter 4.2.

In total, we declared 48 definitions and proved 143 lemmas and theorems, resulting in 2009 lines of Lean code!

5.1 Simplification A of $\mathcal{A}$

It is difficult to reason about $\mathcal{A}\_acc$ and thus $\mathcal{A}$, as we are only interested in certain well behaving accumulator values (in particular homomorphisms) and not arbitrary functions. Let us have another look at the definition of $\mathcal{A}$:

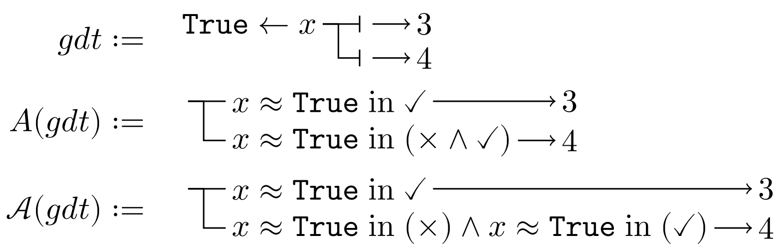
```
def 𝒜_acc : (Φ → Φ) → Gdt → Ant Φ
| acc (Gdt.rhs rhs) := Ant.rhs (acc Φ.true) rhs
| acc (Gdt.branch tr1 tr2) := Ant.branch
        (𝒜_acc acc tr1)
        (𝒜_acc ((𝒰_acc acc tr1).and ∘ acc) tr2)
| acc (Gdt.grd (Grd.bang var) tr) := Ant.diverge
        (acc (Φ.var_is_bottom var))
        (𝒜_acc (acc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

def 𝒜 : Gdt → Ant Φ := 𝒜_acc id
```

Since $\mathcal{A}$ is central to many propositions, we define a much simpler 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.tgrd grd) tr) := (A tr).map (Φ.tgrd_in grd)
```

However, $\mathcal{A}$(gdt) is not syntactically equal to A(gdt) for every gdt, as the following example shows:



$gdt :=$
True ← x ⟶ 3
⟶ 4
$A(gdt) :=$
$x \approx$ True in ✓ ⟶ 3
$x \approx$ True in $(\times \wedge \checkmark)$ ⟶ 4
$\mathcal{A}(gdt) :=$
$x \approx$ True in ✓ ⟶ 3
$x \approx$ True in $(\times) \wedge x \approx$ True in $(\checkmark)$ ⟶ 4

This counterexample can easily be verified by Lean:

```
theorem A_neq_𝒜 (r: Rhs) (g: TGrd): A ≠ 𝒜 :=
begin
    intro,
    replace a := congr_fun a (Gdt.grd (Grd.tgrd g)
        (
                    (Gdt.rhs r)
            .branch (Gdt.rhs r)
        )),
    finish [A, 𝒜, 𝒜_acc, Ant.map],
end
```

We are unsure whether the definition of A and $\mathcal{A}$ can be adapted to get syntactical equality while maintaining the simplicity of A and aligning the recursion structure of $\mathcal{A}$ and $\mathcal{U}$ (see chapter 3.6).

Instead, we define a semantics on `Ant Φ` and show that A and $\mathcal{A}$ have the same semantics:

```
def Ant.eval_rhss (ant: Ant Φ) (env: Env): Ant bool :=
    ant.map (λ ty, ty.eval env)

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

When only relying on semantical equivalence, care has to be taken when getting insights into $\mathcal{A}$ by studying A, as `can_prove_empty` does not have to be *well defined* on refinement types modulo semantical 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 $\times$ - it would need to return `true` for all refinement types that are empty! Thus, `can_prove_empty` must operate on the refinement types of $\mathcal{A}$.

5.2 Redundant RHSs Can Be Removed Without Changing Semantics

5.2.1 Proof Idea

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 semantics. 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 5.1 for why this is important). This internal statement forms the second part of the correctness property defined in chapter 4.2 and is formalized as follows:

```
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
```

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 given guard tree on env and safely remove all RHSs except the one the evaluation ended with. We call RHSs that play no role in the evaluation on env *inactive*, the resulting RHS is called *active*. If the evaluation diverged however, the diverging bang guard must not be removed; thus, all RHSs behind the diverging bang operator except one can be removed. In this case, the bang guard is *active* and all RHSs are inactive. Clearly, at most one node (RHS or bang guard) is active.

