The Agda UHC Backend

Philipp Hausmann

M.Sc. thesis ICA-4003373

Supervisors: Dr. Wouter Swierstra
Dr. Atze Dijkstra

October 9, 2015

Universiteit Utrecht

Department of Computing Science

Abstract

This thesis studies how we can facilitate combined Haskell/Agda developments. As foundation for our research we have created the Agda UHC backend, targeting the intermediate Core language of the Utrecht Haskell Compiler. We will present a formal description of our translation scheme, which now powers all major Agda backends.

Building upon our new backend, we introduce a Contract framework specifically aimed at the Foreign Function Interface (FFI). As with most FFI implementations, a major challenge are the different type systems of the languages involved. Our contract library provides a concise and powerful way to translate data between Agda foreign languages like Haskell. We also provide a formal specification of our contract framework, making it feasible to implement a similar scheme in other languages.

Furthermore, we provide an improved Agda FFI interface for function calls which combines well with our Contract framework.

Our contributions make Agda a more viable choice for applied dependently-typed programming and provide an elegant and novel solution for the FFI problem in a dependently typed setting.

Contents

Chapter 1

Introduction

The importance of computer programs in society is growing steadily; most parts of our modern civilization grind to a standstill when our electronic gadgets fail us. This increasing reliance on computer programs causes a rising demand for assuring the correctness of programs. To meet this increased need for correct programs, a lot of effort is being spent on different ways to reach this goal. Testing is often the first approach chosen to meet this goal. Its major drawback is that exhaustive testing of all possible program inputs is rarely possible, especially with the ever-increasing size and complexity of programs. In consequence, non-exhaustive testing can only prove the absence of certain errors but never the correctness of a program. And while testing is certainly a valuable tool for assuring correctness, it is often not sufficient by itself.

One avenue seeing a lot of research activity are type-systems and dependent types in particular, which could well be the tool solving this issues. Dependent types provide a way to formally state the expected behavior and verify that the program adheres to the specification. Dependent types only admit programs which are correct by construction, thereby assuring the correctness of the program for all possible inputs.

However, often programs have to interact with some external entities and other programs running on the same or other computers. Creating a formal specification for a program often requires knowledge about this other external systems, but the specification of this external systems are often written in an informal way and cannot be readily incorporated in a formal specification. This problem is not new, and every programming language which interacts with any outside system with a more relaxed type system faces this difficulty.

The additional challenge in interoperating with the outside world from Agda is because Agda does *not* readily admit programs failing at runtime; there is no exception or error mechanism in Agda. We strive to remedy this problem by providing a controlled way to perform unsafe operations embedded in Agda.

This thesis documents our work and the solutions we have implemented:

- We will describe our experience implementing a new compiler backend for Agda. We survey the existing Agda backends and the design criteria we set ourselves for our new compiler.

  Our new backend enables easy experimentation with combined Agda/Haskell developments. The principal algorithms implemented may not be novel and many elements have been described before, but we do provide a more detailed formal semantics of the translation from Agda to UHC's Core language (Chapter 3).

- We have built a new Foreign Function Interface for calling Haskell functions from Agda. Having a shared intermediate language for both the Utrecht Agda Compiler and the Utrecht Haskell Compiler makes it possible to have a common runtime data representation and exchange data between the two languages freely (Chapter 4).
- We have created a contracts framework enabling us to annotate FFI calls with their invariants and verifying them at runtime. Building upon our compiler backend and FFI work, this feature increases the safety of FFI calls while making them more concise and convenient to use at the same time. Our work shows how Contracts can be implemented and used in a dependently typed setting and we explore some of the available design space (Chapter 5).

The Agda UHC backend is available as part of the official Agda codebase [1]. All other code has been published on public Github repositories, namely the Contracts library [2], the new FFI for Agda [3] and the accompanying standard library version [4].

[1]`https://github.com/agda/agda`
[2]`https://github.com/phile314/agda-contracts/`
[3]`https://github.com/phile314/agda/tree/new-ffi`
[4]`https://github.com/phile314/agda-stdlib/tree/ffi`

Chapter 2

Background

2.1 Agda - the Language

Agda is the language used throughout this Master Thesis, hence we will give a very brief introduction into it. More in-depth introductions are available online [1].

Agda is a functional language, whose main difference compared to traditional functional languages such as Haskell and ML is its dependently typed nature. This mix of dependent types with functional programming makes Agda suitable for creating programs, specifications *and* proofs; unifying all these applications in one language. We will use Agda mostly as programming language in this thesis and only touch on the specification/proofing parts lightly.

The Agda language is quite similar in spirit and syntax to Haskell, but not without some major differences. Let us begin with data type declarations. In Agda we can define data types similar to the GADT syntax in Haskell. However, Agda allows us to index data types, which for example means that we can encode the length of a list in it's type. The following vector type for example mimics a Haskell list, where we additionally encode the length:

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A 0
  _::_ : ∀ {n} → A → Vec A n →  Vec A (suc n)
```

The definition follows the usual List definition, where we have a base case nil constructing the empty Vector of length 0, and the cons case constructing a Vector with the length incremented by one.

Taking advantage of this more expressive data type, we can define a safe head function where the type will ensure that it is only called on non-empty lists:

```
head : ∀ {n A} → Vec A (suc n) → A
head (x :: v) = x
```

In fact, Agda requires that all functions are total, it is hence impossible to define the traditional non-total head function. This restriction is necessary to ensure that Agda is a consistent logic, admitting non-total functions would make it inconsistent.

[1]http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Othertutorials

The remainder of this thesis assumes that the reader is familiar with the basics of dependently typed programming and Agda. The Agda tutorial by Norell [24] provides a good introduction to dependently typed programming and Agda for those familiar with functional programming.

Chapter 3

The Agda UHC Backend

3.1 Existing Agda backends

The current version of Agda has three backends. The MAlonzo backend targets Haskell as intermediate language, which is then compiled using GHC. The JS backend, as it's name says, targets JavaScript and is intended for compiling web applications. Lastly, the Epic backend targets the Epic language and uses the Epic compiler to produce an executable. We shall now look at these three backends a bit more in depth.

The most notable compiler for Agda is MAlonzo [21]. It targets the Haskell language, using the Glasgow Haskell Compiler (GHC) to produce executables from the generated Haskell code. While a substantial part of the Agda language can easily be translated into valid Haskell code, this does not hold for the entire language; after all, Haskell is not dependently typed. The MAlonzo compiler works around this problem by inserting (unsafe) type coercions. This coercions increase the code size significantly, as can be seen in the Hello World example in Figure 3.1.

This behavior, together with the lack of optimizations in MAlonzo, can lead to a blowup in the size of the generated Haskell code [1]. The inserted type-coercions also prevent GHC from applying certain type-directed optimization, which is unfortunate as MAlonzo relies solely on GHC for optimizing the generated code. Although these coercions can have a performance impact, they do not affect the correctness of the program, as the Agda source code has already been type checked by Agda.

MAlonzo also provides a Foreign Function Interface (FFI) to Haskell. Using this FFI, programmers can call Haskell functions from Agda, export Agda functions to Haskell and reuse Haskell data types in Agda. Only the common subset of both programming languages can be used in the FFI. Dependently typed functions, for example, are not supported. The current FFI relies on the programmer to specify the exact mapping between Agda and Haskell using pragmas. We will discuss the FFI more thoroughly in chapter 4.

Besides the MAlonzo compiler, there are several more experimental Agda backends targeting JavaScript [17] and Epic [25, 7].

3.1.1 What should be the target language?

While targeting Haskell directly is certainly a viable approach, the numerous coercions inserted by MAlonzo push GHC to its limits. There are examples of modest Agda files

```
main = run (putStrLn (show (10 + 10)))
```

(a) A small Agda program.

```
main = d1
name1 = "HelloWorld.main"
d1
  = MAlonzo.RTE.mazCoerce
      (MAlonzo.Code.IO.d21
         (MAlonzo.RTE.mazCoerce MAlonzo.Code.Agda.Primitive.d3)
         (MAlonzo.RTE.mazCoerce MAlonzo.Code.Data.Unit.Base.d3)
         (MAlonzo.RTE.mazCoerce
            (MAlonzo.Code.IO.d75
               (MAlonzo.RTE.mazCoerce
                  (MAlonzo.Code.Data.Nat.Show.d11
                     (MAlonzo.RTE.mazCoerce
                        (MAlonzo.Code.Data.Nat.Base.d14
                           (MAlonzo.RTE.mazCoerce
                              (MAlonzo.Code.Data.Nat.Base.↩
                                  ↪ mazIntegerToNat (10 :: Integer)))
                           (MAlonzo.RTE.mazCoerce
                              (MAlonzo.Code.Data.Nat.Base.↩
                                  ↪ mazIntegerToNat
                                 (10 :: Integer)))))))))
```

(b) The generated Haskell code for the above Agda program. The mazCoerce expressions are type coercions.

Figure 3.1: A example Agda program and its translation to Haskell.

generating huge Haskell files, that require unacceptable compilation time. We felt that it might be worthwhile to explore alternative approaches.

We explicitly wanted to avoid creating a full compiler from scratch; instead, we wanted to reuse existing infrastructure to deal with the low-level aspects of compilation. Foremost, reusing an existing language as our target language saves a lot of work and allows us to leverage all existing tooling and infrastructure. Secondly, we hope to experiment further with a Foreign Function Interface between two high-level languages such as Agda and Haskell. Having a common target language is almost a necessity for such experiments.

The obvious candidate for an intermediate language would have been GHC Core [27]. GHC Core, however, is a typed intermediate language, based on System FC [29]. While all of Haskell's high-level language features, including type families [10] and GADTs [28], can be translated to System FC, it is *not* dependently typed. As a result, translating Agda to System FC is a non-trivial exercise.

This mismatch in type systems is the same problem the MAlonzo backend exhibits. We could solve the problem in the same way: inserting numerous type coercions. These coercions, however, will also cause the same problems already present in the current MAlonzo backend. We could, perhaps, do a slightly better job by only inserting those type coercions that are strictly necessary, as opposed to naively inserting them everywhere where they *might* be required. Coq's extraction mechanism takes this approach [18]. By replicating an ML type checker inside the Coq compiler, the extraction mechanism only inserts coercions at places where the generated ML code would not type check otherwise. Nonetheless, following the same approach would have required a relatively large effort and would have seriously limited our freedom to experiment and modify the target language to suit our needs.

Although there have been proposals for a dependently-typed core language [4], there is no mature implementation that we could use off the shelf. Instead, we chose to resolve this mismatch between type systems by targeting an untyped core language.

Adapting the existing Epic backend would have been another possibility. It's target language Epic is a simple untyped lambda calculus extended with data types. The main drawback of this approach would have been that the existing Haskell FFI could not have been retained. The Agda Standard library and most executable Agda code assumes the existence of a Haskell FFI; not being able to support it would require significant changes to many existing Agda projects. Furthermore, there is no other high-level language with which the Epic backend can interact, necessitating that the runtime system is built mostly in C.

The candidate target language we chose in the end is the Utrecht Haskell Compiler's Core language. The Utrecht Haskell Compiler (UHC), developed by Dijkstra et al. [12], is a Haskell Compiler which has an intermediate Core language similar to Epic. The advantage of choosing UHC Core over Epic is that the Utrecht Haskell Compiler also compiles Haskell to executables via UHC Core. This allows us to retain the existing Haskell FFI and use it as a vehicle for exploring the possibility of mixed Agda-Haskell developments. Additionally, the UHC project also focuses primarily on experimenting with new ideas which fits well with the the experimental nature of our new Agda Compiler.

3.2 Translating between Agda and UHC

Both Agda and UHC consist of a series of transformations between intermediate languages; starting from the high-level Agda or Haskell input and going towards more simplistic core languages. Our compiler links Agda's Internal Syntax language with the UHC Core language; reusing Agda's type checker and UHC's code generation.

While UHC's code generation already existed, its compilation pipeline required significant changes to support our use case. We will describe these changes in more detail in Section 3.3.1.

Reusing Agda's type checker by itself is easy; the hard part has been translating Agda's Internal Syntax language to the UHC Core language. We were able to reuse some code from the existing Agda Epic backend, but differences in the target language and new Agda features posed new challenges. For example, the support for varying arity functions described later in this section is new compared to the Epic backend.

The task of translating Agda's Internal Syntax into the relevant target language is one that all backends have to carry out. Yet at the time when we started implementing the UHC backend, every backend had it's own translation with it's own specific set of limitations and bugs. This is clearly not a scaleable approach and increases the maintenance burden considerably. We have thus decided to introduce an additional intermediate language in Agda, called Treeless. The Treeless language is modeled after UHC Core and is a slightly more high-level, locally-nameless version of UHC Core. This new language allows us to share the complicated conversion of case trees to a cascade of case expressions by all backends. A welcome side-effect of this approach is that this also helped fixing a major bug in the code generator of the MAlonzo backend [16]. How our new Treeless Syntax fits into Agda can be seen in Figure 3.2.

We will give a formal description of the translation from Agda's Internal syntax to the Treeless syntax in the remainder of this chapter. We will omit the translation from Treeless syntax to UHC Core, as it mainly consists of replacing De Bruijn indices by names.

3.2.1 Treeless Intermediate Syntax

Our new Treeless Syntax is an untyped lambda calculus using a locally-nameless representation extended with data types. We do not specify the evaluation regime of Treeless syntax, to keep open the possibility of targeting strict languages in the future. The grammar of Treeless syntax is given in Figure 3.3.

Local variables γ use De Bruijn indices. The $\mathsf{Con}\ n\ \psi \rightarrow t$ alternative matches on the constructor with the name n which has arity ψ.

All Treeless syntax terms in the rest of this paper will be written in red to distinguish them from terms in Agda's Internal Syntax.

3.2.2 Translating Agda's Internal Syntax to Treeless Syntax

The input for our Compiler is Agda's Internal Syntax (AIS), which is produced by the existing Agda frontend and presented in Figure 3.4. The description of the translation uses a simplified version of AIS. To help distinguish AIS from Treeless, we will use the color blue for AIS expressions in the rest of this chapter.

Figure 3.5 contains the complete formal semantics of the translation. In the following subsections, we will go through all constructs of AIS and explain the translation in detail. The translation from the AIS term A to the Treeless term B is written as $\mathsf{A} \rhd \mathsf{B}$.

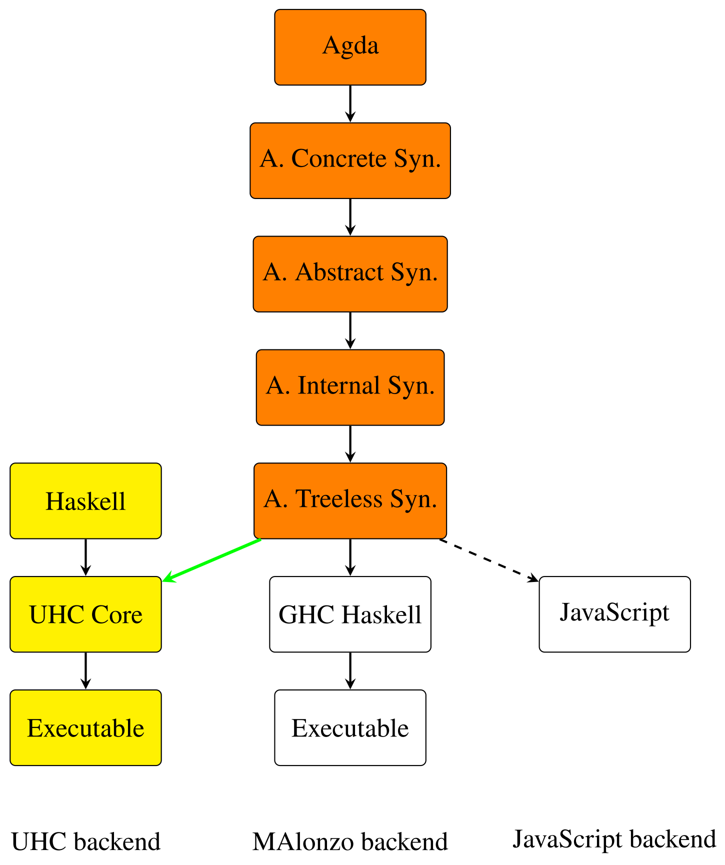


Figure 3.2: The Agda architecture and how it fits together with the UHC backend. Parts of UHC are marked yellow, parts of Agda orange. Note that the drawing omits some of the intermediate languages present in the UHC and GHC compilers. The JavaScript backend currently uses Agda's Internal Syntax as its input, but the goal is to make it use Treeless syntax as input too.

Terms	t	$::=$	γ	Variable
		$\mid$	n	Definition
		$\mid$	$t\ \vec{t}$	Application
		$\mid$	$\lambda \rightarrow t$	Lambda Abstraction
		$\mid$	$\mathrm{Con}\ n\ \vec{t}$	Constructor Application
		$\mid$	$\mathrm{case}\ \gamma\ \mathrm{of}\ \vec{alt}\ \mathrm{else}\ t$	Case with default
		$\mid$	$\top$	Unit
		$\mid$	$\bot$	Failure
Alternatives	alt	$::=$	$\mathrm{Con}\ n\ \psi\ \rightarrow\ t$	Constructors

Figure 3.3: The abstract syntax of Treeless. The notation $\vec{t}$, $\vec{alt}$ refers to a list of terms and alternatives respectively.

Name	n			
Definition	def	$::=$	Fun c	Function
		$\mid$	Axiom	Axiom
CaseTree	c	$::=$	Case $\alpha\ \vec{b}$	Inspect
		$\mid$	Done $\psi\ t$	Finished
		$\mid$	Fail	Absurd case
Branch	b	$::=$	Con $n\ \psi \rightarrow c$	Constructor
		$\mid$	CatchAll $\rightarrow c$	Default
Term	t	$::=$	Var $\beta\ \vec{t}$	Variable / Application
		$\mid$	Def $n\ \vec{t}$	Application / Projection
		$\mid$	$\lambda \rightarrow t$	Abstraction
		$\mid$	Con $n\ \vec{t}$	Constructor Application
		$\mid$	$\Pi\ t\ t$	(Dependent) Function Type
		$\mid$	Set	Set / Type

Figure 3.4: The abstract syntax of Agda. Irrelevant parts of the syntax have been omitted for brevity.

Axioms

Axioms are one of the simplest definitions to translate. They arise from postulates, for example:

