Compiling Dependent Type Preconditions to Runtime Checks With Agda2Hs

Master's Thesis

Jakob Naucke

Compiling Dependent Type Preconditions to Runtime Checks With Agda2Hs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jakob Naucke
born in Altdorf bei Nürnberg, Germany

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.tudelft.nl/eemcs

Compiling Dependent Type Preconditions to Runtime Checks With Agda2Hs

Author: Jakob Naucke
Student ID: 5845483

Date of defence: 12 December 2024

Abstract

Formally verified programs can be embedded in larger non-verified code bases by means of syntactically faithful source-to-source translation: systems like *Agda2Hs* make it possible to translate verified code written in a dependently typed programming language to a general-purpose functional programming language, Agda and Haskell in this case. Such systems can enable verification for critical functionality while keeping wider ecosystem access and easier to write code for peripheral functionality. However, this interfacing leaves a gap in the formal guarantees of the verified code: preconditions that were only expressible as a dependent type and have thus got erased upon translation might not be met. We present *runtime checking* these preconditions as a way to close this gap and ensure that computation does not continue on ill-formed input. As an extension to Agda2Hs, we have implemented a solution to automatically insert runtime checks for the preconditions and only make those definitions accessible in the output that are also checkable. The runtime check insertions do not only cover functions, but also data types and non-class records in the form of smart constructors; higher-order erased arguments are supported as well. We make a formal completeness analysis of a simplified version of the checks we generate for well- and ill-typed programs.

In our solution, class instances are not supported for runtime checking due to their different nature, and capabilities for recovering from a failed runtime check are still rudimentary. Despite these limitations, we conclude that a closure of the input precondition verification gap is possible, and that the development time trade-off in comparison to handwriting checks can be worthwhile.

Thesis Committee:

Chair: Dr. J.G.H. Cockx, Faculty EEMCS, TU Delft
Committee Member: B. Liesnikov, Faculty EEMCS, TU Delft
Committee Member: Dr. A. Panichella, Faculty EEMCS, TU Delft

Contents

List of Listings

Chapter 1

Introduction

Today's world has grown dependent on computer software in various aspects of modern life. Many of these software applications exist not only for convenience, but for the economic order of the world, humans' safety in the built environment, and in transit. Software bugs can lead to severe ramifications, so among other fields, *formal* or *mechanised verification* aims to remove them before programs are run. Verification is a long-standing and active field of computer science (Meyer 1986; Eremondi 2023).

The overhead of verifying software as opposed to merely writing it is often very large. This has lead to approaches like *Agda2Hs* (Cockx et al. 2022), a way to integrate verified software into a more mainstream, unverified programming language code base. As the name implies, Agda2Hs translates between Agda, a dependently typed functional language, and Haskell, a general functional programming language. However, this method leaves a gap in the verification of the software written in Agda: preconditions to definitions that are expressible at type level cannot be translated to Haskell, which can lead to errors without obvious origin. We present an approach to insert *runtime checks* in these cases automatically.

1.1 Agda2Hs

Agda (Norell 2007) is a dependently typed programming language. In particular, this means that arguments to a function can arbitrarily be quantified over in its type specification. Like most languages of this kind, it is also a proof assistant: the Curry–Howard correspondence (Howard 1980) shows that a function in such a language that type checks is simultaneously a proof of its type. This type could represent a mathematical theorem based on some axiomatic system, or e.g. an assertion about the behaviour of another function written in this language. By default, Agda definitions are terminating (i.e. recurse only sub-structurally), total (i.e. be defined on their entire domain), and type safe.

These properties frequently lead to a need to prove behaviours of functions, unlike (optional) test cases: for example, array access at a particular index must be proven to be sound, i.e. the index must be mathematically proven to be within the array's bounds. Thus, languages like Agda present one of the approaches to mechanised verification of software. Note that these languages are functional programming languages and that this method of verification is not to be confused with the verification of imperative programs, which is usually based on Hoare logic.

Creating proofs in a proof assistant like Agda is often perceived as difficult (Juhošová 2024) and can considerably overwhelm the time spent implementing functionality. Additionally, the style in which proofs must be written is more formal than what most mainstream mathematics literature does on paper, although having much stronger guarantees. These difficulties have hindered the more wide-spread adoption of such languages.

This observation drove the introduction of the Agda2Hs source-to-source translation system by Cockx et al. 2022. The target language, Haskell, is a pure functional programming language: all side effects of functions are part of their type, which makes them easier to reason about. These guarantees are a subset of those of dependently typed languages. Among the (relatively) mainstream styles of programming, this is the one most closely related to dependently typed programming. Agda2Hs then allows translating Agda programs into Haskell while, unlike Agda's existing Haskell backend, remaining close in syntax to the original Agda source and maintaining readability. This enables embedding some logic that has been formally verified in a larger, potentially preexisting project, with a richer ecosystem access than would be available in native Agda.

1.2 Problem Illustration

As mentioned in the introducing words, this approach has a drawback: on the Agda side, there can be a precondition on the input that was expressed as a dependent type, but was then erased upon translation to Haskell. Such input would then unsafely be given to code translated from Agda as an assumed truth.

In the following example, an Agda function intended to be compiled to a Haskell function subtracts a natural number y from a natural number x, yielding a natural number. With non-modified subtraction, this is only possible when the result is not less than zero. This means that x must be greater than y. The proof is marked as erased with `@0` because this data cannot be supplied from Haskell as it is a dependent type. Thus, even when the function would always produce correct output when called from Agda, this guarantee is lost when calling from Haskell after passing the function through Agda2Hs.

```
subtractFromGreater : (x y : Nat) → @0 ⦃ IsFalse (x < y) ⦄ → Nat
subtractFromGreater x y = x - y
{-# COMPILE AGDA2HS subtractFromGreater #-}
```

Listing 1.1: Motivating example: subtracting numbers where one is greater than the other

This shortcoming has also been acknowledged by Cockx et al. 2022 under Future Work. In these situations, they suggested to use runtime checks on the Haskell side instead. We present a solution that adds checks for all such arguments.

1.3 Contributions and Limitations

We show how the rigour of pure Agda can be retrieved back, albeit at a performance cost, by adding automatic runtime checks to Agda2Hs. We restrict the exposition of functions on the Haskell side to those that have such a deciding function for *all* erased arguments. This restriction guarantees that no unchecked input can be passed. These changes to Agda2Hs are implemented as an optional feature since they are generally incompatible with existing code. They align with the goals of Agda2Hs in that the source output remains human-readable. Specifically:

- We extend Agda2Hs with a command line flag which, when enabled, uses *instance search* to find deciding predicates:
  - For erased arguments to functions, turning them into a human-readable, runtime checked version that calls the unchecked function only if this check passes. Otherwise, an error is thrown. The unchecked function is placed in a separate "post-runtime checks" module and not exported otherwise. An exception is made

for callers that were also translated from Agda and thus will not call the function with unverified input.

- For erased arguments in data type and record constructors, turning them into an exposed *smart constructor* that performs the runtime check similarly, and an internal unchecked constructor for other Agda functions to use.
- For erased arguments within other types, i.e. nested preconditions used inside a lambda, turning them into checked functions inside the `where` block of functions and smart constructors. This separation is made to avoid Haskell expressions with very deep nesting.

Under this flag, functions and constructors that lack respective instances deciding their erased arguments remain hidden in the respective post-runtime checks module. These modules are not to be used by code written in Haskell natively. We could make the unchecked definitions inaccessible altogether, but this would make it impossible to continue calling the unchecked versions from code translated from Agda, which would be inefficient.

- We show how it becomes impossible to compute on input that does not satisfy the conditions expressed in Agda in the transpiled Haskell code without yielding an error when this flag is enabled. This analysis is not mechanised and based on some assumptions about Agda2Hs. Establishing a formal, let alone mechanical, analysis of the translation guarantees of existing Agda2Hs itself and these new features remains future work.

Agda2Hs is meant to be a research vehicle for such checks here. We expect a similar approach to be equally feasible in other systems, such as Coq's program extraction. However, Coq is not as focused on being a programming language and its program extraction has less focus on a readable output than Agda2Hs. Our work covers our Agda2Hs implementation, but intends to discuss the general concept of using additional runtime checks at the gap between verified and pure functional programming.

We will re-address limitations in the discussion—for things we believe could only be built at the cost of other drawbacks—and conclusion—for things considered future work, but for a brief and limited introduction:

- Failure is expressed as an `error`. This provides an easy way of showing how gaps in preconditions can be closed, but is not ideal from a software engineering perspective. This could be extended to layered error handling (cf. section 8.1).
- Records compiled to classes cannot contain erased arguments because unlike with smart constructors, for classes, there is no obvious way to compute a runtime check. It would, however, be possible to compute runtime checks for at least the instances to classes (section 6.2.1).
- Parametric undecidable data types and records that become decidable by their parameters are not checked. This can generally be worked around by rewriting functions accordingly (section 6.2.2).
- We cannot maintain evaluation order and laziness of the arguments (section 6.3).

1.4 Overview

In this thesis, we begin by establishing preliminaries about Agda2Hs itself, specific concepts used in this implementation, and runtime checks (chapter 2). We then present the feature itself from a user's perspective (chapter 3). Due to their complexity, we give a rundown of

some more involved excerpts of our implementation (chapter 4). We provide a formal analysis of a simplified version of our checks regarding their completeness (chapter 5). Following this, we make a comparison of the automatic check generation to a handwritten checks approach, discuss potential further features, their drawbacks, and a general drawback to this approach with respect to lazy evaluation (chapter 6). At the end, related (chapter 7) and future work (chapter 8) is discussed.

Chapter 2

Background

In this chapter, we briefly go over the fundamentals of the Agda language, and how its dependent type system sets it apart from other languages. We explain the motivation and general workings of the Agda2Hs translation system. Furthermore, we explain two important challenges in creating automatic runtime checks for preconditions written in Agda, and how we resolved them: finding a term that decides a precondition, and hiding unchecked translated definitions from the user.

This chapter features both Agda and Haskell code, which can look similar at surface level. To distinguish between them, Haskell code is shown with a tinted background. Syntax highlighted Agda code depicted in code blocks has been type checked.

2.1 Agda

Agda is a dependently typed programming language, which means that it can be used as a proof assistant that checks proofs written in it, ensuring that they are correct. If Agda type checked a definition that formulates an incorrect proof, this would be a bug in Agda. These proofs can be about anything expressed in the language, e.g. mathematics or programs.

```
zeroLtSuc : (a : Nat) → IsTrue (0 < (a + 1))
zeroLtSuc zero = IsTrue.itsTrue
zeroLtSuc (suc _) = IsTrue.itsTrue
```

Listing 2.1: Proving in Agda: pattern matching is case distinction

Listing 2.1 displays a simple proof that was created as a lemma in the test suite to our contributions. In mathematics, it would be expressed as $\forall a \in \mathbb{N} : 0 < (a + 1)$. The pattern matching in this function is effectively a proof by case distinction on natural numbers (zero and successor). `IsTrue` is a data type that has `itsTrue` as its sole constructor for decidably true propositions. Agda is a total programming language, meaning that omitting one of the cases would make the proof incomplete and prohibit the program from type checking.

The proof on the behaviour of a function is also a function; there is no border between definitions that compute data and definitions that exist just to show a property. This non-distinction is a manifestation of the Curry–Howard correspondence (Howard 1980): a dependently typed program that type checks can be seen as a proof of its type.

To be sound, an Agda definition may require arguments that would typically not be found in other programming languages. Listing 2.2[1] shows a function that retrieves a list item at a given index, additionally requiring the precondition that the index does not exceed the list's

[1]To avoid confusion: `a : Set` does not mean that `a` is a set in set theory, but is of a type in the universe of small types which is called `Set`.

```
_!!ᴺ_ : {a : Set} (xs : List a) (n : Nat) → @0 ⦃ IsTrue (n < lengthNat xs) ⦄ → a
(x :: xs) !!ᴺ zero = x
(x :: xs) !!ᴺ suc n = xs !!ᴺ n
```

Listing 2.2: Preconditions in a total programming language: list lookup

bounds. Removing this precondition will result in the definition no longer type checking: the left-hand side pattern matching will be incomplete because it does not include the case of the empty list. In Agda, recursive definitions must also pass *termination checking* in order to make type checking decidable (Norell 2007). A successful termination check can typically be achieved by ensuring that a recursion call is sub-structural, e.g. operating only on the tail of a list.

As a side note, in that definition, `a` is an *implicit argument*, which means that it can usually be inferred from a dependent type in the definition type and does not generally need to be specified upon calling.

2.2 Agda2Hs

Requirements such as mandatory proofs and termination checking are often perceived as difficult to work with. This has contributed to the fact that dependently typed languages like Agda have remained relatively niche. As a result, the ecosystem of libraries, tooling, and interfacing with other languages is also lacking.

This observation was a driving factor in the invention of one approach to *gradual verification* that we use in this thesis, being the Agda2Hs source-to-source translation system (Cockx et al. 2022). What makes this gradual is the fact that verified Agda code can be used inside a larger unverified, although very strongly typed Haskell code base. This code base may already exist, allowing for a verification retrofit, and/or correctness may be more important to be guaranteed in some parts of the logic than in others. Despite native Agda code (i.e. code not intended to be fed to Agda2Hs) also compiling to Haskell by default, Agda2Hs's approach is very different: it uses native Haskell data types as much as possible and creates human-readable source output for better integration. Centrally, Agda2Hs is based on a partial reimplementation of Haskell's `Prelude` base library in Agda, treating it as its trusted computing base (TCB).

Agda2Hs leverages the Agda feature of *erased arguments* to translate code. Only unerased arguments are translated to Haskell. A definition's type may include arguments that cannot be translated, but are also required exclusively to make the program sound, *not* to compute output. Agda's type checker ensures that an erased argument is of no runtime relevance (Agda Development Team 2024b, Run-time Irrelevance). The introductory listing 1.1 provided a simple example of an erased argument: `IsFalse (x < y)` plays no role in the computation and is marked erased. A use case of erased arguments beyond Agda2Hs is saving memory in recursive calls.