The function `Gdt.mark_inactive` directly computes a boolean annotated tree that marks inactive nodes for a given guard tree and environment. 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. This function equals the negation of the semantic of trees annotated with refinement types!

It remains to relate the set of RHSs $r := \mathcal{R}(\mathcal{A}(\text{gdt})).\text{red}$ to the RHSs that can be removed when focusing on a particular environment.

Figure 5.1 sketches the proof idea. Thin arrows mark the data flow, fat arrows the flow of reasoning.

Step 1: Defining gdt and $\mathcal{A}(\text{gdt})$

We start with a guard tree gdt and its annotated tree $\mathcal{A}(\text{gdt})$.

As a detail in the formal proof, we actually use `Agdt` instead of $\mathcal{A}(\text{gdt})$, but since $\text{ant}_2 := \mathcal{A}(\text{gdt}).\text{map}(\neg \circ \Phi.\text{eval}_{\text{env}})$ only depends on the semantic of $\mathcal{A}(\text{gdt})$ and Agdt has the same semantic, this does not change the proof idea.

Step 2: Decomposing $\mathcal{R}$ into R and Ant.map(can_prove_empty), Defining ant_1

To better understand $\mathcal{R}$, we decompose $\mathcal{R}$, which takes an Ant Φ and needs a function `can_prove_empty`, into a function R that takes an Ant bool and a function $f := \text{map}(\text{can\_prove\_empty})$ that computes an Ant bool from an Ant Φ so that $\mathcal{R} = R \circ f$.

In figure 5.1, $\text{ant}_1 := f(\mathcal{A}(\text{gdt}))$ represents the object that R works on. Clearly, $\mathcal{R}(\mathcal{A}(\text{gdt})).\text{red} = R(\text{ant}_1).\text{red}$. In this particular example, only the refinement type associated with RHS 1 is recognized as empty and we have $\mathcal{R}(\mathcal{A}(\text{gdt})).\text{red} = \{\texttt{RHS 1}\}$, as indicated by the ellipsis.

Step 3: Defining ant_3 and ant_2

ant_3 in figure 5.1 is a boolean annotated tree whose nodes indicate inactivity under env (true if they are inactive, otherwise false). It is much easier to reason about the effect of removing selected RHSs from this tree due to the closely related definitions of `Gdt.mark_inactive` and `Gdt.eval`, especially if the selection of RHSs is done by only looking at ant_3.