```
    postulate axiom-0==1 : 0 == 1
```

Postulates can be used to introduce new assumptions by declaring their type, without providing the associated definition. Such postulates are only interesting during proofs; if such an axiom needs to be evaluated at runtime, the program will crash. We hence translate it to the crashing Treeless expression $\perp$, as can be seen in rule E-Def-Axiom.

Functions and case splitting trees

The AIS Fun construct introduces a new function. Most such functions are defined using pattern matching, just as in other functional languages such as Haskell or OCaml. During type checking this patterns are converted to case splitting trees, as first described by Augustsson [5]. The AIS case tree generated by Agda's type checker is the input to our compiler.

Nameless references AIS uses a locally nameless representation for variable references [23, 19, 11]. Somewhat surprisingly, AIS uses both De Bruijn indices *and* levels. This unusual choice is motivated by the varying arity feature of Agda explained later in this section. The Treeless syntax on the other hand uses exclusively De Bruijn indices for local references, which considerably simplifies code generation. Importantly, the AIS De Bruijn indices do *not* necessarily coincide with the Treeless De Bruijn indices. We will use the letter γ to denote Treeless De Bruijn indices, β for AIS De Bruijn indices and α for AIS De Bruijn levels.

$$grow(\Gamma, \Delta) = [0..(\Delta - 1)] +\!\!+ \; wk(\Delta, \Gamma)$$

$$\frac{}{\text{Axiom} \rhd \bot}\ \text{E-Def-Axiom} \qquad \frac{[]; \bot; c \rhd c'}{\text{Fun } c \rhd c'}\ \text{E-Def-Fun}$$

$$\frac{\begin{array}{l}\text{let } \Delta = max(0, \alpha + 1 - |\Gamma|) \\ \text{let } \Gamma'[\gamma_n..\gamma_\alpha..\gamma_0] = grow(\Gamma, \Delta) \qquad \Gamma'; \phi\ 0..(\Delta - 1); \vec{b} \rhd \phi' \qquad \Gamma'; \phi'; \alpha; \vec{b} \rhd \vec{b}'\end{array}}{\Gamma; \phi; \text{Case } \alpha\ \vec{b} \rhd \lambda_0 \lambda_1..\lambda_{\Delta-1} \rightarrow \text{case } \gamma_\alpha \text{ of } \vec{b}' \text{ else } \phi'}\ \text{E-Case}$$

$$\frac{\Gamma; \phi; c \rhd c'}{\Gamma; \phi; [b_0, b_1, ..(\text{CatchAll} \rightarrow c).., b_n] \rhd c'}\ \text{E-CatchAll}$$

$$\frac{\vec{b} \text{ contains no CatchAll}}{\Gamma; \phi; \vec{b} \rhd \phi}\ \text{E-NoCatchAll}$$

$$\frac{(wk\ \psi\ [\gamma_n..\gamma_{\alpha+1}]) +\!\!+ [0..(\psi - 1)] +\!\!+ (wk\ \psi\ [\gamma_{\alpha-1}..\gamma_0]); \phi; c \rhd c'}{[\gamma_n..\gamma_\alpha..\gamma_0]; \phi; \alpha; \text{Con } n\ \psi \rightarrow c \rhd [\text{Con } n\ \psi \rightarrow c']}\ \text{E-Branch-Con}$$

$$\frac{}{\Gamma; \phi; \alpha\ \text{CatchAll} \rightarrow \text{c} \rhd []}\ \text{E-Branch-CatchAll}$$

$$\frac{}{\Gamma; \phi; \text{Fail} \rhd \bot}\ \text{E-Fail}$$

$$\frac{\text{let } \Delta = max(0, \psi - |\Gamma|) \qquad grow(\Gamma, \Delta); t \rhd t'}{\Gamma; \phi; \text{Done } \psi\ t \rhd \lambda_0 \lambda_1..\lambda_{\Delta-1} \rightarrow t'}\ \text{E-Done}$$

$$\frac{\Gamma; \vec{t} \rhd \vec{t'}}{\Gamma[\gamma_0..\gamma_\alpha..\gamma_n]; \text{Var } \alpha\ \vec{t} \rhd \gamma_\alpha\ \vec{t'}}\ \text{E-VarApp} \qquad \frac{grow(\Gamma, 1); t \rhd t'}{\Gamma; \lambda \rightarrow t \rhd \lambda \rightarrow t'}\ \text{E-Abs}$$

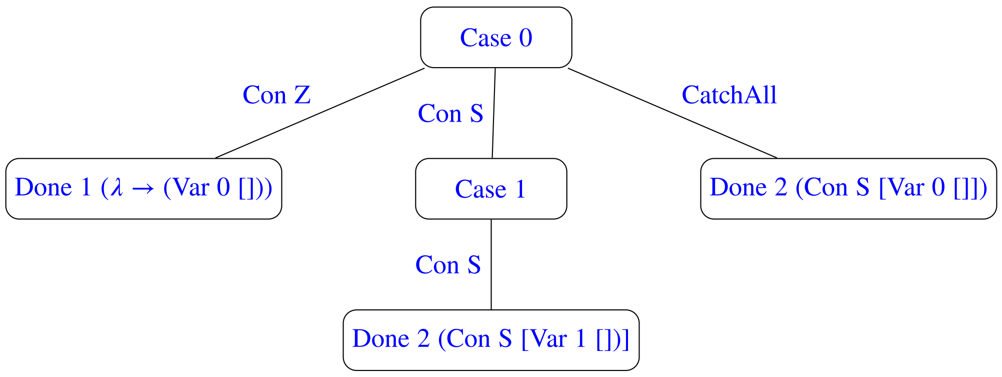
$$\frac{\Gamma; \vec{t} \rhd \vec{t'}}{\Gamma\ \text{Def } n\ \vec{t} \rhd n\ \vec{t'}}\ \text{E-DefApp}$$

$$\frac{\Gamma; \vec{x} \rhd \vec{x}'}{\Gamma; \text{Con } n\ x \rhd \text{Con } n\ \vec{x}'}\ \text{E-Con} \qquad \frac{}{\Gamma; \Pi\ t_1\ t_2 \rhd \top}\ \text{E-Pi} \qquad \frac{}{\Gamma; \text{Set} \rhd \top}\ \text{E-Set}$$

Figure 3.5: The formal description of the translation from Agda's internal Syntax to Treeless Syntax.

```
f : ℕ → ℕ → ℕ
f Z = λ y → y
f (S x) (S y) = S x
f x y = (S y)
```

(a) Example Agda function.



(b) Case-tree translation of the above Agda function. The branches $\vec{b}$ of a Case non-terminal are drawn as edges, and case trees as nodes.

Figure 3.6: A simple example of the case splitting tree translation.

The conversion from AIS De Bruijn levels and indices to Treeless De Bruijn indices is achieved by passing down an environment Γ, containing the mapping from AIS levels/indices to Treeless indices. We will use the convention that the left side of the environment contains the closest bound AIS variable, and the rightmost entry is the most distant bound AIS variable.

We will use this environment to look up the Treeless De Bruijn index of any nameless AIS variable. Initially, we set Γ to the empty environment in rule E-Def-Fun, as the function has not yet received any arguments or defined any local variables.

Case splitting trees Unlike Treeless case expressions which scrutinize exactly one variable at a time, a case tree can inspect any number of variables. The case splitting tree translation of an example Agda function can be seen in Figure 3.6.

A case tree consists of non-terminal Case nodes that scrutinizes one variable at a time; and Done and Fail terminal nodes representing the result of the given branch.

Terminal nodes

The Fail non-terminal occurs whenever the Agda type checker believes that this node should never be reached. These nodes correspond to absurd patterns in Agda. As Fail terminal nodes should never be reached at runtime, we translate them to the Treeless error term ⊥:

$$\frac{}{\Gamma;\phi;\mathsf{Fail} \rhd \bot}\ \text{E-Fail}$$

Second, a $\text{Done}\ \psi\ t$ non-terminal represents a successful pattern match, containing the body of the corresponding function clause for this terminal node. We ignore the ψ value for now and simply translate the body term t:

$$\frac{\Gamma; t \rhd t'}{\Gamma; \text{Done}\ \psi\ t \rhd t'}\ \text{E-Done'}$$

Term translation

The terms Π and Set are significant for type checking only. In Agda, a value of type Set cannot be inspected or pattern matched on. As Agda enforces that it is impossible to observe any value of these types, they cannot affect the runtime semantics. For executing a program, it is thus safe to replace all occurrences of such values by the unit value $\top$.

One could also be tempted to completely remove any values of these kind. This could potentially alter the semantics of the translated program. Agda does not evaluate expressions under lambdas; dropping lambda abstractions taking type expressions could remove lambdas blocking evaluation. A sound way to erase types is to only do so where it doesn't affect soundness, for example in saturated function applications. A more detailed description of when such types may be soundly erased can be found in previous work by Letouzey [18].

The lambda construct translated by the E-Abs rule is the first place where we grow the environment. The rule uses the $grow(\Gamma, \Delta)$ function, which grows the environment Γ by Δ entries. As we are introducing new bound variables, we also have to weaken the existing environment. Let $wk(\Gamma, \Delta)$ be the function which weakens the environment Γ by Δ in the definition for the $grow$ function:

$$grow(\Gamma, \Delta) = [0..(\Delta - 1)] \mathbin{+\!\!+} wk(\Delta, \Gamma)$$

The remaining term constructs have a clear one-to-one translation from AIS to Treeless Syntax and can hence easily be translated.

Case non-terminals $\text{Case}\ \alpha\ \vec{b}$ non-terminals use the argument at the De Bruijn level α to branch on. De Bruijn levels count arguments from the outermost towards the innermost binder; as the environment Γ grows to the left, the name has to be looked up at position α from the right.

In the example case tree in Figure 3.6, the root non-terminal branches on the first argument of the function `f`. If the first variable has the value Con S, another Case non-terminal is encountered and the second argument of the function is examined. A case tree may contain any number of such Case non-terminals.

Each $\text{Case}\ \alpha\ \vec{b}$ non-terminal is translated to one case expression. Given a way to translate the branches $\vec{b}$ to Treeless alternatives and assuming that the environment Γ already contains the variable matched on and ignoring non-exhaustive cases, the translation is defined as follows:

$$\frac{\Gamma; \vec{b} \rhd \vec{b}'}{\Gamma[\gamma_n..\gamma_\alpha..\gamma_0]; \text{Case}\ \alpha\ \vec{b} \rhd \text{case}\ \gamma_\alpha\ \text{of}\ \vec{b}'}\ \text{E-Case'}$$

Varying arity One peculiarity of Agda complicating the translation to Treeless is that, in contrast to Haskell, Agda doesn't require all functions clauses to take the same number of arguments. Agda function clauses may accept a varying number of arguments; for instance, when assigning a dependent type to functions such as `printf`, the value of one argument may determine the number of subsequent arguments.

This behavior makes the conventional way of compiling functions by putting as many lambda abstractions as there are function arguments around the whole function body impossible. Instead, we need to check at all Case non-terminals if enough lambda-abstractions are already present; if not, additional lambda abstractions need to be inserted. To this purpose we let Δ be the number of lambda abstractions to insert, and grow the environment accordingly:

$$\frac{\text{let } \Delta = \mathit{max}(0, \alpha + 1 - |\Gamma|) \qquad \text{let } \Gamma'[\gamma_n..\gamma_\alpha..\gamma_0] = \mathit{grow}(\Gamma, \Delta) \qquad \Gamma'; \alpha; \vec{b} \rhd \vec{b}'}{\Gamma; \text{Case } \alpha\ \vec{b} \rhd \lambda_0\lambda_1..\lambda_{\Delta-1} \to \text{case } \gamma_\alpha \text{ of } \vec{b}'} \text{ E-Case''}$$

The varying arity feature motivates using De Bruijn levels in the case splitting trees; with De Bruijn indices it would be impossible to tell when additional lambdas need to be inserted.

We also need to extend the rule for the *Done* ψ t non-terminal in a similar way. The ψ value we have ignored beforehand tells us how many arguments the function body expects to have already been abstracted over:

$$\frac{\text{let } \Delta = \mathit{max}(0, \psi - |\Gamma|) \qquad \mathit{grow}(\Gamma, \Delta); t \rhd t'}{\Gamma; \phi; \text{Done } \psi\ t \rhd \lambda_0\lambda_1..\lambda_{\Delta-1} \to t'} \text{ E-Done}$$

Constructor branches Previously, we assumed that branches could be translated to Treeless alternatives, but have not gone into any details. A Con ψ c branch translates directly to a Treeless alternative. The number ψ records the number of arguments the constructor takes, and we replace the current case scrutinee γ_α in the environment Γ with ψ Treeless De Bruijn indices. As this enlarges the environment, the whole remaining environment needs to be weakened by ψ. This updated environment makes the pattern matched variables available to be used in the child case splitting tree stored inside the current Con non-terminal. The body of the generated Treeless alternative is the sub tree c, translated to Treeless syntax.

$$\frac{(\mathit{wk}\ \psi\ [\gamma_n..\gamma_{\alpha+1}]) +\!\!+ [0..(\psi - 1)] +\!\!+ (\mathit{wk}\ \psi\ [\gamma_{\alpha-1}..\gamma_0]); \phi; c \rhd c'}{[\gamma_n..\gamma_\alpha..\gamma_0]; \phi; \alpha; \text{Con } n\ \psi \to c \rhd [\text{Con } n\ \psi \to c']} \text{ E-Branch-Con}$$

Note that this replacement inside the environment can change the size of Γ and may change the indexes of variables inside it. This is already taken into account when the case splitting tress are generated and requires no further consideration during the translation.

CatchAll branches So far we have ignored the CatchAll c branches. These come into play when none of the other branches match and provide a default value. This default c in itself is another case splitting tree.

CatchAll branches apply to a whole sub tree. To come back to the example in Figure 3.6, the CatchAll branch at the root node also applies to the Case 1 non-terminal. This can be useful when the same CatchAll applies to many Case non-terminals. For