Without our extension to Agda2Hs, listing 1.1 translates to what is shown in listing 2.3. The `COMPILE AGDA2HS` pragma in the former listing is required to trigger translation. We omit the pragma in subsequent examples except where its use is unclear.

```
import Numeric.Natural (Natural)

subtractFromGreater :: Natural -> Natural -> Natural
subtractFromGreater x y = x - y
```

Listing 2.3: Agda2Hs: non-runtime-checked output of listing 1.1

Despite Haskell being not as popular for industry projects as many imperative languages, it is a reasonable middle-ground to work in for a translation system like Agda2Hs. Like Agda, Haskell has pattern matching, lazy evaluation, support for currying, and is overall syntactically similar. Haskell is also a pure functional programming language that relies on monads to exert side effects.

Agda2Hs's approach has some drawbacks compared to native Agda programming. Agda's standard library (Agda Development Team 2024c), which implements many useful data structures and proofs related to them, can be used in principle, but its data types do not align with those of Agda2Hs. Only Agda's `Builtin` modules have such implementations, making e.g. rational numbers, finite numbers, trees or maps not natively available.

In this thesis, we consider a gap in the soundness of Agda2Hs: the correctness of the definition of constructs like functions and data types can depend on a precondition expressed in a dependent type, which cannot be translated due to a lack of dependent types in Haskell. Consequently, functions can be called with input not satisfying the precondition.

We present a novel optional Agda2Hs feature that generates runtime checks for violations to any erased argument. These checks prevent errors from happening at a later point in the computation where their cause may no longer be obvious, as well as prohibiting invalid output of functions occurring due to invalid input. They also enable handling and recovering from this violation, although these capabilities are extremely rudimentary as of now.

2.3 Finding terms that decide preconditions

To generate such checks, Agda types must be converted to Haskell terms in a correct way. Consider `subtractFromGreater` from listing 1.1 again. It has `IsFalse (x < y)` as an erased precondition. We must now find a way to *decide* this type and turn it into a term such as `not (x < y)` which we can then use as a Haskell guard.

Generally, deciding a type means deciding whether it can be inhibited from context or not. For our purposes, we can understand decidability as a Boolean, and a proof that this Boolean reflects the type decided over, i.e. the type can be inhibited from context if and only if this Boolean is **true**. Indeed, Agda2Hs defines the `Dec` type as a dependent pair consisting of these elements. Our mechanism for finding a term that decides a precondition then is to wrap the precondition in this `Dec` type, finding a suitable term with *instance search*, and compiling it to Haskell.

Instance search is an Agda compiler technology that is normally used to resolve instance arguments, which are marked by the double braces (`⦃…⦄`). Like implicit arguments (`{…}`), instance arguments can be used to specify arguments that are expected to be able to be inferred, and specified explicitly only when necessary. However, instance arguments are more powerful than implicit arguments and can e.g. be used for type class constraints (Agda Development Team 2024b, Instance Arguments).

The `IsFalse (x < y)` from above was also given as an instance argument, which means that the program will only type check if, for every mention of this function, instance search finds a proof of this condition. This proof can also be supplied explicitly, for example if an additional lemma is required to show that this precondition holds. Listing 2.4 shows the cases of implicit and explicit instance argument supply in action, requiring a small proof by induction for the latter to show that $x + 1 \not\leq x$. Listing 2.5 has no syntax highlighting because it also does not type check, it is an invalid call because $1 < 2$.

Recall that we also use instance search in our approach to generating runtime checks, but instead to turn e.g. `IsFalse (x < y)` into something resembling `not (x < y)`. For this instance search to have a result, it is necessary to have a `Dec` instance of `IsFalse`. We ship deciders for all decidable predicates already existing in Agda2Hs's Haskell-emulating Agda library

```
haveInstance : Nat
haveInstance = subtractFromGreater 2 1

s≮ : (x : Nat) → IsFalse (suc x < x)
s≮ zero = IsFalse.itsFalse
s≮ (suc x) = s≮ x

provingInstance : Nat → Nat
provingInstance x = subtractFromGreater (suc x) x ⦃ s≮ x ⦄
```

Listing 2.4: Persuading the type checker: supplying an instance to `subtractFromGreater` when necessary

```
haveNoInstance : Nat
haveNoInstance = subtractFromGreater 1 2
-------------------
True != False of type Bool
when checking that 2 is a valid argument to a function of type
(y : Nat) ⦃ @0 _ : IsFalse ((iOrdNat Ord.< 1) y) ⦄ → Nat
```

Listing 2.5: Counterexample: non-type-checking program due to lack of appropriate instance

including `IsFalse`[2]. Here, we show a slightly more interesting instance: the one for `NonEmpty`. Listing 2.6 shows the definition of `NonEmpty` itself. It has a constructor for lists with a cons (`::`), but not for empty lists.

```
data NonEmpty {a : Set} : List a → Set where
  itsNonEmpty : ∀ {x xs} → NonEmpty (x :: xs)
```

Listing 2.6: Predicates by construction: non-empty list

The decidable instance is shown in listing 2.7. The reflection proof is given in the `⟨⟩` brackets on the right-hand side of the definition, showing that the decider yields true if and only if a proof for the type exists. Thus, for the empty list case, the reflection proof is the absurd lambda: there is nothing to show because the empty list is not an instance. For the cons case, the instance defined in listing 2.6 proves the correspondence.

```
instance
  decNonEmpty : {xs : List a} → Dec (NonEmpty xs)
  decNonEmpty {xs = []} = False ⟨ (λ ()) ⟩
  decNonEmpty {xs = _ :: _} = True ⟨ NonEmpty.itsNonEmpty ⟩
```

Listing 2.7: Reflection proof: decidable instance for non-empty list

To get back to the example, instance search for `Dec (IsFalse (x < y))` will not precisely give `not (x < y)`, but `decIsFalse (x < y)`, where `decIsFalse` is equivalent to `not`.

It should be noted that in some cases, the most useful way to phrase a precondition with respect to proving may not be one that has a decidable instance. For example, when requiring equality of two terms `a` and `b`, the precondition `a ≡ b` uses Agda's `≡` operator, denoting propositional equality. This equality is a property that can be used in proofs directly, but is not decidable for all types, such as `Nat → Nat` (cf. Rice's theorem). To have decidable equality,

[2] `IsTrue`, `IsFalse`, `NonEmpty`, `All`, `Any`. The `IsTrue` decider could be compiled transparent in theory, but this does not work with the insertion of instances that we do.

it would be necessary to e.g. have `IsTrue (a == b)` as a precondition instead, using the `==` operator for Boolean equality.

Such a `==` operator must be implemented first, usually as an instance of the `Eq` type class. This type class requires a proof that the Boolean equality check returns true if and only if the two inputs are propositionally equal. This proof can outgrow the definition itself in complexity by a large margin, which is a drawback to our approach compared to writing entire projects in Agda. Listing 2.8 shows an equality instance to a simple binary tree, with the lawfulness proof in appendix A for brevity.

```
data Tree (a : Set) : Set where
  Leaf : a → Tree a
  Branch : a → Tree a → Tree a → Tree a

instance
  iEqTree : ∀ {a} → ⦃ Eq a ⦄ → Eq (Tree a)
  iEqTree ._==_ (Leaf x) (Leaf y) = x == y
  iEqTree ._==_ (Branch x xl xr) (Branch y yl yr) = x == y && xl == yl && xr == yr
  iEqTree ._==_ (Leaf _) (Branch _ _ _) = False
  iEqTree ._==_ (Branch _ _ _) (Leaf _) = False

  iLawfulEqTree : ⦃ iEqA : Eq a ⦄ → ⦃ iLEqA : IsLawfulEq a ⦄ → IsLawfulEq (Tree a)
  -- definition of iLawfulEqTree was another 24 LOC in our implementation
```

Listing 2.8: Interoperating between equality and congruence: defining `Eq` on a simple tree

2.4 Effective hiding of unchecked definitions

Having established a way to generate checks, we also want to ensure that they are not bypassed. We only want to make those definitions available in the Haskell transpilation target that have runtime checkability for all their preconditions. All definitions that were marked for transpilation should still be transpiled because other definitions might depend on them. We solve this by emitting two versions of each definition that requires checking. One is the regular, existing Agda2Hs transpilation, and one is the checked version, which passes the arguments on to the former if the checks pass. For definitions that are not completely checkable, only the former exists; for definitions that require no checking, the name is re-exported directly instead of creating the latter, which would only be the identity function. Because performing the runtime check is potentially expensive, we only do so when necessary, i.e. the former, "post-check" definition will only call other post-check definitions.

Listing 2.9 illustrates the output before and after using our flag for enabling runtime checks. In the export list of the pre-check file, checkable definitions must be disambiguated to be the pre-check version by a qualifier.

This does not make it impossible to call unchecked code from Haskell, which has no infrastructure to make a name available to only some other modules. However, it is only possible to do so when importing a module with `PostRtc` in its qualifier. We suggest making this string a forbidden word in any code not ignored by version control on a test/continuous integration (CI)-level or any similar measure to ensure it is not used.

Stronger shielding could have been provided by using a build system like Cabal and its `exposed-modules` restrictor. We did not go this way because Agda2Hs does not create multi-package output. Doing so would also conflict with Agda2Hs's goal of readably integrating Agda2Hs code in Haskell code bases, as IDE support for jumping to definitions outside the current module is still poor (The Haskell IDE Team 2019).

```
.
└── Module.hs
###################
.
├── Module
│   └── PostRtc.hs # verbatim contents of Module.hs when compiled without checks
└── Module.hs # checks, exporting the checked versions
              # and those that need no checking
```

Listing 2.9: Separating checked and unchecked compilation: file output without and with runtime checks

Chapter 3

Usage

In this chapter, we present our Agda2Hs extension from a user perspective. Basic usage is straightforward, and revolves around the concepts introduced in sections 2.3 and 2.4. We also cover advanced concepts, like what happens to data types, records and nested dependent types. We conclude with further remarks on which compilation targets are unsupported and why, and why some types are considered erased types when they might not seem so at surface level and vice versa.

3.1 General usage

Our feature is enabled by calling `agda2hs --runtime-check`. The introductory example from listing 1.1, `subtractFromGreater`, becomes what is shown in listing 3.1: the call of the actual function is guarded by a check reflecting the erased precondition from the Agda type. If the check fails, an `error` is raised.

To skip checking when calling from within Agda, references to other modules from Agda modules will contain the same `PostRtc` qualifier in the Haskell transpilation target. Putting the definition of `haveInstance` from listing 2.4 in a separate module `HaveInstance` yields the compilation shown in listing 3.2. This is done to avoid potentially costly runtime checks where they are not necessary. The example showcases another feature: if a definition has no erased arguments, it is written to the export list in the module declaration directly, employing a "fall-through" approach.

3.2 Data types, records, and nested dependent types

For data types, a similar approach is employed, treating each constructor as a smart constructor, but still exporting the constructor directly if it has no erased arguments. This has the negative side effect of making pattern matching unavailable in Haskell. A workaround is to create a deconstructor in Agda and compile it with Agda2Hs.

Listing 3.3 shows a simple data type defined for Agda2Hs compilation, a list together with a predicate that the list is not empty, and its runtime checked compilation. In the export list, the data type is exported, but not its constructor; only the smart constructor is exported. If we defined the type without its erased precondition, the constructor would appear in the parentheses of `NonEmptyList()`.

Records receive a single smart constructor for all their fields if any are erased or contain erased types. Listing 3.4 shows input and output for a record that essentially amounts to a dependent pair with a proof and a value. Because it only has one non-erased field, it can be compiled to a `newtype`. Note that the export list on the Haskell side specifies `ErasedField(value)`, which includes the type and field, but *not* the constructor.

```haskell
-- Haskell/Extra/Dec/Instances.hs
module Haskell.Extra.Dec.Instances where

...

decIsFalse :: Bool -> Bool
decIsFalse False = True
decIsFalse True = False

...
-- Subtract/PostRtc.hs
module Subtract.PostRtc where

import Haskell.Extra.Dec.Instances (decIsFalse)
import Numeric.Natural (Natural)

subtractFromGreater :: Natural -> Natural -> Natural
subtractFromGreater x y = x - y

-- Subtract.hs
module Subtract (Subtract.subtractFromGreater) where

import Haskell.Extra.Dec.Instances (decIsFalse)
import Numeric.Natural (Natural)

import Subtract.PostRtc

subtractFromGreater :: Natural -> Natural -> Natural
subtractFromGreater x y
  | decIsFalse (x < y) = Subtract.PostRtc.subtractFromGreater x y
  | otherwise = error "Runtime check failed: decIsFalse (x < y)"
```

Listing 3.1: Basic checking: runtime checked output of listing 1.1

Finally, we show an example that is less useful for real-world programming but is still important for the completeness of our emissions: we also generate according checks for nested function types. In most cases, we would consider examples to this poor software engineering, although under the employment of *continuation-passing style*, functions of very high orders can occur (e.g. six in Okasaki 1998). In any case, support for higher-order types is included because we want to show how complete shielding can be achieved.

The `silly` function in listing 3.5 takes a function with a precondition, a precondition on the function, and a named and an unnamed number. Its runtime-checked output showcases how this input is treated. For a precondition in a nested type, runtime checking is executed by creating a numbered `go` function in the `where`-declarations. On check success, control is returned to the caller, marked by the `up` function.