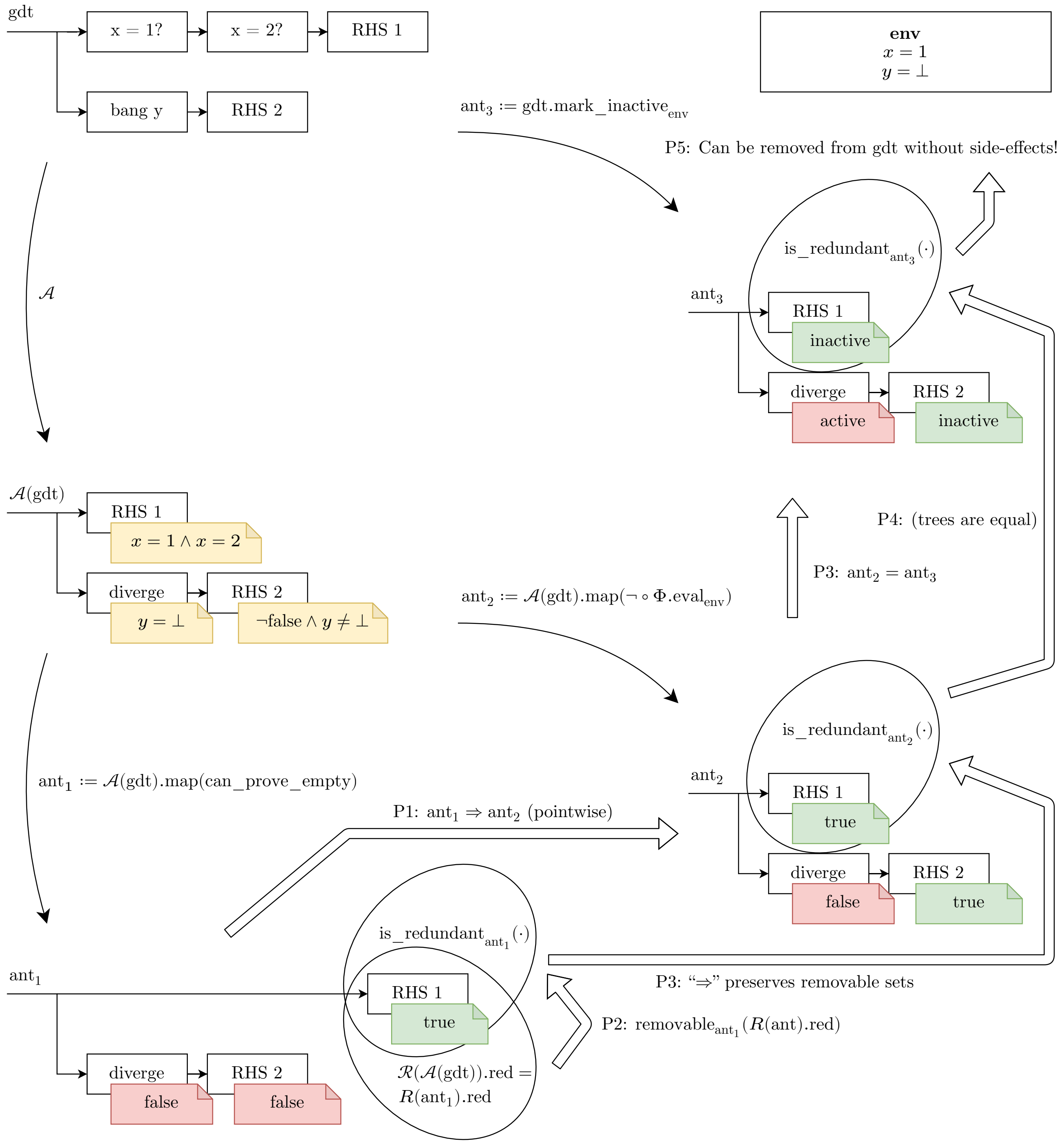
It is easy to relate ant_1 with ant_3 if we define $\text{ant}_2 := \mathcal{A}(\text{gdt}).\text{map}(\neg \circ \Phi.\text{eval}_{\text{env}})$ as the negation of the evaluation of each refinement type under env.

Step 4: Relating ant_1, ant_2 and ant_3

We can show that each boolean annotation in ant_1 implies ("$\Rightarrow$") the corresponding boolean annotation in ant_2 pointwise (P1): If a refinement type is empty, it must not match any environment.

We can also show $\text{ant}_2 = \text{ant}_3$ (P3), since a node is active under env if and only if the corresponding refinement type matches env.

Figure 5.1: Proof Overview: Redundant RHSs can be removed without changing semantics.



gdt
x = 1?
x = 2?
RHS 1
bang y
RHS 2
env
$x = 1$
$y = \bot$
$\text{ant}_3 := \text{gdt.mark\_inactive}_{\text{env}}$
P5: Can be removed from gdt without side-effects!
$\text{is\_redundant}_{\text{ant}_3}(\cdot)$
ant_3
RHS 1
inactive
diverge
active
RHS 2
inactive
$\mathcal{A}$
$\mathcal{A}(\text{gdt})$
RHS 1
$x = 1 \land x = 2$
diverge
$y = \bot$
RHS 2
$\neg\text{false} \land y \neq \bot$
P4: (trees are equal)
P3: $\text{ant}_2 = \text{ant}_3$
$\text{ant}_2 := \mathcal{A}(\text{gdt}).\text{map}(\neg \circ \Phi.\text{eval}_{\text{env}})$
$\text{is\_redundant}_{\text{ant}_2}(\cdot)$
ant_2
RHS 1
true
diverge
false
RHS 2
true
$\text{ant}_1 := \mathcal{A}(\text{gdt}).\text{map}(\text{can\_prove\_empty})$
P1: $\text{ant}_1 \Rightarrow \text{ant}_2$ (pointwise)
$\text{is\_redundant}_{\text{ant}_1}(\cdot)$
ant_1
RHS 1
true
diverge
false
RHS 2
false
$\mathcal{R}(\mathcal{A}(\text{gdt})).\text{red} = R(\text{ant}_1).\text{red}$
P3: "⇒" preserves removable sets
P2: $\text{removable}_{\text{ant}_1}(R(\text{ant}).\text{red})$