```
1  f = λ → case 0 of
2       Con Z 0 → (λ → 0)
3       Con S 1 → (λ →
4            case 0 of
5                 Con S 1 → Con S 2
6                 else ((λ → Con S 0) 0)
7            )
8       else (λ →  Con S 0)
```

Figure 3.7: Translation to Treeless of the example Agda function from Figure 3.6.

example, when compiling the following Agda function, a failing pattern match on the first *or* second argument will yield the same result:

```
f : ℕ → ℕ → ℕ
f zero zero = zero
f x y = x
```

Our earlier rule E-Case" plainly ignore CatchAll branches. To fix this, we need to pass around the default value to use when generating Treeless case expressions in E-Case". To this end, we pass down the value ϕ, which contains the Treeless term to use as default value. Initially, we set ϕ to $\bot$, causing the produced program to crash when a pattern match fails.

We may also need to update the default value ϕ. If the current non-terminal contains a CatchAll branch, rule E-CatchAll triggers and returns the new default value ϕ'. If there is no CatchAll branch, rule E-NoCatchAll fires and the current default value is used unchanged.

Furthermore, the updated default value ϕ' needs also to be passed along when translating the case tree branches. Doing this correctly requires careful attention though. The rule E-Case may introduces new lambda abstractions, which the Treeless expression ϕ does not expect. The expression ϕ has been compiled in the environment Γ, where the corresponding CatchAll branch has been defined, rather than the extended environment $grow(\Gamma, \Delta)$. To remedy this, the ϕ needs to ignore all newly-introduced lambda abstractions. This is achieved by applying the default value immediately to the newly introduced lambda arguments before passing it on. The changed rule E-Case looks like this:

$$\frac{\begin{array}{l}\text{let } \Delta = max(0, \alpha + 1 - |\Gamma|) \\ \text{let } \Gamma'[\gamma_n..\gamma_\alpha..\gamma_0] = grow(\Gamma, \Delta) \qquad \Gamma'; \phi\ 0..(\Delta - 1); \vec{b} \rhd \phi' \qquad \Gamma'; \phi'; \alpha; \vec{b} \rhd \vec{b}'\end{array}}{\Gamma; \phi; \text{Case } \alpha\ \vec{b} \rhd \lambda_0\lambda_1..\lambda_{\Delta-1} \to \text{case } \gamma_\alpha \text{ of } \vec{b}' \text{ else } \phi'} \text{ E-Case}$$

3.2.3 Example

The translation of the Agda example from Figure 3.6 is shown in Figure 3.7.

One side-effect of the inherited default values can be seen in the repetition of the term $(\lambda \to \text{Con S } 0)$ on line 6 and 8. The higher the sub tree in question and the larger the default expression, the more significant this duplication becomes. To avoid this problem, our compiler shares the default expression between all occurrences using a let binding.

3.3 Lessons learned

Apart from constructing the necessary translation and compilation scheme, making our compiler work also gave rise to some engineering and technical challenges. In this section we will go into more detail about what these difficulties were and the solutions we have adopted.

3.3.1 Adapting UHC to our needs

Build System At the start of this project, UHC was a standalone Haskell compiler. It had not been used before as component of another compiler. Using UHC as part of our Agda compiler required using the build system in an unsupported way and was fairly fragile. The long dependency lists of both Agda and UHC, combined with the custom configuration and build system of UHC made versioning conflicts in the Cabal Package Database a recurring problem.

UHC Core In contrast to the build system, exposing the UHC Core language as a public API not only required technical changes, but also posed challenges in understanding the behavior of the generated code. Due to its origin as an internal intermediate language, there is no formal specification of UHC Core. While most of the syntactic constructs are self-explanatory, a few of the invariants are harder to discover. For example, UHC Core requires case alternatives to be in lexicographical order. This is not immediately obvious from the syntax of UHC Core and was poorly documented. Consequently, early versions of our compiler violated this invariant and the generated code exhibited unexpected behavior.

In the rare case that we encountered such unexpected behavior, the difficult part often lay in identifying the actual cause of the issue; fixing the issue after the root cause had been discovered usually was straightforward.

3.3.2 Testing our Compiler

As with all sufficiently complex software, verifying the correctness of our Agda compiler is non-trivial. We have not tried to formally prove the correctness of our compiler. Instead, we rely on a test suite to validate the correctness of our compiler.

To this end, we have created a test suite consisting of 40 example Agda programs in total. Each of these programs can be run, and the output can be compared to a golden standard. Most of the examples are inspired by existing programs; in some cases by adapting existing Agda programs for our need, in others by porting examples from languages such as Haskell.

A somewhat unexpected issue we ran into was that none of the popular Haskell test frameworks supported our main use cases directly out of the box. The best candidate was the tasty-golden [31] package, but it does not have an interactive mode for inspecting the test results. Nonetheless, we used tasty-golden as basis for our tests and created the new tasty-silver [32] package which adds an interactive test running mode and a simpler API.

The test collection itself, very roughly, consists of two kinds of programs. The first kind are small Agda programs, each testing a single Agda feature. These programs also serve as regression tests to avoid re-introducing fixed bugs.

The second kind of test programs are more complicated programs, which involve more computations. The intent here is to stress test all parts of the compiler and runtime system. For example, we have a port of the interactive Eliza[1] program from GHC's nofib benchmark suite.

These test cases target both our compiler and the existing MAlonzo compiler. This allows us to compare the output and use the MAlonzo backend as baseline for development. In fact, our principled approach to testing revealed an existing bug in the MAlonzo backend that the old test suite missed [20].

3.4 Related work

A lot of work has been done on compiling dependently typed languages in the last few years, without which this project would not have been feasible.

Other Agda Compilers The existing Epic backend [25] was very important in the development of our compiler. While the implementation of our backend by now significantly diverges from the Epic backend, it nonetheless provided us with a good starting point from where on we could incrementally improve our backend. It is also the most ambitious existing Agda compiler in terms of optimizations, and clearly shows the viability of compiling Agda to an untyped core language. Sadly, it is not maintained anymore and does not work with up-to-date Agda code.

The MAlonzo backend, written by Benke [21], on the other hand is better maintained and works reasonably well. Compared to the Epic backend, it has the advantage that it can interface freely with Haskell libraries. However, it does not support many of the optimizations that the Epic backend does implement.

Last but not least, there is also a JavaScript backend [17]. It supports some of the optimizations of the Epic backend and has a mature FFI to JavaScript. The motivation for targeting JavaScript was to use Agda for developing web applications. Using JavaScript as a target language for running code locally, however, may not be the best choice.

Coq Leaving the world of Agda, Coq supports the extraction of OCaml, Haskell and Scheme from Gallina. It suffers from similar problems as compiling Agda to Haskell as discussed in Section 3.1.

Idris Idris, created by Brady [8], is the dependently typed language aimed specifically at writing real-world runnable programs. To this end, it currently features a compiler targeting the LLVM intermediate language. This is a different design compared to the extractions to high-level languages used by Agda and Coq, as it targets a low-level language. While this approach may yield favorable performance, it severely limits the possibility of interacting with other high-level languages – an important constraint to support the existing FFI between Agda and Haskell.

3.5 Future work

While our new UHC backend creates correct code, it would be worthwhile to further improve it's performance. A major issue is that currently Agda's typechecker destroys

[1]https://github.com/ghc/nofib/tree/master/spectral/eliza

sharing even when explicitly introduced by using an Agda let-binding [33]. This behavior occurs both, at typechecking time *and* at runtime. Fixing this issue in the backends first requires the Agda typechecker to be fixed, which may happen in the near future.

As with all compilers, it would also be nice if more optimizations were added than currently exist. For example, some of the optimizations implemented in the Idris language [9] could be ported to Agda. Our newly introduced Treeless syntax should make it feasible to implement those optimizations in Agda.

Last but not least, Agda currently does not properly support per-module/per-package compilation. While the backends try to avoid recompilation as much as possible inside a given source code collection, there is no means of easily sharing compilation results between different projects. Compilation speeds could probably be improved significantly by compiling libraries, such as the Agda standard library, only once instead of once per project. The root cause of this problem is that Agda currently has no package management system at all, and it would be a great improvement to add such a system.

Chapter 4

FFI

As a short interlude before introducing a Contract framework for Agda Foreign Function calls, we will discuss the current state of Agda's Foreign Function Interface (FFI) and some improvements we have implemented. This improvements will allow us to introduce a nice syntax for combining Contracts and FFI calls later on in chapter 5.

Each of the existing Agda backends features a Foreign Function Interface (FFI) to their target language. The basic functionality of all these FFI interfaces is similar; they bind an Agda definition to a definition in the foreign language. This involves specifying the identity of the relevant Agda and foreign definitions, and declaring the type of the binding. As all three existing backends use a similar approach, we will discuss the limitations and challenges for the Haskell FFI of the UHC backend only. The findings should nevertheless carry over to the other backends.

4.1 Existing FFI

Functions

The basic functionality of any FFI is to call functions from the foreign language. This is done by exposing a matching Agda function definition, which will call the foreign Haskell function at runtime when evaluated.

We assume that the Agda compiler provides a way to pass primitive data types to and from Haskell, e.g. integers. We can then define a FFI call; first we postulate an Agda definition matching the Haskell definition we want to expose. Second, using a pragma we bind this Agda definition to a Haskell implementation.

```
postulate
  abs : HsInteger → HsInteger
{-# COMPILED_UHC abs (Prelude.abs) #-}
```

Note that the Agda definition is a postulate. This means that Agda will *never* try to reduce the _+_ function for type checking, it treats it as a black box.

Data types

While we can assume that the compiler provides some way to pass primitive data types to Haskell, we would prefer to be able to pass arbitrary data types around. In general, this is not possible for arbitrary Agda data types; however, for the Haskell case it is possible to define proper FFI semantics for a large subset of Haskell data types and their Agda counterpart.

Depending on the compiler backend, this may include GADTs, but will always include non-GADT data types without class constraints. To enable the FFI machinery to generate correct code, a FFI pragma mapping the Agda definition to a Haskell definition needs to be used:

```
data List (A : Set) : Set where
  [] : List A
  _::_ : A → List A → List A
{-# COMPILED_DATA_UHC List Data.List.List [] (:) #-}
```

4.2 Level/Set arguments

The current FFI of Agda is type-directed to some degree, but only in a very limited fashion. The MAlonzo backend verifies that FFI function bindings only involve valid FFI types, but none of the other backends do any verification whatsoever. And while it in general is not possible to completely guarantee the correctness of FFI bindings, it would be preferable to include as many checks as possible.

Furthermore, depending on the target language the builtin Level and Set Agda types need to be treated specially. For example, Haskell has no notion of explicit Level and Set values; types are completely separated from terms. Compare the Agda identity function, which takes the explicit type argument A:

```
id : (A : Set) → A → A
id A x = x
```

with the corresponding Haskell function:

```
id :: forall a . a -> a
id x = x
```

Note that the type a in the Haskell code is *not* a normal argument, but is treated differently and erased at runtime.

If we want to expose the Haskell identity function in Agda, we might now be tempted to write the following FFI definition:

```
postulate
  id′ : (A : Set) → A → A
{-# COMPILED_UHC id′ The.Haskell.id #-}
```

This, however, will not work. As described in the formal semantics in section 3.2.2, Set terms are translated to the unit value, but they are *not* completely removed. Agda thus expects that the compiled representation of id takes two argument. The Haskell identity function doesn't satisfy this invariant. This can easily be resolved by wrapping the Haskell identity function in an additional lambda, discarding the superfluous Set argument:

```
postulate
  id″ : (A : Set) → A → A
{-# COMPILED_UHC id″ (\_ -> The.Haskell.id) #-}
```

The user is required to insert this additional lambdas for all Level and Set arguments occurring in the Agda type. The process of inserting the lambdas is completely type-directed and mechanical, and doesn't require any human interaction in principle. It is thus preferable that we extend the Agda compilers to handle this translation internally, instead of burdening the user.

4.3 Syntax

A more syntactical problem with the current FFI system is that it only admits top-level FFI calls. However, especially when generating FFI calls automatically it would be preferable to allow FFI calls to be used like normal expressions. Furthermore, the current pragma-based approach will grow cumbersome as additional FFI target languages are added.

4.4 The new foreign calls

To remedy the problems we have outlined, we have extended the Agda language with a special foreign call expression:

```
id‴ = foreign (hsCall "Prelude.id") ((A : Set) → A → A)
```

The foreign keyword takes an ordinary Agda record describing the Haskell entity as first argument, and an Agda type as second argument. This new syntax emphasizes the fact that the FFI interface is type-directed. Apart from the new syntax, this new FFI interface also uses the type information to automatically handle Level and Set arguments correctly. This becomes especially handy when we want to call higher-ranked Haskell functions. As a simple example, lets say we want to call the following Haskell functions from Agda:

```
f :: (forall a . a -> a) -> Bool
f = ...
```

To do so, we can just give the corresponding type- and universe-polymorphic Agda type for the foreign call:

```
f = foreign (hsCall "f")
  (({a : Level} {A : Set a} → A → A)
    → Bool)
```

Another advantage is that the specification of the foreign entity is a normal Agda record. This lets the Agda interactive editing mode give helpful hints about the expected information in a convenient fashion.

4.5 Related work

The dependently typed language Idris has gained a new FFI recently as well [6]. Idris has chosen a slightly different approach, in that they use a universe encoding in Idris itself to represent the type of a foreign call, whereas we just take normal Agda types as input. However, this is only a minor difference in implementation strategy. Our choice of avoiding a universe encoding in Agda itself is motivated by a restriction in Agda's implementation of universe polymorphism. It is currently impossible to represent universe-polymorphic types inside Agda itself using a universe encoding. We will encounter this Agda restriction again in the next chapter and discuss it in more detail in section 5.4.2.

The main difference to the Haskell FFI [22] is that our foreign calls are expressions instead of declarations, which makes them more composable. Additionally, we represent the naming information required by the FFI as a normal Agda record, thereby reducing the amount of special syntax we have to introduce.

4.6 Future Work

We have only introduced a new FFI for function calls in this chapter, but the Agda FFI also contains pragmas to have a common runtime representation for Agda and Haskell data types. Some of the concepts shown in this chapter could also be used to improve the other parts of the Agda FFI.

An issue more specific to the UHC backend FFI lies in how the names of Haskell functions need to be specified. Haskell modules often use re-exports to hide the internal implementation details of a package. However, these re-exports are removed when UHC compiles Haskell code to UHC Core and replaces all uses of re-exported names with their original names. The consequence of this is that the Haskell FFI of the UHC backend requires that the Haskell names are the original names; it is not possible to use re-exported names. While it is not difficult to resolve the names manually, it can be inconvenient and it would be preferable if re-exported names could be used in the FFI of the UHC backend.

Lastly, linking Agda code with Haskell code which is not part of the Haskell base libraries currently requires some manual interaction during the compilation when the UHC backend is used. Contributing to this problem are the lack of a package system for Agda, and the outdated nature of the Cabal UHC support.

Chapter 5

Contracts

In the previous chapter we have demonstrated a Foreign Function Interface which allows us to call Haskell code. This is very useful by itself, but a recurring problem with Foreign Function calls is that the type systems on both sides are not an exact match. Agda's type system is more powerful than Haskell's, which results in an impedance mismatch. In this chapter, we discuss how to handle this impedance mismatch in a graceful and controlled fashion.

5.1 Data Contracts

5.1.1 The problem

Agda programs often use richer types than Haskell programs, as it enables us to prove the correctness of programs. The enriched Agda types are often similar to the Haskell or C representation, with additional information or guarantees added.

Let us start with a simple example. Most languages have special builtin numeric types with associated primitive operations. Not so in Agda, where for example natural numbers are normally represented as Peano numbers:

```
data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ
```

This definition is well-suited for conducting inductive reasoning and programming in Agda itself. However, if we want to call code written in other languages such as Haskell or C, the Peano number representation from Agda needs to be converted to the binary numerical types used by this languages. For example, natural numbers are usually represented as arbitrary-precision integers in Haskell; thus we need to convert between Agda's Peano numbers and Haskell's integers.

However, this conversion is not fool-proof, as Haskell's integers have a bigger range than Agda's Peano numbers. If the Haskell integer representation has a negative value, there is no sensible way to convert this into an Agda Peano number. In other languages this problem could easily be solved by emitting a runtime failure if necessary, but Agda does not have any such facility readily available; it forces *all* functions to be total. A reasonable solution could hence be to lift the conversion functions into an error monad such as Maybe. However, doing so is contagious and forces us to lift many calls using the foreign function interface (FFI) into the error monad. For example, if we want to expose Haskell's integer addition as an addition function on natural numbers in Agda, the resulting function is forced into the chosen error monad:

```
hsAdd = foreign (hsCall "UHC.Agda.Builtins.primAdd") (ℤ → ℤ → ℤ)

nat2integer : ℕ → ℤ
nat2integer = <<omitted>>

integer2nat : ℤ → Maybe ℕ
integer2nat = <<omitted>>

_+_ : ℕ → ℕ → Maybe ℕ
x + y = integer2nat (hsAdd (nat2integer x) (nat2integer y))
```

While this is correct and safe code, it tends to force the entire program to live in the chosen error monad. This is often inconvenient and makes code more verbose, even though it is impossible that the _+_ function should ever return nothing.

5.1.2 Unsafe Conversions

In light of the fact that FFI calls themselves are highly unsafe and dangerous territory, we propose to approach the problem by using dynamic checks at runtime instead of lifting all computations into an error monad. The intention is to make checking the invariants of FFI calls feasible and accessible, but it should in no way replace proper error handling. Instead, it should be used to verify invariants where proper error handling would be prohibitively expensive. In other words, it provides a middle ground between fully handling all possible errors, and just blindly assuming that invariants are never violated.

To this end, we first need to model the data conversions in Agda itself. A conversion is a decision function, which decides if the given conversion succeeds. This can be encoded simply by returning a Maybe:

```
Conversion : Set → Set → Set
Conversion FROM TO = FROM → Maybe TO
```

The trickier part is in actually performing the conversion and inserting the dynamic checks as necessary.

As we want to hide the potential runtime failure from the type signature, we are required to introduce a new ConversionFailure axiom. Importantly, this axiom makes Agda potentially unsound! However, we use this axiom as an implementation detail only and do not expose it to the outside world; it is just used to hide the potential failure. Using this axiom, we can define the ↯ function executing the conversion:

```
postulate
  ConversionFailure : {A : Set} → A

↯ : {A B : Set} → Conversion A B → A → B
↯ co x with co x
↯ co x | just x₁ = x₁
↯ co x | nothing = ConversionFailure
```

Using the ConversionFailure axiom in this restricted way ensures that we make Agda programs unsound only in a strictly controlled fashion. Any usage of the axiom will be surrounded by the necessary dynamic checks, thus making sure that introducing *and* evaluating an unsound conversion will lead to a runtime failure.

Example Revisiting the addition example discussed earlier, we can now rewrite the addition function. The newly introduced ↯ function enables us to remove the Maybe monad from the type signature, resulting in the following code:

```
_+_ : ℕ → ℕ → ℕ
x + y = ↯ int2nat (hsAdd (nat2int x) (nat2int y))
```

5.2 Type-Indexed contracts

5.2.1 Why type-indexed

While the conversion primitive introduced in the previous section makes it possible to convert data as necessary, doing so for non-trivial functions yields a significant amount of boilerplate code and obfuscates the intention of the code.

Assume we want to expose a map-like Haskell function in Agda. For simplicity, we assume that we map over a list of integers and that we want to strengthen/weaken the domain/range of the mapping function to natural numbers. Doing so using the primitive conversion operator results in the following code, assuming we are given the two conversions ℤ⇒ℕ and ℕ⇒ℤ:

```
hsMap = foreign (hsCall "The.Haskell.map")
  ((ℤ → ℤ) → List ℤ → List ℤ)

map : (ℕ → ℕ) → List ℤ → List ℤ
map f xs = hsMap (λ x → ↯ ℕ⇒ℤ (f (↯ ℤ⇒ℕ x))) xs
```

This verbose syntax makes it hard to understand what the exact conversions are and what the assumed invariants of the whole function are.

More concise and expressive code can be achieved by using a small Domain Specific Language (DSL) to create a description of the necessary conversions. Note that in the above example, all conversions can be related to certain positions in the type signature. This relation can be exploited by giving the DSL the same structure as the type signature. This approach has been successfully applied to implementing contracts in other languages such as Haskell [15] or Scheme [13].

For our earlier map example, we aim to write a contract in approximately the following fashion:

```
map′ = assert ((⟨ ℕ⇔ℤ ⟩ ⇒ ⟨ ℕ⇔ℤ ⟩) ⇒ List ℤ ⇒ List ℤ) hsAdd
```

This contract lifts the Haskell map on integer lists function to Agda's natural numbers. Compared to the explicit, manual insertion of conversions shown earlier, our proposed syntax requires a lot less boilerplate code and expresses the expected invariants in a much clearer fashion.

In the remainder of this chapter, we will show how we can implement such a solution in Agda.

5.2.2 Non-dependent Isomorphisms

A crucial observation to make about the examples used so far is that we often want to relate a specific Agda data type with a Haskell or C data type, or to be more precise the Agda representation of it. This relation forms an isomorphism, consisting of the conversion from the Agda to the Haskell/C data type and the conversion in the opposite direction.

There is a catch though, having a proper isomorphism would require the conversions to be total functions. As we explicitly want to allow conversions to fail, we have a restricted form of an isomorphism which only holds under certain additional conditions. We will call such an isomorphism with additional conditions a partial isomorphism. Note that we do *not* require the isomorphism to be partial. If one or both of the conversion functions are total, the partial isomorphism may be in reality a retraction or proper isomorphism.

It is also possible to use isomorphisms between arbitrary Agda data types and is in no way restricted to the FFI. To honor this, we will call the Agda type we want to expose the high type, and the Haskell/C type we will call the low type.

Having discussed the motivation for using partial isomorphisms, we require a way to encode this concept in Agda. This can be easily done using records. The record contains the low and high types, and the conversion functions to go from one representation to the other:

```
record PartIso : Set where
  field
    LOW : Set
    HIGH : Set
    up : Conversion LOW HIGH
    down : Conversion HIGH LOW
```

An example definition for the isomorphism between natural numbers and integers could then look like this:

```
ℕ⇔ℤ : PartIso
ℕ⇔ℤ = record { LOW = ℤ ; HIGH = ℕ ; up = ℤ⇒ℕ ; down = ℕ⇒ℤ }
  where
    ℕ⇒ℤ : ℕ → Maybe ℤ
    ℕ⇒ℤ n = just (+ n)
    ℤ⇒ℕ : ℤ → Maybe ℕ
    ℤ⇒ℕ −[1+ n ] = nothing
    ℤ⇒ℕ (+ n) = just n
```

Finally, given such an isomorphism we can define two functions to extract the necessary information from the record and apply the conversion. This is implemented using the conversion primitive introduced in section 5.1.2:

```
goDown : (p : PartIso) → HIGH p → LOW p
goDown p x = ⚡ (down p) x

goUp : (p : PartIso) → LOW p → HIGH p
goUp p x = ⚡ (up p) x
```

5.2.3 Contract language

Building upon the Agda encoding of partial isomorphisms discussed in the previous section, we can construct a contract language to describe the conversions we want to apply to a function. We will use the term *contract* to denote such a specification. We will start by defining a formal model for our contracts, and will explain our Agda implementation in more detail in section 5.4.

The full syntax of the expression, type and contract language we are going to use can be seen in figure 5.1. Our contract language C uses a syntax closely mirroring the syntax of normal Agda types. The only extension is the "$\langle I \rangle$" construct, indicating that the partial isomorphism I should be used at the denoted location. Note that we allow mixing contracts with normal Agda types, as it may very well be that some arguments need not be converted.

An isomorphism is defined by the four projections giving the low and high type, and the up and down conversions. In the actual Agda code, the conversion functions are normal Agda functions; for the formal semantics we will ignore how the conversion functions are constructed and will just assume they are given.

As our main intention is to create a contract language suitable for the foreign interface, our contract language is only able to represent a subset of all Agda types. More precisely, we do not admit lambdas to our contract or type language, as neither Haskell nor C support types containing lambdas. However, note that this restrictions applies to the *normalized* type only, fully-applied lambda abstractions pose no problem as they disappear during type/contract normalization. We also restrict ourselves to non-dependent contracts/types for now, but we will later add those to the contract language in section 5.3.

Examples The newly introduced contract language already allows us to annotate functions and FFI calls with contracts. Our examples in this section will all use the partial isomorphism between natural numbers and integers. The formal specification of the partial isomorphism $I_{\mathbb{N}\mathbb{Z}}$ can be seen in equation 5.1. As the specific definition of the conversions functions $\mathbb{N}{\Rightarrow}\mathbb{Z}$ and $\mathbb{Z}{\Rightarrow}\mathbb{N}$ is not relevant for the contract language, we will omit their definition:

$$I_{\mathbb{N}\mathbb{Z}} = \{\tau_{low} = \mathbb{Z}; \tau_{high} = \mathbb{N}; \alpha_{down} = \mathbb{N}{\Rightarrow}\mathbb{Z}; \alpha_{up} = \mathbb{Z}{\Rightarrow}\mathbb{N}\} \tag{5.1}$$

Taking advantage of this isomorphism, we can express the before mentioned contract lifting an addition function from integers to natural numbers. This gives us the following expression, wrapping the hsAdd FFI call:

$$add = \text{assert}\ (\langle\ I_{\mathbb{N}\mathbb{Z}}\ \rangle \Rightarrow \langle\ I_{\mathbb{N}\mathbb{Z}}\ \rangle \Rightarrow \langle\ I_{\mathbb{N}\mathbb{Z}}\ \rangle)\ hsAdd \tag{5.2}$$

Contracts	C	$::=$	$\langle\ I\ \rangle$	Apply Isomorphism
		$\mid$	σ_C	Type
Types	τ	$=$	σ_τ	Plain Agda Type
Type-Like	σ_r	$::=$	Set	Set
		$\mid$	$n\ \vec{\tau}$	Variable / Application
		$\mid$	$r_1 \Rightarrow r_2$	Function Type
Isomorphism	I	$=$	$\{\tau_{low} = \tau;\ \tau_{high} = \tau;\ \alpha_{down} = \alpha;\ \alpha_{up} = \alpha\}$	
Conversion	α			
Expressions	e	$::=$	Let $n\ =\ \sigma_e$ in σ_e	Let binding
		$\mid$	$\lambda n \rightarrow e$	Lambda Abstraction
		$\mid$	$e\ :\ \tau$	Type annotation
		$\mid$	$\lightning\ \ \alpha\ e$	(Unsafe) conversion primitive
		$\mid$	assert $C\ e$	Assertion
		$\mid$	$I.\tau_{low}\ \mid\ I.\tau_{high}\ \mid\ I.\alpha_{down}\ \mid\ I.\alpha_{up}$	Isomorphism Projection

Figure 5.1: The contract, type and expression language for non-dependent contracts.

Our contract language also allows us to specify contracts on higher order functions. To demonstrate this, we will reuse the map-like Haskell function introduced in section 5.2.1 with the following Agda representation:

```
hsMap = foreign (hsCall "The.Haskell.map")
  ((ℤ → ℤ) → List ℤ → List ℤ)
```

Our intent is to lift the first argument from the type ($\mathbb{Z} \rightarrow \mathbb{Z}$) to the type ($\mathbb{N} \rightarrow \mathbb{N}$). We hence have to apply the $I_{\mathbb{N}\mathbb{Z}}$ partial isomorphism on the function argument, yielding the following expression:

$$map = \text{assert}\ ((\langle\ I_{\mathbb{N}\mathbb{Z}}\ \rangle \Rightarrow \langle\ I_{\mathbb{N}\mathbb{Z}}\ \rangle) \Rightarrow\ List\ \mathbb{Z} \Rightarrow List\ \mathbb{Z})\ hsMap \tag{5.3}$$

Note that the above expression mixes contracts with normal Agda types. We specify that a contract needs to be applied to the argument/result of the function passed as the first argument to *map* ; for the list argument/result we did not specify any contracts and just use the normal Agda data types directly.

5.2.4 Contract desugaring

So far we have shown how a type-indexed contract language can be defined, but we have not yet discussed it's precise semantics nor how to implement it. We can accomplish both goals by formally specifying how a contract C desugars into normal Agda code.

In the end, the goal is to replace all terms of the form assert $C\ e$ by a plain Agda expression performing the necessary checks and conversions.

A slight complication arises from the fact that we intend to use contracts mainly together with the FFI. Agda's FFI works in a type-directed fashion, and requires that the type of a FFI call is explicitly given. It would thus be nice if we could reuse the

$$\frac{}{\rho; Set \blacktriangleright Set}\text{ T-Set} \qquad \frac{}{\rho; n\ \vec{\tau} \blacktriangleright n\ \vec{\tau}}\text{ T-Var}$$

$$\frac{\rho; C_1 \blacktriangleright \tau_1 \qquad \rho; C_2 \blacktriangleright \tau_2}{\rho; C_1 \Rightarrow C_2 \blacktriangleright \tau_1 \Rightarrow \tau_2}\text{ T-Pi}$$

$$\frac{}{L; \langle I \rangle \blacktriangleright I.\tau_{low}}\text{ T-Iso-Low}$$

$$\frac{}{H; \langle I \rangle \blacktriangleright I.\tau_{high}}\text{ T-Iso-High}$$

Figure 5.2: Derivation rules for the high and low types.

contract specification C to derive the type of an FFI call; thereby exposing one syntax for both, defining the contracts for an FFI call, and the FFI call itself.

In the formal model, we take this into account by inserting type annotations for the scrutinee of an assertion. A nice side effect of this approach is that it enables us to also annotate the whole generated term with the expected type. This allows the type checker of Agda to check that the generated expression matches the expected type.

In the formal description of the translation which follows, we will refer to the expression targeted by an assertion as the scrutinee. Following the terminology introduced earlier, we will call the type of the scrutinee the *low* type, while calling the type of the whole assertion term the *high* type.

Type derivation

We will start explaining the full translation by discussing how the high/low types are derived. We will use the syntax $L; C \blacktriangleright \tau_{low}$ to denote the translation of the contract C to the low type τ_{low}. Note that the leading L value signals that we want to derive the low type. Replacing L with H will represent the derivation of the high type correspondingly. The full rules can be seen in figure 5.2.

The rules for all Type-Like σ constructs are trivial; these constructs are already valid parts of Agda types and no conversion is thus needed. The only case where special care needs to be taken is the isomorphism case. Depending on if the high or low type should be derived, the proper type needs to be selected.

Example For the addition example given in equation 5.2, the derived low/high types are as follows:

$$add - \tau_{low} = \mathbb{Z} \Rightarrow \mathbb{Z} \Rightarrow \mathbb{Z} \tag{5.4}$$
$$add - \tau_{high} = \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N} \tag{5.5}$$

Similarly, we can derive the following low/high types for the map-like function from equation 5.3:

$$map - \tau_{low} = (\mathbb{Z} \Rightarrow \mathbb{Z}) \Rightarrow List\ \mathbb{Z} \Rightarrow List\ \mathbb{Z} \tag{5.6}$$
$$map - \tau_{high} = (\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow List\ \mathbb{Z} \Rightarrow List\ \mathbb{Z} \tag{5.7}$$

Inserting dynamic checks

The final missing piece is the code actually translating the assertion terms, inserting the necessary checks and conversions. To be more precise, the translation function needs to construct a term of type τ_{high}, being given the contract C and an expression to wrap of type τ_{low}.

Expressions language The target expression language e is given in figure 5.1. It features a basic lambda calculus, extended with an unsafe conversion primitive $\lightning$ which is used to apply a conversion. In the actual Agda implementation, the conversion primitive is a normal Agda function, but for clarity we choose an explicit representation in our formal expression language. Furthermore, we introduce the syntax $\text{assert}\ C\ e$ to apply the contract C to the expression e.

Translation rules The aim of the translation rules now is to replace all assertions by a suitable new expression using the unsafe conversion primitive instead.

The complete set of rules can be found in figure 5.3. Most of the rules are fairly straightforward and we will focus our description on the complicated cases, starting from the top-level.

Top-level translation The translation of assertion starts at the $\text{assert}\ C\ e$ expressions. Computing the low/high types for the generated expression builds upon the already introduced rules for the type derivation. The more interesting part is the generation of the actual new expression to splice in.

Most translation rules require additional information about their context. First, for applying partial isomorphisms correctly it is important to know if they are used in a co- or contravariant context. We encode the contravariant context as N (negative) and the covariant context as P (positive). Second, we need to know the expression which shall be wrapped. We will use the syntax $\omega; \phi; C \triangleright e$ to describe how to derive the conversion term e, given the contract C, the co- or contravariant context ω, and the term to wrap ϕ.

Using this preliminaries, we can introduce the assertion rule:

$$\frac{L; C \blacktriangleright \tau_{low} \qquad H; C \blacktriangleright \tau_{high} \qquad P; (e : \tau_{low}); C \triangleright e'}{\text{assert}\ C\ e \ \triangleright (e' : \tau_{high})}\ \text{C-Assert}$$

A top-level contract is always covariant, we thus mark it as P (positive) and set ϕ to the expression containing the assertion scrutinee. We also add two type annotations using the derived high/low types.

Trivial cases A contract may contain plain Agda types. By their definition, these contract terms do not establish any additional property or conversion of the given argument, hence we can just return the original assertion scrutinee given as ϕ.

Isomorphisms If an isomorphism is encountered, an upward or downward conversion of the ϕ term needs to be inserted. The position dictates the direction of the conversion:

$$\frac{}{P; \phi; \langle I \rangle \triangleright \lightning\ \ I.\alpha_{up}\ \phi}\ \text{C-Iso-Pos}$$

$$\frac{}{N; \phi; \langle I \rangle \triangleright \lightning\ \ I.\alpha_{down}\ \phi}\ \text{C-Iso-Neg}$$

Function Type Finally, the missing piece and most complex rule handles the function type/contract:

$$\frac{\text{let } x, x' \text{ be fresh} \qquad \omega^{-1}; x'; C_1 \rhd e_1 \qquad \omega; app\ (\phi, x); C_2 \rhd e_2}{\omega; \phi; C_1 \Rightarrow C_2 \rhd \lambda x' \rightarrow \text{Let } x = e_1 \text{ in } e_2} \text{ C-Fun}$$

Notably, we have to introduce a new lambda abstraction to be able to inspect the newly given argument. To derive the term for converting x' we walk over the C_1 contract, giving x' as the term to transform. We also invert the context ω using the syntax ω^{-1}. This results in the term e_1, which performs the necessary checks on x' and which we bind in a Let expression to the fresh name x.

For the right hand side of the function contract, the contract C_2 needs to be inspected. Importantly, we apply the original wrapped term ϕ to the transformed term x. This gives us the body e_2, which we then insert into the Let binding.

Examples Once again, we will first give the solution for the addition example. The whole translation including the type signatures gives the following result:

$$\begin{aligned} add = &\lambda x' \rightarrow \text{Let } x = \lightning\ \mathbb{N}{\Rightarrow}\mathbb{Z}\ x' \text{ in} \\ &\quad \lambda y' \rightarrow \text{Let } y = \lightning\ \mathbb{N}{\Rightarrow}\mathbb{Z}\ y' \text{ in} \\ &\qquad \lightning\ \mathbb{Z}{\Rightarrow}\mathbb{N} \\ &\qquad\quad ((hsAdd : \mathbb{Z} \Rightarrow \mathbb{Z} \Rightarrow \mathbb{Z})\ x\ y) \\ &: \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N} \end{aligned} \tag{5.8}$$

More interestingly, we can do the same transformation for the higher-order map example:

$$\begin{aligned} map = &\lambda f' \rightarrow \\ &\quad \text{Let } f = (\lambda x' \rightarrow \\ &\qquad \text{Let } x = \lightning\ \mathbb{N}{\Rightarrow}\mathbb{Z}\ x' \text{ in} \\ &\qquad\quad \lightning\ \mathbb{Z}{\Rightarrow}\mathbb{N}\ (f'\ x) \\ &\qquad) \text{ in} \\ &\qquad\quad \lambda xs' \rightarrow \text{Let } xs = xs' \text{ in} \\ &\qquad\qquad (hsMap : (\mathbb{Z} \Rightarrow \mathbb{Z}) \Rightarrow List\ \mathbb{Z} \Rightarrow List\ \mathbb{Z})\ f\ xs \\ &\quad) \\ &: (\mathbb{N} \Rightarrow \mathbb{N}) \Rightarrow List\ \mathbb{Z} \Rightarrow List\ \mathbb{Z} \end{aligned} \tag{5.9}$$

Importantly, the lambda $\lambda x' \rightarrow ...$ is introduced in the body of the let-binding for f. This lambda allows us to inspect the argument passed to the function f', applying the appropriate contract to x' as well as to the result of f' being applied to x.

5.3 Dependent Contracts

The mechanism discussed so far already gives us first-order and higher-order contracts, but one crucial bit is still missing. Contracts are so far not allowed to depend on other

$$app(e_1, \vec{e_2}) = \mathit{Let}\ x = e_1\ \mathit{in}\ x\ \vec{e_2}, \text{where x is fresh}$$

$$\frac{}{\omega; \phi; \mathsf{Set} \rhd \phi}\ \text{C-Set} \qquad \frac{}{\omega; \phi; n\ \vec{\tau} \rhd \phi}\ \text{C-Var}$$

$$\frac{\text{let } x, x' \text{ be fresh} \qquad \omega^{-1}; x'; C_1 \rhd e_1 \qquad \omega; app\ (\phi, x); C_2 \rhd e_2}{\omega; \phi; C_1 \Rightarrow C_2 \rhd \lambda x' \to \text{Let } x = e_1 \text{ in } e_2}\ \text{C-Fun}$$

$$\frac{}{P; \phi; \langle I \rangle \rhd ↯\ I.\alpha_{up}\ \phi}\ \text{C-Iso-Pos}$$

$$\frac{}{N; \phi; \langle I \rangle \rhd ↯\ I.\alpha_{down}\ \phi}\ \text{C-Iso-Neg}$$

$$\frac{L; C \blacktriangleright \tau_{low} \qquad H; C \blacktriangleright \tau_{high} \qquad P; (e : \tau_{low}); C \rhd e'}{\mathsf{assert}\ C\ e\ \rhd (e' : \tau_{high})}\ \text{C-Assert}$$

Figure 5.3: Non-dependent contracts translation.

arguments; they are not *dependent* contracts. In the setting of Agda, this is especially problematic as type polymorphism is done using type arguments. To give an example, the normal Haskell map function for lists can be imported into Agda with the following Agda code:

```
hsMap = foreign (hsCall "Data.List.map")
  ((A B : Set) → (f : A → B) → List A → List B)
```

The types A and B are passed as arguments, and the remainder of the type depends on these values.

A further complication is that the Agda representation of a function may take additional arguments compared to the Haskell version. For example, we may want to expose Haskell Lists as Agda Vectors of a certain length. We would like to be able to write a dependent contract stating that mapping a list does not change its length. Assuming we are given a partial isomorphism vec⇔list between vectors and lists, the syntax for specifying such a contract could look like the following pseudo-code:

```
map = assert
  ((n : ℕ) ⇒ (A B : Set) ⇒ (f : A ⇒ B) ⇒ ⟨ vec⇔list A n ⟩ ⇒ ⟨ vec⇔list B n ⟩)
  hsMap
```

This example shows that the lifted / dependently-typed version of a function may take more arguments than its low-level counterpart; in the map example, the n argument is only used by the high-level definition and *not* by the original low function.

5.3.1 Extended Syntax

To properly support dependent contracts, we need to embellish our contract and type language; making it powerful enough to encode dependent types. To this end, the grammar itself does no longer distinguish between expressions and types as can be seen in

Figure 5.4.

We also let the partial isomorphism projections be lambdas taking arguments, and the partial isomorphism application now takes the corresponding argument expressions. Note that we split the arguments of a partial isomorphism application into three parts. All fields of an isomorphism may depend on the x_a argument. The high type, on the other hand, is not allowed to depend on the x_l argument, and the low type must not depend on the x_h argument. This separation is necessary to support the erasing Pi construct, as it causes arguments to only be in scope in certain contexts.

For example, a partial isomorphism I_{lv} between lists and length-indexed vectors might look like this given the two conversions $vec \Rightarrow list$ and $list \Rightarrow vec$, where the $list \Rightarrow vec$ conversion takes the expected length n as argument. Note that n is a high argument, as it *must* not be used in the derived low type later on:

$$
\begin{aligned}
I_{lv} = \{ & \tau_{low} = \lambda A \; \_ \to List \; A \\
; & \tau_{high} = \lambda A \; n \to Vec \; A \; n \\
; & \alpha_{down} = \lambda A \; \_ \; n \to vec{\Rightarrow}list \\
; & \alpha_{up} = \lambda A \; \_ \; n \to list{\Rightarrow}vec \; n \\
\} &
\end{aligned}
\tag{5.10}
$$

Furthermore, we introduce the dependent function type Pi, subsuming the existing non-dependent function type. We also add a special Pi type to annotate functions where the argument shall be erased. We will call this an erasing Pi.

As a small example, let us show how we could apply the I_{lv} partial isomorphism defined earlier to expose the Haskell List map function as Vector map function in Agda. This also makes use of the erasure annotations, as we want to erase the length argument n before calling the Haskell function. We also pass the unit value as the low argument x_l, as the isomorphism I_{lv} does not take any low arguments:

$$
\begin{aligned}
map = \; & \mathrm{assert} \; ((A : Set) \Rightarrow (B : Set) \Rightarrow (n : \mathbb{N}) \\
& \not\Rightarrow (f : A \Rightarrow B) \Rightarrow \langle \; I_{lv} \; A \; () \; n \; \rangle \Rightarrow \langle \; I_{lv} \; B \; () \; n \; \rangle) \; hsMap
\end{aligned}
\tag{5.11}
$$

5.3.2 Embellished Translation Rules

Given the extended syntax defined in the previous section, the translation rules need embellishment as well. The updated rules for the type derivation are given in figure 5.5. Note that even though we have merged the expression and type grammar, we only allow the subset for which we have given type derivation rules to be used in contracts.

Furthermore, we need to extend the type derivation rules to parametrize them over whether we derive the low or high type. We use the syntax $\omega; \rho; A \blacktriangleright B$ to denote the translation of the term A to the term B, in the covariant (P) or contravariant (N) context ω, where ρ takes the value L/H for deriving the low/high type. This allows us to introduce the two new rules handling the erasing Pi construct.

Erasing Pi

For first-order contracts this is relatively easy; for producing the high-type, the T-Pi rule applies without any change. On the other hand, for the low type we just need to ignore the left hand side of the erasing Pi:

Contracts	C	$::=$	$\langle\ I\ e_a\ e_l\ e_h \rangle$	Apply Isomorphism
		$\mid$	$(n\ :\ C_1) \nRightarrow C_2$	Erasing Pi
		$\mid$	e_C	Type
Isomorphism	I	$=$	$\{\tau_{low} = \lambda x_a\ x_l \rightarrow e$	
			$;\tau_{high} = \lambda x_a\ x_h \rightarrow e$	
			$;\alpha_{down} = \lambda x_a\ x_l\ x_h \rightarrow \alpha$	
			$;\alpha_{up} = \lambda x_a\ x_l\ x_h \rightarrow \alpha\}$	
Conversion	α			
Expressions	τ_r, e_r	$::=$	$(n\ :\ r) \Rightarrow r$	Pi
		$\mid$	Set	Set
		$\mid$	$n\ \vec{e_r}$	Variable / Application
		$\mid$	$\text{Let } n\ =\ e_r \text{ in } e_r$	Let binding
		$\mid$	$\lambda n \rightarrow e_r$	Lambda Abstraction
		$\mid$	$e_r\ :\ \tau$	Type annotation
		$\mid$	$\lightning\ \alpha\ e_r$	(Unsafe) conversion primitive
		$\mid$	$\text{assert } C\ e_r$	Assertion
		$\mid$	$()$	Unit
		$\mid$	$e_r, e_r \mid proj_1\ e_r \mid proj_2\ e_r$	(Dependent) pairs
		$\mid$	$I.\tau_{low} \mid I.\tau_{high} \mid I.\alpha_{down} \mid I.\alpha_{up}$	Isomorphism Projection
	e, τ	$=$	e_e, τ_τ	

Figure 5.4: The terms of the dependent contracts.

$$\frac{\omega; L; C_2 \blacktriangleright \tau_2}{\omega; L; C_1 \not\Rrightarrow C_2 \blacktriangleright \tau_2}\ \text{T-Erase-Pi1'}$$

Beware though, this rule breaks down for higher-order contracts. Assume we are given the following contract, with the corresponding low/high type computed using rule T-Erase-Pi1' :

$$\begin{aligned} C &= (A : Set) \Rightarrow (A \not\Rightarrow A) \Rightarrow A \\ \tau_{low} &= (A : Set) \Rightarrow (f_{low} : A) \Rightarrow A \\ \tau_{high} &= (A : Set) \Rightarrow (f_{high} : A \Rightarrow A) \Rightarrow A \end{aligned}$$

According to the types, if we want to call the low function we have to produce an expression f_{low} of type A while being given the expression f_{high} of type $A \Rightarrow A$. The only way to do this is to apply f_{high} to an expression of type A, but we do not have any such value.

The crucial observation to make here is that the erasing Pi is in *contravariant* position in the given example. The proper solution is thus to reverse the effect of the erasing Pi in a contravariant context; instead of erasing arguments in the low type, we do so in the high type.

Coming back to the example, this means that the types of f_{low} and f_{high} are switched around:

$$\begin{aligned} C &= (A : Set) \Rightarrow (A \not\Rightarrow A) \Rightarrow A \\ \tau_{low} &= (A : Set) \Rightarrow (f_{low} : A \Rightarrow A) \Rightarrow A \\ \tau_{high} &= (A : Set) \Rightarrow (f_{high} : A) \Rightarrow A \end{aligned}$$

Accordingly, we now need to produce an expression f_{low} of type $A \Rightarrow A$ being given the expression f_{high} of type A. This is now easily possible by wrapping f_{high} in an additional lambda abstraction: $\lambda\_ \rightarrow f_{high}$.

We can generalize this observation and the rules for the type derivation we thereby obtain erase an argument in the low or high type depending on the co-/contravariant context ω and the low (L) / high (H) target type ρ:

$$\frac{(\omega, \rho) \in \{(P, L), (N, H)\} \qquad \omega; \rho; C_2 \blacktriangleright \tau_2}{\omega; \rho; C_1 \not\Rrightarrow C_2 \blacktriangleright \tau_2}\ \text{T-Erase-Pi1}$$

$$\frac{(\omega, \rho) \in \{(P, H), (N, L)\} \qquad \omega^{-1}; \rho; C_1 \blacktriangleright \tau_1 \qquad \omega; \rho; C_2 \blacktriangleright \tau_2}{\omega; \rho; C_1 \not\Rrightarrow C_2 \blacktriangleright \tau_1 \Rightarrow \tau_2}\ \text{T-Erase-Pi2}$$

The rule for checking the contract established by a erasing Pi closely follows the rule for normal Pis, the only difference is that the wrapped term ϕ is *not* applied to the argument x:

$$\frac{\text{let } x' \text{ be fresh} \qquad \omega^{-1}; x'; C_1 \triangleright t_1 \qquad \omega; \phi; C_2 \triangleright t_2}{\omega; \phi; (x : C_1) \not\Rrightarrow C_2 \triangleright \lambda x' \rightarrow \text{Let } x = t_1 \text{ in } t_2}\ \text{C-Erase-Pi}$$

Dependent Isomorphisms

The rules T-Iso-Low/High need to be adapted to apply the arguments of the now *dependent* partial isomorphism correctly. Importantly, the T-Iso-Low rule must apply only

$$\frac{}{\omega; \rho; Set \blacktriangleright Set}\ \text{T-Set}$$

$$\frac{}{\omega; \rho; n\ \vec{\tau} \blacktriangleright n\ \vec{\tau}}\ \text{T-Var} \qquad \frac{\omega^{-1}; \rho; C_1 \blacktriangleright \tau_1 \qquad \omega; \rho; C_2 \blacktriangleright \tau_2}{\omega; \rho; C_1 \Rightarrow C_2 \blacktriangleright \tau_1 \Rightarrow \tau_2}\ \text{T-Pi}$$

$$\frac{}{\omega; L; \langle I\ e_a\ e_l\ e_h \rangle \blacktriangleright I.\tau_{low}\ e_a\ e_l}\ \text{T-Iso-Low}$$

$$\frac{}{\omega; H; \langle I\ e_a\ e_l\ e_h \rangle \blacktriangleright I.\tau_{high}\ e_a\ e_h}\ \text{T-Iso-High}$$

$$\frac{(\omega, \rho) \in \{(P, L), (N, H)\} \qquad \omega; \rho; C_2 \blacktriangleright \tau_2}{\omega; \rho; C_1 \not\Rightarrow C_2 \blacktriangleright \tau_2}\ \text{T-Erase-Pi1}$$

$$\frac{(\omega, \rho) \in \{(P, H), (N, L)\} \qquad \omega^{-1}; \rho; C_1 \blacktriangleright \tau_1 \qquad \omega; \rho; C_2 \blacktriangleright \tau_2}{\omega; \rho; C_1 \not\Rightarrow C_2 \blacktriangleright \tau_1 \Rightarrow \tau_2}\ \text{T-Erase-Pi2}$$

Figure 5.5: Derivation rules for the high and low types for the dependent contracts.

the x_a and x_l arguments, whereas the T-Iso-High rule only the x_a and x_h arguments. Simply applying all arguments in either rule would lead to invalid terms, as an erasing Pi may have dropped the respective argument in the current context.

It is also worth noting that even though our model only permits exactly one argument per argument kind, we can easily encode any number of arguments by using pairs or the unit value as arguments.

For the contract checking itself in rule C-Iso-Pos/C-Iso-Neg, all arguments are in scope and we can just apply all arguments and apply the resulting up/down conversions as beforehand.

5.4 An actual Agda Implementation

Having formulated a formal model of our contracts, we can now go ahead and discuss some of the key design points in our Agda implementation.

5.4.1 Dependent Partial Isomorphisms

First, we need to extend our earlier Agda definition of partial isomorphisms to dependent partial isomorphisms. Crucially, it must be relatively easy to define a specific instance of such a dependent partial isomorphism. To aid the programmer to use our contracts properly, we also strive to enforce the correctness of any such partial isomorphism definition using Agda's type system.

A partial isomorphism will always contain a conversion in both the upward and downward direction, we hence give bidirectional conversions an explicit name:

```
-- bi-directional conversions
Conversions : Set → Set → Set
Conversions A B = Conversion A B × Conversion B A
```

$$\frac{}{\omega; \phi; \text{Set} \rhd \phi}\ \text{C-Set}$$

$$\frac{}{\omega; \phi; n\ \vec{\tau} \rhd \phi}\ \text{C-Var}$$

$$\frac{\text{let } x' \text{ be fresh} \qquad \omega^{-1}; x'; C_1 \rhd t_1 \qquad \omega; (apply\ \phi\ x); C_2 \rhd t_2}{\omega; \phi; (x : C_1) \Rightarrow C_2 \rhd \lambda x' \to \text{Let } x = t_1 \text{ in } t_2}\ \text{C-Pi}$$

$$\frac{}{P; \phi; \langle I\ e_a\ e_l\ e_h \rangle \rhd \lightning\ (I.\alpha_{up}\ e_a\ e_l\ e_h)\ \phi}\ \text{C-Iso-Pos}$$

$$\frac{}{N; \phi; \langle I\ e_a\ e_l\ e_h \rangle \rhd \lightning\ (I.\alpha_{down}\ e_a\ e_l\ e_h)\ \phi}\ \text{C-Iso-Neg}$$

$$\frac{\text{let } x' \text{ be fresh} \qquad \omega^{-1}; x'; C_1 \rhd t_1 \qquad \omega; \phi; C_2 \rhd t_2}{\omega; \phi; (x : C_1) \Rightarrow C_2 \rhd \lambda x' \to \text{Let } x = t_1 \text{ in } t_2}\ \text{C-Erase-Pi}$$

$$\frac{L; C \blacktriangleright \tau_{low} \qquad H; C \blacktriangleright \tau_{high} \qquad P; (e : \tau_{low}); C \rhd e'}{\text{assert } C\ e\ \rhd (e' : \tau_{high})}\ \text{C-Assert}$$

Figure 5.6: Dependent contracts translation.

In our formal model, partial isomorphisms take exactly three arguments. As we want to support arbitrary numbers of arguments in our Agda formulation, we might be tempted to encode the argument types as a list of types:

```
ArgTypes : Set
ArgTypes = List Set
```

This however has the big drawback of requiring that all argument types must be independent; the argument types may not depend on each other! Therefore we instead adopt the aforementioned approach using (dependent) pairs and unit values to encode arbitrary number of arguments.

We then go on to define the embellished PartIso record. The arguments follow the split into the three kinds discussed beforehand, and the low/high types and the conversions may depend on these arguments. This yields the following Agda code:

```
record PartIso : Set where
  constructor mkPartIso
  field
    -- the common arguments
    ARG-a : Set
    -- the low arguments
    ARG-l : ARG-a → Set
    -- the high arguments
    ARG-h : ARG-a → Set
    -- the low type
    τ-l : (aa : ARG-a) → (ARG-l aa) → Set
    -- the high type
```

```
τ−h : (aa : ARG−a) → (ARG−h aa) → Set
- the conversions
⇅ : (aa : ARG−a) → (al : ARG−l aa) → (ah : ARG−h aa)
  → Conversions (τ−l aa al) (τ−h aa ah)
```

To see if this indeed works, let us build a dependent partial isomorphism between lists and vectors. We first define a conversion function converting a list into a vector of a certain length:

```
- decision function for converting lists
- to vectors of a certain length
list⇒vec : ∀ {n : ℕ} {A : Set} → List A → Maybe (Vec A n)
list⇒vec {n} xs with n ≟ length xs
list⇒vec xs | yes refl = just (Data.Vec.fromList xs)
list⇒vec xs | no ¬p = nothing
```

The definition of the partial dependent isomorphism given this preliminaries is then straightforward:

```
- the partial dependent isomorphism
vec⇔listI : PartIso
vec⇔listI = record
  - the element type
  { ARG−a = Set
  - dummy unit type
  ; ARG−l = λ _ → ⊤
  - the vector length type
  ; ARG−h = λ _ → ℕ
  - the low list type
  ; τ−l = λ A _ → List A
  - the high vector type
  ; τ−h = λ A n → Vec A n
  - the conversions
  ; ⇅ = λ A _ n → list⇒vec , (just ∘ toList)
  }
```

This demonstrates that our chosen representation allows us to encode dependent partial isomorphisms.

5.4.2 Internal Syntax

The syntax for describing contracts is split into two parts; the internal syntax used to generate the proper code, and the surface syntax exposed to the user.

The internal syntax closely mirrors the contract language *C* introduced in the formal model before hand. An important design aspect is whether this internal language should be strongly-typed inside Agda itself, thereby taking advantage of Agda's type system to guarantee that terms are correct by construction. While certainly a worthwhile goal, this is more challenging than apparent at first as the contract language includes dependent types.

The first main issue is related to the universe hierarchy of Agda. Agda code needs to be explicit in which universe a type belongs to, and this universes are *not* cumulative. As long as the term we want to describe in our contract language are universe-homogeneous, meaning that all Abstract Syntax Tree (AST) nodes live in the same universe, this causes no difficulties. This would be overly restrictive though, as it would severely limit the expressiveness of our language.

For example, the type of the identity function already violates this restriction:

```
id1 : (A : Set0) → A → A
id1 _ x = x
```

The first argument is in the universe Set1, whereas the second argument and the result are in universe Set0. While it is possible to encode such universe-heterogeneous terms in Agda using a clever encoding, this requires some additional effort. [2]

Furthermore, there are still certain contracts terms we can not represent. Reusing the identity function example, we may want to make the identity function universe polymorphic:

```
id2 : (l : Level) → (A : Set l) → A → A
id2 _ _ x = x
```

There is however currently no way known to express an universe-polymorphic data structure in Agda itself. It's unclear if this Agda restriction will be lifted in the future, and what the implications of doing so would be. [3]

Due to this complications with the universe hierarchy in Agda we chose a shallow, untyped embedding for the internal syntax, where the well-formedness of terms is *not* enforced by the Agda type checker. The untyped nature of our internal syntax also implies that we have to use reflection for this purpose if we want to interact with other Agda code.

Our untyped approach is a reasonable strategy, as it simplifies code generation; furthermore, we expect that the public surface syntax will perform the necessary checks to ensure that all contract terms are valid.

5.4.3 Surface Syntax

Creating a suitable surface syntax is deeply related to how we have defined the internal syntax. Nevertheless, the focus here lies on creating a safe and easy-to-use language. The two major design goals are:

1. Concise and clear syntax
2. Only admit well-formed contracts

One option to achieve this is to extend the Agda compiler to support some special syntax for our contracts. This is a rather invasive approach and makes it difficult to evolve our contracts over time. It would also require extending the Agda compiler for this specific use-case, which is not desirable.

Instead, we have chosen to create a sufficiently sophisticated typed AST in Agda itself, taking advantage of Agda's typesystem to guarantee well-formedness. However, this approach leads us to immediately re-encounter the universe hierarchy/polymorphism problems mentioned in the previous section. We resolve this problem by simply disabling the universe check in Agda (`--type-in-type`), as this is the only possibility to be able to express the full power of our contracts in Agda itself. Beware that this makes Agda inconsistent due to Girard's paradox! It is however highly unlikely that one writes this paradox by accident, and disabling the universe check is normally not a problem in practice.

The basic structure of the surface syntax can then be implemented in Agda using induction-recursion.

First, let us introduce a helper function for the isomorphism application construct. Given a partial isomorphism, the function will return the type of a dependent pair which we will use to store the argument values. Note that the type of the low/high argument may depend on the common argument ARG−a:

```
withArgs : PartIso → Set
withArgs i = Σ (ARG−a i) (λ aa → ARG−l i aa × ARG−h i aa)
```

We can then define the contract data type, where the applyIso constructor takes both, an isomorphism and the corresponding arguments:

```
data C : Set
el : C → Set

data C where
  -- a pi contract
  pi : (c : C) → (el c → C) → C
  -- a plain Agda type
  type : Set → C
  -- (partial) isomorphism application
  applyIso : (i : PartIso)
    -- the arguments for the isomorphism i
    → withArgs i
    → C

el (pi c x) = (a : el c) → el (x a)
el (type x) = x
el (applyIso iso args) = ⊤
```

However, some complications arise when we try to encode the erasing Pi construct in the surface syntax. An argument introduced by an erasing Pi construct is by definition only in scope for the low *or* high type, but never for both. We thus need to ensure that arguments introduced by erasing pis are only used in the proper context.

It is worth noting that an erased argument may only be used by a partial isomorphism application, and never in plain Agda types. The reason for this is that partial isomorphism applications are the only terms constructing differing low/high types, where the low type may depend on other arguments than the high type.

To enforce this invariant, we introduce two new data types representing the high and low context. It is crucial that we do *not* publicly export any eliminators of these data types, as we will use them to wrap arguments which may only be used in a high or low context. On the other hand, it is perfectly fine to export functions to lift any value into the High/Low data types, as lifting an argument valid in all contexts into the low or high context is always sound.

The definition of the two data types is as follows:

```
data High (A : Set) : Set where
  wrap : A → High A
data Low (A : Set) : Set where
  wrap : A → Low A
```

We also need to slightly extend the withArgs′ function, as we will need to specify in which context the expected arguments of a partial isomorphism application shall live:

```
withArgs′ : PartIso → Set
withArgs′ i = Σ (ARG−a i) (λ aa → Low (ARG−l i aa) × High (ARG−h i aa))
```

Furthermore, we will need to keep track if we are in a co- or contravariant position to determine into which high/low wrapper erased arguments need to be lifted. First, let us introduce a Context data type and a function to invert the context:

```
data Context : Set where
  Pos : Context -- Covariant
  Neg : Context -- Contravariant

invert : Context → Context
invert Pos = Neg
invert Neg = Pos
```

This lets us define a small helper function, which will wrap an erased argument in the proper data type depending on the context:

```
ω→Wrap : Context → Set → Set
ω→Wrap Pos = Low
ω→Wrap Neg = High
```

The basic contract data type is similar to our previous version. The first difference is that the C′ is now indexed by the Context. Second, we introduce a erasing−pi constructor where the argument is wrapped in the Low/High data types using the γ→Wrap function:

```
data C′ (ω : Context) : Set
el′ : ∀ {ω} → C′ ω → Set

data C′ (ω : Context) where
  pi : (c : C′ (invert ω))
    → (el′ c → C′ ω)
    → C′ ω
  type : Set → C′ ω
  applyIso : (i : PartIso)
    → withArgs′ i
    → C′ ω
  erasing−pi : (c : C′ (invert ω))
    → (ω→Wrap ω (el′ c) → C′ ω)
    → C′ ω
```

Our embellished contract language now also allows us to express contracts with erased arguments, and applied partial isomorphisms may depend on such arguments. The extended el′ function follows the same pattern, we add a clause for the erasing−pi constructor and wrap the argument depending on the context:

```
el′ {ω} (erasing−pi c x) = (a : ω→Wrap ω (el′ c)) → el′ (x a)
el′ (pi c x) = (a : el′ c) → el′ (x a)
el′ (type x) = x
el′ (applyIso i (aa , al , ah)) = ⊤
```

We now finally have a complete surface syntax for our internal syntax, which guarantees that all contracts are correct by construction. Importantly, this solution works without any changes to the Agda compiler and language.

The actual Agda implementation also features some additional syntactic sugar using Agda's Macro and Syntax features, but this are only minor additions and the implementation mirrors closely the solution described in this section.

5.4.4 Fitting everything together

What's left to be done is fitting all the pieces together to present a clean interface to the user. We have already discussed most parts necessary for this; for example, the low/high type and wrapper function derivation follows exactly the formal model. We thus are not going to discuss the Agda implementation of this functions in detail, as the formal model introduced in section 5.3 already fully captures how these functions work. This gives us the following three Agda functions implementing the formal model, where Term is a special Agda data type representing reflected terms and InternalSyn represents our internal syntax:

```
- derive the low/high type from a contract
deriveLowType : InternalSyn → Term
deriveLowType = <<omitted>>

deriveHighType : InternalSyn → Term
deriveHighType = <<omitted>>

- lifts a term from the low type to the high type
contract−apply : InternalSyn - the contract
  → Term - the assertion scrutinee
  → Term - the produced wrapper
contract−apply = <<omitted>>
```

The translation between the surface syntax and the internal syntax uses Agda's reflection mechanism. It takes a reflected term representing the surface syntax as input and produces the internal syntax as output. The translation itself is not very complicated per se, only tedious. As Agda's reflection mechanism is untyped, we cannot rely on Agda's type system to check our translation. At the same time, the reflected terms are often quite large which makes debugging complicated. The translation from the reflected terms to internal syntax itself does not perform any checks, but only converts the reflected terms to our internal syntax. The implementation is hence straightforward and mechanical. It's type signature is as follows:

```
- convert the surface syntax in reflected term form
- to internal syntax
surface⇒internal : Term → InternalSyn
surface⇒internal = <<omitted>>
```

This gives us all the necessary building blocks to add a nice syntax. We take advantage of Agda's macro system for this, which is a shorthand notation for inserting the necessary reflection calls. It requires that the macro function takes arguments of type Term, and returns a result of type Term and will take care of converting these terms to/from actual Agda code.

This allows us to define the assert macro, which we can use to assert a contract on any arbitrary Agda expression:

```
macro
  assert : Term - reflected surface syntax term
    → (lowDef : Term) - assertion scrutinee
    → Term
  assert ast lowDef = result
    where
      open import Function
      - convert surface to internal syntax
      int = surface⇒internal ast
      - annotate the scrutinee with the low type
      low = forceTy′ (deriveLowType int) lowDef
      - apply the contract
      lifted = contract−apply int low
      - annotate the result with the high type
      result = forceTy′ (deriveHighType int) lifted
```

We have finally constructed a complete contract library. In the following section, we will demonstrate how this can be put to good use.

5.4.5 Examples

Having discussed the design of our contracts library, it is time to show some examples using the actual Agda implementation. The syntax used follows closely the syntax introduced in the previous sections, but may differ in some details. All examples shown in this section are directly usable and work as-is.

Starting with the easiest example, the Haskell addition function can be lifted to natural numbers as follows, using the ∅ shorthand to indicate that the partial isomorphism does not take any arguments:

```
hsAdd = foreign (hsCall "Prelude.add")
  (ℤ → ℤ → ℤ)

add : ℕ → ℕ → ℕ
add = assert (makeContract (
  ⟨ _ :: ⟦ ℕ⇔ℤ ⇋ ∅ ⟧ ⟩⇒
  ⟨ _ :: ⟦ ℕ⇔ℤ ⇋ ∅ ⟧ ⟩⇒
  ⟨ ⟦ ℕ⇔ℤ ⇋ ∅ ⟧ ⟩
  )) hsAdd
```

A more interesting example is lifting the map function from lists to vectors, where tt is the unit constructor. This also demonstrates how the erasing Pi feature can be used to discard additional arguments; the *n* argument in this case. :

```
hsMap = foreign (hsCall "Data.List.map")
   ((A B : Set) → (A → B) → List A → List B)

map = assert (makeContract (
   ⟨ n :: ⟦ ℕ ⟧ ⟩⇏
   ⟨ A :: ⟦ Set ⟧ ⟩⇒
   ⟨ B :: ⟦ Set ⟧ ⟩⇒
   ⟨ f :: (⟨ _ :: ⟦ A ⟧ ⟩⇒ ⟨ ⟦ B ⟧ ⟩) ⟩⇒
   ⟨ _ :: ⟦ vec⇔list′ ⇋ A , wrap tt , n ⟧ ⟩⇒
   ⟨  ⟦ vec⇔list′ ⇋ B , wrap tt , n ⟧ ⟩
   )) hsMap
```

Also notice that we have not given the map function an explicit type. This is possible because the assert macro will annotate the generated body with the type derived from the given contract, which allows Agda to easily infer the type of the whole definition.

5.5 Witness Objects

The contracts we introduced so far work very well for translating between different drapes; however, we often want to have separate witness objects in Agda.

List indexing example

A simple example is the list indexing function. Importing this function from Haskell is easy enough:

```
hsIndex = foreign (hsCall "Data.List.!!")
   ((A : Set) → (n : ℕ) → (xs : List A) → A)
```

However, the Haskell list indexing function will throw a runtime error if the index is out of bounds. Using our contracts and the erasure annotation, we can easily create a safer version of the same function which takes a proof object that the index is valid for the given list. Assuming we have a less-than relation $<$, we can write the following contract:

```
index : (A : Set) → (n : ℕ) → (xs : List A) → n < length xs → A
index = assert (makeContract (
   ⟨ A :: ⟦ Set ⟧ ⟩⇒
   ⟨ n :: ⟦ ℕ ⟧ ⟩⇒
   ⟨ xs :: ⟦ List A ⟧ ⟩⇒
   ⟨ _ :: ⟦ n < length xs ⟧ ⟩⇏
   ⟨ ⟦ A ⟧ ⟩)) hsIndex
```

Even numbers example

While erasure annotation are very useful for specifying additional constraints for arguments, we may also want to annotate the result with additional proof objects.

Let us first introduce a witness data type stating that a number is even:

```
data Even : ℕ → Set where
  Z : Even 0
  SS : {n : ℕ} → Even n → Even (suc (suc n))
```

We can now annotate the addition function type with this witness data type using dependent pairs:

```
add : Σ ℕ Even → Σ ℕ Even → Σ ℕ Even
```

It would now be desirable to lift an addition function on plain natural numbers to the above refined type. To do so, we need to define a partial isomorphism between ℕ and Σ ℕ Even.

But instead of defining a partial isomorphism specifically for our current example, we strive to define a partial isomorphism which allows us to refine any type with additional proof objects.

In general, we will need a decision function to decide if the invariant holds on the given object. We introduce the Dec data type to represent the result of such a decision function:

```
data Dec (P : Set) : Set where
  yes : ( p :  P) → Dec P
  no  : (¬p : ¬ P) → Dec P
```

We introduce the type synonym DEC to denote decision functions:

```
DEC : (A : Set) → (P : A → Set) → Set
DEC A P = (a : A) →  Dec (P a)
```

Using this type synonym, we can for example define the type of the function deciding whether a number is even:

```
even? : DEC ℕ Even
```

For all decision functions of this form, we can define an isomorphism between the non-refined type A and the refined type Σ A P with the proof object P. We can implement such a parametrized partial isomorphism using the framework for dependent contracts introduced earlier. The definition goes as follows:

```
⇔Witness′ : PartIso
⇔Witness′ = record
  { ARG−a = Σ
    Set - the unrefined type
    (λ A → Σ
      (A → Set) - the proof type
```

```
          (λ P → DEC A P) -- the decision function
        )
      ; ARG−l = λ _ → ⊤
      ; ARG−h = λ _ → ⊤
      ; τ−l = λ aa _ → proj₁ aa
      ; τ−h = λ aa _ → Σ (proj₁ aa) (proj₁ (proj₂ aa))
      ; ⇅ = λ aa _ _ →
          let dec = proj₂ $ proj₂ aa
              up = λ x → case dec x of λ
                  { (yes p) → just (x , p)
                  ; (no ¬p) → nothing }
    in withMaybe up , total proj₁
      }
```

We can then instantiate the parametrized partial isomorphism, for example to refine the addition function to even numbers:

```
add : Σ ℕ Even → Σ ℕ Even → Σ ℕ Even
add = assert (makeContract (
    ⟨ _ :: ⟦ ⇔Witness ⇋ evenDec ⟧ ⟩⇒
    ⟨ _ :: ⟦ ⇔Witness ⇋ evenDec ⟧ ⟩⇒
    ⟨  ⟦ ⇔Witness ⇋ evenDec ⟧ ⟩)) hsAdd
    where
        evenDec = (ℕ , Even , even?) , wrap tt , wrap tt
```

Gcd example

Witness objects may also depend on other arguments, going into the realm of dependent types. For example, we may want to annotate the gcd function with the proof that the gcd divides the given two integers.

Let us first introduce the Haskell gcd function:

```
hsGcd = foreign (hsCall "Prelude.gcd")
    (ℕ → ℕ → ℕ)
```

And the type synonym IsGCD which defines the proof object we are interested in:

```
IsGCD : ℕ → ℕ → ℕ → Set
IsGCD x y z = z Divides x × z Divides y
```

We also assume that we are given the following decision function:

```
divs? : (x : ℕ) → (y : ℕ) → DEC ℕ (IsGCD x y)
```

We can then apply the parametrized partial isomorphism we have defined in the even example, and just pass in the divs? decision function among other things:

```
gcd : (x : ℕ) → (y : ℕ) → Σ ℕ (IsGCD x y)
gcd = assert (makeContract (
⟨ x :: ⟦ ℕ ⟧ ⟩⇒
⟨ y :: ⟦ ℕ ⟧ ⟩⇒
⟨ ⟦ ⇔Witness ⇋ (ℕ , IsGCD x y , divs? x y) , wrap tt , wrap tt ⟧ ⟩
)) hsGcd
```

This shows that our mechanism for dependent contracts is very powerful, and also allows some degree of abstraction inside the contracts themselves.

5.6 Contracts and the FFI

The Contract framework we have defined so far cannot only be used with normal Agda definitions, but most importantly also combines nicely with the new FFI introduced in chapter 4. We introduce a new Agda reflection macro called assert−foreign for this purpose.

The big advantage of this new macro is that it is able to infer the type of the foreign call from the contract; we don't need to specify the type twice! If we revisit the map example from earlier, we can rewrite this as:

```
map″ = assert−foreign (hsCall "Data.List.map")
  (makeContract (
  ⟨ n :: ⟦ ℕ ⟧ ⟩⇏
  ⟨ A :: ⟦ Set ⟧ ⟩⇒
  ⟨ B :: ⟦ Set ⟧ ⟩⇒
  ⟨ f :: (⟨ _ :: ⟦ A ⟧ ⟩⇒ ⟨ ⟦ B ⟧ ⟩) ⟩⇒
  ⟨ _ :: ⟦ vec⇔list ⇋ A , wrap tt , n ⟧ ⟩⇒
  ⟨  ⟦ vec⇔list ⇋ B , wrap tt , n ⟧ ⟩
  ))
```

The combination of our new FFI interface together with our contract library yields a concise, precise and powerful tool for expressing FFI calls in Agda.

5.7 Related work

There has been a lot of research into dynamically verified contracts. The contracts studied by Hinze et. al [15], which are implemented in Haskell, have inspired some of our own design choices. They provide a succinct and elegant way of encoding contracts in a functional setting. However, their contracts do not support dependent types and are hence less powerful than our own implementation.

How contracts can be leveraged for interaction between a dependently and a non-dependently typed language has been studied by Osera et. al [26]. He studies the behavior of the dynamic conversions, but only discusses a basic conversion primitive without adding contracts for functions.

How such a conversion primitive can be implemented has been discussed by Tanter [30], who independently discovered a similar conversion primitive to the one we are using. The fact that we have independently discovered a very similar solution may mean that this is a preferable spot in the design space. In comparison, our solution extends the basic primitives with a contract layer yielding a more high-level design more suited for wide-scale application.

Furr et. al [14] have studied a different approach, whereby they have created a multi-lingual type inference system for the OCaml/C language combination. Having such a system would greatly reduce the potential use cases for contracts; however, implementing a multi-lingual type inference system for the Agda/Haskell combination would be non-trivial. Furthermore, their solution is not able to cover all possible FFI use cases; a more traditional FFI would still be necessary.

5.8 Future work

Our contract solution works fine for many use cases, but there is still room for improvement.

One restriction we would hope to lift in the future is that currently a contract may not depend on arguments using isomorphisms. Consider the following example:

```
hsMinus = foreign (hsCall "Prelude.-") (ℤ → ℤ → ℤ)

minus : (x : ℕ) → (y : ℕ) → (x ≥ y) → ℕ
minus = assert (makeContract (
  ⟨ x :: ⟦ ℕ⇔ℤ ⥱ ∅ ⟧ ⟩⇒
  ⟨ y :: ⟦ ℕ⇔ℤ ⥱ ∅ ⟧ ⟩⇒
  ⟨ _ :: ⟦ x ≥ y ⟧ ⟩⇒
  ⟨ ⟦ ℕ⇔ℤ ⥱ ∅ ⟧ ⟩
  )) hsMinus
```

We expose a minus function and require that a proof object is given that the minuend x is greater or equal to the subtrahend y. We wish that this proof object is defined in terms of the high level $\mathbb{N}$ values x and y. However, our current implementation does not allow dependencies on arguments introduced by isomorphism application, the above example hence is currently not supported. It should be feasible to allow such dependencies, but this requires further investigation.

Our current implementation is based on Agda's reflection mechanism. This has two major drawbacks; first, this complicates the contract library code. Secondly, it makes using the library by a user slightly more complicated than necessary. Creating an alternative implementation abstaining from using reflection, if possible, could maybe solve some of this issues.

A minor omission from our current contract library is that it does not support Agda implicit arguments. Adding support for implicit arguments should not complicate the underlying translation in any way, we hence expect that it will be straightforward to add this feature in the future.

Chapter 6

Conclusion

The Agda UHC Backend We have developed a working compiler for Agda, targeting UHC Core. We are able to compile the whole Agda standard library, together with a test suite of executable Agda programs and we now consider the UHC backend ready to use. Our modifications to UHC have been incorporated in the last UHC release; the changes to Agda have been integrated into the development version of Agda and will be part of the next release.

The UHC Agda backend provides an excellent foundation for exploring the interaction between Agda and Haskell. Our new Agda backend was crucial in enabling us to create a new FFI and contract framework.

This thesis shows how to tackle some of the challenges inherent in writing a compiler for Agda, piggybacking on the existing technology provided by UHC. We hope that providing a more robust backend for Agda, with excellent interoperability with an existing Haskell compiler, will provide the technology for more 'real world' dependently typed programs.

FFI We implemented a prototype of a new Foreign Function Interface (FFI) for Agda, which improves greatly on the old implementation. While this is an important achievement by itself, it also lays the foundation for further research into the area. Especially the combination with our Agda UHC backend enables the investigation of a more complete Haskell-Agda FFI, for example adding support for Haskell's class system.

Contracts Our work on contracts demonstrates that contracts are a viable and elegant approach to expressing FFI calls in a dependently-typed setting. They provide an easy-to-use solution to integrate Agda developments with external systems, offering a novel trade-off between static verification and development effort.

Our formal model shows that contracts can be implemented in a dependent system like Agda itself with only one additional axiom or primitive operation, hence limiting the inherent unsafety of FFI calls to specific locations.

A more fundamental problem with Agda itself is that universe-polymorphic types cannot be represented in Agda itself in a typed fashion, except by disabling the universe hierarchy and hence making Agda inconsistent. This is not a satisfying solution, but pending a more powerful Agda language cannot be avoided.

And while there is room for further improvement, our contract framework already solves many of the use cases it is aimed at and provides an elegant way to use the FFI in Agda in a concise and safe manner.

Acknowledgments

I would like to thank both my supervisors, Dr. Wouter Swierstra and Dr. Atze Dijkstra, for their support, helpful guidance and the fruitful discussions we had. My gratitude also goes to the Agda community and the Agda development team, which always provided helpful answers to my questions and made it ease to contribute to Agda and become part of the community.

With looming deadlines and a lot of work to do, my friends made the pressure much more bearable and I would like to thank all of them for the great times we had. Finally, a great thank you to my family and especially Dani, who has always supported me throughout my studies.

Bibliography

[1] *[Agda] Size limit on generated code?* URL: `https://lists.chalmers.se/pipermail/agda/2014/006990.html` (visited on 03/04/2015).

[2] *[Agda] Universe-Heterogeneous tree*. URL: `https://lists.chalmers.se/pipermail/agda/2015/007961.html` (visited on 09/03/2015).

[3] *[Agda] Universe-Heterogeneous tree*. URL: `https://lists.chalmers.se/pipermail/agda/2015/007966.html` (visited on 08/16/2015).

[4] Thorsten Altenkirch et al. "ΠΣ: Dependent Types without the Sugar." In: *Functional and Logic Programming*. Ed. by Matthias Blume, Naoki Kobayashi, and Germán Vidal. Lecture Notes in Computer Science 6009. Springer Berlin Heidelberg, 2010, pp. 40–55. ISBN: 978-3-642-12250-7 978-3-642-12251-4. URL: `http://link.springer.com/chapter/10.1007/978-3-642-12251-4_5` (visited on 02/14/2015).

[5] Lennart Augustsson. "A Compiler for Lazy ML." In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. LFP '84. New York, NY, USA: ACM, 1984, pp. 218–227. ISBN: 0-89791-142-3. DOI: `10.1145/800055.802038`. URL: `http://doi.acm.org/10.1145/800055.802038` (visited on 02/14/2015).

[6] Brady, Edwin. "Cross-platform Compilers for Functional Languages." In: *Under consideration for Trends in Functional Programming* (2015).

[7] Edwin Brady. "Epic—A Library for Generating Compilers." In: *Trends in Functional Programming*. Ed. by Ricardo Peña and Rex Page. Lecture Notes in Computer Science 7193. Springer Berlin Heidelberg, 2012, pp. 33–48. ISBN: 978-3-642-32036-1 978-3-642-32037-8. URL: `http://link.springer.com/chapter/10.1007/978-3-642-32037-8_3` (visited on 02/14/2015).

[8] Edwin Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation." In: *Journal of Functional Programming* 23.05 (September 2013), pp. 552–593. ISSN: 1469-7653. DOI: `10.1017/S095679681300018X`. URL: `http://journals.cambridge.org/article_S095679681300018X` (visited on 02/14/2015).

[9] Edwin C. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. 2005.

[10] Manuel M. T. Chakravarty et al. "Associated Types with Class." In: *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '05. New York, NY, USA: ACM, 2005, pp. 1–13. ISBN: 1-58113-830-X. DOI: `10.1145/1040305.1040306`. URL: `http://doi.acm.org/10.1145/1040305.1040306` (visited on 03/13/2015).

[11] Arthur Charguéraud. “The Locally Nameless Representation.” In: *Journal of Automated Reasoning* 49.3 (May 6, 2011), pp. 363–408. ISSN: 0168-7433, 1573-0670. DOI: 10.1007/s10817-011-9225-2. URL: http://link.springer.com/article/10.1007/s10817-011-9225-2 (visited on 03/13/2015).

[12] Atze Dijkstra, Jeroen Fokker, and S. Doaitse Swierstra. “The Architecture of the Utrecht Haskell Compiler.” In: *Proceedings of the 2Nd ACM SIGPLAN Symposium on Haskell*. Haskell ’09. New York, NY, USA: ACM, 2009, pp. 93–104. ISBN: 978-1-60558-508-6. DOI: 10.1145/1596638.1596650. URL: http://doi.acm.org/10.1145/1596638.1596650 (visited on 02/14/2015).

[13] Robert Bruce Findler and Matthias Felleisen. “Contracts for Higher-order Functions.” In: *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*. ICFP ’02. New York, NY, USA: ACM, 2002, pp. 48–59. ISBN: 978-1-58113-487-2. DOI: 10.1145/581478.581484. URL: http://doi.acm.org/10.1145/581478.581484 (visited on 10/08/2015).

[14] Michael Furr and Jeffrey S. Foster. “Checking Type Safety of Foreign Function Calls.” In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’05. New York, NY, USA: ACM, 2005, pp. 62–72. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065019. URL: http://doi.acm.org/10.1145/1065010.1065019 (visited on 02/14/2015).

[15] Ralf Hinze, Johan Jeuring, and Andres Löh. “Typed Contracts for Functional Programming.” In: *Functional and Logic Programming*. Ed. by Masami Hagiya and Philip Wadler. Lecture Notes in Computer Science 3945. Springer Berlin Heidelberg, 2006, pp. 208–225. ISBN: 978-3-540-33438-5 978-3-540-33439-2. URL: http://link.springer.com/chapter/10.1007/11737414_15 (visited on 03/30/2015).

[16] *Issue 1414 - agda - MAlonzo returns wrong result for FlexibleInterpreter - Agda is a dependently typed programming language - Google Project Hosting*. URL: https://code.google.com/p/agda/issues/detail?id=1414 (visited on 03/04/2015).

[17] Alan Jeffrey. “Dependently Typed Web Client Applications.” In: *Practical Aspects of Declarative Languages*. Ed. by Kostis Sagonas. Lecture Notes in Computer Science 7752. Springer Berlin Heidelberg, 2013, pp. 228–243. ISBN: 978-3-642-45283-3 978-3-642-45284-0. URL: http://link.springer.com/chapter/10.1007/978-3-642-45284-0_16 (visited on 02/14/2015).

[18] Pierre Letouzey. “A New Extraction for Coq.” In: *Types for Proofs and Programs*. Ed. by Herman Geuvers and Freek Wiedijk. Lecture Notes in Computer Science 2646. Springer Berlin Heidelberg, 2003, pp. 200–219. ISBN: 978-3-540-14031-3 978-3-540-39185-2. URL: http://link.springer.com/chapter/10.1007/3-540-39185-1_12 (visited on 02/14/2015).

[19] Andres Löh, Conor McBride, and Wouter Swierstra. “A Tutorial Implementation of a Dependently Typed Lambda Calculus.” In: *Fundamenta Informaticae* 102.2 (January 1, 2010), pp. 177–207. DOI: 10.3233/FI-2010-304. URL: http://dx.doi.org/10.3233/FI-2010-304 (visited on 03/01/2015).

[20] *MAlonzo returns wrong result for FlexibleInterpreter · Issue #1414 · agda/agda.* URL: `https://github.com/agda/agda/issues/1414` (visited on 10/07/2015).

[21] Marcin Benke. *Alonzo - a compiler for Agda.* 2007. URL: `http://www.mimuw.edu.pl/~ben/Papers/TYPES07-alonzo.pdf` (visited on 02/14/2015).

[22] Simon Marlow. *Haskell 2010 Language Report.*

[23] Conor McBride and James McKinna. "Functional Pearl: I Am Not a Number–i Am a Free Variable." In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell.* Haskell '04. New York, NY, USA: ACM, 2004, pp. 1–9. ISBN: 1-58113-850-4. DOI: `10.1145/1017472.1017477`. URL: `http://doi.acm.org/10.1145/1017472.1017477` (visited on 03/02/2015).

[24] Ulf Norell. "Dependently Typed Programming in Agda." In: *Proceedings of the 6th International Conference on Advanced Functional Programming.* AFP'08. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 230–266. ISBN: 978-3-642-04651-3. URL: `http://dl.acm.org/citation.cfm?id=1813347.1813352` (visited on 10/09/2015).

[25] Olle Fredriksson and Daniel Gustafsson. "A totally Epic backend for Agda." Göteborg, Sweden: Chalmers University of Technology, May 2011. URL: `http://publications.lib.chalmers.se/records/fulltext/146807.pdf`.

[26] Peter-Michael Osera, Vilhelm Sjöberg, and Steve Zdancewic. "Dependent Interoperability." In: *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification.* PLPV '12. New York, NY, USA: ACM, 2012, pp. 3–14. ISBN: 978-1-4503-1125-0. DOI: `10.1145/2103776.2103779`. URL: `http://doi.acm.org/10.1145/2103776.2103779` (visited on 02/14/2015).

[27] Simon Peyton Jones and Simon Marlow. "Secrets of the Glasgow Haskell Compiler Inliner." In: *J. Funct. Program.* 12.5 (July 2002), pp. 393–434. ISSN: 0956-7968. DOI: `10.1017/S0956796802004331`. URL: `http://dx.doi.org/10.1017/S0956796802004331` (visited on 03/03/2015).

[28] Tom Schrijvers et al. "Complete and Decidable Type Inference for GADTs." In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming.* ICFP '09. New York, NY, USA: ACM, 2009, pp. 341–352. ISBN: 978-1-60558-332-7. DOI: `10.1145/1596550.1596599`. URL: `http://doi.acm.org/10.1145/1596550.1596599` (visited on 03/13/2015).

[29] Martin Sulzmann et al. "System F with Type Equality Coercions." In: *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation.* TLDI '07. New York, NY, USA: ACM, 2007, pp. 53–66. ISBN: 1-59593-393-X. DOI: `10.1145/1190315.1190324`. URL: `http://doi.acm.org/10.1145/1190315.1190324` (visited on 02/18/2015).

[30] Éric Tanter and Nicolas Tabareau. "Gradual Certified Programming in Coq." In: (June 12, 2015). arXiv: `1506.04205`. URL: `http://arxiv.org/abs/1506.04205` (visited on 07/15/2015).

[31] *tasty-golden: Golden tests support for tasty | Hackage.* URL: `http://hackage.haskell.org/package/tasty-golden` (visited on 10/07/2015).

[32] *tasty-silver: A fancy test runner, including support for golden tests. | Hackage*. URL: `http://hackage.haskell.org/package/tasty-silver` (visited on 10/07/2015).

[33] *Use sharing in generated code · Issue #1528 · agda/agda*. URL: `https://github.com/agda/agda/issues/1528` (visited on 10/07/2015).