The type is so large because it demonstrates two specific features:

- The `go` and `a` prefixes receive running numbering with definition-level uniqueness. This is implemented in the simplest way possible, which leads to the somewhat odd scheme where `go` prefixes and those `a` prefixes on the left-hand side of these `go` definitions are numbered bottom-up, but arguments to deeper `go` functions are numbered top-down.

```haskell
-- HaveInstance/PostRtc.hs
module HaveInstance.PostRtc where

import Numeric.Natural (Natural)
import Subtract.PostRtc (subtractFromGreater)

haveInstance :: Natural
haveInstance = subtractFromGreater 2 1

-- HaveInstance.hs
module HaveInstance (haveInstance) where

import Numeric.Natural (Natural)
import Subtract.PostRtc (subtractFromGreater)

import HaveInstance.PostRtc
```

Listing 3.2: Calling post-runtime check versions from within Agda

```haskell
data NonEmptyList {a : Set} : Set where
  NE : (xs : List a) → @0 ⦃ NonEmpty xs ⦄ → NonEmptyList
{-# COMPILE AGDA2HS NonEmptyList newtype #-}
```

```haskell
-- Datatype/PostRtc.hs
module Datatype.PostRtc where

import Haskell.Extra.Dec.Instances (decNonEmpty)

newtype NonEmptyList a = NE [a]

-- Datatype.hs
module Datatype (NonEmptyList(), Datatype.scNE) where

import Haskell.Extra.Dec.Instances (decNonEmpty)

import Datatype.PostRtc

scNE :: [a] -> NonEmptyList a
scNE xs
  | decNonEmpty xs = NE xs
  | otherwise = error "Runtime check failed: decNonEmpty xs"
```

Listing 3.3: Runtime checking data types: generating a smart constructor

```
record ErasedField : Set where
  field
    value : Nat
    @0 proof : IsTrue (value > 0)
open ErasedField public
{-# COMPILE AGDA2HS ErasedField newtype #-}
```

```
-- Record/PostRtc.hs
module ErasedField.PostRtc where

import Haskell.Extra.Dec.Instances (decIsTrue)
import Numeric.Natural (Natural)

newtype ErasedField = ErasedField{value :: Natural}

-- Record.hs
module Record (ErasedField(value), Record.scErasedField) where

import Haskell.Extra.Dec.Instances (decIsTrue)
import Numeric.Natural (Natural)

import Record.PostRtc

scErasedField :: Natural -> ErasedField
scErasedField value
  | decIsTrue (value > 0) = ErasedField value
  | otherwise = error "Runtime check failed: decIsTrue (value > 0)"
```

Listing 3.4: Runtime checking records

Other numbering schemes, such as top-down for everything, would require a more complex symbol replacement algorithm deemed not worth the code overhead for this already niche feature.

- Checks descending two or more levels without any checks at the higher-up level are inlined (seen in `a0 -> f (go0 a0)`). The alternative is a pass-through function without guards, increasing verbosity.

Because the definition has two erased arguments, it has two checks, and can go wrong in two different places. Listing 3.6 shows how to trigger both of them, using `catch` to continue after error. The listing also shows the output. Because of Haskell's lazy evaluation, printing the variables is necessary to trigger the checks.

3.3 Further usage remarks

The pre-supplied decidable instances, such as `decNonEmpty` from listing 2.7 or `decIsFalse`, are imported automatically when runtime checking is enabled. However, because they will appear with full qualifiers in the Agda2Hs output (`Haskell.Extra.Dec.Instances.decIsFalse`), it is recommended to import `Haskell.Extra.Dec.Instances` from Agda anyhow.

```
silly : (f : ((x : Nat) → @0 ⦃ IsTrue (x < 2) ⦄ → Nat) → Nat)
        → @0 ⦃ IsTrue (f (λ x → x) > 0) ⦄ → Nat → Nat
silly f y = f (λ x → x + y)
```

```
module Silly (Silly.silly) where

import Haskell.Extra.Dec.Instances (decIsTrue)
import Numeric.Natural (Natural)

import Silly.PostRtc

silly :: ((Natural -> Natural) -> Natural) -> Natural -> Natural
silly f a1
  | decIsTrue (f (\ x -> x) > 0) =
    Silly.PostRtc.silly (\ a0 -> f (go0 a0)) a1
  | otherwise =
    error "Runtime check failed: decIsTrue (f (\\ x -> x) > 0)"
  where
    go0 up x
      | decIsTrue (x < 2) = up x
      | otherwise = error "Runtime check failed: decIsTrue (x < 2)"
```

Listing 3.5: Erased lambdas in types: generation of nested runtime checks

```
module Main where

import Control.Exception (catch, SomeException)
import Silly (silly)

tryPrint :: Show a => a -> IO ()
tryPrint f = print f `catch` \e -> print (e::SomeException)

main :: IO ()
main = do
  tryPrint $ silly (const 0) 0
  tryPrint $ silly ($ 2) 0
```

```
Runtime check failed: decIsTrue (f (\ x -> x) > 0)
CallStack (from HasCallStack):
  error, called at ./Silly.hs:13:5 in main:Silly
Runtime check failed: decIsTrue (x < 2)
CallStack (from HasCallStack):
  error, called at ./Silly.hs:17:21 in main:Silly
```

Listing 3.6: Triggering both checks from listing 3.5

Some constructs are prohibited from runtime checks, generally because there is no obvious way to insert checks for them. Attempting to do so with a type that contains erased arguments will result in an error.

- Functions marked for inlining. Checking them would be possible, but create very long lines, and undermine the separation into pre- and post-check definitions that we keep.
- Functions marked for transparent compilation. Transparent compilation is an Agda2Hs feature for Agda functions that, after erasure, collapse to the identity function, and calls to them are consequently stripped away in the output. This makes checking impossible.
- Records marked for unboxed compilation. These are analoguous to transparent functions, having a single unerased field and their constructors stripped away in the output.
- Records to be compiled to type classes, cf. section 6.2.1.

Not only the types that are marked erased at top level are considered to require runtime checking. Consider a precondition embedded in a nonempty list[1]; then, it could be used by Agda to complete a proof, but would also require checking. Thus, data types and records are recursed into upon deciding whether a type should be checked. Generally, no decidable instance will exist, and it is recommended to refactor code to not use such structures. While we do not see an immediate use case for the list of preconditions, a notable example to this is the `∃` record from the Agda2Hs library. It is also used by `Dec` and poses a value-proof pair like `ErasedField`, but with parametric types for its fields. The according recommendation in this case is to use a regular argument paired with the precondition instead. Note that structures that compile to an existing class, such as `Functor`, can still be used in the ordinary way, as none of these structures would induce a check in any case.

Regardless, definitions from the Agda2Hs TCB are exempt from runtime checking because many definitions would be no longer available due to a lack of instances. This is different from the checking described in the paragraph above, as it concerns definitions imported and marked with the `AGDA2HS` pragma in the TCB. This exemption does not interfere with our goal, which is to block input to Agda2Hs definitions that does not pass preconditions, but any such definition using native Agda2Hs definitions would still provide proof to their preconditions based on its checked input. However, this non-checking does make it possible to bypass checks by writing out-of-spec functions in a module named `Haskell`, but this can be mitigated likewise by prohibiting such code on a CI level. Caution is advised if this is not possible.

[1] More specifically, e.g. a precondition wrapped in the `Haskell.Extra.Erased` record; just a data type ranging over erased types is not valid Agda.

Chapter 4

Implementation

This chapter exists mostly for reference and is not necessary for understanding the remainder of this thesis.

In this chapter, we give a brief rundown of our approach to the implementation of the features shown in chapter 3 with respect to the existing Agda2Hs code base. We also explain some intricacies of this implementation. Additionally, we provide a more detailed explanation of the relatively complex process of generating a runtime check for a given type. This process was built independent of the kind of the definition, be it a function, data type, or record. The code is available at `https://github.com/naucke/agda2hs/tree/strict-runtime-check` for reference.

4.1 Agda2Hs architecture background

Agda2Hs is implemented as an *Agda backend* and, like the Agda compiler, is written in Haskell. In the architecture of Agda, this means that Agda2Hs produces a separate binary, but relies on Agda as a library for type checking. Centrally, it implements its own compile functionality, receiving type checked definitions and writing Haskell files. For Agda2Hs, compiling is based on pattern matching on the type of Agda definition and pragma, and producing a Haskell definition accordingly: e.g. compiling a function, a data type, a record, a record marked to be compiled to a class, or a class instance. Some combinations will not yield any definition, such as transparent functions and unboxed records, definitions marked as reimplementing an existing Haskell definition (primarily found in the Agda2Hs TCB), or records compiled to a tuple. The compilation definitions for functions, data types, etc. use common functionality for compiling types and terms.

4.2 Overall structure

Our extension requires changes to the Agda2Hs code base in several places, but usually, the changes to the existing structure are not fundamental. The actual check insertion is very similar regardless of the type of definition it checks, and is thus common functionality. Many of the Haskell outputs computed and passed around in the Agda2Hs compiler become two outputs with our feature, one with the translated definition and one with its respective runtime check. For better legibility, this double-tracking is solved using an abstract data type `WithRtc`, so that it can be used for all compilation stages, which are not all the same type. Some refactoring is induced by this change in any case, but can be made minimal by giving `WithRtc` a functor instance.

Agda2Hs's read-write-state monad `C`, a wrapper monad to Agda's type checking monad (TCM), receives some extensions:

- whether runtime checks should be emitted, as set by the `--runtime-check` option, which is defined as an additional Agda2Hs backend option
- the definition names that had no erased arguments and can thus be passed on upon export
- the definition names that a valid runtime check was found for and whose pre-runtime check names can be exported

For the consideration of the different kinds of Agda definitions, different outcomes are possible:

- To some definition kinds, runtime checks are not applicable because they emit no output:
  - definitions that are compiled to no output at all
  - definitions marked to represent an existing Haskell class
- To other definition kinds, runtime checks are also not applicable, but we must still mark the emitted names for export (see chapter 3):
  - definitions of class instances (see section 6.2.1)
  - postulated definitions, as they are unsafe independent of runtime checking
- The same applies to some other definition kinds that we also prohibit to contain erased arguments when runtime checking is enabled:
  - definitions of records marked for unboxing, meaning that their sole non-erased field is unboxed
  - definitions of records marked for compilation to a tuple
  - definitions of records marked for compiling to a type class
  - definitions of functions marked transparent, meaning that they compile to the identity function
  - definitions of functions marked for inlining
- Finally, we do apply runtime checks to the definitions as described in chapter 3. We cover the process of generating a runtime-checking function based on the type to check and expression on success in section 4.3.
  - For functions, we simply insert `PostRtc` to the function's qualified name in the successful case. The function name is exposed in the module declaration when runtime checks can be found for all erased arguments, or when there were no erased arguments.
  - For data types, we attempt to generate a smart constructor for each constructor. The data type itself is always exposed, but only those constructors that have no erased arguments are exposed, alongside the smart constructors.
  - For those records to be compiled to a Haskell record instead of a class, we also generate the respective smart constructor similarly. The field names to a record are always exposed.

Besides the check determination, it is also necessary to insert the `PostRtc` deviation into Agda2Hs's name resolution logic. Furthermore, the automatic import of the decider instances is based on an existing function in the Agda compiler code base (Agda Development Team 2024a, ConcreteToAbstract). Before check generation, it is tested whether the name to be checked is not part of the TCB by testing if its top-level qualifier is `Agda` or `Haskell`.

4.3 Check insertion

As mentioned, the logic for creating a runtime check with respect to the type to be checked is the same for all structures we consider. We cover this part in more detail because it is the most involved part of our solution.

The central functionality of finding a deciding term for a precondition is based on instance search (cf. section 2.3). For a given erased type, e.g. the one from the introductory listing 1.1, instance search is run upon that type wrapped in the `Dec` type. Compiling its result gives a Haskell term that can be used for guarding. The associated steps are shown in listing 4.1.

```
type: IsFalse ((iOrdNat Ord.< x) y)
dec type: Haskell.Extra.Dec.Dec (IsFalse ((iOrdNat Ord.< x) y))
instance: decIsFalse
term: Just "decIsFalse (x < y)"
```

Listing 4.1: Steps to finding a decidable instance

However, upon receiving the type telescope, it is insufficient to run through it and generate a check for each erased argument. Consider the post-check code from listing 4.2 being called with `\f -> f 0`. The check for such nested preconditions is shown in the same listing below, using a `where` declaration that returns control to the caller if the check passes. Note that for checks necessary on multiple levels of nesting, it would not be possible to flatten all the `where`-declarations to one level as a deeper check might depend on an argument introduced in a shallower check.

```
doubleOdd : (((n : Nat) → @0 ⦃ IsFalse (n < 1) ⦄ → Nat) → Nat) → Nat
doubleOdd f = f (λ n → n - 1)
```

```
-- DoubleOdd/PostRtc.hs
module DoubleOdd.PostRtc where

import Haskell.Extra.Dec.Instances (decIsFalse)
import Numeric.Natural (Natural)

doubleOdd :: ((Natural -> Natural) -> Natural) -> Natural
doubleOdd f = f (\ n -> n - 1)
```

```
-- DoubleOdd.hs
module DoubleOdd (DoubleOdd.doubleOdd) where

import Haskell.Extra.Dec.Instances (decIsFalse)
import Numeric.Natural (Natural)

import DoubleOdd.PostRtc

doubleOdd :: ((Natural -> Natural) -> Natural) -> Natural
doubleOdd a0 = DoubleOdd.PostRtc.doubleOdd (\ a1 -> a0 (go0 a1))
  where
    go0 up n
      | decIsFalse (n < 1) = up n
      | otherwise = error "Runtime check failed: decIsFalse (n < 1)"
```

Listing 4.2: Checking nested erased arguments

This approach introduces new names:

- `a0`, `a1`, … arguments for each unnamed argument
- `go0`, `a1`, … functions, as wrapper functions for any argument with erased arguments inside it
- an `up` argument to the `go` function to return control after the check, the name stemming from the metaphor of trampolining back up

The former two require keeping track of which numbers have been used. Instead of giving them placeholders and filling these in a second pass, we set these names in one go.