Step 5: Exploiting the Relationship

It is easy to show that any subset of RHSs $R(\text{ant}_3)$.red can be removed from *gdt* without changing its semantic on env. We have to show the same for R(ant$_1$).red. We hoped that R(ant$_1$).red would be a subset of R(ant$_2$).red (due to ant$_1 \Rightarrow$ ant$_2$) to complete the proof. However, this is not the case! See chapter 5.2.2 for a counterexample.

To repair the proof idea, we defined a predicate `is_redundant_set` (for brevity called `is_redundant` in figure 5.1) on sets of RHSs for a given boolean annotated tree. This predicate has the property that $R(\text{ant}_1)$.red is a redundant set (P2, hence {`RHS 1`} is a redundant set) and that if r is a redundant set in ant$_1$ and if ant$_1 \Rightarrow$ ant$_2$, then r is also a redundant set in ant$_2$ (P3).

Finally, we show that RHSs that are redundant in `gdt.mark_inactive`$_{\texttt{env}}$ can be removed from guard trees without changing their semantic under `env` (P5). This finishes the proof!

5.2.2 Generalization of $R(\_)$.red

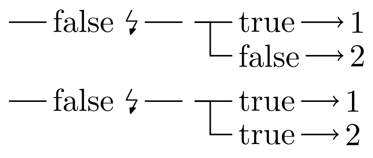
Given two boolean annotated trees ant$_a$ and ant$_b$ with ant$_a \Rightarrow$ ant$_b$, we would like to transfer insights into redundant sets in ant$_a$ to ant$_b$ as stated in the previous chapter.

R is not suitable

We cannot use R directly: R(ant$_a$).red does not need to be a subset of R(ant$_b$).red! In fact, they can be disjoint, as the following counterexample shows.

ant$_a$:=

ant$_b$:=



Clearly, it is ant$_a \Rightarrow$ ant$_b$, but R(ant$_a$) = {1} and R(ant$_b$) = {2}. This counterexample can easily be verified with Lean, a proof is included in [7].

This issue is caused by the freedom of how *critical sets* of RHSs can be avoided and that R does not always consider this freedom. A set of RHSs is critical if removing all its RHSs necessarily also removes a bang guard associated with a non-empty refinement type. Clearly, a set of redundant right hand sides must not contain a critical set - otherwise, a possibly active bang guard might be removed. This could be observable and would contradict the definition of a redundant set to not cause observable side effects on removal!

Hence, if all RHSs behind a possibly active bang guard are inaccessible (as in ant$_b$ in the counterexample), not all of them can be marked as redundant. In such cases, R marks all RHSs as redundant except the first. However, R could have also excluded any other RHS instead (i.e. the second one in the example), which would equally avoid the critical set caused by the bang guard! If such a RHS is not inaccessible (as in ant_a in the counterexample), R does not have to exclude any inaccessible RHS from being redundant (and thus marks the first as redundant in the example). In this case, R makes use of the freedom of how critical sets can be avoided by cleverly using an active RHS instead of just the first RHS as witness.

Definition of `is_redundant_set`

To overcome this issue, we generalize $R(\_).\text{red}$ to a predicate `is_redundant_set` as follows:

```
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. The purpose of critical sets is to ensure 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: Clearly, R avoids all critical sets and only marks inactive RHSs as redundant.

We believe that $\mathcal{R}(\_).\text{red}$ actually computes a largest redundant set given a boolean annotated tree. However, a largest redundant set does not need to be unique! If R would exclude the last inaccessible RHS instead of the first from being redundant, R would compute a different redundant set of equal size.

It is also simple to show that the predicate becomes less strict the more nodes are marked as inactive, as the amount of critical sets decreases and the amount of inactive RHSs increases.

5.2.3 Formal Proof

The complete formal proof follows. All used lemmas and their proofs can be found at [7].