The checks are computed recursively. However, the logic at the top level differs from the logic of all levels below: at the top, definitions from below must be put into the `where`-block, and the name must be marked as exportable. This leads to the definitions of the types returned by the top-level and lower-level functions respectively in listing 4.3. `NameIndices` is an integer tuple to track which `a` and `go` numbers have been used. `NestedLevel` is a two-constructor data type to track whether the type inspected is on an odd or an even level of nesting, as only odd levels incur checks (cf. section 5.2).

```haskell
type NameIndices = (Int, Int)

data NestedLevel = Odd | Even

data RtcResult
  = NoneErased
  | Uncheckable
  | Checkable [Hs.Decl ()]
```

```haskell
data RtcResult'
  = NoneErased'
  | Uncheckable'
  | Checkable'
      { theirLhs :: [Hs.Pat ()],
        theirChks :: [Hs.Exp ()],
        theirRhs :: [Hs.Exp ()],
        theirDecls :: [Hs.Decl ()]
      }
```

Listing 4.3: Return types for runtime check computations

The fields in `Checkable'` are named “theirs” to indicate they come from further below, unlike the fields being computed at current level (“ours”). The module `Hs` denotes the Haskell source extensions, a Haskell formalisation of Haskell's language structures (Broberg and Burton 2020). `Pat` is a pattern for pattern matching, `Exp` is an expression, and `Decl` is a declaration, which includes definitions in this context. The type parameter is empty in our case, it would otherwise be used for denoting the position of the Haskell construct in a piece of source code. In the `nest` example from above, `theirLhs` is `[a0]`, `theirRhs` is `[go0 a0]`, `theirDecls` is the definition of `go0`, and `theirChks` is `[]` since there are no erased arguments at top level.

In order to compute checks for all unerased, but potentially nested types in a telescope, we could use `map`. However, to also keep track of the `go` and `a` numbers used, we need an accumulator along the way. For this purpose, we can use the `mapAccumL` higher-order function or, more precisely, its monadic variant from GHCUtils (The GHC Team 2024). `mapAccumLM` takes a combining function, initial state, and inputs, returning a final state and its outputs. This gives it the signature `mapAccumLM :: (Monad m, Traversable t) => (acc -> x -> m (acc, y)) -> acc -> t x -> m (acc, t y)`. In our instance, these symbols are of different data types:

- `acc` is `NameIndices`.
- `x` is `(Dom (ArgName, Type), Telescope)`.
  - `Dom` is a domain featuring the argument name, so that instead of writing `a`, we can use the argument name given in the Agda source if it exists; and the actual type to be checked.

  - `Telescope` is a type telescope, but only up to `Type`. It is used to provide context to the de Bruijn indices in the runtime checking terms when they are compiled.

- `m` is the `C` monad.
- `y` is `Maybe (Hs.Pat (), Hs.Exp (), [Hs.Decl ()])`. This is akin to the `RtcResult'` type, except there are no `Chks` because there are no current-level erased types, and there is only one `Lhs` and `Rhs` each because we are map-accumulating over the types.
- `t` is the list type.

These signatures bring us to the three central functions to compute runtime checks with respect to a type.

- `checkRtc' :: NameIndices -> Telescope -> [NestedLevel] -> C (NameIndices, RtcResult')`, partitioning the received telescope into current-level erased arguments and unerased, but potentially nested arguments. The latter are `mapAccumLM`ed upon with `checkRtc''`. Additionally, a list of level nestings is received. The reason that a list is given instead of simply toggling the state is that records pass a single telescope of all their fields, and both the fields and the field types should be considered to be at an odd (i.e. to be checked) level. Consequently, the logic for records gives a list that starts with two `Odd`s and then cycles between `Even` and `Odd`. If all current-level and lower-level erased arguments are checkable, the `RtcResult'` is assembled according to that data alongside the checks found by instance search at the current level. The incremented `NameIndices` are also passed back.
- `checkRtc'' :: NameIndices -> ((Dom (ArgName, Type), Telescope), [NestedLevel]) -> C (NameIndices, Maybe (Hs.Pat (), Hs.Exp (), [Hs.Decl ()]))`'s signature results from the bulleted list above. It is mutually recursive with `checkRtc'`, having reserved the number for the `a` argument where necessary, and analysing `checkRtc'`'s result. If that is `Checkable'`, the number for the `go` function is also advanced, the `Lhs` is prepended by an `up`, and that `up` is applied to `theirRhs`.
- `checkRtc :: Telescope -> QName -> Hs.Exp () -> [NestedLevel] -> C RtcResult`, taking a telescope of the type to check, name of the runtime checked function, expression on success, and returning the result: no erased types, uncheckable, or a complete runtime checking declaration. It calls `checkRtc'`, initialising `NameIndices` with `(0, 0)` and passing the telescope directly. On success, it assembles the actual check and marks the name as checkable for export.

In addition, `checkRtc''` as well as the smart constructor name generation check for name conflicts and give an error accordingly if there is a conflict. `checkRtc''` is also capable of inlining a function when there is no check. We give an example to this feature in the form of a step-by-step illustration of the architecture described in appendix C.

The code to the central functions `checkRtc`, `checkRtc'`, and `checkRtc''` is given in appendix B. They also use the functions

- `createGuardExp :: Dom (a, Type) -> Telescope -> C (Maybe (Hs.Exp ()))` which finds a guarding expression when possible using instance search
- `checkTopOrDataErased :: Dom (a, Type) -> [NestedLevel] -> C Bool` which checks if a type is erased or is a type or record containing an erased argument at any (odd) level
- `binds :: [Hs.Decl ()] -> Maybe (Hs.Binds ())` which creates `where`-bindings except when empty

- `errorWhenConflicts :: [String] -> C ()` which gives an error message when appropriate, taking a list of conflicted names

Additionally, the actual creation of the declaration is computed by the separate `createRtc` function, which is also required at the levels below for the `where`-declarations. It has the signature `createRtc :: Hs.Name () -> [Hs.Pat ()] -> [Hs.Exp ()] -> Hs.Exp () -> Maybe (Hs.Binds ()) -> Hs.Decl ()`, taking the function name, left-hand side patterns, check expressions, expression on success, and optionally `where` binds. It connects all checks by `&&` for the successful case, and gives an appropriate error for each failure of a precondition. Listing 4.4 shows a transformation sequence using multiple checks.

```
addHeads : (xs : List Nat) → @0 ⦃ NonEmpty xs ⦄ →
           (ys : List Nat) → @0 ⦃ NonEmpty ys ⦄ → Nat
addHeads (x :: _) (y :: _) = x + y
```

```
addHeads :: [Natural] -> [Natural] -> Natural
addHeads xs ys
  | decNonEmpty xs && decNonEmpty ys = And.PostRtc.addHeads xs ys
  | not (decNonEmpty xs) =
    error "Runtime check failed: decNonEmpty xs"
  | otherwise = error "Runtime check failed: decNonEmpty ys"
```

Listing 4.4: Transformation step of a function with multiple checks

4.4 Testing

To ease updates and refactoring, which took place several times in the development of this thesis, a set of integration tests of our feature was kept. All the core features are tested, such as generating checks at odd levels, not exporting definitions without complete checkability, and checking data types and records. It is also ensured that only the checkable constructors are exported, that records checking works at field and sub-field level, and that data types and records ranging over erased types are themselves considered erased. There is also a test case to ensure that internal calls, i.e. calls from translated Agda code rather than external Haskell, do not incur a runtime check. A number of test cases also focus on catching prohibited cases, such as having conflicting module, definition, or variable names, or compiling inlined, unboxed, or to a tuple with erased arguments. The tests also ensure that these compilation targets remain available when not using erased arguments in odd positions.

Chapter 5

Analysis

Without being specific to our implementation, we create an argument for the completeness of the types of checks we emit. We limit this analysis to functions, treating the creation of smart constructors for data types as an implementation detail. Centrally, we show that with our model of runtime checks, the execution of a function translated from a language with verified proofs to one with runtime checks will only occur if the relevant proof exists.

This analysis makes some simplifications in comparison to the actual implementation. Even then, we found that a minimal and complete check insertion is complicated and reaches beyond "each erased argument should be checked". We show that it is sufficient to insert checks for every argument in an *odd-nested position*. This observation warrants the analysis beyond the mere self-verification.

Initially, we set up a theory to work in based on Norell 2007. We then present an abstract model of what checks are emitted. We conclude with a proof that these checks guard the function implementations from running without meeting the checks.

5.1 A rudimentary model for Agda2Hs and runtime check emissions

To reason about our output, we require a theoretical model for Agda and Haskell. We base this on Norell's dependent type theory for *both* languages. Thus, we simplify the differences between the languages and assume that the output language will not have dependent types. We leave out most of the details surrounding the formalisation of this translation, as these would be more in scope for a general analysis of Agda2Hs, but not for our runtime checks.

The type judgements in this type theory are using a context Γ, consisting of all relevant variables. For example, there is a typing rule

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : (x : A) \to B}$$

where if Γ alongside $x : A$ derives $t : B$, Γ derives the lambda term. Norell writes telescopes explicitly, we assume them implicit.

We require some extensions to Norell's type system as marked in the definition. Most importantly, dependent functions are extended with erased arguments. Lambdas taking an erased argument are denoted by λ_0. In the translation target, we will also use an *assert* construct as a simplification of the guards and error messages that we emit. To be able to assert, we introduce the Bool type and **true** and **false** values with the usual typing rules. Alongside assertion, there is also an if-then-else construct to define decidable types and derive non-trivial Boolean terms. To represent false propositions, we add the bottom type ($\bot$).

For dependent pairs, we make another simplification. Agda2Hs can compile two kinds of pairs: one with the second part in erased position, as seen in listing 2.7, and one with

the dependency in erased position and with no universe levels, i.e. `Σ (a : Set) (b : @0 a → Set) : Set` instead of Agda's `Σ {a b} (A : Set a) (B : A → Set b) : Set (a ⊔ b)`. We only consider the former.

Definition 1.

$$
\begin{aligned}
s, t, A, B \quad &:= x \\
&\mid (x : A) \to B \mid (@0\, x : A) \to B \\
&\mid \lambda x.\, t \mid \lambda_0 x.\, t \mid s\, t \\
&\mid (x : A) \times @0\, B \\
&\mid \langle s, t \rangle \mid \pi_1\, t \mid \pi_2\, t \\
&\mid \mathbf{Set}_i \\
&\mid 1 \mid \langle\rangle \mid \bot \\
&\mid \mathrm{Bool} \mid \mathrm{true} \mid \mathrm{false} \\
&\mid \mathrm{assert}\ s; t \\
&\mid \mathrm{if}\ s\ \mathrm{then}\ t\ \mathrm{else}\ u
\end{aligned}
$$

We require some reduction rules, especially for the constructs we added. Reduction steps for assertion are denoted by $\longrightarrow_e$, and can lead to an error, which is not a term in the general sense, but still a possible result.

Definition 2.

$$\frac{}{(\lambda x.\, s)t \longrightarrow_\beta s[x := t]} \quad \text{where } t \text{ is not erased}$$

$$\frac{}{(\lambda_0 x.\, s)t \longrightarrow_\beta s[x := t]} \qquad \frac{}{\pi_1 \langle s, t \rangle \longrightarrow_\iota s} \qquad \frac{}{\pi_2 \langle s, t \rangle \longrightarrow_\iota t}$$

$$\frac{}{\mathrm{if}\ \mathbf{true}\ \mathrm{then}\ s\ \mathrm{else}\ t \longrightarrow_\iota s} \qquad \frac{}{\mathrm{if}\ \mathbf{false}\ \mathrm{then}\ s\ \mathrm{else}\ t \longrightarrow_\iota t}$$

$$\frac{s \rightsquigarrow \mathbf{true}}{\mathrm{assert}\ s; t \longrightarrow_e t} \qquad \frac{s \rightsquigarrow \mathbf{false}}{\mathrm{assert}\ s; t \longrightarrow_e \mathrm{error}}$$

We generalise single small-step reduction by $\longrightarrow$ and denote multi-step reduction by $\rightsquigarrow$. We slightly **deviate** from the common understanding of $\longrightarrow$ by always reducing a term that is asserted over completely, e.g.

$$(\mathrm{assert}\ (\mathrm{if}\ a\ \mathrm{then}\ b\ \mathrm{else}\ \mathbf{false}); (\lambda c.\, d)e) \longrightarrow (\lambda c.\, d)e$$
$$\text{where } a = b = \mathbf{true}$$

For denoting an Agda2Hs translation, we use the notation $[\![...]\!]$, as in the following definition. The compilation of erased arguments in Agda is based on Quantitative Type Theory (QTT) (Atkey 2018). Agda2Hs compilation of a dependent type ensure that there is no dependent variable in an unerased position. Because full reasoning about them would be off-topic for our analysis, we simply state that it is ignored in compilation and any further use of the erased argument results in a compile-time error ($\bot$). To distinguish from meta-level notation, object-level notation is written in `coloured monospace`. As a result of our sole consideration of dependent pairs with an erased second component, $[\![\pi_2\, \langle s, t \rangle]\!]$ is undefined.

Definition 3.

$$
\begin{aligned}
[\![s]\!]_A &:= \texttt{s} \\
&\text{where} \quad s \text{ is a variable, Boolean, or unit value} \\
[\![\lambda x.\, t]\!]_{(x:A)\to B} &:= \lambda \texttt{x.}\ [\![t]\!]_B \\
[\![\lambda_0 x.\, t]\!]_{(@0\, x:A)\to B} &:= [\![t]\!]_{B[x:=\bot]} \\
[\![s\, t]\!]_B &:= [\![s]\!]_{A\to B}\, [\![t]\!]_A \\
&\text{where} \quad s \text{ has an unerased domain and} \\
&\qquad\quad t \text{ is of an unerased type} \\
[\![s\, t]\!]_{B[x:=t]} &:= [\![s]\!]_{(@0\, x:A)\to B} \\
&\text{where} \quad s \text{ has an erased domain} \\
[\![\langle s, t\rangle]\!]_{(x:A)\times @0\, B} &:= [\![s]\!]_A \\
[\![\pi_1 s]\!]_A &:= [\![s]\!]_A \\
[\![\pi_2 s]\!]_A &:= \bot \\
[\![\text{if } s \text{ then } t \text{ else } u]\!]_A &:= \texttt{if}\ [\![s]\!]_{\text{Bool}}\ \texttt{then}\ [\![t]\!]_A\ \texttt{else}\ [\![u]\!]_A
\end{aligned}
$$

We assume that Agda2Hs always creates execution-equivalent output:

Assumption 1. *For any well-typed Agda terms s and v, if $s \rightsquigarrow v$, then $[\![s]\!] \rightsquigarrow [\![v]\!]$.*

The generation of our checks depends heavily on the reflects predicate and the decidability type. The latter is defined as a dependent pair:

Definition 4. **Reflects** $P\ b :=$ if b then P else $(P \to \bot)$

Definition 5. **Dec** $P := (b : \text{Bool}) \times (@0\ \textbf{Reflects}\ P\ b)$

We also postulate the existence of instance search. Instead of the `Maybe` approach of the real implementation, we treat it as a partial function, aborting further compilation if it fails to find a result as a simplification.

Assumption 2. *If instance search finds a term i of a type A ($i = \text{inst}\ A$), it will be a valid term of that type ($i : A$).*

We denote the runtime checked transpilation by notation of $[\![...]\!]^{\text{rtc}}$. At surface level, only one rule is modified from the Agda2Hs definition; the one covering erased argument lambdas. This rule reflects e.g. the translation of the introductory example from listing 3.1. The checking of nested arguments is achieved by an assumption:

Assumption 3. *All terms in $[\![...]\!]^{\text{rtc}}$ are in η-long form, not expanding heads of application terms.*

This assumption makes it possible to forego the consideration of terms like $[\![s]\!]^{\text{rtc}}_{(x:A)\to B}$ because such a term s will be expanded to $\lambda x.\, sx$. Likewise, when a dependent pair is in argument position, e.g. $\lambda_0 y.\, ty$ where $y : (x : A) \times @0\, B$, it will be expanded to $\lambda_0 \langle y_1, y_2\rangle.\, t\langle y_1, y_2\rangle$ and y_2 will be checked accordingly. The addition with respect to plain Agda2Hs is also marked. Note that this is a compilation of terms, and Agda2Hs's compilation of types is not modified.

Definition 6.

$$
\begin{aligned}
[\![s]\!]^{\mathrm{rtc}}_{A} &:= \texttt{s} \\
\text{where} \quad & s \text{ is a variable, Boolean, or unit value} \\
[\![\lambda x.\, t]\!]^{\mathrm{rtc}}_{(x:A)\to B} &:= \lambda \texttt{x}.\, [\![t]\!]^{\mathrm{rtc}}_{B} \\
[\![\lambda_0 x.\, t]\!]^{\mathrm{rtc}}_{(@0\, x:A)\to B} &:= \texttt{assert}\ [\![i]\!]_{\mathbf{Dec}\ A}; [\![t]\!]^{\mathrm{rtc}}_{B[x:=\bot]} \\
\text{where} \quad & i = \mathrm{inst}(\mathbf{Dec}\ A) \\
[\![s\ t]\!]^{\mathrm{rtc}}_{B[x:=t]} &:= [\![s]\!]^{\mathrm{rtc}}_{(x:A)\to B}\ [\![t]\!]^{\mathrm{rtc}}_{A} \\
\text{where} \quad & s \text{ has an unerased domain and} \\
& t \text{ is of an unerased type} \\
[\![s\ t]\!]^{\mathrm{rtc}}_{B[x:=t]} &:= [\![s]\!]^{\mathrm{rtc}}_{(@0\, x:A)\to B} \\
\text{where} \quad & s \text{ has an erased domain} \\
[\![\langle s, t\rangle]\!]^{\mathrm{rtc}}_{(x:A)\times @0\, B} &:= [\![s]\!]^{\mathrm{rtc}}_{A} \\
[\![\pi_1 s]\!]^{\mathrm{rtc}}_{A} &:= [\![s]\!]^{\mathrm{rtc}}_{(x:A)\times @0\, B} \\
[\![\pi_2 s]\!]^{\mathrm{rtc}}_{A} &:= \bot \\
[\![\text{if } s \text{ then } t \text{ else } u]\!]^{\mathrm{rtc}}_{A} &:= \texttt{if}\ [\![s]\!]^{\mathrm{rtc}}_{\mathrm{Bool}}\ \texttt{then}\ [\![t]\!]^{\mathrm{rtc}}_{A}\ \texttt{else}\ [\![u]\!]^{\mathrm{rtc}}_{A}
\end{aligned}
$$

The direct insertion of runtime checks upon the η-expanded form of the term is different from the Haskell implementation, as Haskell will read all arguments and only then evaluate the guards instead of evaluating the guards in order of appearance in the source type. This means that the Haskell implementation can error at a later point than suggested by theorem 2, cf. the trigger points in listing 3.6.

To reiterate on the simplifications made:

- Agda and Haskell are treated as the same language.
- The language has a basic dependent type theory with erased arguments, Booleans, and assertions.
- The second element of dependent pair types is always erased.
- Data types and records are not considered.
- An instance for a condition-checking term can always be found.
- Checks are inserted directly instead of taking all arguments of a definition first.

5.2 Odd and even positions

We show how this definition inserts runtime checks by the example of definitions from listing 5.1. We will not cover all of them in full length; the latter ones are included only to illustrate the conditions under which a function should be checked.

The function `sub` is defined only as an instance-argument unroll to Agda2Hs's `_-_`. Our focus will be on the `odd`, `even`, `doubleOdd`, etc. functions. Their names denote what position the erased argument is in.

These examples show how only the erased arguments in even positions will require checking. This is much akin to the odd-even rule by Findler and Felleisen 2002. In their publication, they established a model for runtime-checking higher order functions in the Scheme dialect of Lisp. Their analysis also affects postconditions, which are not part of our checks because most postconditions would be formulated as indexed data types in Agda,

which does not translate well to Agda2Hs. They establish how a function is responsible for violations in even positions, but the function's caller is responsible in odd positions. Similarly, `doubleOdd` can be violated by calling with the argument $\lambda f.f\,0$, but there is no way to violate `even` from the caller's side.

```
sub : (n m : Nat) → (@0 p : IsFalse (n < m)) → Nat
sub n m prf = _-_ n m {{ prf }}

odd : (m : Nat) → (@0 p : IsFalse (m < 1)) → Nat
odd = λ n → λ q → ((sub n) 1) q

even : (s : ((m : Nat) → (@0 p : IsFalse (m < 1)) → Nat)) → Nat
even = λ f → (f 1) IsFalse.itsFalse

doubleOdd : (s : ((t : ((m : Nat) → (@0 p : IsFalse (m < 1)) → Nat)) → Nat)) → Nat
doubleOdd = λ f → f (λ n → λ q → ((sub n) 1) q)

-- omitting unnecessarily named variables from here for brevity
doubleEven : ((((m : Nat) → @0 IsFalse (m < 1) → Nat) → Nat) → Nat) → Nat
doubleEven = λ f → f (λ g → (g 1) IsFalse.itsFalse)

tripleOdd : (((((m : Nat) → @0 IsFalse (m < 1) → Nat) → Nat) → Nat) → Nat) → Nat
tripleOdd = λ f → f (λ g → g (λ n → λ q → ((sub n) 1) q))
```

Listing 5.1: Different positions of an erased argument in η-long form

`odd` is the simplest case:

$$
\begin{aligned}
&[\![\lambda n.\lambda_0 q.\ ((\mathbf{sub}\ n)\ \mathbf{1})\ q]\!]^{\mathrm{rtc}}_{(m:\mathbb{N})\to(@0\,p:m\not<\mathbf{1})\to\mathbb{N}} \\
&= \lambda \mathtt{n}.\ [\![\lambda_0 q.\ ((\mathbf{sub}\ n)\ \mathbf{1})\ q]\!]^{\mathrm{rtc}}_{(@0\,p:m\not<\mathbf{1})\to\mathbb{N}} \\
&= \lambda \mathtt{n}.\ \mathtt{assert\ n} \not< \mathtt{1};\ [\![((\mathbf{sub}\ n)\ \mathbf{1})\ q]\!]^{\mathrm{rtc}}_{\mathbb{N}[p:=q]} \\
&= \lambda \mathtt{n}.\ \mathtt{assert\ n} \not< \mathtt{1};\ [\![(\mathbf{sub}\ n)\ \mathbf{1}]\!]^{\mathrm{rtc}}_{((@0\,p:m\not<\mathbf{1})\to\mathbb{N})[n:=\mathbf{1}]} \\
&= \lambda \mathtt{n}.\ \mathtt{assert\ n} \not< \mathtt{1};\ [\![\mathbf{sub}\ n]\!]^{\mathrm{rtc}}_{\mathbb{N}\to(@0\,p:m\not<\mathbf{1})\to\mathbb{N}}[\![\mathbf{1}]\!]^{\mathrm{rtc}}_{\mathbb{N}} \\
&= \lambda \mathtt{n}.\ \mathtt{assert\ n} \not< \mathtt{1};\ [\![\mathbf{sub}]\!]^{\mathrm{rtc}}_{((m:\mathbb{N})\to\mathbb{N}\to(@0\,p:m\not<\mathbf{1})\to\mathbb{N})[n:=o]}[\![n]\!]^{\mathrm{rtc}}_{\mathbb{N}}\,\mathtt{1} \\
&= \lambda \mathtt{n}.\ \mathtt{assert\ n} \not< \mathtt{1}; \mathtt{sub\ n\ 1}
\end{aligned}
$$

`even` takes a function of `odd`'s signature, but critically, its input will not be checked because there is no way to violate its precondition when calling from Haskell:

$$
\begin{aligned}
&[\![\lambda s.\ (s\ \mathbf{1})\ \mathbf{itsFalse}]\!]^{\mathrm{rtc}}_{(s:((m:\mathbb{N})\to(@0\,p:m\not<\mathbf{1})\to\mathbb{N}))\to\mathbb{N}} \\
&= \lambda \mathtt{s}.\ [\![(s\ \mathbf{1})\ \mathbf{itsFalse}]\!]^{\mathrm{rtc}}_{\mathbb{N}[p:=\mathbf{itsFalse}]} \\
&= \lambda \mathtt{s}.\ [\![s\ \mathbf{1}]\!]^{\mathrm{rtc}}_{((@0\,p:m\not<\mathbf{1})\to\mathbb{N})[n:=\mathbf{1}]} \\
&= \lambda \mathtt{s}.\ [\![s]\!]^{\mathrm{rtc}}_{(m:\mathbb{N})\to(@0\,p:m\not<\mathbf{1})\to\mathbb{N}}[\![1]\!]^{\mathrm{rtc}}_{\mathbb{N}} \\
&= \lambda \mathtt{s}.\ \mathtt{s\ 1}
\end{aligned}
$$

`doubleOdd`, on the other hand, is a case that warrants checking its erased argument. This only differs from the implementation in the way that the check is always inlined.

$$
\begin{aligned}
&[\![\lambda f.\, f\,(\lambda n.\, \lambda_0 q.\, ((\mathbf{sub}\ n)\ \mathbf{1})\ q)]\!]^{\mathrm{rtc}}_{(s:((t:((m:\mathbb{N})\to(@0\,p:m\not<\mathbf{1})\to\mathbb{N}))\to\mathbb{N}))\to\mathbb{N}} \\
&= \lambda \mathtt{f}.\ [\![f\,(\lambda n.\, \lambda_0 q.\, ((\mathbf{sub}\ n)\ \mathbf{1})\ q)]\!]^{\mathrm{rtc}}_{\mathbb{N}[t:=\lambda n.\,\lambda_0 q.\,((\mathbf{sub}\ n)\ \mathbf{1})\ q]} \\
&= \lambda \mathtt{f}.\ [\![f]\!]^{\mathrm{rtc}}_{(t:((m:\mathbb{N})\to(@0\,p:m\not<\mathbf{1})\to\mathbb{N}))\to\mathbb{N}}\,[\![\lambda n.\, \lambda_0 q.\, ((\mathbf{sub}\ n)\ \mathbf{1})\ q]\!]^{\mathrm{rtc}}_{(m:\mathbb{N})\to(@0\,p:m\not<\mathbf{1})\to\mathbb{N}} \\
&= \lambda \mathtt{f}.\ \mathtt{f(\lambda n.\ assert\ n\ \not<\ 1;\ sub\ n\ 1)}\ \text{by compilation of } \mathtt{odd}
\end{aligned}
$$

We do not list the compilations of `doubleEven` and `tripleOdd` here, but only `tripleOdd` would incur a check.

5.3 Completeness of the emission

Intuitively, we want to show that

- for each program, compiled from Agda to Haskell, the result should be the same as if executed in Agda,
- when a precondition does not hold, the checked version should error, and
- when the checked version errors, a precondition does not hold.

The inverse of the first statement, "for each runtime-checked execution, there is a proof for the type checked execution", is not included because the Agda type checker is assumed to be correct.

For the positive statement, the formal statement and proof are relatively straightforward:

Theorem 1. *For well-typed terms t and v, if $t \rightsquigarrow v$, then $[\![t]\!]^{\mathrm{rtc}} \rightsquigarrow [\![v]\!]^{\mathrm{rtc}}$.*

Proof. We show a lemma to relate Agda2Hs execution of well-typed terms to the runtime-checked one.

Lemma 1. *For well-typed terms s and v, if $[\![s]\!] \rightsquigarrow [\![v]\!]$, then $[\![s]\!]^{\mathrm{rtc}} \rightsquigarrow [\![v]\!]^{\mathrm{rtc}}$.*

Proof. The only case of s that we need to consider is the one where definition 6 differs from definition 3. This case is the β-reduction $s\,u$ with $s = \lambda_0 x.\, t, s : (@0\, x : A) \to B, u : A$. By case distinction on $\pi_1 i$ where $i = \mathrm{inst}(\mathbf{Dec}\ A)$:

- *true*, then by definition 2, $\mathtt{assert}\ \mathbf{true};\ [\![t]\!]^{\mathrm{rtc}}_{B[x:=\bot]}$ reduces to $[\![t]\!]^{\mathrm{rtc}}_{B[x:=\bot]}$. In turn, this term is equivalent to $[\![t]\!]_{B[x:=\bot]}$ by induction.
- *false*, then by assumption 2, $\mathrm{inst}(\mathbf{Dec}\ A) \rightsquigarrow \langle \mathbf{false}, q\rangle$ where $q : \neg A$. However, $\neg A$ contradicts $t : A$.