```
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 :=
begin

ext env:1,

-- `can_prove_empty` approximates emptiness for a
-- single refinement type.
-- `ant_empt` approximates emptiness of the
-- refinement types in `Agdt` for every `env`.
-- It also approximates inactive rhss of `gdt` in
-- context of `env` (ant_empt_imp_gdt).
let ant_empt := Agdt.map can_prove_empty.val,
have ant_empt_imp_gdt := calc
    ant_empt ⇒ Agdt.mark_inactive_rhss env
        : can_prove_empty_implies_inactive can_prove_empty Agdt env
    ...       ⇒ (A gdt).mark_inactive_rhss env
        : by simp [Ant.implies_refl, ant_def]
    ...       ⇒ gdt.mark_inactive_rhss env
        : by simp [Ant.implies_refl, A_mark_inactive_rhss gdt env],

-- Since `gdt` has disjoint rhss, `ant_empt` has disjoint rhss too.
have ant_empt_disjoint : ant_empt.disjoint_rhss
    := by simp [Ant.disjoint_rhss_of_gdt_disjoint_rhss gdt_disjoint,
            Ant.disjoint_rhss_iff_of_mark_inactive_rhss_eq
                (function.funext_iff.1 ant_def env)],

-- The set of rhss `R_red` is redundant in `ant_empt` (red_in_ant_empt).
-- This means that these rhss are inactive and
-- not all rhss of possibly active diverge nodes are redundant.
let R_red := (R ant_empt).red.to_finset,
have red_in_ant_empt: ant_empt.is_redundant_set R_red
```

```
    := R_red_redundant ant_empt_disjoint,

-- Since `redundant_in` is monotone and `ant_empt`
-- approximates inactive rhss on `gdt`,
-- `R_red` is also redundant in `gdt` (red_in_gdt).
have red_in_gdt: (gdt.mark_inactive_rhss env).is_redundant_set R_red
    := is_redundant_set_monotone _ ant_empt_imp_gdt red_in_ant_empt,

-- Since `R_red` is a redundant set, it can be removed from `gdt` without
-- changing the semantics. Note that `R_red` is independent of env.
show Gdt.eval_option (Gdt.remove_rhss R_red gdt) env = gdt.eval env,
from redundant_rhss_removable gdt gdt_disjoint env _ red_in_gdt,

end
```

5.3 Accessible RHSs Must Be Detected as Accessible

For the correctness of the inaccessible/redundant analysis, it remains to show that the analysis correctly identifies all potentially accessible RHSs.

This is formalized by the following lemma:

```
lemma R_acc_mem_of_reachable
    { gdt: Gdt } { env: Env } { rhs: Rhs } { ant: Ant Φ }
    (gdt_disjoint: gdt.disjoint_rhss)
    (can_prove_empty: CorrectCanProveEmpty)
    (Agdt: ant.mark_inactive_rhss env = (A gdt).mark_inactive_rhss env)
    (h: gdt.eval env = Result.value rhs)
    { r: RhsPartition }
    (r_def: r = R (ant.map can_prove_empty.val)):
    rhs ∈ r.acc \ (r.inacc ++ r.red)
```

As in chapter 5.2, `Agdt` abstracts from the syntactical structure of the refinement types in `A` *gdt*. Given that *gdt* evaluates to *rhs* under *env*, we want to show that R marks *rhs* as accessible and not as inaccessible or redundant.

This proof is very technical, so we concentrate on the key insights.

First, we show that the accessible, inaccessible and redundant RHSs as identified by R form a partition of all RHSs. This is simple to prove and expressed by the following lemma ($a \sim b$ denotes that the list a is a permutation of b):

```
lemma R_rhss_perm { ant: Ant bool }:
    ((R ant).acc ++ (R ant).inacc ++ (R ant).red) ~ ant.rhss_list
```

Clearly, *rhs* is a RHS in *gdt* and thus *ant* and `ant.map can_prove_empty.val`. Since R computes a partition of all RHSs, it remains to show that *rhs* is neither contained in `r.inacc` nor in `r.red`.

With the following lemma we only need to show that *rhs* is not an inactive RHS in `ant.map can_prove_empty.val`:

```
lemma R_inacc_unon_R_red_subseteq_inactive (ant: Ant bool):
    (R ant).inacc.to_finset ∪ (R ant).red.to_finset
    ⊆ ant.inactive_rhss
```

In fact, *rhs* is the only active RHS in `gdt.mark_inactive_rhss env`, as the following lemma shows:

```
lemma gdt_mark_inactive_rhss_inactive_rhss_of_rhs_match
    { gdt: Gdt } { env: Env } { rhs: Rhs }
    (gdt_disjoint: gdt.disjoint_rhss):
    gdt.rhss \ (gdt.mark_inactive_rhss env).inactive_rhss = { rhs }
    ↔ gdt.eval env = Result.value rhs
```

From chapter 5.2, we know that (`ant.map can_prove_empty.val`) pointwise implies (`gdt.mark_inactive_rhss env`): empty refinement types imply inactivity. When we proved that ant_b is a redundant set if ant_a is a redundant set and $ant_a \Rightarrow ant_b$ (as discussed in chapter 5.2.2), we first showed a stronger result that we can reuse now to relate inactive RHSs of `ant.map can_prove_empty.val` and `gdt.mark_inactive_rhss env`:

```
lemma is_redundant_set_monotone' { a b: Ant bool } (h: a ⇒ b):
        a.inactive_rhss ⊆ b.inactive_rhss
        ∧ b.critical_rhs_sets ⊆ a.critical_rhs_sets
```

We can use this fact and the previous lemmas to show that *rhs* must be an active RHS in `ant.map can_prove_empty.val`. This completes the proof.

6 Conclusion

We refined and formalized several correctness properties of LYG and successfully proved them in Lean. However, we parametrized these correctness properties over a correct function $\mathcal{G}$ that semi-decides emptiness of refinement types. While LYG defines such a function, we did not prove that it indeed is such a correct function $\mathcal{G}$.

Even though these correctness properties look seemingly easy to prove, it turned out to be a very involved undertaking. After all, it took us 48 definitions and 143 lemmas and theorems to formalize these proofs in Lean!

We believe that this complexity is caused by the amount of details required to describe LYG and the rigorousness of Lean. In fact, we discovered a minor flaw in LYG's definition of $\mathcal{U}$, buried in the details of the let binding semantics. Luckily, this flaw has no impact on the GHC implementation of LYG, as the implementation uses a different encoding of refinement types. Still, this flaw was not discovered in peer reviews of the LYG paper, showing that LYG's correctness is not obvious at all and making a strong point for verification, yet formal verification.

As our proofs are formally verified by Lean, it is highly unlikely that LYG as formalized by this thesis has any other flaws, except in the definition of the presented function $\mathcal{G}$ that we did not check. Nonetheless, our formally verified proofs alone cannot guarantee that our formalization correctly reflects LYG and that our chosen abstractions are general enough, as this is subject to human interpretation. We hope that our informal clarifications support our claim that they do.

We can strongly recommend to use Lean for formal verification and suggest to formally verify the correctness of $\mathcal{G}$ too!

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] T. Freeman and F. Pfenning, "Refinement types for ml," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, (New York, NY, USA), p. 268–277, Association for Computing Machinery, 1991.

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

[4] 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.

[5] 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.

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

[7] H. Dieterichs, "Lean proof." `https://github.com/hediet/masters-thesis/tree/9524e79f09771a6d9d74f75556a3adbff683ed35/code`. Retrieved: 20 Jan. 2021.

[8] G. Karachalias, T. Schrijvers, D. Vytiniotis, and S. P. Jones, "Gadts meet their match: Pattern-matching warnings that account for gadts, guards, and laziness," *SIGPLAN Not.*, vol. 50, p. 424–436, Aug. 2015.

Erklärung

Hiermit erkläre ich, Henning Dieterichs, dass ich die vorliegende Masterarbeit 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. Außerdem will ich mich bei der Lean Community bedanken, die mir bei Fragen zu Lean viel geholfen hat.