□

The claim of the theorem itself follows directly with this lemma and assumption 1. □

For the negative statements, we need a workaround for the fact that ill-typed terms would not type check in Agda in the first place. We solve this by creating a context Γ with some bindings that will be used as (wrong) arguments to checked functions. This creation bears some similarity to *postulates*, an Agda construct to declare that an element is of a type without defining it. This context will also be an erased context, meaning that each binding is erased, in order to make its bindings available to Agda application, but not to runtime-relevant compilation output.

For the former direction of the negative statement, we explore the cases of a term and the conditions under which it evaluates to an error. For these conditions, it is important

to note that the term stating the opposite of the argument derives from an empty context, not Γ. When the term does not (immediately) evaluate to an error, we base the claim only on single-step reduction ($\longrightarrow$) because the premises and proofs surrounding "will error but later" are much more difficult. There are no cases given for tuples and if-then-else because they cannot reduce to an error in one reduction step and as such would be specific cases of lemma 1.

Theorem 2. *For a well-typed term $s : S$, in an erased context Γ:*

1. *$s = t\ u$ where $t : (@0\ x : A) \to S, u : A$:*

$$(\exists b.\, \varepsilon \vdash b : \neg A) \implies [\![s]\!]^{\mathrm{rtc}}_S \rightsquigarrow \text{error}$$

2. *$s = t\ u$ where $t : (x : A) \to S, u : A$ and some $v : S$, i.e. not an error:*

$$[\![s]\!]^{\mathrm{rtc}}_S \longrightarrow v \vee [\![t]\!]^{\mathrm{rtc}} \rightsquigarrow \text{error}$$

Proof. By case from the statement:

1. If $[\![t]\!]^{\mathrm{rtc}} \rightsquigarrow$ error, the statement follows directly. Otherwise, one interesting case of t remains:
   - t cannot be a variable because Γ has no unerased bindings.
   - t cannot be a Boolean or unit because those cannot be in an application head.
   - t cannot be a pair or an unerased lambda because its type does not match.
   - t can be an application or a projection on a pair. In both cases, the theorem applies inductively.
   - $t = \lambda_0 x.\, w$. Then, $[\![(\lambda_0 x.\, w)\ u]\!]^{\mathrm{rtc}} = \texttt{assert}\ [\![i]\!]_{\mathbf{Dec}\ A}\texttt{;}\ [\![w]\!]^{\mathrm{rtc}}$ where $i = \mathrm{inst}(\mathbf{Dec}\ A)$. The instance search for i does *not* depend on Γ, as it may find unproven postulates otherwise. By case distinction on the normal form of $\pi_1\ i$:
     - **false**, then $\texttt{assert}\ [\![i]\!]_{\mathbf{Dec}\ A}\texttt{;}\ [\![u]\!]^{\mathrm{rtc}} \longrightarrow$ error.
     - **true**. Then, $\pi_2\ i$ is some q where $q :$ **Reflects** A **true**. The type of q reduces to A, but this contradicts $b : \neg A$.

2. If $[\![t]\!]^{\mathrm{rtc}} \rightsquigarrow$ error, the statement follows directly. Otherwise, by distinction on t:
   - For the reasons described in (1), t cannot be a variable, Boolean, unit, pair, or erased lambda.
   - t can be an application or a projection on a pair. In both cases, the theorem applies inductively because $[\![t]\!]^{\mathrm{rtc}} \rightsquigarrow$ error has been ruled out.
   - $t = \lambda x.\, w$. Then, $[\![(\lambda x.\, w)\ u]\!]^{\mathrm{rtc}}_{S[x:=u]} = (\lambda \mathtt{x}.\, [\![w]\!]^{\mathrm{rtc}}_S)\ [\![u]\!]^{\mathrm{rtc}}_A$. From this, an error is not reachable within one step of reduction, resulting in w.

□

For the latter direction of the negative statement, we begin with a lemma on single-step reduction. We can then generalise this to multi-step reduction with a lemma on the existence of equivalent unchecked terms for each reduction of a checked term.

Lemma 2. *For a well-formed context Γ where $\Gamma \vdash s : S$:*

$$[\![s]\!]^{\mathrm{rtc}}_S \longrightarrow \text{error} \implies \exists p.\, \Gamma \vdash p : \bot$$

Proof. Because we only consider single-step reduction, and in the premise, checked s evaluates to an error, checked s must be an assertion. Only erased lambdas compile to assertions, thus, the only case of s that we must consider is $s = (\lambda_0 x.\, w)u$ where $(\lambda_0 x.\, w) : @0\, A \to S, u : A$. Then, $[\![(\lambda_0 x.\, w)u]\!]^{\text{rtc}}_{S[x:=u]} = \texttt{assert}\ [\![i]\!]_{\textbf{Dec}\ A}; [\![w]\!]^{\text{rtc}}_S$ where $i = \text{inst}(\textbf{Dec}\ A)$. By case distinction on the normal form of $\pi_1\, i$:

- **false**. Then, $\pi_2\, i$ is some q where $q : \textbf{Reflects}\ A\ \textbf{false}$. This type reduces to $A \to \bot$. Thus, $qu : \bot$ gives p.
- **true**, but with the assertion over i, an error is not reachable within one step of reduction.

□

The lemma on intermediate representations needs a small helper lemma:

Lemma 3. *For well-typed terms $t : T, u : U$ with x free in t:*

$$[\![t(x := u)]\!]^{\text{rtc}}_{T[x:=u]} = [\![t]\!]^{\text{rtc}}_T(x := [\![u]\!]^{\text{rtc}}_U)$$

Proof. By definition 6, all mentions of x in t will be compiled to $\texttt{x}$. Because the runtime checked compilation of terms recurses into its sub-terms, this is equivalent to compiling u separately and inserting it for all $\texttt{x}$. □

Lemma 4. *For well-typed terms $s, v : S$, where v is not an error:*

$$[\![s]\!]^{\text{rtc}}_S \longrightarrow v \implies \exists v' : S.\, [\![v']\!]^{\text{rtc}}_S = v$$

Proof. By case distinction on s.

- $s = t\, u$ where $t : (x : A) \to S, u : A$. By case distinction on t:
  - t cannot be a Boolean, unit, pair, or erased lambda because the type does not match.
  - t cannot be a variable because there is no reduction towards any v.
  - t can be an application or a projection on a pair. In both cases, the theorem applies inductively.
  - $t = \lambda x.\, w$. Then,

    $$[\![(\lambda x.\, w)u]\!]^{\text{rtc}}_{S[x:=u]} = [\![\lambda x.\, w]\!]^{\text{rtc}}_{(x:A)\to S}[\![u]\!]^{\text{rtc}}_A = (\lambda \texttt{x}.\ [\![w]\!]^{\text{rtc}}_S)[\![u]\!]^{\text{rtc}}_A$$

    and

    $$(\lambda \texttt{x}.\ [\![w]\!]^{\text{rtc}}_S)[\![u]\!]^{\text{rtc}}_A \longrightarrow_\beta [\![w]\!]^{\text{rtc}}_S(x := [\![u]\!]^{\text{rtc}}_A)$$

    and by lemma 3

    $$[\![w]\!]^{\text{rtc}}_S(x := [\![u]\!]^{\text{rtc}}_A) = [\![w(x := u)]\!]^{\text{rtc}}_{S[x:=u]}$$

    giving $w(x := u)$ for v'.
- $s = t\, u$ where $t : (@0\, x : A) \to S, u : A$. We can eliminate all other cases of t similarly to the bullet above and remain with $t = \lambda_0 x.\, w$. Then,

  $$[\![(\lambda_0 x.\, w)\ u]\!]^{\text{rtc}}_{S[x:=u]} = [\![\lambda_0 x.\, w]\!]^{\text{rtc}}_{(x:A)\to S} = \texttt{assert}\ [\![i]\!]_{\textbf{Dec}\ A}; [\![w]\!]^{\text{rtc}}$$

  Because $[\![s]\!]^{\text{rtc}}_S \longrightarrow v$, where v is not an error, $[\![i]\!]_{\textbf{Dec}\ A} \rightsquigarrow \textbf{true}$, giving w for v'.
- $s = \text{if}\ t\ \text{then}\ u_1\ \text{else}\ u_2$ where $t \in \{\textbf{true}, \textbf{false}\}, u_1, u_2 : S$. v' is u_1 or u_2 respectively.

□

With these lemmas, we come to the theorem itself:

Theorem 3. *For a well-formed context* Γ *where* $\Gamma \vdash s : S$*:*

$$[\![s]\!]_S^{\text{rtc}} \rightsquigarrow \text{error} \implies \exists p. \Gamma \vdash p : \bot$$

Proof. $[\![s]\!]_S^{\text{rtc}}$ multi-step reduced to an error through some $s_1 \longrightarrow ... \longrightarrow s_n$ where $s_n \longrightarrow_e$ error. By lemma 4, there also exist $s_1, ..., s_n$ such that $[\![s_1]\!]^{\text{rtc}} \longrightarrow ... \longrightarrow [\![s_n]\!]^{\text{rtc}}$. For $[\![s_n]\!]^{\text{rtc}}$, lemma 2 applies. □

Chapter 6

Discussion

We have shown how it is possible to generate complete and automatic runtime checks for a verified-to-functional source-to-source translation system in order to guard against input that does not meet criteria only expressible in dependent types. In this chapter, we discuss what place our Agda2Hs extension takes in an ecosystem of verified and unverified code, especially with regard to how it compares to handwriting the emitted checks instead. Furthermore, we discuss some additional features that our solution could have included, but that also come with drawbacks.

6.1 Comparison to writing checks by hand

We acknowledge that our approach of automatically generating checks at translation level is not necessary for having them per se. Listing 6.1 shows two Agda files reimplementing the introductory example from listing 1.1, but with a check spelled out in Agda, and its **native** Agda2Hs compilation. This approach should be used in tandem with a technical solution to prohibit erased arguments in files imported from unverified Haskell code, again, e.g. on CI level.

The most obvious drawback to handwriting is that the relatively mundane checks have to be written first. There are some issues of a more technical nature with this approach, but workarounds could be created for them:

- The output has an if-then-else instead of guards. Agda's `case` pattern matching is somewhat akin to guards, but translation to guards is not supported in Agda2Hs at this time.
- We used `error` through a hack by defining an untranslated function of type `Nat`. A generic `error` like in Haskell cannot be implemented in Agda as is because it creates arbitrary types: Agda2Hs's signature for it is `error : {a : Set} {@0 @(tactic absurd) i : ⊥} → String → a`, i.e. to create arbitrary types, you must first have a proof of ⊥. More advanced error handling (see section 8.1 for more detail) would overcome the need to `error` altogether, but probably introduce more complex pragmas that could also be used here.

6.2 Potential features left unimplemented and their challenges

6.2.1 Omission of runtime checking class instances

Type classes is an Agda and Haskell feature to implement functions on a data type according to a fixed set of signatures, akin to interfaces in object-oriented programming. A structure that implements a type class is called an instance. In Agda, the type class is no separate

```
-- PostRtc.agda
module PostRtc where
open import Haskell.Prelude

subtractFromGreater : (x y : Nat) → @0 ⦃ IsFalse (x < y) ⦄ → Nat
subtractFromGreater x y = x - y
{-# COMPILE AGDA2HS subtractFromGreater #-}

-- SubtractManual.agda
module SubtractManual where
open import Haskell.Prelude hiding (error)
open import Haskell.Extra.Dec
open import Haskell.Extra.Dec.Instances
-- hack: put into same level (Agda2Hs handles common prefixes of modules poorly)
import PostRtc

-- hack: simulate Haskell error
error : String → Nat
error _ = 0

subtractFromGreater : (x y : Nat) → Nat
-- could also use instance search (`it`) with implicit argument
-- specification, but this is not translated correctly by Agda2Hs
subtractFromGreater x y = ifDec decIsFalse
  (PostRtc.subtractFromGreater x y)
  (error "check failed")
{-# COMPILE AGDA2HS subtractFromGreater #-}
```

```
-- PostRtc.hs
module PostRtc where

import Numeric.Natural (Natural)

subtractFromGreater :: Natural -> Natural -> Natural
subtractFromGreater x y = x - y

-- SubtractManual.hs
module SubtractManual where

import Haskell.Extra.Dec.Instances (decIsFalse)
import Numeric.Natural (Natural)
import qualified PostRtc (subtractFromGreater)

subtractFromGreater :: Natural -> Natural -> Natural
subtractFromGreater x y
  = if decIsFalse (x < y) then PostRtc.subtractFromGreater x y else
      error "check failed"
```

Listing 6.1: Comparison to the manual approach: generating similar checks without the need for Agda2Hs modification

construct, but simply a record with instance fields (Agda Development Team 2024b, Record Types, Instance fields). For Agda2Hs, a record can be marked as to be compiled to a type class by stating `class` in the pragma. Class fields can contain erased arguments, for which we do not support automatic runtime checking.

Classes have no logic in themselves, but instances translated from Agda2Hs do, and could have runtime checking. Listing 6.2 shows a minified definition of Agda2Hs's `Num` record, which is to be compiled to a Haskell typeclass, and the instance for `Nat`. Agda2Hs's `native` compilation of these definitions is shown below.

```
open import Haskell.Prelude
open import Agda.Builtin.Nat using (_-_)

record MiniNum (a : Set) : Set₁ where
  field
    @0 MinusOK : a → a → Set
    sub : (x y : a) → @0 ⦃ MinusOK x y ⦄ → a
open MiniNum public

instance
  iMiniNumNat : MiniNum Nat
  MinusOK iMiniNumNat n m = IsFalse (n < m)
  sub iMiniNumNat n m = Agda.Builtin.Nat._-_ n m

{-# COMPILE AGDA2HS MiniNum class #-}
{-# COMPILE AGDA2HS iMiniNumNat #-}
```

```
module Class where

import Numeric.Natural (Natural)

class MiniNum a where
    sub :: a -> a -> a

instance MiniNum Natural where
    sub n m = n - m
```

Listing 6.2: An Agda record and instance with their native Agda2Hs compilation to a class and instance

Listing 6.3 shows a theoretical runtime-checked version of the instance. We did not implement this approach for two reasons:

- Instances cannot be separated into pre- and post-check versions like we do with other definitions. It is not possible to have two instances to the same class and data type in one namespace in Haskell, although it is in Agda.
- This only works for instances of data types also defined in Agda, and as such leaves a gap for instances defined in Haskell.

6.2.2 Types that become decidable by their parameters

The checks that we generate omit a specific kind of erased types that is not checkable directly, but can become checkable indirectly in some cases. One example is the `∃` record, which, while

```
instance MiniNum Natural where
    sub n m | IsFalse (n < m) = n - m
            | otherwise = error "…"
```

Listing 6.3: Instance-level checking: a hypothetical approach to runtime checking listing 6.2

not a Haskell construct, is shipped by the Agda2Hs library as a useful extension. Essentially, it is the Σ type from dependent type theory with the second part erased; the kind we also considered in chapter 5. Thus, it is a value and a proven statement about that value (see listing 6.4).

```
record ∃ {ℓ ℓ' : Level} (a : Set ℓ) (@0 P : a → Set ℓ') : Set (ℓ ⊔ ℓ') where
  field
    value : a
    @0 proof : P value
```

Listing 6.4: Abridged ∃ record from the Agda2Hs library

Consider the function `useExists` from listing 6.5. Our implementation treats this as an uncheckable definition because the function type uses a record that contains an erased field and the record is not decidable.

```
useExists : ∃ Nat (λ n → IsFalse (1 < n)) → Nat
useExists (n ⟨ p ⟩) = _-_ 1 n ⦃ p ⦄
```

Listing 6.5: Simple use of ∃

However, the parametrised ∃ record becomes decidable when given a decidable type in the `proof` position, such as in this example. Listing 6.6 shows a hypothetical checking of `useExists`. While this seems reasonably straightforward to implement at surface level, we decided against doing so because such an approach would imply an additional complexity regarding avoiding double checking. Smart constructors for the data types and records that *are* checkable at constructor level already exist. Thus, not double checking them would require either tracking which types have been given smart constructors, or testing whether the definition behind the type used can also be checked with a smart constructor. Alternatively, it would be possible to generate all checks at function level and skip smart constructors entirely. This approach would lead to extra checks likewise, as data types and records would be runtime checked on every use. Note that this limitation of not checking parameter-level decidable types can generally be worked around by e.g. unrolling the record to a Π type.

```
subtractFromGreater :: Natural -> Natural
subtractFromGreater n
  | decIsFalse (1 < n) = UseExists.PostRtc.useExists n
  | otherwise = error "Runtime check failed: decIsFalse (1 < n)"
```

Listing 6.6: Parameter-level checking: a hypothetical approach to runtime checking listing 6.5

6.3 Eagerness of runtime checks

One drawback to consider when working with this extension is that on top of the time that a runtime check takes in itself, the check may evaluate more eagerly than the execution would have made necessary. Listing 6.7 show the Agda definition and checked output excerpt of

an example: the first ten elements of a list will be added, requiring a precondition due to output types. Logically, only those first ten would need to be checked, and due to Haskell's laziness, the rest would not be evaluated, which works even for infinite lists. However, with the generated check, the entire list will be checked, which will not work with infinite lists.

```
-- Agda2Hs implements a `take` for lists, but it does not have universe
-- polymorphism. The `All` predicate, however, has a universe polymorphic
-- definition, so we implement `take` with universe polymorphism.
take : ∀ {a} {A : Set a} → Nat → List A → List A
take n [] = []
take zero nil = []
take (suc n) (x :: xs) = x :: take n xs

Positive : Integer → Set
Positive i = IsTrue (i >= 0)

i>=0⇒NIOK : (i : Integer) → @0 ⦃ Positive i ⦄ → iNumNat .Num.FromIntegerOK i
i>=0⇒NIOK (pos n) = tt

-- Explicit fold to provide relevant instance arguments.
intSum : (is : List Integer) → @0 ⦃ All Positive is ⦄ → Nat
intSum [] = 0
intSum (i :: is) ⦃ allCons ⦄ = fromInteger i ⦃ i>=0⇒NIOK i ⦄ + intSum is ⦃ it ⦄
{-# COMPILE AGDA2HS intSum #-}

take⁺ : ∀ {n a b} {A : Set a} {P : A → Set b}
          (xs : List A) → All P xs → All P (take n xs)
take⁺ [] allNil = allNil
take⁺ {zero} (_ :: _) _ = allNil
take⁺ {suc n} (_ :: xs) allCons = allCons ⦃ is = take⁺ xs it ⦄

takeSum : (is : List Integer) → @0 ⦃ All Positive is ⦄ → Nat
takeSum is = intSum (take 10 is) ⦃ take⁺ is it ⦄
{-# COMPILE AGDA2HS takeSum #-}
```

```
takeSum :: [Integer] -> Natural
takeSum is
  | decAll is (\ x -> decIsTrue (x >= 0)) = Lazy.PostRtc.takeSum is
  | otherwise =
    error
      "Runtime check failed: decAll is (\\ x -> decIsTrue (x >= 0))"
```

Listing 6.7: No lazy evaluation: checks more eager than the computation itself

6.4 Closing remarks

In short, we have shown how unsafe input to translated verified functions can be checked, and how this check can be inserted automatically. We have also shown where this check insertion is truly necessary—something the manual technique from section 6.1 does not handle correctly. However, we do not present this as a solution that would be strictly necessary for safe use of Agda2Hs.

The sentiment from the Agda2Hs development team seems to be that considering the overall current mission of Agda2Hs, a merge of the feature is not worth the incurring technical burden at this time. Although a large share of the code of the implementation is in a separate module, there also are a number of changes that were necessary to put into existing Agda2Hs logic, which can affect the maintainability of the project. I believe that it was interesting to show that this principle can work, but in its current state, it may not align with Agda2Hs's goal of integrating verified code in larger software projects perfectly.

Chapter 7

Related work

In this chapter, we explore related work. Much of the related work we found is not based on dependent types, but was created for the much more popular imperative languages. Reasoning about such languages is generally based on *Hoare logic*, which is also based on pre- and postconditions (Hoare 1969). At the end of the chapter, we also discuss related work for dependent types.

To begin, it was not a novel idea to increase safety by making runtime checks of properties accessible. Specifying pre- and postconditions has been a first-class feature of the object-oriented Eiffel language since its beginning (Meyer 1986). The obvious drawback is that formal arguments that would make a costly runtime check obsolete are never available.

Eiffel's rigour had been partially inspired by the Ada language. The formally defined SPARK subset of Ada attempts to automatically create a proof for pre- and postconditions based on Hoare logic (Praxis High Integrity Systems Ltd. 2005, dating back to at least 1994). Failure to find such a proof is a compile-time error and must be rectified by rewriting the program in a more obviously correct way. Because these conditions are undecidable, a desirable condition may have to be skipped entirely. Compared to our work, their approach seems to sometimes require extra nudging to pass a condition. The effect, however, may be similar, being an increased development time and lack of possible optimisations.

An early verification standard which supported both mechanised proofs, again with Hoare logic, and runtime checking was the Java Modeling Language (JML). Frameworks like JACK (Burdy, Requet, and Lanet 2003) and JMLC (Cheon 2003) have been created to support both methods separately, the former computing proofs automatically like SPARK. In comparison, our approach is more hybrid in the way that interoperation of code translated from Agda receives verification, but no runtime checking, and code interoperating between Haskell and Agda receives runtime checking.

The Prusti system (Astrauskas et al. 2019) is a more modern approach to automatic verification of imperative programs. It is written for Rust and leverages Rust's ownership tracking. Hegglin 2023 then used runtime checks to improve upon Prusti's verification, notably for the cases of unverified functions and unverified contexts, the same gaps that our solution tackles. Assumed theorems from Prusti can also be runtime checked—more on this in section 8.1. He uses Rust's `panic!` in case of a failed runtime check, much akin to our `error`.

Using a dependently typed programming language is quite different from the Hoare logic-related approaches above: the overall proof style is more akin to mathematical logic, and Turing completeness is sacrificed for decidability and termination guarantee. When working exclusively in dependently typed languages, runtime checks are not necessary, at least not without using unsafe language features. However, we are not aware of any efforts to e.g. runtime check dependent arguments in Coq's program extraction (Inria 2024).

Regarding formal analysis, there is some remote resemblance to a polymorphic blame calculus (Ahmed et al. 2011), in which incorrect type casts lead to blame. However, this work

has not been extended to dependent types. It is noteworthy that they too present a theorem on how well-typed programs will not go wrong (or blame), but not on how ill-typed programs will. This difficulty may be analogue to our restriction on the negative statements (theorem 2).

Chapter 8

Conclusion

We have demonstrated how the gap between verified and unverified code can be bridged with runtime checking. We have also shown formally these checks are complete and make further execution on input not in accordance with the erased arguments impossible. Regardless, we acknowledge the invasiveness of this feature as definitions with undecidable preconditions are made unavailable.

We conclude with the further work that could be built on top of our contributions. The most central piece is a more nuanced way to produce and handle errors than `error`. We close with some further, less immediate or smaller potential improvements.

8.1 Future work

The biggest piece of future work is providing more error handling capability than giving a simple `error`. It can be caught as `SomeException` with `Control.Exception.catch` as in listing 3.5, but working with just an error string is unsuitable for non-trivial programs: actual `Exception`s would be better. One difficulty of providing advanced error handling is the presumable need to design a usable programmer interface for error handling.

Agda2Hs's compilation pragmas could be leveraged here; listing 8.1 shows an unimplemented mock-up of such an approach: a data type is defined for exceptions, deriving `Exception` and having one string in each constructor. For a function with erased arguments, each must be named, and the desired exception is specified in the compilation pragma to the function. The proposed output is shown in the same listing.

On the other hand, this explicit exception insertion has the drawback of necessary specification on the Agda side. Furthermore, any system more advanced than `error` would have implications on the types of definitions: exceptions, as in the example above, can only be caught from code in the `IO` monad, which may turn code that would have been pure otherwise into monadic code.

Further areas of future work are:

Formalising Agda2Hs itself The statement of the analysis from chapter 5 would also be more reliable if a formalisation of Agda2Hs itself existed. With such a formalisation, it would no longer be necessary to depend on simplifications like pretending to work in just one language or assumptions like all outputs being identical. This formalisation is, however, difficult to create and was considered to be beyond the scope of this thesis.

Postulate tracing Because of the completeness guarantees of the checks that we emit, working with unsafe Agda, namely Agda containing `postulate` statements, was not considered here. In practice, Agda code will often contain postulates due to the hardness of creating mechanised proofs. It could be useful to also check postulates with a decidable output type.

```
data RuntimeException : Set where
  Empty : String → RuntimeException
  SubNat : String → RuntimeException
{-# COMPILE AGDA2HS RuntimeException deriving (Show, Exception) #-}

subtractFromGreaterList :
  (x : Nat) (ys : List Nat)
  ⦃ @0 ne : NonEmpty ys ⦄ ⦃ @0 gt : IsFalse (x < head ys) ⦄
  → Nat
subtractFromGreaterList x (y :: _) = x - y
{-# COMPILE AGDA2HS subtractFromGreaterList
    ne=RuntimeException.Empty,gt=RuntimeException.SubNat #-}
```

```
-- SubtractException/PostRtc.hs
{-# LANGUAGE DeriveAnyClass #-}
module SubtractException.PostRtc where

import Control.Exception

import Numeric.Natural (Natural)

data RuntimeException = Empty String
                      | SubNat String
                          deriving (Show, Exception)

subtractFromGreaterList :: Natural -> [Natural] -> Natural
subtractFromGreaterList x (y : _) = x - y
-- SubtractException.hs
module SubtractException (RuntimeException(Empty, SubNat),
                          SubtractException.subtractFromGreaterList) where

import Control.Exception

import Haskell.Extra.Dec.Instances (decIsFalse, decNonEmpty)
import Numeric.Natural (Natural)

import SubtractException.PostRtc

subtractFromGreaterList :: Natural -> [Natural] -> Natural
subtractFromGreaterList x ys
  | decNonEmpty ys && decIsFalse (x < head ys) =
    return $ SubtractException.PostRtc.subtractFromGreaterList x ys
  | not (decNonEmpty ys) =
    throw $ Empty "Runtime check failed: decNonEmpty ys"
  | otherwise =
    throw $ SubNat "Runtime check failed: decIsFalse (x < head ys)"
```

Listing 8.1: `Exception`: a proposed way of more nuanced error handling capabilities

An extension to this would be postulate tracing, i.e. checking preconditions even on internal calls if proofs given to them depend on a postulate.

Delinting The output created by our check insertion system is sometimes more verbose than it would need to be. Listing 3.5, for instance, could have `\x -> x` and `\a0 -> f (go0 a0)` replaced by `id` and `f . go0` respectively. An easy to apply, but only partial fix would be to run the output through a linter like *hlint* at the last stage (Mitchell 2024). However, it may be more consistent to apply such a fix to Agda2Hs as a whole, which can produce similar η-reducible outputs, not least because Haskell has a greater emphasis on supporting a point-free style. On the other hand, this may lead to confusion with respect to comparability with the Agda source.

8.2 Closing remarks

This thesis has shown how it is possible to runtime check input to source-translated verified programs: the gap between unverified and verified software really can be closed with only few unsupported constructs left. The concept could pose an important and helpful addition to the complex task of writing correct software. However, *handling* the case of a failed check better remains a difficult piece of future work, and this feature would be required for real-world projects.

Acknowledgements I want to thank my supervisors Jesper and Bohdan for the various and effortful ways in which they helped me and made this thesis possible. I also want to thank my co-chair Annibale and the programming languages group at TU Delft, especially my fellow master students who eased making this field feel more accessible and supported me through the life of this thesis. Thanks also go out to all my musical companions for the well-needed balance they gave me in this period. Finally, I want to thank my parents, my brother, my friends, and my girlfriend Aurelia for her love, support, and review of my stylistic offences.

Bibliography

Agda Development Team (2024a). *Agda 2*. URL: https://github.com/agda/agda.

— (2024b). *Agda 2.7.0.1 documentation*. URL: https://agda.readthedocs.io/en/v2.7.0.1/.

— (2024c). *The Agda standard library*. URL: https://github.com/agda/agda-stdlib.

Ahmed, Amal et al. (2011). "Blame for all". In: *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '11. Association for Computing Machinery. URL: https://dx.doi.org/10.1145/1926385.1926409.

Astrauskas, Vytautas et al. (2019). "Leveraging rust types for modular specification and verification". In: *Proceedings of the ACM on Programming Languages*. Vol. 3. OOPSLA (Athens, Greece). New York, NY: Association for Computing Machinery, pp. 1–30. URL: https://dx.doi.org/10.1145/3360573.

Atkey, Robert (2018). "Syntax and Semantics of Quantitative Type Theory". In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '18 (Oxford, UK). New York, NY: Association for Computing Machinery. URL: https://dx.doi.org/10.1145/3209108.3209189.

Broberg, Niklas and Dan Burton (2020). *haskell-src-exts: Manipulating Haskell source: abstract syntax, lexer, parser, and pretty-printer*. URL: https://github.com/haskell-suite/haskell-src-exts.

Burdy, Lilian, Antoine Requet, and Jean-Louis Lanet (2003). "Java Applet Correctness: A Developer-Oriented Approach". In: *FME 2003: Formal Methods*. Ed. by Keijiro Araki, Stefania Gnesi, and Dino Mandrioli. Berlin, Heidelberg, Germany: Springer, pp. 422–439. URL: https://doi.org/10.1007/978-3-540-45236-2_24.

Cheon, Yoonsik (2003). "A runtime assertion checker for the Java Modeling Language". PhD thesis. Iowa State University. URL: https://dx.doi.org/10.31274/rtd-180813-9872.

Cockx, Jesper et al. (2022). "Reasonable Agda is correct Haskell: writing verified Haskell using agda2hs". In: *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*. Haskell 2022 (Ljubljana, Slovenia). New York, NY: Association for Computing Machinery, pp. 108–122. URL: https://doi.org/10.1145/3546189.3549920.

Eremondi, Joseph S. (2023). "On The Design of a Gradual Dependently Typed Language for Programming". PhD thesis. University of British Columbia.

Findler, Robert Bruce and Matthias Felleisen (2002). "Contracts for higher-order functions". In: *SIGPLAN Not.* 37.9, pp. 48–59. URL: https://doi.org/10.1145/583852.581484.

Hegglin, Cedric (2023). "Contract Checking at Runtime and Verification-based Optimizations for a Rust Verifier". Master's thesis. ETH Zürich.

Hoare, Charles Antony Richard (1969). "An axiomatic basis for computer programming". In: *Commun. ACM* 12.10, pp. 576–580. URL: https://doi.org/10.1145/363235.363259.

Howard, William Alvin (1980). "The Formulae-as-Types Notion of Construction". In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Ed. by Haskell Curry et al. Academic Press.

Inria, CNRS and contributors (2024). *Coq 8.19.1 documentation*. URL: https://coq.inria.fr/doc/V8.19.1/refman.

Juhošová, Sára (2024). "How Novices Perceive Interactive Theorem Provers (Extended Abstract)". In: *Proceedings of Workshop on Type-Driven Development*. TyDe '24 (Milan, Italy). New York, NY: Association for Computing Machinery. URL: https://icfp24.sigplan.org/details/tyde-2024-papers/1/How-Novices-Perceive-Interactive-Theorem-Provers-Extended-Abstract-.

Meyer, Bertrand (1986). "Genericity versus inheritance". In: *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA '86 (Portland, OR). New York, NY: Association for Computing Machinery, pp. 391–405. URL: https://doi.org/10.1145/28697.28738.

Mitchell, Neil (2024). *HLint: Haskell source code suggestions*. URL: https://github.com/ndmitchell/hlint.

Norell, Ulf (2007). "Towards a practical programming language based on dependent type theory". PhD thesis. Chalmers University of Technology and Göteborg University.

Okasaki, Chris (1998). "Even higher-order functions for parsing or Why would anyone ever want to use a sixth-order function?" In: *Journal of Functional Programming* 8.2, pp. 195–199. URL: https://doi.org/10.1017/S0956796898003001.

Praxis High Integrity Systems Ltd. (2005). *SPARK 95 – The SPADE Ada 95 Kernel*. Ed. by Rod Chapman and Peter Amey. URL: https://didawiki.cli.di.unipi.it/lib/exe/fetch.php/magistralesicurezza/sss/spark95.pdf.

The GHC Team (2024). *ghc: The GHC API*. URL: https://hackage.haskell.org/package/ghc.

The Haskell IDE Team (2019). *Can not go to definition of non local libraries*. URL: https://github.com/haskell/haskell-language-server/issues/708.

Acronyms

CI continuous integration

JML Java Modeling Language

QTT Quantitative Type Theory

TCB trusted computing base

TCM type checking monad

Appendix A

Lawfulness of the tree equality definition

This is the omitted lawfulness proof of the equality instance to a binary tree from listing 2.8.

```
cong₃ : ∀ {A B C D : Set} (f : A → B → C → D) {a b c x y z} →
        a ≡ x → b ≡ y → c ≡ z → f a b c ≡ f x y z
cong₃ _ refl refl refl = refl

instance
  iLawfulEqTree .isEquality (Leaf x) (Leaf y) with (x == y) in h
  ... | True = cong Leaf $ equality x y h
  ... | False = flip exFalso h ∘ equality' x y ∘ leaf-injective
    where
      leaf-injective : Leaf x ≡ Leaf y → x ≡ y
      leaf-injective refl = refl
  iLawfulEqTree .isEquality xt@(Branch x xl xr) yt@(Branch y yl yr)
    with (x == y) in h
  ... | True = mapReflects branch tree-injective
      (reflects-&& (isEquality xl yl) (isEquality xr yr))
    where
      branch : xl ≡ yl × xr ≡ yr → xt ≡ yt
      branch (refl , refl) = cong₃ Branch (equality x y h) refl refl
      tree-injective : xt ≡ yt → xl ≡ yl × xr ≡ yr
      tree-injective refl = refl , refl
      reflects-&& : ∀ {e f} → Reflects (xl ≡ yl) e → Reflects (xr ≡ yr) f
                            → Reflects (xl ≡ yl × xr ≡ yr) (e && f)
      reflects-&& {False} re1 _ = re1 ∘ fst
      reflects-&& {True} {False} _ = _∘ snd
      reflects-&& {True} {True} refl refl = refl , refl
  ... | False = flip exFalso h ∘ equality' x y ∘ branch-injective
    where
      branch-injective : xt ≡ yt → x ≡ y
      branch-injective refl = refl
```

Appendix B

Core runtime check determination functions

This is the full definition of the `checkRtc` functions from section 4.3.

```
-- Creates a runtime check if necessary and possible, informing C accordingly.
-- Takes telescope of type to check, name, level of nesting,
-- and expression on success.
checkRtc :: Telescope -> QName -> Hs.Exp () -> [NestedLevel] -> C RtcResult
checkRtc tel name success lvls = do
  (_, chk) <- checkRtc' (0, 0) tel lvls
  case chk of
    NoneErased' -> return NoneErased
    Uncheckable' -> return Uncheckable
    Checkable' {..} -> do
      tellAllCheckable name
      let rhs = eApp success theirRhs
          chkName = hsName $ prettyShow $ qnameName name
          chk = createRtc chkName theirLhs theirChks rhs $ binds theirDecls
      return $ Checkable [chk]

-- Recursively check for runtime checkability in nested types.
-- Accumulates on name indices for `go` function and `a` argument.
-- Takes telescope of type to check.
checkRtc' ::
  NameIndices ->
  Telescope ->
  [NestedLevel] ->
  C (NameIndices, RtcResult')
checkRtc' idcs tel lvls = do
  -- Partition out arguments that are erased and at top level
  -- (those we will attempt to check)
  (erased, call) <- partitionM (checkTopOrDataErased . fst) $ zip doms telsUpTo
  ourChks <- uncurry createGuardExp `mapM` if head lvls == Odd then erased else []
  -- Recursively accumulate checks on arguments below top level
  (belowIdcs, belowChks) <- mapAccumLM checkRtc'' idcs $ map (,lvls) call
  (belowIdcs,)
    <$> if not $ all isJust belowChks && all isJust ourChks
      then return Uncheckable'
      else -- all checkable or none erased
```

```
        let (theirLhs, theirRhs, decls) = unzip3 $ catMaybes belowChks
            theirDecls = concat decls
            -- all checks below found an instance
            theirChks = catMaybes ourChks
         in if null theirDecls && null erased
              then return NoneErased'
              else return Checkable' {..}
  where
    doms = telToList tel
    telsUpTo = map (\i -> fst $ splitTelescopeAt i tel) [0 ..]

-- Check a single type for runtime checkability.
-- Accumulates on name indices for `go` function and `a` argument.
-- Takes domain of type and telescope up to that point for context.
-- If checkable, returns lhs and rhs at that point
-- plus declarations from checks below.
checkRtc'' ::
  NameIndices ->
  ((Dom (ArgName, Type), Telescope), [NestedLevel]) ->
  C (NameIndices, Maybe (Hs.Pat (), Hs.Exp (), [Hs.Decl ()]))
checkRtc'' (goIdx, argIdx) ((d, tUpTo), _ : lvls) =
  -- Mutual recursion with checkRtc'
  addContext tUpTo (checkRtc' (goIdx, ourArgIdx) tAt lvls) >>= \case
    (idcs, NoneErased') -> return (idcs, Just (ourLhs, argVar, []))
    (idcs, Uncheckable') -> return (idcs, Nothing)
    ((theirGoIdx, theirArgIdx), Checkable' {..}) -> do
      let go = "go" ++ show theirGoIdx
          conflicts = tAtNames `intersect` [go, arg, up]
      errorWhenConflicts conflicts
      let (ourGoIdx, ourRhs, ourRtc) =
            if null theirChks
              then
                -- inline if nothing to check at this level (consumes no `goIdx`)
                -- e.g. `\ a3 -> a2 (go0 a3)`, continuing the example above
                ( theirGoIdx,
                  Hs.Lambda () theirLhs $ argVar `eApp` theirRhs,
                  theirDecls
                )
              else
                let -- e.g. `up m a2`
                    lhs = hsPat up : theirLhs
                    -- e.g. `up m (\ a3 -> a2 (go0 a3))`
                    rhs = hsVar up `eApp` theirRhs
                    rtc =
                      createRtc (hsName go) lhs theirChks rhs $
                        if theirDecls
                 in -- e.g. `go1 a1`
                    (succ theirGoIdx, hsVar go `eApp` [argVar], [rtc])
      return ((ourGoIdx, theirArgIdx), Just (ourLhs, ourRhs, ourRtc))
  where
    tAt = domToTel $ snd <$> d
    tAtNames = map (fst . unDom) $ telToList tAt
```

```
    name = fst $ unDom d
    -- Use arg name if available, otherwise insert one (consumes one on `argIdx`)
    (arg, ourArgIdx) =
      if name == "_"
        then ("a" ++ show argIdx, succ argIdx)
        else (name, argIdx)
    ourLhs = hsPat arg
    argVar = hsVar arg
    up = "up"
```

Appendix C

Full example of nested check determination

This is an illustration what happens in the mutual recursions of appendix B. It is based on this Agda type:

```
tripleOdd : (((m : Nat) → @0 IsTrue (m > 0) →
              (((n : Nat) → @0 IsFalse (n < 1) → Nat) → Nat) → Nat) → Nat
```

The checked output is this Haskell definition:

```
tripleOdd ::
          ((Natural -> ((Natural -> Natural) -> Natural) -> Natural) ->
             Natural)
            -> Natural
tripleOdd a0 = TripleOdd.PostRtc.tripleOdd (\ a1 -> a0 (go1 a1))
  where
    go1 up m a2
      | decIsTrue (m > 0) = up m (\ a3 -> a2 (go0 a3))
      | otherwise = error "Runtime check failed: decIsTrue (m > 0)"
      where
        go0 up n
          | decIsFalse (n < 1) = up n
          | otherwise = error "Runtime check failed: decIsFalse (n < 1)"
```

These are the actions and results of `checkRtc'` at each level:

| (((m : Nat) → @0 … → (((n : Nat) → @0 … → Nat) → Nat) → Nat) → Nat ~(0, 0)

Reserve a0 for ((Nat → ((Nat → Nat) → Nat) → Nat) → Nat):

| ((m : Nat) → @0 … → (((n : Nat) → @0 … → Nat) → Nat) → Nat ~(0, 1)

Reserve a1 for (Nat → ((Nat → Nat) → Nat) → Nat):

| | | (m : Nat) → @0 … → (((n : Nat) → @0 … → Nat) → Nat) → Nat ~(0, 2)

Reserve a2 for ((Nat → Nat) → Nat):

| | | ((n : Nat) → @0 … → Nat) → Nat ~(0, 3)

Reserve a3 for (Nat → Nat):

Reserve go0 for (((Nat → Nat) → Nat) → Nat):

Inline:

Reserve go1 for ((Nat → ((Nat → Nat) → Nat) → Nat) → Nat → ((Nat → Nat) → Nat) → Nat):

Inline: