Eötvös Loránd University

Faculty of Informatics

Dept. of Programming Languages and Compilers

Agda formalisation of an elaborator for a language based on simply typed lambda calculus

Supervisor:
Kaposi Ambrus
Associate professor

Author:
Zahorán Barnabás
Computer Science MSc

Budapest, 2024

Contents

1 Introduction **2**

1.1 Motivation . . . 2

1.2 Glossary . . . 4

2 Outline **5**

3 Related work **6**

4 Implementation **9**

4.1 Lexical analysis . . . 10

4.2 Parsing . . . 14

4.3 Scope checking . . . 21

4.4 Bidirectional type checking . . . 25

4.5 Algebraic definition quotiented by equations . . . 31

4.6 Standard interpretation and normalisation . . . 36

4.7 Running the elaborator . . . 38

4.8 Examples . . . 41

5 Conclusion **50**

5.1 Results . . . 50

5.2 Discussion . . . 51

5.3 Future work . . . 52

Acknowledgements **53**

Bibliography **53**

List of Codes **56**

Chapter 1

Introduction

1.1 Motivation

Our goal was to formalise an elaborator using Agda that presents all steps of describing a simple language in a mathematically rigorous, yet practical and easy to run framework. Elaboration means that we refine the concept of the language from broader ideas to stricter ones, while moving from more concrete representations towards more abstract ones.

The language - which we will often refer to as our *object language*, *object theory* or *well-typed syntax with quotients* - is based on simply typed lambda calculus (STLC) à la Church and à la Curry. The description stands closer to Curry's system. It has function space, i.e., abstraction and application, and a few extensions like finite types: booleans, products and sums; inductive types: naturals, lists, trees; and coinductive types like streams and simple state machines.

The novelty of our study is the elaboration stack that we implemented as an extension to the pre-existing formalisation [1] of the language. The elaboration consists of lexical analysis, parsing, scope checking and bidirectional type checking. We also refer to these as *compilation steps*. In addition, we have a standard model interpretation into our meta language and normalisation done by Agda. We call these the *evaluation steps*.

By rigorousness we mean that our code is correct by construction in multiple aspects. Agda - our *meta theory* or *metatheoretic language* - is a purely functional and total language, so we cannot get unwanted side effects, unhandled cases, runtime exceptions, or even non-terminating computations. The latter two could be potential sources of issues if we used Haskell, for example. Another aspect is that our representations are algebraic theories giving us strong guarantees. For example, our abstract binding trees cannot be badly scoped, or our well-typed terms cannot be badly typed by the very definitions of these constructs. Moreover, the theorem proving nature of Agda also helps us with formalising and proving statements about our language, e.g., program equivalence can be verified.

Ease of use and transparency comes in the form of our top-level functions: `elaborate` returns each intermediate step leading up to the final compilation and evaluation results, or until an error occurs, e.g., syntax error, scope error, etc. This provides the user deeper insight into the abstract representations and reasons of potential errors. We can also run `compile` or `eval` when only caring about the compilation or evaluation results, or their `compileM` and `evalM` versions when a `Maybe` monadic return value is desired.

On Code 1.1, we present a few small examples to give the reader a quick taste of our language's syntax and our elaborator's capabilities. We omit all the `_ = refl` proof lines for brevity.

```
_ : compile "((10)"                    ≡ inj₂ syntax-error
_ : compile "(λ foo. bar) : ℕ → ℕ"     ≡ inj₂ scope-error
_ : compile "if true then 0 else false" ≡ inj₂ type-error

+1       = "(λ x. x+1)   : ℕ → ℕ"
double   = "(λ x. x+x)   : ℕ → ℕ"
triple   = "(λ x. x+x+x) : ℕ → ℕ"
plus     = "(λ x y. iteℕ x (λz.z + 1) y) : ℕ → ℕ → ℕ"
multiply = "(λ x y. iteℕ 0 (λz.z + x) y) : ℕ → ℕ → ℕ"
twice    = "(λ f x. f f x)   : (ℕ → ℕ) → ℕ → ℕ"
3-times  = "(λ f x. f f f x) : (ℕ → ℕ) → ℕ → ℕ"
∘        = "(λ f g x. f g x) : (ℕ → ℕ) → (ℕ → ℕ) → ℕ → ℕ"

_ : eval (triple ++ₛ "8")                      ≡ inj₁ (Nat , λ γ* → 24)
_ : eval (plus ++ₛ "3" ++ₛ "8")                ≡ inj₁ (Nat , λ γ* → 11)
_ : eval (multiply ++ₛ "6" ++ₛ "20")           ≡ inj₁ (Nat , λ γ* → 120)
_ : eval (3-times ++ₛ +1 ++ₛ "10")             ≡ inj₁ (Nat , λ γ* → 13)
_ : eval (∘ ++ₛ double ++ₛ triple ++ₛ "10")    ≡ inj₁ (Nat , λ γ* → 60)
```

Code 1.1: Introductory examples of compilation and evaluation results

1.2 Glossary

`STLC` = Simply Typed Lambda Calculus

`AST` = Abstract Syntax Tree

`ABT` = Abstract Binding Tree

`constructor` = type introduction rule, e.g., true, false

`destructor` = type elimination rule, e.g., if_then_else_

`Ty` = Type of our object language

`Tm` = Term of our object language

`Con` = Context of our object language

`Sub` = Substitution in our object language

`St` = Standard model of our object language

`ite` = ”if then else”, or ”iterator of” when used as a prefix, e.g., in `iteℕ`

`nil` = constructor of the empty list

`cons` = head-tail constructor of non-empty lists

`zero, suc` = constructors of Peano arithmetic

Chapter 2

Outline

This study expects the reader to have some familiarity with λ-calculus, for we will not go into details about basic concepts like substitution, alpha equivalence or beta reduction.

The reader will also need some expertise with reading code written in functional languages, e.g., Haskell, including concepts like recursion, pattern matching and algebraic data types. Possessing familiarity with dependent types and Agda is an advantage, but not an absolute necessity for following our key points.

Firstly, we discuss the work of others related to our study in Chapter 3, where we cite several sources that can assist the reader in understanding our work. Then, from Chapter 4.1 to 4.4, we explain our methods of formalising the STLC elaborator. Chapters 4.5 and 4.6 describe the qutiented well-typed syntax and its standard model evaluation. Chapter 4.7 presents our high-level functions for running both the elaboration and the evaluation steps, while Chapter 4.8 goes through practical examples demonstrating the framework. Finally, in Chapter 5, we summarise and discuss our results and offer some insight about future work that could improve the implementation.

This thesis, all examples it presents, and the whole codebase is publicly accessible on GitHub [2].

Chapter 3

Related work

Agda and underlying theories

Agda [3] is a dependently typed programming language and theorem prover based on Martin-Löf's intuitionistic type theory [4]. We could use any language, e.g., Haskell, as our meta theory, implementing a compiler from the set of strings to some algebraic data type, however we would miss out the *mathematical rigorousness* we mentioned in the introduction.

The expressiveness and rigorousness of Agda's type system comes from the Curry-Howard isomorphism, which gives us *propositions-as-types* and *proofs-as-programs*. For example, defining a function of type `A → B → C` stands as a proof of `C`, presuming that `A` and `B` hold. More precisely, given a constructive proof for proposition `A` and one for `B`, this function can construct us a proof for `C`.

Extending this with dependent types allows us to formalise and prove statements in first-order predicate logic. Sørensen and Urzyczyn had published a great collection of literature about the isomorphism and related theories [5]. In particular, we recommend reading Chapters 1-4, 6, 9 and 10 of their work the most. They introduce the intuitionistic logical foundations and the basis of simply typed λ-calculus. Also, in Chapter 11.6 they present Gödel's System T, which stands even closer to our language than the baseline simply typed λ-calculus.

In practice, the above means that by using dependent functions and products, usually notated `Π` and `Σ`, respectively, we can encode universally and existentially quantified, i.e., first-order, statements in Agda's types, then prove said statements by constructing terms of these types.

Furthermore, having dependent types that correspond to propositions as firstclass citizens in our meta language allows us to index our inductive types (e.g., our binding trees) or to build models quotiented by equations (e.g., our well-typed syntax). This in essence, means that we can write record fields with types that are propositions of equality and need equality proofs when instancing, giving us a means to formalise categories and inference rules for operational semantics, discussed in Chapter 4.5.

As a result, the algebraic terms in our object language cannot be badly scoped or badly typed by definition and we also get decidable program equality, interpretation into our metatheoretic language, and even normalisation, i.e., a form of evaluation as seen in Chapter 4.6

agda-stdlib

Agda standard library [6] is a project that collects the most commonly used constructions for both programming and theorem proving under one easy to access umbrella. We chose to reuse the fundamental types and functions, e.g., `Bool`, `Nat`, `Maybe` and `List`, from this library instead of reimplementing these concepts.

agdarsec

Agdarsec [7] plays a crucial role in our toolchain. It is a total parser combinator library written in Agda. Parser combinators give us a high level interface for building complex parsers by composing simpler primitives. Totality means that by the definition of its type, we cannot write non-terminating parsers, i.e., Agda's termination checker would not accept such constructions. In order to circumvent this, the library builds fixpoints by using a form of guarded recursion and sized types. This gives us strong guarantees: non-advancing parsers or problematic left recursive grammar rules are simply not type correct in this framework.

Veltri and von Weide [8] not only discuss guarded recursion, sized types and their relation, but they also work in Agda on an object language similar to ours, e.g., their terms, substitution calculus, quotients are all similar.

Type systems formalisation

The formalisation we use as our object theory was written by Ambrus Kaposi as course notes for the Type systems lecture at ELTE-IK [1]. We took his `STT`, `Fin`, `Ind` and `Coind` languages - standing for *Simple Type Theory*, *finite*, *inductive* and *coinductive* types, respectively - and merged them into a single model we call `STLC`, i.e., Simply Typed Lambda Calculus.

Our extension to his work, which is also the uniqueness of our study, is the elaboration toolchain [9] that we will present in greater detail.

Note that, while our language contains some less common constructions - mostly for demonstration purposes - like trees and streams, these are still usually considered simple types. We do not support dependent or polymorphic types or even full recursion (which would require a fixpoint combinator), but our formalisation could be extended in the future with some of these concepts.

Chapter 4

Implementation

First we present a bird's-eye overview of our elaboration and evaluation stacks on Figure 4.1, then show the process on a concrete example on Figure 4.2. Subsequent chapters detail the depicted representations and the steps between them.

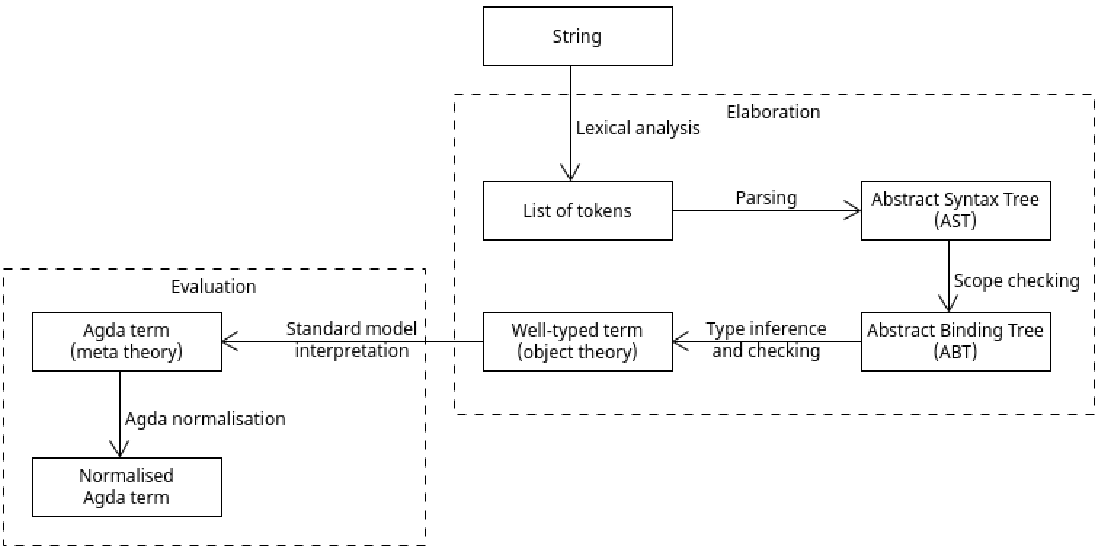

Figure 4.1: High-level overview of the elaboration and evaluation steps

From our seven representations: string and list of tokens are trivial; AST and ABT are simple inductive data types (albeit the latter is indexed by $\mathbb{N}$); and Agda terms, whether in normal form or not, are in the metatheoretic domain, which we do not have to deal with. The well-typed term level, i.e., our object theory, is the only one that bears a more complex definition.

From the six transitions between our levels of abstractions: lexical analysis and parsing are solved in a high-level approach using the agdarsec library; scope checking, type inference and standard model interpretation bear more in-depth explanations; and Agda normalisation, as mentioned earlier, is not part of our domain.

We give prefixes to our constructors on all levels for clarity, e.g., `t-true` for the token, `s-true` for the AST (syntactic) term, `p-true` for its parser, and `abt-true` for the ABT term. Well-typed terms have no prefixes, e.g., `true`. We also often use the "o" suffix for distinguishing between Agda terms and our own, e.g. in `zeroo` and `suco`.

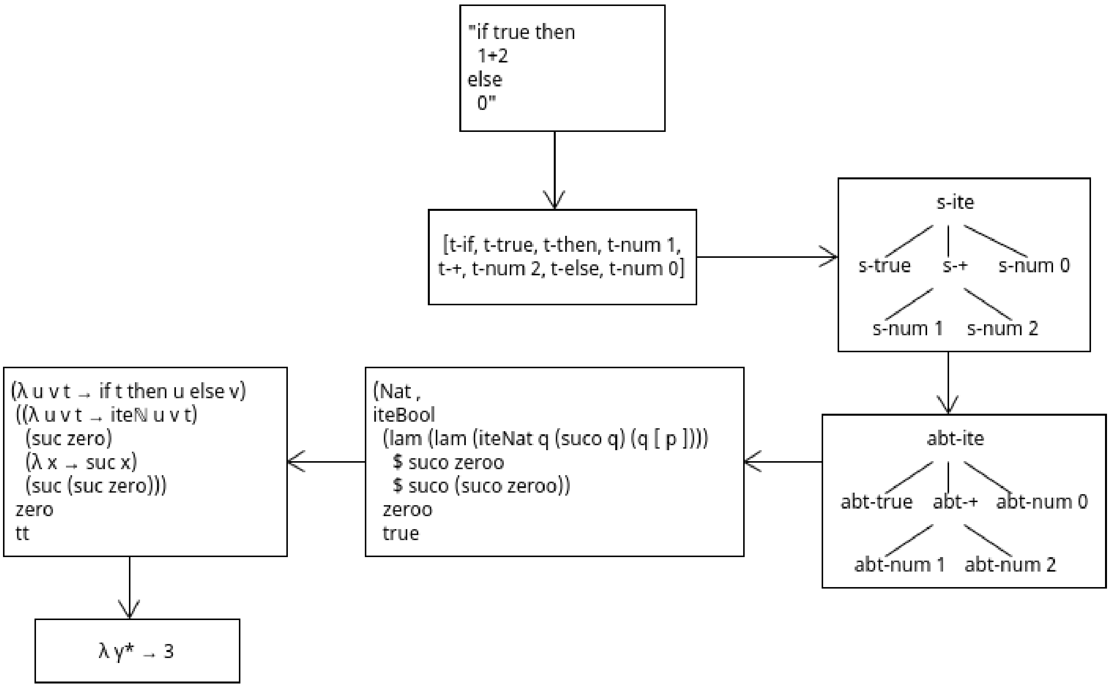

Figure 4.2: A simple elaboration example

4.1 Lexical analysis

First of all, it is worth mentioning that we initially implemented our own lexer which was functionally the same as the final implementation relying on agdarsec. It was a simple algorithm: read the input by characters; accumulate them to a word until a whitespace or separator is read; if it is a valid token put it into a list, otherwise return an error; repeat the steps until end of input.

Since agdarsec already ships with a high-level interface for lexical analysis, we chose to abstract this step similarly to parsing.

```
data Tok : Set where
  t-true   : Tok
  t-false  : Tok
  t-if     : Tok
  t-then   : Tok
  t-else   : Tok
  t-isZero : Tok
  t-+      : Tok
  t-num    : ℕ → Tok
  t-λ      : Tok
  t-var    : String → Tok
```

Code 4.1: Portion from our vocabulary of tokens

We have 50 tokens in total. This includes symbols like `t-dot`, `t-lpar` and `t-rpar`, standing for the '.' in a λ binding and for the '(' and ')' parentheses, respectively. On Code 4.1, and also in other code examples to follow in our thesis, we do not list all cases, only a few that can serve as examples and the more notable ones. We insert "..." comments where code is omitted from the middle of a snippet.

Tokens that need information attached are indexed. In particular, `t-num` and `t-var` store the natural numbers and variable names read from the source code, respectively.

We supply the `Token = Position × Tok` type to the lexical analyser, so it also outputs the character positions for all tokens.

The agdarsec library requires decidable equality over our token type, which we formalise in the following way: we list all 50 cases of propositional equality and postulate that the other combinations are unequal, instead of listing all the thousand and more cases. For `t-num` and `t-var` we simply rely on the decidable equality of the `Nat` and `String` types, as well as `cong`, i.e., congruence of equality in Agda.

```
eq-tok : Decidable {A = Tok} _≡_
eq-tok t-true  t-true  = yes refl
eq-tok t-false t-false = yes refl
eq-tok t-if    t-if    = yes refl
-- ...
eq-tok (t-num n) (t-num m) with n ≟ℕ m
... | yes eq = yes (cong t-num eq)
... | no ¬eq = no λ hyp → ¬eq (cong (λ { (t-num n) → n ; _ → 0}) hyp)
eq-tok t-λ t-λ = yes refl
eq-tok (t-var name) (t-var name') with name ≟str name'
... | yes eq = yes (cong t-var eq)
... | no ¬eq = no λ hyp → ¬eq (cong (λ { (t-var s) → s ; _ → ""}) hyp)
-- ...
eq-tok _ _ = no no-more-eq where
  private postulate no-more-eq : ∀ {A} → A
```

Code 4.2: Decidability of token equivalence

```
keywords : List⁺ (String × Tok)
keywords = ("true", t-true) ∷ ("false", t-false) ∷ ("if", t-if) ∷ {- ... -} ∷ []
```

Code 4.3: Mapping strings to our token type

```
breaking : Char → ∃ λ b → if b then Maybe Tok else Lift _ ⊤
breaking c = case c of λ where
  '+' → true , just t-+
  'λ' → true , just t-λ
  '.' → true , just t-dot
  '→' → true , just t-→
-- ...
  c   → if isSpace c then true , nothing else false , _
```

Code 4.4: Special tokens that also work as separators

As expected, all whitespace characters act as valid separators. Additionally, our tokens which are symbols, i.e., the ones that are not alphanumeric, are also accepted as separators. For example, `"λx.x+10"` is lexically valid. On Code 4.4, we can see that `breaking` takes a character and returns a pair. The first field indicates if it is a separator and the second optionally contains a `Tok` depending on whether it is also a token.

```
default : String → Tok
default s = case (listch⇒ℕ (toList s)) of λ where
  (just n) → t-num n
  nothing  → t-var s
```

Code 4.5: Fallback function for words that are not keywords or separators

If the lexer encounters a word that is not a valid keyword or a separator, then it calls the `default` function on it. Our implementation tests whether the word is a valid natural number. If it is, then it constructs a `t-num`, otherwise we treat the word as a variable name using `t-var`. `listch⇒ℕ` has a standard definition as seen on Code 4.6.

```
listch⇒ℕ : List Char → Maybe ℕ
listch⇒ℕ [] = nothing
listch⇒ℕ = step 0 where
  step : ℕ → List Char → Maybe ℕ
  step n [] = just n
  step n (c ∷ cs) = if isDigit c then
                      step (n + (pow 10 (length cs)) * (toℕ c - 48)) cs
                    else
                      nothing where
    pow : ℕ → ℕ → ℕ
    pow b = λ { 0 → 1 ; (suc e) → b * (pow b e)}
```

Code 4.6: Reading natural numbers from lists of characters

Accepting arbitrary strings as variable names is uncommon. We chose this approach because agdarsec's interface, i.e., the `default` function, does not support returning an error. It returns `Tok` and not a type like `Maybe Tok`. Of course, we could return an ”error token”, but that would pollute our `Tok` type, also the error would surface during parsing which would not be elegant. A potential clean solution would be to bring back our self-made tokenizer (that only accepted alphanumeric variable names) and inject that to agdarsec's parsing toolchain. This method is supported by the library and could be a potential improvement in the future.

The lexer itself is parametrized with the three functions discussed above using the `import` statement:

```
open import Text.Lexer keywords breaking default
```

4.2 Parsing

Just as in lexical analysis, our first approach was to implement our own parsing algorithm. This meant a recursive function that simulated a state machine. There were two problems with this approach. Firstly, Agda's termination checker rejected recursive calls at binary and ternary operators with no trivial way around, i.e., without implementing some form of fixpoint combinator. Secondly, our implementation could only handle the Polish notation of STLC, i.e., operators had to be written in prefix order: `+ 1 + 2 3` instead of `1 + 2 + 3`. Implementing infix syntax for operators like `_+_`, function application with no symbol required, and support for parenthesis proved to be far from trivial. In essence, we would have had to implement a parser combinator library of our own. In order to avoid reinventing the wheel, we opted to use agdarsec to build our AST.

Setting up agdarsec's parser takes substantial importing and argument instancing work, mostly concerning monadic constructions. We chose to omit these implementation details from this thesis. Those who are interested can find the code in our repository [2], or can see similar use cases among the examples distributed with the library. Instead, we will focus on the high-level construction of the syntax itself by building and composing combinators with the library.

Firstly, we present our syntax on Code 4.7. We have nullary nodes like the Bool constructors `s-true`, `s-false` or `s-nil` which creates the empty list. Unary nodes include the product elimination rules `s-fst` and `s-snd`, and the sum introduction rules `s-inl` and `s-inr`. STLC has several binary operators, for example, addition of naturals `_s-+_`, function elimination `_s-$_`, type former for products `_s-,_` or the cons operator of lists `_s-∷_`. Ternary nodes include iterators like `s-ite` (this is the Bool destructor "if then else"), `s-iteℕ`, `s-iteList` and `s-iteTree`.

Code 4.8 depicts the syntax we use for our type annotations, which we will need to clarify types of terms like the empty list `"nil"` or identity function `"λx.x"`. Their code might look like `"nil : [ℕ]"` and `"(λx.x) : ℕ → ℕ"`, respectively.

```
data AST : Set where
  s-true       : AST
  s-false      : AST
  s-ite        : AST → AST → AST → AST
  s-isZero     : AST → AST
  _s-+_        : AST → AST → AST
  s-num        : ℕ → AST
  s-λ          : List String → AST → AST
  s-var        : String → AST
  _s-$_        : AST → AST → AST

  s-triv       : AST
  _s-,_        : AST → AST → AST
  s-fst        : AST → AST
  s-snd        : AST → AST

  s-inl        : AST → AST
  s-inr        : AST → AST
  s-case       : AST → AST → AST

  s-nil        : AST
  _s-∷_        : AST → AST → AST

  s-leaf       : AST → AST
  _s-node_     : AST → AST → AST

  s-iteℕ       : AST → AST → AST → AST
  s-iteList    : AST → AST → AST → AST
  s-iteTree    : AST → AST → AST → AST

  s-head       : AST → AST
  s-tail       : AST → AST
  s-genStream  : AST → AST → AST → AST

  s-put        : AST → AST → AST
  s-set        : AST → AST
  s-get        : AST → AST
  s-genMachine : AST → AST → AST → AST → AST

  s-ann        : AST → SType → AST
```

Code 4.7: Syntax of STLC

```
data SType : Set where
  s-Nat     : SType
  s-Bool    : SType
  _s-→_     : SType → SType → SType
  s-⊤       : SType
  s-⊥       : SType
  _s-×_     : SType → SType → SType
  _s-⊎_     : SType → SType → SType
  s-List    : SType → SType
  s-Tree    : SType → SType
  s-Stream  : SType → SType
  s-Machine : SType
```

Code 4.8: Syntax for type annotations

At this level `s-λ` does not mean Church's usual lambda, but an abstract node that stores a list of variable names it binds, since we support the compact notation `"λ x y z. x+y+z"`. We will unroll this to properly nested lambdas when introducing De Bruijn indices [10] on our ABT level. Analogue to this, `s-var` stores the parsed variable name in its index which will be turned to a De Bruijn index later.

```
p-tok : Tok → ∀[ Parser P [ Token ] ]
p-tok t = maybeTok $ λ where
  tok@(_ , t') → case eq-tok t t' of λ where
    (yes eq) → just tok
    (no ¬eq) → nothing

p-parens : ∀ {A} → ∀[ □ Parser P A ⇒ Parser P A ]
p-parens rec = p-tok t-lpar &> rec <& box (p-tok t-rpar)

p-name : ∀[ Parser P [ String ] ]
p-name = maybeTok λ where (_ , t-var s) → just s; _ → nothing
```

Code 4.9: Parsing exact tokens, parenthesised terms and variable names

`p-tok` parses an exact token by using `eq-tok` introduced in the previous chapter. `p-parens` parses an opening parenthesis, followed by a recursive parsing of an arbitrary term, and then by a closing parenthesis. We always call the parameter passed for recursive parsing `rec` which has type `□ Parser P A`. Here □ means the type is "boxed" or "guarded" as in *guarded recursion* [8]; `P` stands for the parameters of the parsing module; and `A` is an arbitrary type, so `p-parens` is polymorphic and can be used to parse any type between parentheses. `p-name` can parse arbitrary strings, which we use in `p-var` and `p-λ`.

```
p-type : ∀[ Parser P [ SType ] ]
p-type = fix _ $ λ rec →
  let p-nat     = s-Nat     <$ p-tok t-ℕ
      p-bool    = s-Bool    <$ p-tok t-𝕃
      p-unit    = s-⊤       <$ p-tok t-⊤
      p-empty   = s-⊥       <$ p-tok t-⊥
      p-machine = s-Machine <$ p-tok t-Machine
      p-list'   = s-List    <$> (p-tok t-[ &> rec <& box (p-tok t-]))

      p-atom = p-nat <|> p-bool <|> p-unit <|> p-empty <|> p-machine <|>
               p-list' <|> p-parens rec
      p-×    = chainr1 p-atom (box (_s-×_ <$ p-tok t-×))
      p-+    = chainr1 p-×    (box (_s-⊎_ <$ p-tok t-⊎))
      p-→    = chainr1 p-+    (box (_s-→_ <$ p-tok t-→))

      p-list   = s-List   <$> (p-tok t-List   &> box p-atom)
      p-tree   = s-Tree   <$> (p-tok t-Tree   &> box p-atom)
      p-stream = s-Stream <$> (p-tok t-Stream &> box p-atom)

  in p-→ <|> p-list <|> p-tree <|> p-stream
```

Code 4.10: Parsing type annotations

We use agdarsec's fixpoint combinator `fix` for recursively parsing the syntax of type annotations as seen on Code 4.10. The parsing of nullary and unary nodes are trivial, though note the required use of the `box` operator for the type correctness of the total parsing. For binary nodes `p-→`, `p-+` and `p-×`, we use the library's `chainr1` combinator which parses right associative chains of operators. The order of embedding among them determines the precedence between these operators. Also, these chains can consist of single elements, and as a result, `"ℕ"` is parsed as a single-element `→` chain, of a single-element `+` chain of a single-element `×` chain of an atom, which gets accepted by `p-nat`.

We guarantee guardedness by wrapping the recursive call in parentheses, i.e., the `p-parens rec` case in `p-atom`. This introduces requirement for some parentheses in certain situations. However, this lifts some ambiguity, for example, in `"List ℕ → 𝕃"` between the choices of `"List (ℕ → 𝕃)"` and `"(List ℕ) → 𝕃"`.

Our list types support two separate syntaxes: the classical `"List ℕ"` and the Haskell-like `"[ℕ]"`.

For `AST`, parsers of non-recursive nodes like `p-true`, `p-false` or `p-num` have type `∀[ Parser P [ AST ] ]`, while parsers of recursive nodes are `∀[ □ Parser P [ AST ] ⇒ Parser P [ AST ] ]`, where the guarded parameter is the recursive parser we name `rec`. Without listing them all, we include a few more examples on Code 4.11.

```
p-λ rec = (λ l⁺ → s-λ (toList l⁺)) <$>
          (p-tok t-λ &> box (list⁺ p-name) <& box (p-tok t-dot)) <*> rec

p-$ rec = _s-$_ <$> p-subexp rec <*> rec

p-+ rec = chainl1 (p-subexp rec) (box (_s-+_ <$ p-tok t-+))
p-, rec = chainr1 (p-+ rec)      (box (_s-,_ <$ p-tok t-,))
p-:: rec = chainr1 (p-, rec)     (box (_s-::_ <$ p-tok t-::))

p-list rec = (add-nil <$> (p-tok t-[ &> box (chainr1 (p-subexp rec)
             (box (_s-::_ <$ p-tok t-,))) <& box (p-tok t-])))
             <|>
             (s-nil <$ p-tok t-[ <& box (p-tok t-])) where
  add-nil : AST → AST
  add-nil (l s-:: r) = l s-:: (add-nil r)
  add-nil = _s-:: s-nil

p-ann rec = s-ann <$> p-subexp rec <*> box (p-tok t-: &> box p-type)
```

Code 4.11: Parsers of some nodes in our AST

There is no combinator in agdarsec that would allow us writing a parser that accepts an empty input, or an empty list to be more precise, since non-advancing parsers would violate totality. This aligns with our need for `s-λ`, since we expect at least one variable name to bind in a lambda expression. For this we use the `list⁺` combinator. Converting it back to a simple list with `toList` is just an implementation trick, a liberty we took, so we could more easily pattern match later in the scope checking step.

The case of `p-::`, `p-,` and `p-+` is similar to the binary nodes presented in our `p-type` example.

In `p-list` we handle the empty list and non-empty list cases separately. Since we cannot have a parser that accepts the empty string, we need to manually add `nil` to the end of our non-empty list case using `add-nil`.

Finally, to conclude our parsing stack, we present the definitions of `p-subexp` and `p-exp` on Code 4.12.

```
p-subexp rec =
  p-true       <|>
  p-false      <|>
  p-num        <|>
  p-var        <|>
  p-triv       <|>
  p-nil        <|>
  p-leaf   rec <|>
  p-parens rec

p-exp = fix _ $ λ rec →
  p-ann        rec <|>
  p-list       rec <|>
  p-node       rec <|>
  p-$          rec <|>
  p-∷          rec <|>
  p-ite        rec <|>
  p-isZero     rec <|>
  p-fst        rec <|>
  p-snd        rec <|>
  p-inl        rec <|>
  p-inr        rec <|>
  p-case       rec <|>
  p-λ          rec <|>
  p-iteℕ       rec <|>
  p-iteList    rec <|>
  p-iteTree    rec <|>
  p-head       rec <|>
  p-tail       rec <|>
  p-genStream  rec <|>
  p-put        rec <|>
  p-set        rec <|>
  p-get        rec <|>
  p-genMachine rec <|>
  p-subexp     rec
```

Code 4.12: p-subexp and p-exp parsers

`p-subexp` has all nullary nodes listed, because they do not need to be guarded by parentheses. `p-leaf` is included too, since it is guarded in itself, because of its unique prefix, e.g., leaves of `Tree ℕ` look like `"<42>"`. `p-exp` is the top-level parser of STLC.

Omitting the cumbersome work of argument instancing, we show the final parsing algorithm on Code 4.13.

```
module _ (open Parameters P)
  -- ...
  parse_by_ : ∀ {A : Set≤ l} → String → ∀[ Parser P A ] → Maybe (theSet A)
  parse s by parser =
   let input = Vec.fromList $ Tokenizer.fromText t s
       parse = runParser parser (n≤1+n _) (lift $ 𝕃.into input)
       check = λ s → if ⌊ Success.size s Nat.≟ 0 ⌋
                     then just (Success.value s) else nothing
   in case List.TraversableM.mapM MaybeCat.monad check $ runM ℝ parse of λ where
        (just (a ∷ _)) → just (lower a)
        _              → nothing
  parse-exp : String → Maybe AST
  parse-exp s = parse s by p-exp

parse : String → Maybe AST
parse = parse-exp
```

Code 4.13: Monadic total implementation of the parsing stack

Without going too much into detail, we can see that our function starts from a string (the source code) and a parser (`p-exp` in our case). It runs the injected tokenizer for which we use the one provided by agdarsec, parametrized with our three functions discussed in the previous chapter. It is also clear that the parsing can fail which is indicated by returning `nothing`. The need for the `n≤1+n` proof suggests that the internal code of agdarsec uses the fact that parsing always consumes some non-zero amount of characters from the input, thus reducing the sized type used.

```
_ : parse "10 + 20 + 30"   ≡ just (s-num 10 s-+ s-num 20 s-+ s-num 30)
_ = refl
_ : parse "(10 + 20) + 30" ≡ just ((s-num 10 s-+ s-num 20) s-+ s-num 30)
_ = refl
_ : parse "10 + (20 + 30)" ≡ just (s-num 10 s-+ (s-num 20 s-+ s-num 30))
_ = refl
_ : parse "λ f x. if f x then x else 0" ≡
          just (s-λ ("f" ∷ "x" ∷ []) (s-ite (s-var "f" s-$ s-var "x")
                                            (s-var "x") (s-num 0)))
_ = refl
```

Code 4.14: Parsing examples

4.3 Scope checking

```
data ABT (n : ℕ) : Set where
  abt-true        : ABT n
  abt-false       : ABT n
  abt-ite         : ABT n → ABT n → ABT n → ABT n
  abt-isZero      : ABT n → ABT n
  _abt-+_         : ABT n → ABT n → ABT n
  abt-num         : ℕ → ABT n
  abt-λ           : ABT (suc n) → ABT n
  _abt-$_         : ABT n → ABT n → ABT n
  abt-var         : Fin n → ABT n

  abt-triv        : ABT n
  _abt-,_         : ABT n → ABT n → ABT n
  abt-fst         : ABT n → ABT n
  abt-snd         : ABT n → ABT n

  abt-inl         : ABT n → ABT n
  abt-inr         : ABT n → ABT n
  abt-case        : ABT n → ABT n → ABT n

  abt-nil         : ABT n
  _abt-::_        : ABT n → ABT n → ABT n

  abt-leaf        : ABT n → ABT n
  _abt-node_      : ABT n → ABT n → ABT n

  abt-iteℕ        : ABT n → ABT n → ABT n → ABT n
  abt-iteList     : ABT n → ABT n → ABT n → ABT n
  abt-iteTree     : ABT n → ABT n → ABT n → ABT n

  abt-head        : ABT n → ABT n
  abt-tail        : ABT n → ABT n
  abt-genStream   : ABT n → ABT n → ABT n → ABT n

  abt-put         : ABT n → ABT n → ABT n
  abt-set         : ABT n → ABT n
  abt-get         : ABT n → ABT n
  abt-genMachine : ABT n → ABT n → ABT n → ABT n → ABT n

  abt-ann         : ABT n → SType → ABT n
```

Code 4.15: Syntax for abstract binding trees of STLC

The most apparent difference between our AST and ABT definitions is that the latter is indexed by ℕ. This index indicates the number of free variables in the expression, i.e., the size of context the corresponding typed term needs.

Another notable difference is in `abt-λ`. It no longer stores the list of variable names obtained from the source code. Instead, we unroll these shorthand lambdas to separate constructors which we will treat as binders that use De Bruijn indices [10]. For example, `s-λ ("x" ∷ "y" ∷ []) (s-var "x" s-+ s-var "y")` is turned into `abt-λ (abt-λ (abt-var 1 abt-+ abt-var 0))`. Note that the first variable `x` got the index one, and `y` got index zero. This is because De Bruijn indices indicate the *distance* from the represented variable's binder in the tree.

Naturally, our `abt-var` node is also indexed by natural numbers instead of strings. To be more rigorous, we chose to use `Fin n` (set of n elements, i.e., subset of naturals from 0 to n-1) in our definition `abt-var : Fin n → ABT n`, as can be seen on Code 4.15. This ensures that we cannot create an ABT which has a variable reference that is out of bounds, i.e., one that uses an index not present in its context.

Our scope checking formalisation rely on a few lemmas depicted on Codes 4.16 and 4.17. Here we omit the proofs for the arithmetic ones, but they can be found in our codebase.

```
lift-abt : {m n : ℕ} → (m ≤ n) → ABT m → ABT n
lift-abt m≤n (u abt-+ v) = (lift-abt m≤n u) abt-+ (lift-abt m≤n v)
lift-abt m≤n (abt-num n) = abt-num n
lift-abt m≤n (abt-λ t)   = abt-λ (lift-abt m≤n t)
lift-abt m≤n (u abt-$ v) = (lift-abt m≤n u) abt-$ (lift-abt m≤n v)
lift-abt {m} {n} m≤n (abt-var f) = abt-var (inject≤ {m} {n} f m≤n) where
  inject≤ : {m n : ℕ} → Fin m → m ≤ n → Fin n
  inject≤ {_} {suc n} zero    _     = Fin.zero
  inject≤ {_} {suc n} (suc i) (m≤n) = suc (inject≤ i m≤n)
-- ...
```

Code 4.16: Lemma: an ABT can always be embedded into a bigger context

```
≤-max2 : (n m : ℕ) → (n ≤ max n m) ×
                     (m ≤ max n m)

≤-max3 : (n m k : ℕ) → (n ≤ max (max n m) k) ×
                       (m ≤ max (max n m) k) ×
                       (k ≤ max (max n m) k)

≤-max4 : (n m k l : ℕ) → (n ≤ max (max (max n m) k) l) ×
                         (m ≤ max (max (max n m) k) l) ×
                         (k ≤ max (max (max n m) k) l) ×
                         (l ≤ max (max (max n m) k) l)
```

Code 4.17: Arithmetic lemmas needed for scope checking

```
scopeinfer : AST → Maybe (Σ ℕ λ n → ABT n)
scopeinfer = sinfer [] where

  sinfer : List String → AST → Maybe (Σ ℕ λ n → ABT n)

  sinfer ss s-true  = just (0 , abt-true)
  sinfer ss s-false = just (0 , abt-false)

  sinfer ss (s-ite t u v) with sinfer ss t | sinfer ss u | sinfer ss v
  ... | just (i , t') | just (j , u') | just (k , v') =
  just (max (max i j) k , abt-ite (lift-abt (π₁     (≤-max3 i j k))  t')
                                  (lift-abt (π₁ (π₂ (≤-max3 i j k))) u')
                                  (lift-abt (π₂ (π₂ (≤-max3 i j k))) v'))
  ... | _ | _ | _ = nothing
-- ...
  sinfer ss (s-λ       [] t) = sinfer ss t
  sinfer ss (s-λ (v ∷ vs) t) with sinfer (v ∷ ss) (s-λ vs t)
  ... | just (0     , t') = just (0 , abt-λ (lift-abt tt t'))
  ... | just (suc n , t') = just (n , abt-λ t')
  ... | nothing = nothing

  sinfer ss (u s-$ v) with sinfer ss u | sinfer ss v
  ... | just (i , u') | just (j , v') =
  just (max i j , _abt-$_ (lift-abt (π₁ (≤-max2 i j)) u')
                          (lift-abt (π₂ (≤-max2 i j)) v'))
  ... | _ | _ = nothing

  sinfer ss (s-var name) = case (lookup ss name) of λ where
    nothing  → nothing -- not in scope
    (just i) → just (suc i , abt-var (fromℕ i)) where
      fromℕ : (n : ℕ) → Fin (suc n)
      fromℕ = λ { 0 → Fin.zero ; (suc n) → suc (fromℕ n) }
```

Code 4.18: Scope inference algorithm

`scopeinfer` takes an `AST`, then if it is well-scoped, it returns a dependent pair: some natural `n` and an `ABT n`, otherwise it returns `nothing`. Basically what happens is that we decorate our syntax tree with context size indices. We use `sinfer` as a helper function to iterate over the tree. It accumulates the bound variable names in its first argument from the `s-λ` nodes. Then, at `s-var` nodes we look up the variable from the list: if not found, we return with error; otherwise we use the resulting list index as the De Bruijn index. Note that the `suc` for the ABT index is needed here only because `lookup` indexes from zero, i.e., a De Bruijn index 0 means that we need a context of at least size one.

For nodes with arity two or more, such as `_s-$_` and `s-ite` on Code 4.18, we recursively infer the scope of all subterms; choose the biggest scope size among them; then lift all operands to that scope for constructing the ABT node. To achieve this, we use our lemmas `≤-max2` and `≤-max3` which state that the maximum of two or three numbers is always larger than or equal to the individual numbers.

```
scopecheck : AST → Maybe (ABT 0)
scopecheck ast with scopeinfer ast
... | just (0 , abt) = just abt
... | _              = nothing
```

Code 4.19: Scope checking algorithm

The module's top-level function `scopecheck` simply runs the scope inference and checks whether we get a closed term, i.e., a binding tree with index zero. This is to ensure that there are no free variables in the term, or in other words: it is well-scoped. Note that, again we encode a strong requirement in our type: the scope checking cannot return a badly scoped tree by definition.

```
_ : scopecheck (s-var "foo") ≡ nothing
_ = refl
_ : scopecheck (s-λ ("foo" ∷ []) (s-var "bar")) ≡ nothing
_ = refl
_ : scopecheck (s-λ ("foo" ∷ []) (s-var "foo")) ≡ just (abt-λ (abt-var Fin.zero))
_ = refl
_ : scopecheck (s-λ ("x" ∷ "y" ∷ []) (s-var "x" s-+ s-var "y")) ≡
              just (abt-λ (abt-λ (abt-var (suc Fin.zero) abt-+ abt-var Fin.zero)))
_ = refl
```

Code 4.20: Scope checking examples

4.4 Bidirectional type checking

In this chapter we first give a brief introduction to our well-typed syntax for easier understanding. However, we will discuss our object theory in more detail in the next chapter.

```
data Ty    : Set where
  _⇒_      : Ty → Ty → Ty
  _×o_     : Ty → Ty → Ty
  Unit     : Ty
  _+o_     : Ty → Ty → Ty
  Empty    : Ty
  Bool     : Ty
  Nat      : Ty
  List     : Ty → Ty
  Tree     : Ty → Ty
  Stream   : Ty → Ty
  Machine  : Ty

data Con : Set where
  ◇      : Con
  _▹_    : Con → Ty → Con

Tm : Con → Ty → Set
-- ...
```

Code 4.21: Types, contexts and terms in the well-typed syntax

`Ty` has the exact same definition as our syntactic types `SType` as visible when comparing Code 4.21 with 4.8. Contexts (`Con`) are simply a list of types, where ◇ corresponds to nil, and `_▹_` is similar to the cons constructor, takes a prefix list and a type to form a new context. So ◇ represents the empty context, and `◇ ▹ Nat ▹ Bool` stands for the context that has a `Bool` and a `Nat` variable in De Bruijn indices zero and one, respectively. The indices are inverted, because when we append a new variable with a `λ` introduction it becomes De Bruijn index zero.

Terms (`Tm`) are indexed by their contexts and their types. Later we will see that we cannot construct arbitrary terms from arbitrary contexts. For example, the term `q`, which stands for the ”last bound variable”, cannot be constructed in the empty context.

Codes 4.22, 4.23 and 4.24 show portions from the code used for type checking: decidable equality over `Ty`; mapping from the syntactic `SType` to `Ty`; and a few additional helper functions, respectively.

```
_≟_ : (A B : Ty) → Dec (A ≡ B)
Nat  ≟ Nat       = yes refl
Nat  ≟ Bool      = no λ ()
Nat  ≟ Unit      = no λ ()
Nat  ≟ Empty     = no λ ()
Nat  ≟ (_ ⇒  _) = no λ ()
Nat  ≟ (_ ×o _) = no λ ()
Nat  ≟ (_ +o _) = no λ ()
-- ...
(A₁ ×o A₂) ≟ (B₁ ×o B₂) with A₁ ≟ B₁ | A₂ ≟ B₂
... | yes e₁ | yes e₂ = yes (cong₂ _×o_ e₁ e₂)
... | yes e₁ | no ¬e₂ = no λ hyp → ¬e₂ (cong ×snd hyp) where
  ×snd : Ty → Ty
  ×snd = λ { (A ×o B) → B ; X → X }
... | no ¬e₁ | yes e₂ = no λ hyp → ¬e₁ (cong ×fst hyp) where
  ×fst : Ty → Ty
  ×fst = λ { (A ×o B) → A ; X → X }
... | no ¬e₁ | no ¬e₂ = no λ hyp → ¬e₁ (cong ×fst hyp) where
  ×fst : Ty → Ty
  ×fst = λ { (A ×o B) → A ; X → X }
-- ...
```

Code 4.22: Decidable equality of types

```
infer-ty : SType → Maybe Ty
infer-ty s-Nat  = just Nat
infer-ty s-Bool = just Bool
infer-ty s-⊤    = just Unit
infer-ty s-⊥    = just Empty
infer-ty (A s-→ B) with infer-ty A | infer-ty B
... | just A' | just B' = just (A' ⇒ B')
... | _       | _       = nothing
```

Code 4.23: Mapping syntactic types SType to Ty

```
length : Con → ℕ
length = λ { ◇ → 0 ; (Γ ▹ _) → suc (length Γ) }

lookup : (Γ : Con) → Fin (length Γ) → (Σ Ty λ A → Tm Γ A)
lookup (Γ ▹ A) zero    = A , q
lookup (Γ ▹ A) (suc n) = proj₁ rest , proj₂ rest [ p ] where
  rest : Σ Ty λ A → Tm Γ A
  rest = lookup Γ n

fconv : {Γ : Con}{A B : Ty} → Tm Γ (A ⇒ B) → Tm (Γ ▹ A) B
fconv f = f [ p ] $ q
```

Code 4.24: Helper functions for type checking `lookup` is a safe function, due to the index argument's type `Fin (length Γ)` that searches the nth type from a `Con`. It also constructs the term that extracts said nth variable from the context. For this, it builds a chain of `[ p ]` substitutions with length equal to the queried index. Informally, all `[ p ]` substitutions "peel off" the last bound variable from the list. Finally, the function puts a `q` to the term, that extracts the last variable from the truncated list. For example, if we run `lookup Γ 3`, then we get a pair like: `A , q [ p ] [ p ] [ p ]`, where `A` is the type of the variable at De Bruijn index three in `Γ`. We have a shorthand notation for the previous term: `q [ p ⊚ p ⊚ p ]`.

`fconv` is interesting when we consider our introduction rule for abstraction:

```
lam : ∀{Γ A B} → Tm (Γ ▹ A) B → Tm Γ (A ⇒ B)
```

It means that when we can explain a term of type `B` using a context that is some `Γ` plus a variable `A`, we can also always explain a function in context `Γ` that goes from `A` to `B`. `fconv` proves the opposite direction by weakening the function's context in `f [ p ]` and then applying the first variable from this context to it with `$ q`. We explain `q`, `p`, `_[_]` and `_$_` alongside the whole substitution calculus in Chapter 4.5 in more detail.

Corollary: `Tm (Γ ▹ A) B` and `Tm Γ (A ⇒ B)` are isomorphic in our model.

Note the difference between `→` and `⇒` in the definition above. The former denotes function types in our metatheoretic language, while the latter means functions in our object theory.

Bidirectional type checking consists of the usual [11] two functions: `infer` and `check`. Both take an ABT term as input. The former, if well-typed, returns its inferred type and the corresponding well-typed term. The latter takes the expected type as argument and checks whether the input term has that type; if yes, it returns the well-typed term.

```
infer : (Γ : Con) → ABT (length Γ) → Maybe (Σ Ty λ A → Tm Γ A)
check : (Γ : Con) → (A : Ty) → ABT (length Γ) → Maybe (Tm Γ A)
```

Code 4.25: Types of functions infer and check

From Code 4.26 to 4.31, we present a non-exhaustive subset of our inference and checking rules.

```
infer Γ abt-true  = just (Bool , true)
infer Γ abt-false = just (Bool , false)

infer Γ (abt-isZero t) with infer Γ t
... | just (Nat , t') = just (Bool , iteNat true false t')
... | _               = nothing
```

Code 4.26: Type inference: some trivial cases

It is a recurring pattern that type inference of terms usually requires type inference of their subterm(s), like with `isZero` where the subterm must be a `Nat`.

```
infer Γ (abt-ite t u v) with infer Γ t | infer Γ u | infer Γ v
... | just (Bool , t') | just (A , u') | just (B , v') with B ≟ A
... | yes e = just (A , iteBool u' (transp (λ X → Tm Γ X) e v') t')
... | _     = nothing
infer Γ (abt-ite t u v) | _ | _ | _ = nothing

infer Γ (u abt-+ v) with infer Γ u | infer Γ v
... | just (Nat , u') | just (Nat , v') = just (Nat , iteNat v' (suco q) u')
... | _               | _               = nothing
```

Code 4.27: Type inference: if-then-else and addition

For `"if t then u else v"` constructions, we first check whether `t` is of type `Bool` and that `u` and `v` have the same type by using pattern matching. If all criteria hold, we use `Bool`'s iterator `iteBool` for building the well-typed term. It normalises to its first argument if the third does to `true`, or to the second otherwise. The use of `transp` in this proof and in ones to follow is simply needed for technical reasons.

```
transp : ∀{ℓ}{A : Set ℓ}{ℓ'}(P : A → Set ℓ'){a a' : A} → a ≡ a' → P a → P a'
```

With this we can transport propositions over *propositional equality* of `Ty`, because Agda does this implicitly only with its own *definitional equality*. More precisely, if we have a proof for `a ≡ a'` and that `P a` holds, `transp` gives us a proof of `P a'`. So here we rely on the correctness of the trivial ≟ proposition of `Ty` equality from before.

The addition operator is handled similarly, now using the iterator of naturals `iteNat`. Its first argument is the term that `zeroo` will be replaced with; the second is one that given a partial result of the iteration (in the context), produces the next result for each `suco`; and the third is the `Nat` that we wish to iterate on.

```
iteNat : ∀{Γ A} → Tm Γ A → Tm (Γ ▹ A) A → Tm Γ Nat → Tm Γ A
```

In case of our addition, we replace the `zeroo` in the first number with the second number itself, then for each `suco` in the first number we insert a `suco` into the result.

```
infer Γ (abt-λ t) = nothing

infer Γ (u abt-$ v)      with infer Γ u
... | just (A ⇒ B , u') with check Γ A v
... | just v' = just (B , u' $ v')
... | nothing = nothing
infer Γ (u abt-$ v) | _ = nothing

infer Γ (abt-var n) = just (lookup Γ n)
```

Code 4.28: Type inference: abstraction, application and variable lookup

We reject inference of abstractions by returning `nothing`, because we cannot determine types of bound variables in general. For example, the type of "`λx.x`" can be "`ℕ → ℕ`", "`𝕃 → 𝕃`", "`[ℕ] → [ℕ]`" or an infinite amount of other types. It is not only the identity function that is problematic. Any abstraction that does not use all its bound variables has a similar issue, like "`λx.10`", which can be "`ℕ → ℕ`", "`𝕃 → ℕ`", "`⊤ → ℕ`", etc. As a result, lambdas need to be type annotated in our language, or they must be present in a context where we expect a specific type of function through the usage of our `check` function.

When inferring an application, we first test whether the left-hand side is a function `A ⇒ B`, then we check if the right-hand side is of type `A`. If these hold, we can construct the term for application using `$`.

```
infer Γ (u abt-∷ v) with infer Γ u
... | just (A , u') with check Γ (Ty.List A) v
... | just v' = just (Ty.List A , cons u' v')
... | nothing = nothing
infer Γ (u abt-∷ v) | _ = nothing

check Γ (List A) abt-nil = just nil

check Γ A t with infer Γ t
... | nothing = nothing
... | just (B , t') with B ≟ A
... | yes e = just (transp (λ X → Tm Γ X) e t')
... | _      = nothing
```

Code 4.29: Type inference and checking: lists

We present the mutually recursive nature of `infer` and `check` on our list type.

To determine the type of `u :: v`, we first infer the type of the head `u` which will be some type `A`. Then we check if the rest of the list `v` is of type `List A`. If `v` is still a non-empty list, then our third case from Code 4.29 will run, which simply calls inference on the remainder list. `infer` and `check` will call each other until we reach our base case: the empty list. It is clear from the code attached above that no matter what type `A` is in `List A`, we can always construct the term `nil`. This is guaranteed by Agda's polymorphic nature as our meta language:

```
nil : ∀{Γ A} → Tm Γ (List A)
```

`List` is not the only type former for which we need explicit `check` cases, Code 4.30 shows a few more.

```
check Γ (A ⇒ B) (abt-λ t) with check (Γ ▹ A) B t
... | just t' = just (lam t')
... | nothing = nothing

check Γ (A +o _) (abt-inl t) with infer Γ t
... | just (B , t') with B ≟ A
... | yes e = just (inl (transp (λ X → Tm Γ X) e t'))
... | _      = nothing
check Γ (A +o _) (abt-inl t) | nothing = nothing

check Γ (_ +o A) (abt-inr t) with infer Γ t
... | just (B , t') with B ≟ A
... | yes e = just (inr (transp (λ X → Tm Γ X) e t'))
... | _      = nothing
check Γ (_ +o A) (abt-inr t) | nothing = nothing
```

Code 4.30: Type checking: functions and sums

Some `abt-λ t` term will be a valid term of `A ⇒ B` in context `Γ`, if and only if `t` is a valid term of type `B` in context `(Γ ▹ A)`. The extended context here contains the variable bound by the lambda.

Sum types are introduced by the left and right injection rules `inl` and `inr`. With the `(A +o _)` and `(_ +o A)` patterns in the snippet above, we indicate that the type checking does not consider the other types in the sum in these cases.

For annotated terms, we first use `infer-ty` to determine the type from the syntactic annotation. Then we test whether the subject term has that type using `check`, which simply involves an equality test of `Ty`, as seen on Code 4.31.

```
infer Γ (abt-ann t ty) with infer-ty ty
... | just A           with check Γ A t
... | just t' = just (A , t')
... | nothing = nothing
infer Γ (abt-ann t ty) | nothing = nothing

check Γ A (abt-ann t ty) with infer Γ (abt-ann t ty)
... | just (B , t') with B ≟ A
... | yes e = just (transp (λ X → Tm Γ X) e t')
... | no ¬e = nothing
check Γ A (abt-ann t ty) | nothing = nothing
```

Code 4.31: Type inference and checking: annotated terms

4.5 Algebraic definition quotiented by equations

We do not include the whole formalisation of our models neither in this nor in the following chapter since that would entail depicting nearly a thousand lines of code. Instead, we highlight the most fundamental and significant parts of our code, and ask the reader to visit our codebase if they look for a deeper review.

In Chapter 4.4, we already introduced the reader to some of the concepts in our object theory, such as `Ty`, `Con` and `Tm`. Now we will explain the full substitution calculus and some other quotients that formalise desired β and η equivalences between terms of our language. Structures like these are sometimes called *quotient inductive-inductive types*, or QIITs [12].

Each λ-abstraction binds a variable introducing a new scope. We can mathematically express this using substitutions. Usual notations for rewriting `(λx.M)N` to substitution form include `M[x := N]` and `M[x ↤ N]`. This means that we replace all occurrences of the variable `x` in `M` with `N`. If we have a scope with multiple bound variables, e.g., $(\lambda x.(\lambda y.(\lambda z.M)))\ N_1\ N_2\ N_3$, we may write $M[x := N_1,\ y := N_2,\ z := N_3]$. However, on this level we use De Bruijn indices, so we can omit the variable names: $M[N_1,\ N_2,\ N_3]$. The notation we employ is `t [u ,o v ,o w]` since we use the letters `t`, `u`, `v`, etc., for terms and `,o` for syntactic distinction between our object and meta languages.

```
Sub  : Con → Con → Set
_⊚_  : ∀{Γ Δ Θ} → Sub Δ Γ → Sub Θ Δ → Sub Θ Γ
ass  : ∀{Γ Δ Θ Ξ}{γ : Sub Δ Γ}{δ : Sub Θ Δ}{θ : Sub Ξ Θ} → (γ ⊚ δ) ⊚ θ ≡ γ ⊚ (δ ⊚ θ)
id   : ∀{Γ} → Sub Γ Γ
idl  : ∀{Γ Δ}{γ : Sub Δ Γ} → id ⊚ γ ≡ γ
idr  : ∀{Γ Δ}{γ : Sub Δ Γ} → γ ⊚ id ≡ γ

ε    : ∀{Γ} → Sub Γ ◇
◇η   : ∀{Γ}{σ : Sub Γ ◇} → σ ≡ ε

Tm   : Con → Ty → Set
_[_] : ∀{Γ Δ A} → Tm Γ A → Sub Δ Γ → Tm Δ A
[∘]  : ∀{Γ Δ Θ A}{t : Tm Γ A}{γ : Sub Δ Γ}{δ : Sub Θ Δ} →  t [ γ ⊚ δ ] ≡ t [ γ ]
                                                                   [ δ ]
[id] : ∀{Γ A}{t : Tm Γ A} → t [ id ] ≡ t
_,o_ : ∀{Γ Δ A} → Sub Δ Γ → Tm Δ A → Sub Δ (Γ ▹ A)
p    : ∀{Γ A} → Sub (Γ ▹ A) Γ
q    : ∀{Γ A} → Tm (Γ ▹ A) A
▹β₁  : ∀{Γ Δ A}{γ : Sub Δ Γ}{t : Tm Δ A} → p ⊚ (γ ,o t) ≡ γ
▹β₂  : ∀{Γ Δ A}{γ : Sub Δ Γ}{t : Tm Δ A} → q [ γ ,o t ] ≡ t
▹η   : ∀{Γ Δ A}{γa : Sub Δ (Γ ▹ A)} → p ⊚ γa ,o q [ γa ] ≡ γa
```

Code 4.32: The substitution calculus

Code 4.32 shows that substitution: `Sub` is between contexts: `Con`s. The above explained notation appears in `_[_]`. It means that if we have a term of type `A` in context `Γ` and a substitution from `Δ` to `Γ`, then we can construct a term of type `A` in `Δ`. This operator is often called *instantiation*.

Substitutions can be composed with the `_⊚_` operator which is associative, shown by `ass`. `[∘]` says that two consecutive substitutions are equivalent to the single substitution of their composition. We call the substitution that leaves the context untouched `id`. It is both a left and right identity of composition witnessed by `idl` and `idr`, respectively. `[id]` states that substitution with the identity does not change any term.

`ε` means that we can substitute any `Γ` to the empty context. The uniqueness rule `◇η` states that there is exactly one such substitution.

We gave an informal introduction to `p` and `q` in the previous chapter. Their *computation rules* or *beta-rules* - i.e., descriptions of what happens when we apply a destructor to a constructor - formally states what we explained. `p` removes the last bound variable, returning a truncated context; and `q` extracts the last variable. Both work only on non-empty contexts as seen in the common pattern `γ ,o t`.

Finally, we need one more uniqueness rule: `▹η`. It describes that given an arbitrary context, if we remove then reinsert the last variable, the result will only be equivalent to the original context.

Note that the above presented model is also often called a simply typed category with families, sCwF [13].

```
lam   : ∀{Γ A B} → Tm (Γ ▹ A) B → Tm Γ (A ⇒ B)
_$_   : ∀{Γ A B} → Tm Γ (A ⇒ B) → Tm Γ A → Tm Γ B
⇒β    : ∀{Γ A B}{t : Tm (Γ ▹ A) B}{u : Tm Γ A} → lam t $ u ≡ t [ id ,o u ]
⇒η    : ∀{Γ A B}{t : Tm Γ (A ⇒ B)} → lam (t [ p ] $ q) ≡ t
lam[] : ∀{Γ A B}{t : Tm (Γ ▹ A) B}{Δ}{γ : Sub Δ Γ} →
        (lam t) [ γ ] ≡ lam (t [ γ ⊚ p ,o q ])
$[]   : ∀{Γ A B}{t : Tm Γ (A ⇒ B)}{u : Tm Γ A}{Δ}{γ : Sub Δ Γ} →
             (t $ u) [ γ ] ≡ t [ γ ] $ u [ γ ]
```

Code 4.33: Quotients of abstraction and application

`⇒β` is the rule of β-reduction: applying some `t` on some `u` is equivalent to substituting all occurrences of `t`'s bound variable (i.e., term for De Bruijn index zero) with `u`.

`⇒η` formalises the eta conversion of lambda calculus: `λx.f x = f`, where `x` does not appear free in `f`. Naturally, we do not have to take the "does not appear free" part into consideration, since variable names are not used on this level of abstraction.

`lam[]` and `$[]` describe how to substitute abstractions and applications.

```
v0 : {Γ : Con} → {A : Ty} → Tm (Γ ▹ A) A
v0 = q
v1 : {Γ : Con} → {A B : Ty} → Tm (Γ ▹ A ▹ B) A
v1 = q [ p ]
v2 : {Γ : Con} → {A B C : Ty} → Tm (Γ ▹ A ▹ B ▹ C) A
v2 = q [ p ⊚ p ]
v3 : {Γ : Con} → {A B C D : Ty} → Tm (Γ ▹ A ▹ B ▹ C ▹ D) A
v3 = q [ p ⊚ p ⊚ p ]
```

Code 4.34: Terms for De Bruijn indexed variables

```
zeroo    : ∀{Γ} → Tm Γ Nat
suco     : ∀{Γ} → Tm Γ Nat → Tm Γ Nat
iteNat   : ∀{Γ A} → Tm Γ A → Tm (Γ ▹ A) A → Tm Γ Nat → Tm Γ A
Natβ₁    : ∀{Γ A}{u : Tm Γ A}{v : Tm (Γ ▹ A) A} → iteNat u v zeroo ≡ u
Natβ₂    : ∀{Γ A}{u : Tm Γ A}{v : Tm (Γ ▹ A) A}{t : Tm Γ Nat} →
           iteNat u v (suco t) ≡ v [ id ,o iteNat u v t ]
zero[]   : ∀{Γ Δ}{γ : Sub Δ Γ} → zeroo [ γ ] ≡ zeroo
suc[]    : ∀{Γ}{t : Tm Γ Nat}{Δ}{γ : Sub Δ Γ} → (suco t) [ γ ] ≡ suco (t [ γ ])
iteNat[] : ∀{Γ A}{u : Tm Γ A}{v : Tm (Γ ▹ A) A}{t : Tm Γ Nat}{Δ}{γ : Sub Δ Γ} →
           iteNat u v t [ γ ] ≡ iteNat (u [ γ ]) (v [ γ ⊚ p ,o q ]) (t [ γ ])
```

Code 4.35: Quotients of natural numbers

On Code 4.35, the computation rules `Natβ₁` and `Natβ₂` show the iteration semantics we introduced in Chapter 4.4: when we reach `zeroo`, we normalise to the first argument; otherwise for each `suco`, we insert an application of the second argument. In the latter case we see the structurally reducing (and thus total) recursive call, i.e., on the left we see `suco t`, and on the right it is `t`. `zero[]`, `suc[]` and `iteNat[]` are the substitution rules for these respective constructs. Equalities for boolean terms on Code 4.36 are similar.

```
true      : ∀{Γ} → Tm Γ Bool
false     : ∀{Γ} → Tm Γ Bool
iteBool   : ∀{Γ A} → Tm Γ A → Tm Γ A → Tm Γ Bool → Tm Γ A
Boolβ₁    : ∀{Γ A u v} → iteBool {Γ}{A} u v true ≡ u
Boolβ₂    : ∀{Γ A u v} → iteBool {Γ}{A} u v false ≡ v
true[]    : ∀{Γ Δ}{γ : Sub Δ Γ} → true [ γ ] ≡ true
false[]   : ∀{Γ Δ}{γ : Sub Δ Γ} → false [ γ ] ≡ false
iteBool[] : ∀{Γ A t u v Δ}{γ : Sub Δ Γ} →
            iteBool {Γ}{A} u v t [ γ ] ≡ iteBool (u [ γ ]) (v [ γ ]) (t [ γ ])
```

Code 4.36: Quotients of booleans

The equalities we decorate our algebra with can be viewed as a set of *inference rules* that entails a form of *structural operational semantics* [14]. In practice, this means that we can transform terms of the well-typed syntax along series of equalities using Agda's *equational reasoning*. This logic stands as the first building block towards normalisation and semantical meaning of programs.

Code 4.37 presents an example where we prove that the well-typed term compiled from `"if isZero 1 then 0 else 1+1"` is equivalent to the one compiled from `"2"`.

```
s = suco
z = zeroo
--          0             1+1                     isZero      1        2
--          - -------------------------  ------------------------    ------
eq : iteBool z (iteNat (s z) (s q) (s z)) (iteNat true false (s z)) ≡ s (s z)
eq = iteBool z (iteNat (s z) (s q) (s z)) (iteNat true false (s z))
          ≡⟨ cong (λ X → iteBool z (iteNat (s z) (s q) (s z)) X) Natβ₂ ⟩
     iteBool z (iteNat (s z) (s q) (s z)) (false [ id ,o iteNat true false z ])
          ≡⟨ cong (λ X → iteBool z (iteNat (s z) (s q) (s z)) X) false[] ⟩
     iteBool z (iteNat (s z) (s q) (s z)) false
          ≡⟨ Boolβ₂ ⟩
     iteNat (s z) (s q) (s z)
          ≡⟨ Natβ₂ ⟩
     s q [ id ,o iteNat (s z) (s q) z ]
          ≡⟨ suc[] ⟩
     s (q [ id ,o iteNat (s z) (s q) z ])
          ≡⟨ cong (λ X → s X) ▹β₂ ⟩
     s (iteNat (s z) (s q) z)
          ≡⟨ cong (λ X → s X) Natβ₁ ⟩
     s (s z)
          ∎
```

Code 4.37: Proof of semantic equivalence using equational reasoning

In our proof, we use computation rules `Natβ₁`, `Natβ₂`, `Boolβ₂`; as well as substitution rules `false[]`, `suc[]` and `▹β₂`.

Firstly, we rewrite `isZero 1` to `false`. This leaves us with a non-empty substitution in `false [ id ,o iteNat true false z ]`, but `false` is a constant function witnessed by `false[]`, so we can drop the context. Now that we know that the condition is false, we can use `Boolβ₂` to discard the true branch. Finally, we have to compute the `1+1` addition using the iteration rules `Natβ₂` and `Natβ₁`. This also involves `suc[]`, which is similar to `false[]`, since natural numbers are constant terms. We employ `▹β₂` to access the recursive result of the iteration, because opposed to the previous one, this is a two-step iteration.

Note that the quotients do not specify any notion of order between these rewriting steps. For example, we could have started with evaluating the `"1+1"` part of the term, and only then compute the `"isZero 1"` condition. This would have led to the same result, albeit with a different proof in terms of the order of steps. This is similar to the observations in lambda calculus: λ-terms can have multiple β-reduction strategies. The Church-Rosser theorem states that all reduction will eventually reduce to the same term. Of course, we are not yet talking about *reduction* here, since we can apply equalities in arbitrary directions, even in a back and forth manner to write infinite proofs. So when writing proofs by hand on this level as above, we have to be more cautious.

Another aspect of ordering is *optimisation*. Lazy programming languages usually do not evaluate all sub-expressions of constructs like if-then-else, only the condition first, and then the relevant branch. Our proof would have been a lot longer too if instead of `"0"`, we had had a much larger term as the true branch and were inclined to compute it unnecessarily.

4.6 Standard interpretation and normalisation

In the previous chapter we introduced the *syntax*, i.e., *initial model*, of our language. It is initial, because there is a homomorphism from here to any model [1]. What makes an algebraic structure *model* of STLC is that all sorts are specified with types and all equations hold, i.e., we provide proofs for them.

In this chapter we present a model we call *standard model*. It provides the standard meta-language interpretation of our well-typed terms. In essence, we map our object theoretic concepts to our meta theory, i.e., Agda. We can call this a form of evaluation, because the resulting Agda terms can be normalised by Agda. For example, the standard model interpretation of `iteBool zeroo (suco zeroo) false` is `if false then 0 else 1`, which Agda normalises to `1`.

We denote *rewriting*, often called *interpretation*, by enclosing our syntactic terms - terms of the initial model, `I` - in ⟦_⟧. For technical reasons, we have separate operators for all sorts, e.g., `⟦_⟧T` for rewriting types, `⟦_⟧C` for contexts, etc. We use Agda's `REWRITE` pragma as seen on Code 4.38.

```
⟦_⟧T : I.Ty → Ty
⟦_⟧C : I.Con → Con
⟦_⟧S : ∀{Γ Δ} → I.Sub Δ Γ → Sub ⟦ Δ ⟧C ⟦ Γ ⟧C
⟦_⟧t : ∀{Γ A} → I.Tm  Γ A → Tm  ⟦ Γ ⟧C ⟦ A ⟧T
-- ...
⟦zero⟧   : ∀{Γ}                       → ⟦ I.zeroo {Γ} ⟧t    ≈ zeroo
⟦suc⟧    : ∀{Γ}{t : I.Tm Γ I.Nat} → ⟦ I.suco t ⟧t       ≈ suco ⟦ t ⟧t
⟦iteNat⟧ : ∀{Γ A}{u : I.Tm Γ A}{v : I.Tm (Γ I.▹ A) A}{t : I.Tm Γ I.Nat} →
                                     ⟦ I.iteNat u v t ⟧t ≈ iteNat ⟦ u ⟧t ⟦ v ⟧t ⟦ t ⟧t
{-# REWRITE ⟦zero⟧ ⟦suc⟧ ⟦iteNat⟧ #-}
```

Code 4.38: Rewriting rules for interpretation

```
St : Model
St = record
  { Con        = Set
  ; Sub        = λ Δ Γ → Δ → Γ
  ; Ty         = Set

  ; Tm         = λ Γ A → Γ → A
  ; _▹_        = _×_
  ; _,o_       = λ γ t δ* → γ δ* , t δ*
  ; p          = π₁
  ; q          = π₂
  ; ▹β₁        = λ {Γ}{Δ} → refl {A = Δ → Γ}
  ; ▹β₂        = λ {Γ}{Δ}{A} → refl {A = Δ → A}
  ; ▹η         = λ {Γ}{Δ}{A} → refl {A = Δ → Γ × A}

  ; _⇒_        = λ A B → A → B
  ; lam        = λ t γ* α* → t (γ* , α*)
  ; _$_        = λ t u γ* → t γ* (u γ*)
  ; ⇒β         = λ {Γ}{A}{B}{t}{u} → refl {A = Γ → B}
  ; ⇒η         = λ {Γ}{A}{B}{t} → refl {A = Γ → A → B}
  ; lam[]      = λ {Γ}{A}{B}{t}{Δ}{γ} → refl {A = Δ → A → B}
  ; $[]        = λ {Γ}{A}{B}{t}{u}{Δ}{γ} → refl {A = Δ → B}

  ; Bool       = 𝟚
  ; true       = λ _ → tt
  ; false      = λ _ → ff
  ; iteBool    = λ u v t γ* → if t γ* then u γ* else v γ*
  ; Boolβ₁     = refl
  ; Boolβ₂     = refl
  ; true[]     = refl
  ; false[]    = refl
  ; iteBool[] = refl
```

Code 4.39: Portions from the standard model

Most of the standard model is self-evident: we map booleans to Agda's bools, naturals to Agda's Peano numbers, etc. We interpret functions: `A ⇒ B` using the meta theoretic function space: `A → B`. Contexts are implemented using Agda's products, for example, `[t ,o u ,o v]` is mapped to `(t , u) , v`. The terms `p` and `q` then become `π₁` and `π₂`, i.e., left and right projections, respectively. Equality proofs are all provided by `refl`.

We define auxiliary types in Agda for our slightly more complex constructions, like the inductive `List` and `Tree`, or the coinductive `Stream` and `Machine`. We will discuss the semantics of these alongside their examples in Chapter 4.8.

Our elaborator uses the standard model interpretation and Agda's built-in normalisation for evaluating the compiled programs. For insights about *implementing normalisation*, we direct our reader to Ambrus Kaposi's work [1] which we based our elaborator on. It contains normalisation, albeit only up until a subset of our language as of writing this thesis, not including function space, for example.

4.7 Running the elaborator

`Program` is a dependent pair: a type `A` and a term of type `A` that is valid in the empty context `◇`. `Evaluation` is the interpretation of a program in the standard model which Agda can normalise for us. `ProgEval` is the combination of the two: a term of some type `A` along with its evaluation in the standard model.

```
Program    = Σ Ty λ A → Tm ◇ A
Evaluation = Σ Ty λ A → St.Tm St.⟦ ◇ ⟧C St.⟦ A ⟧T
ProgEval   = Σ Ty λ A → (Tm ◇ A × St.Tm St.⟦ ◇ ⟧C St.⟦ A ⟧T)
```

Code 4.40: Top-level return types of the elaboration

Some readers might question why we have not defined `ProgEval` as `Program × Evaluation`. In fact, that was our very first thought for this type. However, we realised that it means a weaker type which would, in theory, allow results like `(A , Tm) , (B , Eval)` where `A` and `B` are separate types. Using our definition we always get `A , Tm , Eval` where the type of `Tm` is `A`, and the type of `Eval` is `St.⟦ A ⟧T`, i.e., the type in the standard model that corresponds to `A`.

```
elaborate : String → Maybe ((Data.List.List Token) × Maybe (AST ×
                     Maybe (ABT 0 × Maybe ProgEval)))
elaborate code = case just (tokenize code)                    of λ where
  nothing               → nothing
  (just tokens)         → case parse code                     of λ where
    nothing             → just (tokens , nothing)
    (just ast)          → case scopecheck ast                 of λ where
      nothing           → just (tokens , just (ast , nothing))
      (just abt)        → case infer ◇ abt                    of λ where
        nothing         → just (tokens , just (ast , just (abt , nothing)))
        (just (A , tm)) → just (tokens , just (ast , just (abt ,
                          just (A , (tm , St.⟦ tm ⟧t)))))
```

Code 4.41: Combining the elaboration stack `elaborate` employs all the conversions between our levels of abstractions that we discussed in the previous chapters. It runs the steps until any one of them returns an error, i.e., `nothing`, or until we reach the well-typed term level. In the latter case, we can always evaluate our resulting term using the standard model interpretation. This is guaranteed by the total definition of `St.⟦_⟧t`, which always returns a `Tm` with some type `A`, in some `Γ` context. A watchful reader will see that `Γ` will always be `◇` in our case.

We chose to return all intermediate results of our process for the sake of transparency to the user, i.e., all levels of abstractions are automatically available for inspection. Also, even if the whole compilation fails, we still return partial results up until the point of the error for easy debugging.

One questionable aspect in the code above is that `tokenize code` is wrapped in `just`. It is because of the issue we mentioned in Chapter 4.1: the lexer provided by agdarsec cannot fail. As a result, `lexical-error` below is only a placeholder as of writing this thesis.

```
_ : elaborate "(λ x. isZero x) : ℕ → 𝕃" ≡ just (
 (record { line = 0 ; offset =  0 } , t-lpar   ) ∷
 (record { line = 0 ; offset =  1 } , t-λ      ) ∷
 (record { line = 0 ; offset =  3 } , t-var "x") ∷
 (record { line = 0 ; offset =  4 } , t-dot    ) ∷
 (record { line = 0 ; offset =  6 } , t-isZero ) ∷
 (record { line = 0 ; offset = 13 } , t-var "x") ∷
 (record { line = 0 ; offset = 14 } , t-rpar   ) ∷
 (record { line = 0 ; offset = 16 } , t-:      ) ∷
 (record { line = 0 ; offset = 18 } , t-ℕ      ) ∷
 (record { line = 0 ; offset = 20 } , t-→      ) ∷
 (record { line = 0 ; offset = 22 } , t-𝕃      ) ∷ [] ,
 just (s-ann (s-λ ("x" ∷ []) (s-isZero (s-var "x"))) (s-Nat s-→ s-Bool) ,
 just (abt-ann (abt-λ (abt-isZero (abt-var Fin.zero))) (s-Nat s-→ s-Bool) ,
 just (Nat ⇒ Bool , lam (iteNat true false q) ,
                   λ γ* x → iteℕ tt (λ _ → ff) x))))
_ = refl
```

Code 4.42: Elaboration example

```
data Error : Set where
  lexical-error : Error
  syntax-error  : Error
  scope-error   : Error
  type-error    : Error
```

Code 4.43: Type for errors

Our `Error` type is used like an enumeration for representing the error cases we get in each elaboration step.

An improved implementation could one day include arguments to some of these constructors for showing more specific information about certain errors to the user. For example, the starting character position of an invalid token; the variable name that is not in scope; or the expected and present types at mismatching types.

```
compile-eval : String → ProgEval ⊎ Error
compile-eval code = case elaborate code of λ where
  nothing                                           → inj₂ lexical-error
  (just (_  , nothing))                             → inj₂ syntax-error
  (just (_ , just (_ , nothing)))                   → inj₂ scope-error
  (just (_ , just (_ , just(_ , nothing))))         → inj₂ type-error
  (just (_ , just (_ , just(_ , just tm-eval)))) → inj₁ tm-eval

compile : String → Program ⊎ Error
compile code = case compile-eval code of λ where
  (inj₂ error)         → inj₂ error
  (inj₁ (A , tm , _)) → inj₁ (A , tm)

eval : String → Evaluation ⊎ Error
eval code = case compile-eval code of λ where
  (inj₂ error)           → inj₂ error
  (inj₁ (A , _ , eval)) → inj₁ (A , eval)

compileM : String → Maybe Program
compileM code = case compile code of λ where
  (inj₂ _)   → nothing
  (inj₁ tm) → just tm

evalM : String → Maybe Evaluation
evalM code = case eval code of λ where
  (inj₂ _)     → nothing
  (inj₁ eval) → just eval
```

Code 4.44: User-level functions built on top of elaborate

The set of functions presented on Code 4.44 provides a convenient interface to the user for compiling source code and evaluating programs. `compile-eval` returns both the compilation and the evaluation results, or an error as a right injection for incorrect programs. `compile` and `eval` work the same way, except they return only the well-typed term and its normalised Agda interpretation, respectively. Finally, `compileM` and `evalM` can be used, when the user would trade off the knowledge about the kind of potential errors for `Maybe` monadic results.

4.8 Examples

We hand-picked a few from our more interesting examples, that we had tested our elaborator with. These, as well as a collection of similar tests, are publicly available in our codebase for those who are interested in gaining further insight of our formalised language.

Note that tests called "_" are all proven with "`_ = refl`" lines, which we omit from the following snippets for the sake of brevity.

```
_ : compile-eval "(λx.x) : ℕ → ℕ" ≡ compile-eval "(λy.y) : ℕ → ℕ"

_ : compile-eval "(λ       x y z. x+y+z) : ℕ → ℕ → ℕ → ℕ" ≡
    compile-eval "(λ x. λ y. λ z. x+y+z) : ℕ → ℕ → ℕ → ℕ"
```

Code 4.45: Example: Alpha equivalence and unrolling lambda notation

```
not  = "((λ a. if a then false else true) : 𝕃 → 𝕃)"

even = "((λ x. iteℕ true (λa." ++ not ++ "a) x) : ℕ → 𝕃)"

odd = "(λ x. " ++ not ++ "(" ++ even ++ "x)) : ℕ → 𝕃"

_ : eval (even ++ₛ "3") ≡ inj₁ (Bool , λ γ* → ff)
_ : eval (odd  ++ₛ "3") ≡ inj₁ (Bool , λ γ* → tt)
```

Code 4.46: Example: even and odd functions

First we define boolean negation `not` in a standard way using the `Bool` elimination rule. For `even`, we use the iterator of ℕ to replace `zero` with `true` and each `suc` with a negation, essentially rewriting the structure, e.g. `suc(suc(suc(zero))) → ¬(¬(¬(true)))`. We simply add one more negation on top in `odd`.

```
xor = "(λ a b.            \
\         if a then        \
\           if b then      \
\             false        \
\           else           \
\             true         \
\         else if b then   \
\           true           \
\         else             \
\           false)         : 𝕃 → 𝕃 → 𝕃"

_ : compile-eval xor ≡ inj₁ (Bool ⇒ Bool ⇒ Bool
    , lam (lam (iteBool
      (iteBool false true q) (iteBool true false q) (q [ p ])))
    , λ γ* a b → if a then if b then ff else tt else (if b then tt else ff))

_ : eval (xor ++ₛ "false" ++ₛ "false") ≡ inj₁ (Bool , λ γ* → ff)
_ : eval (xor ++ₛ "true"  ++ₛ "false") ≡ inj₁ (Bool , λ γ* → tt)
_ : eval (xor ++ₛ "false" ++ₛ "true")  ≡ inj₁ (Bool , λ γ* → tt)
_ : eval (xor ++ₛ "true"  ++ₛ "true")  ≡ inj₁ (Bool , λ γ* → ff)
```

Code 4.47: Example: xor function

The `xor` example shows that we can write multi-line source code, and that we can nest operators like `if_then_else_`. Note that the normalised Agda interpretation syntactically looks almost the same as the source code for our language.

At this point, a curious reader might wonder why all evaluation results start with `"λ γ* → "`. The `γ*` stands for the term's context, however it is never used in any resulting term. This means that the term following the arrow can be put into an arbitrary context. The reason is that we require our compiled λ-terms to be closed as discussed at scope checking. As a result, the empty context suffices, and evidently any larger one does as well.

```
_ : compile-eval "1,2" ≡ inj₁ (Nat ×o Nat
                   , ⟨ suco zeroo , suco (suco zeroo) ⟩ , λ γ* → 1 , 2)
_ : compile-eval "0 , false" ≡ inj₁ (Nat ×o Bool
                   , ⟨ zeroo , false ⟩ , λ γ* → 0 , ff)
_ : compile-eval "1, trivial, (isZero 1)" ≡ inj₁ (Nat ×o (Unit ×o Bool)
               , ⟨ suco zeroo , ⟨ trivial , iteNat true false (suco zeroo) ⟩ ⟩
               , λ γ* → 1 , triv , ff)
```

Code 4.48: Example: constructing products

```
_ : compile-eval "fst (2,3)" ≡
    inj₁ (Nat , fst ⟨ suco (suco zeroo) , suco (suco (suco zeroo)) ⟩ , λ γ* → 2)
_ : compile-eval "snd (2,3)" ≡
    inj₁ (Nat , snd ⟨ suco (suco zeroo) , suco (suco (suco zeroo)) ⟩ , λ γ* → 3)

third = "(λ t. snd snd t) : (ℕ × ℕ × ℕ → ℕ)"

_ : compile-eval third ≡ inj₁ (Nat ×o (Nat ×o Nat) ⇒ Nat
                         , lam (snd (snd q)) , λ γ* t → π₂ (π₂ t))

_ : eval (third ++ₛ "10, 20, 30") ≡ inj₁ (Nat , (λ γ* → 30))
_ : eval (third ++ₛ "5+5, 42, (if (isZero 0) then (6+2) else 0)") ≡
                                inj₁ (Nat , (λ γ* → 8))
```

Code 4.49: Example: destructing products

We can use products to construct pairs, triples or tuples of arbitrary arity by chaining the _,_ constructor. They can be destructed with the `fst` and `snd` elimination rules or a proper chain of them like in our `third` function's implementation.

```
curry = "(λ f. λ x y. f (x,y)) : (ℕ × ℕ → ℕ) → (ℕ → ℕ → ℕ)"

uncurry = "(λ f. λ p. (f (fst p)) (snd p)) : (ℕ → ℕ → ℕ) → (ℕ × ℕ → ℕ)"

add = "(λ x y. x + y) : ℕ → ℕ → ℕ"

_ : compile-eval curry ≡ inj₁ (((Nat ×o Nat) ⇒ Nat) ⇒ (Nat ⇒ Nat ⇒ Nat)
                         , lam (lam (lam (q [ p ] [ p ] $ ⟨ q [ p ] , q ⟩)))
                         , λ γ* f x y → f (x , y))
_ : compile-eval uncurry ≡ inj₁ ((Nat ⇒ Nat ⇒ Nat) ⇒ ((Nat ×o Nat) ⇒ Nat)
                         , lam (lam (q [ p ] $ fst q $ snd q))
                         , λ γ* f p → f (π₁ p) (π₂ p))

_ : eval (uncurry ++ₛ add ++ₛ "(3 , 4)") ≡ inj₁ (Nat , (λ γ* → 7))

_ : eval (curry ++ₛ (uncurry ++ₛ add)) ≡ eval add
```

Code 4.50: Example: implementing curry and uncurry

On Code 4.50, we show currying and uncurrying, i.e., proofs that `n`-ary functions `(n ∈ ℕ⁺)` are isomorphic to unary higher-order ones that return `(n-1)`-ary functions. This means that applying `uncurry` then `curry` on any curried function gets us the original function, as seen on the example with `add`.

For lists, both type annotations and terms support two syntaxes. Types can be written as `"List T"` or `"[T]"` for any type `T`; and terms like `"2 :: 1 :: 0 :: nil"` or

`"[2, 1, 0]"`, i.e., both Agda and Haskell-like forms are valid. We can create nested lists. A list of type `"List (List ℕ)"` can be `"(0 ∷ 2 ∷ nil) ∷ (1 ∷ nil) ∷ nil"`, for example. A limitation from our parser is that nested lists are not supported by our Haskell-like syntax as of writing this thesis.

```
isnil = "(λ xs. iteList true (λ _ _.false) xs) : [ℕ] → 𝕃"

length = "(λ xs. iteList 0 (λ _ x. x+1) xs) : [ℕ] → ℕ"

sum = "(λ xs. iteList 0 (λ x y. x + y) xs) : [ℕ] → ℕ"

concat = "(λ xs ys. iteList ys (λ a as. a ∷ as) xs) : [ℕ] → [ℕ] → [ℕ]"

headM = "(λ xs. iteList ((inl trivial) : ⊤ ⊎ ℕ)
                        (λ a as. ((inr a) : ⊤ ⊎ ℕ))
                        xs) : [ℕ] → ⊤ ⊎ ℕ"

filter = "(λ f xs. iteList (nil : [ℕ])
                           (λ a as. if (f a) then a ∷ as else as)
                           xs) : (ℕ → 𝕃) → [ℕ] → [ℕ]"

map = "(λ f xs. iteList (nil : [ℕ]) (λ a as. (f a) ∷ as) xs)
      : (ℕ → ℕ) → [ℕ] → [ℕ]"

replicate = "(λ n x. iteℕ (nil : [ℕ]) (λ xs. x ∷ xs) n) : ℕ → ℕ → [ℕ]"
```

Code 4.51: Example: list operations

Since we do not support type polymorphism, we decided to implement all generic list functions for `[ℕ]`.

The iterator of lists: `iteList` can fold the list to an arbitrary result type, e.g. 𝕃, ℕ or `[ℕ]`, as seen on the examples above. Its first argument tells what to return for the empty list. The second turns a partial result and the next list element of the iteration to a new result. Finally, the third argument is the list itself we wish to fold.

Note that, unlike most of the time with function types, here we do not have to annotate the type of the second argument. This is because we already annotate the outer functions, so result types are propagated during type inference to the `iteList` terms, for which we only need to *check* whether the types of all arguments match.

`headM` also serves as an example of our sum types. It is a total implementation of the *head* operation, i.e., the function that returns the first element of lists. Here we simulate `Maybe` with the `⊤ ⊎ ℕ` sum. `nil` case in the iteration builds a left injection:

`inl trivial`; and the non-empty list case immediately packs the head element to a right injection: `inr a`, discarding the recursive result `as` of the remainder list.

The reader can find the compilation results in our codebase of all list operations above. On Code 4.52 we only present some evaluation results.

```
_ : eval (isnil ++ₛ "[]")  ≡ inj₁ (Bool , λ γ* → tt)
_ : eval (isnil ++ₛ "[0]") ≡ inj₁ (Bool , λ γ* → ff)

_ : eval (length ++ₛ "[]")          ≡ inj₁ (Nat , λ γ* → 0)
_ : eval (length ++ₛ "[1,2,3,4,5]") ≡ inj₁ (Nat , λ γ* → 5)

_ : eval (sum ++ₛ "[]")              ≡ inj₁ (Nat , λ γ* → 0)
_ : eval (sum ++ₛ "[10, 7, 20, 1]") ≡ inj₁ (Nat , λ γ* → 38)

_ : eval (concat ++ₛ "[]" ++ₛ "[]")          ≡ inj₁ (Ty.List Nat , λ γ* → [])
_ : eval (concat ++ₛ "[3,1]" ++ₛ "[4,1,5]") ≡ inj₁ (Ty.List Nat ,
                                            λ γ* → 3 ∷ (1 ∷ (4 ∷ (1 ∷ (5 ∷ [])))))

_ : eval (headM ++ₛ "[]")      ≡ inj₁ (Unit +o Nat , λ γ* → inj₁ triv)
_ : eval (headM ++ₛ "[2,4,6]") ≡ inj₁ (Unit +o Nat , λ γ* → inj₂ 2)

_ : eval (filter ++ₛ even ++ₛ "[]") ≡ inj₁ (Ty.List Nat , λ γ* → [])
_ : eval (filter ++ₛ even ++ₛ "[1,2,3,4,5,6,7,8]") ≡ inj₁ (Ty.List Nat ,
                                                  λ γ* → 2 ∷ (4 ∷ (6 ∷ (8 ∷ []))))

_ : eval (map ++ₛ double ++ₛ "[3,0,11,23]") ≡ inj₁ (Ty.List Nat ,
                                              λ γ* → 6 ∷ (0 ∷ (22 ∷ (46 ∷ []))))
_ : eval (map ++ₛ double ++ₛ "[]") ≡ inj₁ (Ty.List Nat , λ γ* → [])

_ : eval (replicate ++ₛ "4" ++ₛ "42") ≡ inj₁ (Ty.List Nat ,
                                        λ γ* → 42 ∷ (42 ∷ (42 ∷ (42 ∷ []))))
_ : eval (replicate ++ₛ "0" ++ₛ "42") ≡ inj₁ (Ty.List Nat , λ γ* → [])
```

Code 4.52: Example: evaluation of list operations

Binary trees are inductive types similar to lists. They have two constructors: *leaf* and *node*, in our syntax <_> and _|_, respectively. Branching can be controlled using parentheses as shown on Figure 4.3.

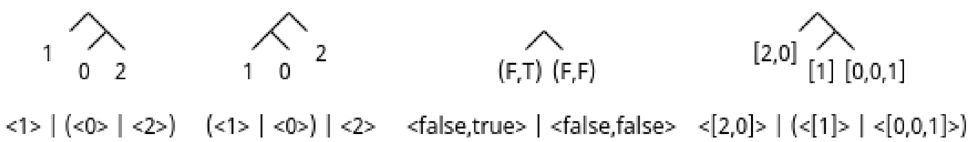

Figure 4.3: Building binary trees of naturals, pairs and lists

Iteration on trees is similar to iteration on lists. The first argument of `iteTree` is a function that can turn a leaf of type `A` to some term `B`. The second argument is a function that, given the two partial results from the recursive processing of the left and right subtrees, gives us a new result. Finally, the third argument is the tree itself to iterate. Code 4.53 shows some example computations on trees.

```
size = "(λ t. iteTree ((λ_. 1):ℕ→ℕ) (λ l r. l + r) t) : (Tree ℕ) → ℕ"

_ : eval (size ++ₛ "<3>")                          ≡ inj₁ (Nat , λ γ* → 1)
_ : eval (size ++ₛ "<3> | ((<10> | <2>) | <3>)") ≡ inj₁ (Nat , λ γ* → 4)

sum = "(λ t. iteTree ((λx. x):ℕ→ℕ) (λ l r. l + r) t) : (Tree ℕ) → ℕ"

_ : eval (sum  ++ₛ "<3>")                          ≡ inj₁ (Nat , λ γ* → 3)
_ : eval (sum  ++ₛ "<3> | ((<10> | <2>) | <3>)") ≡ inj₁ (Nat , λ γ* → 18)

map = "(λ f t. iteTree ((λx.<f x>):ℕ→(Tree 𝕃)) (λ l r. l | r) t)
      : (ℕ → 𝕃) → (Tree ℕ) → (Tree 𝕃)"

_ : eval (map ++ₛ even ++ₛ "(<0> | <7>) | (<1> | <10>)") ≡
    inj₁ (Ty.Tree Bool ,
    λ γ* → Tree.node (Tree.node (Tree.leaf tt) (Tree.leaf ff))
                     (Tree.node (Tree.leaf ff) (Tree.leaf tt)))
```

Code 4.53: Example: tree operations and their evaluation

Coinductive types are the mathematical dual of inductive types. They are specified by their destructors, often called *observers*, which most of the time either return a simpler type, or a new copy of the original object with a different inner state. Our STLC implementation features two coinductive type formers: `Stream` and `Machine`.

```
evens = "genStream ((λn.n):ℕ→ℕ) (λn.n+2) 0"
odds  = "genStream ((λn.n):ℕ→ℕ) (λn.n+2) 1"

_ : eval ("head"           ++ evens) ≡ inj₁ (Nat , λ γ* → 0)
_ : eval ("head tail"      ++ evens) ≡ inj₁ (Nat , λ γ* → 2)
_ : eval ("head tail tail" ++ evens) ≡ inj₁ (Nat , λ γ* → 4)
_ : eval ("head"           ++ odds)  ≡ inj₁ (Nat , λ γ* → 1)
_ : eval ("head tail"      ++ odds)  ≡ inj₁ (Nat , λ γ* → 3)
_ : eval ("head tail tail" ++ odds)  ≡ inj₁ (Nat , λ γ* → 5)
```

Code 4.54: Example: infinite streams of even and odd numbers

`genStream` introduces stream types from three arguments. A *seed* or *initial state* which is the third argument. A function which specifies how the `head` eliminator must turn a stream state to a result. Finally, a specification for the `tail` operation that "advances the stream", i.e., produces a new state from the previous. In the case of `even` and `odd`, the seeds and the operations are all trivial as seen on Code 4.54.

```
first-n = "genStream ((λns.ns):[ℕ]→[ℕ]) (λ ns. (" ++ length ++ " ns) ∷ ns)
                                                                (nil : [ℕ])"

_ : eval ("head"           ++ first-n) ≡ inj₁ (Ty.List Nat , (λ γ* → []))
_ : eval ("head tail"      ++ first-n) ≡ inj₁ (Ty.List Nat , (λ γ* → 0 ∷ []))
_ : eval ("head tail tail" ++ first-n) ≡ inj₁ (Ty.List Nat , (λ γ* → 1 ∷ (0 ∷ [])))

step-n = "(λ start diff. genStream ((λn.n):ℕ→ℕ) (λn. n+diff) start)
                                                      : ℕ → ℕ → (Stream ℕ)"

_ : eval ("head           ((" ++ step-n ++ ") 10) 5") ≡ inj₁ (Nat , λ γ* → 10)
_ : eval ("head tail      ((" ++ step-n ++ ") 10) 5") ≡ inj₁ (Nat , λ γ* → 15)
_ : eval ("head tail tail ((" ++ step-n ++ ") 10) 5") ≡ inj₁ (Nat , λ γ* → 20)

_ : eval ("head           ((" ++ step-n ++ ") 0) 33") ≡ inj₁ (Nat , λ γ* → 0)
_ : eval ("head tail      ((" ++ step-n ++ ") 0) 33") ≡ inj₁ (Nat , λ γ* → 33)
_ : eval ("head tail tail ((" ++ step-n ++ ") 0) 33") ≡ inj₁ (Nat , λ γ* → 66)

get-nth = "(λ s n. head iteℕ s (λ ss. tail ss) n) : (Stream ℕ) → ℕ → ℕ"

_ : eval (get-nth ++ₛ odds  ++ₛ  "8") ≡ inj₁ (Nat , λ γ* → 17)
_ : eval (get-nth ++ₛ evens ++ₛ "13") ≡ inj₁ (Nat , λ γ* → 26)

_ : eval ("((" ++ get-nth ++ ") (((" ++ step-n ++ ") 100) 25))" ++ "3") ≡
                                                inj₁ (Nat , λ γ* → 175)
_ : eval ("((" ++ get-nth ++ ") (((" ++ step-n ++ ")   0) 30))" ++ "5") ≡
                                                inj₁ (Nat , λ γ* → 150)
```

Code 4.55: Example: arithmetic progression and parametric streams

`first-n` generates the (decreasing) sequences of first `n` natural numbers (`n ∈ ℕ`): `[]`, `[0]`, `[1,0]`, `[2,1,0]`, etc.

`step-n` is a function that takes two naturals, `start` and `diff`, and returns a stream that generates the arithmetic progression, which starts at `start` and where the difference is `diff`.

Finally, `get-nth` is a function that takes a stream of naturals and an index `n`, and returns the nth element from the stream, i.e., it applies `n` number of `tail` destructors, then a `head`. For this, we use `iteℕ`: we iterate on the parameter `n`; replace `zero` with the stream parameter; and replace each `suc` with a `tail`. For example, with `n=3` the following informal rewriting happens:

```
get-nth s suc(suc(suc(zero))) →  head tail tail tail s
```

Our `Machine` type works like simple state machines with the following semantics:

- `put`: taking the current state and a number, it advances the state - *input*
- `set`: advances the state - *signal*
- `get`: produces a natural number from the current state and returns it - *output*
- `seed`: initial state which has an arbitrary type, like with `Stream`

```
sum-machine = "genMachine (λs i. s+i) (λ_.0) (λs.s) 0"

_ : eval ("get "                ++ sum-machine)           ≡ inj₁ (Nat , λ γ* →  0)
_ : eval ("get put "            ++ sum-machine ++ " 10") ≡ inj₁ (Nat , λ γ* → 10)
_ : eval ("get put put put "     ++ sum-machine ++ " 10 20 30")
                                                          ≡ inj₁ (Nat , λ γ* → 60)
_ : eval ("get put put set put " ++ sum-machine ++ " 10 20 30")
                                                          ≡ inj₁ (Nat , λ γ* → 50)
```

Code 4.56: Example: Machine for summation

Code 4.56 presents a simple example. `sum-machine`, starting from zero, adds together all numbers input with `put`. `get` can be used to extract the result and `set` to reset the sum to zero.

```
partitioned-sums = "genMachine                                                  \
\                    (λs i. if (" ++ even ++ "i) then                           \
\                      ((fst s) ,  (fst snd s)      , ((snd snd s) + i))        \
\                    else                                                       \
\                      ((fst s) , ((fst snd s) + i) ,  (snd snd s)    ))        \
\                    (λs. (" ++ not ++ "(fst s)) , (fst snd s) , (snd snd s))   \
\                    (λs. if (fst s) then (fst snd s) else (snd snd s))         \
\                    (true , 0 , 0)"

_ : eval ("get put put put put"     ++ partitioned-sums ++ "3 10 1 8") ≡
                                                           inj₁ (Nat , λ γ* → 4)
_ : eval ("get put put put put set" ++ partitioned-sums ++ "3 10 1 8") ≡
                                                           inj₁ (Nat , λ γ* → 18)
```

Code 4.57: Example: Machine that sums odd and even numbers separately

`partitioned-sums` works like `sum-machine`, except it adds together the even and odd numbers separately. Also, now the purpose of `set` is to toggle a boolean flag that determines which sum `get` will return. To implement this we used products to store a triple as internal state: *(flag, odd-sum, even-sum)*. In `put` we add to the 2nd or 3rd component of the state depending on the `even` check's result. `set` toggles the first component and does not touch the other two. `get` returns the 2nd or 3rd depending on the flag component's state.

At this point, the reader can see that constructing inner states of arbitrary complexity is possible. Also, we could support ”machine types” which have more than one input methods (like `put`), output methods (like `get`) and signals (like `set`); essentially simulating constructions like Turing machines, REPLs or even operating systems.

Chapter 5

Conclusion

5.1 Results

This thesis has introduced readers to the formalisation of a small language based on simply typed λ-calculus. We guided them through each elaboration step of the language, going from source code to syntax trees and well-typed terms. The steps include lexing, parsing, scope checking and type checking. We presented algebraic definitions and total conversions between them, formalised in Agda for a solution that is correct by construction. We picked conventional and well-known syntactic elements for both the base STLC constructions and its extensions. We also included a meta theoretic interpretation that serves as a form of semantical evaluation.

The novelty and main value of our work is the elaboration stack presented in Chapters 4.1 to 4.4. It serves both as educational material, as well as techinal and demonstrational extension for the underlying formalisation [1].

This study also explains some inductive types like lists and trees, formal definitions for their iteration, as well as implementations of standard algorithms on them. We also briefly discussed some coinductive types along their introduction and elimination rules through the `Stream` and `Machine` examples.

Our framework provides an easy-to-use interface consisting of a small set of high-level functions for users to test and experiment on STLC terms. We report errors encountered on various levels of the elaboration stack, e.g., syntax errors, scope errors and type errors.

Our codebase remains publicly accessible for anyone on GitHub and is open for future study and development [2].

5.2 Discussion

At an early stage of development, we limited our type annotations to lambda terms which we called "annotated lambdas". This meant that each variable binding in a lambda could be annotated, so we wrote `"λx:ℕ y:ℕ.x+y"` instead of `"(λx y.x+y):ℕ→ℕ"`. This was a working solution, but we later realised that lambdas are not the only terms that need type annotations. For example, the empty list `nil`, or terms of sum types like `inl true` also need annotation, otherwise it is impossible to infer their complete types. That is why we dropped our "annotated lambda" syntax and introduced the general type annotation for arbitrary terms instead.

Note that the "annotated lambda" method is similar to Church's STLC definition where abstractions have *domains*, e.g., `λx:σ.M`; opposed to Curry's one which would instead write `λx.M : σ→σ` [5].

Performance is definitely not the strong suit of our elaborator. Type checking of files with a few dozen examples, or ones with slightly more complex constructions, such as `partitioned-sums`, sometimes took up to a minute on average hardware. We suspect that this is due to the high amount of implicit arguments Agda has to lookup and substitute. It is not obvious which part of the elaboration stack would be a good target for optimisation. Benchmarking shows that parsing takes a substantial amount of time, so it might be a good start. Type checking could potentially be more efficient if we included some "optimisation steps", ones like simplification using the η-reduction principle turning `lam q [ p ] $ q` to `lam q`. This way Agda would get structurally smaller terms earlier.

Many of the topics we explained are also discussed in more detail in *Programming Language Foundations in Agda* [15]. From fundamental concepts like Peano numbers, lists or De Bruijn indices, to complex ones like type inference and big-step semantics, they cover various aspects of programming language formalisation. They also use Agda, and even though some of their constructions are built differently, readers might still find their work a useful supplementary material to ours.

The first volume of *Software foundations* [16] is another great source we recommend. They work in Coq and formalise an imperative language opposed to our functional, but they too include some level of elaboration. The book presents fully in-house implementations of total lexing and parsing in a monadic approach similar to ours with agdarsec.

5.3 Future work

We already mentioned the idea of injecting a lexer independent of agdarsec into our toolchain. The main demand for this comes from the fact that the one parametrised with the functions `keyword`, `breaking` and `default` cannot fail. This currently results in overly permissive variable names and an unused `lexical-error` case in the elaboration.

There are many aspects we could improve upon our parser. A better implementation would probably spare us some parentheses. For example, function application always requires one, like in `"((λx.x):ℕ→ℕ) 1"`. This so far looks reasonable, but looks less right once we apply more than one argument: `"(((λx y.x+y):ℕ→ℕ→ℕ) 1) 2"`. Solving this in the total parser however, is not a trivial task.

Less obvious is the fact that type inference could also be improved upon. Here we mean that a more efficient implementation would alleviate the need for many type annotations. For example, as discussed previously, it is reasonable to expect annotation for `"λx.x"` since `x` can be any type and we do not support polymorphic functions. However, if we look at `"λx.isZero x"`, it is apparent that `x` must be ℕ for the term to be type correct. Dual to this is the case where we would have to analyse not the body, but the context of the abstraction, like in `"(λx.x) 3"`. Here it is obvious that the function's type must be ℕ→ℕ. The current implementation, as shown on Code 4.28, first infers the type of the function and then checks whether the argument has the correct type, not the other way around. It might be reasonable to implement a solution that tries both ways of inference and checking to lower the number of function type annotations required.

The framework would entail a considerably more complete study if we also included a normalisation of our own. We briefly mentioned this in Chapter 4.6. It could be a future update merge from the formalisation codebase we initially forked from [1], if the normalisation ever got finished.

An even greater undertaking would be to boost the expressibility of the presented language by introducing higher order concepts like dependent types, type polymorphism or full recursion for the function space by using some form of fixpoint combinator.

Acknowledgements

First and foremost, I would like to thank my supervisor, Ambrus Kaposi. He formalised the well-typed algebra of the presented language, provided theoretic and technical help, and guided me through the challenges of writing this thesis.

Additionally, I would also like to thank my close family for staying supportive in life throughout the years of my university studies.

The document was compiled using the LaTeX template of Máté Cserép[1].

[1]https://github.com/mcserep/elteikthesis

Bibliography

[1] Ambrus Kaposi. *Type systems (course notes)*. (Accessed 2023-11-24). 2020. URL: https://bitbucket.org/akaposi/typesystems.

[2] Barnabás Zahorán. *Agda formalisation of an elaborator for a simply typed language*. (Accessed 2023-11-24). 2023. URL: https://github.com/seeker04/stlc-agda-elab.

[3] The Agda development team. *Agda wiki*. (Accessed 2023-11-24). 2020. URL: https://wiki.portal.chalmers.se/agda.

[4] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*. Vol. 9. Bibliopolis Naples, 1984.

[5] Morten Heine B Sørensen and Pawe l Urzyczyn. "Curry-Howard Isomorphism". In: *Studies in Logic and the Foundations of Mathematics* 149 (1998), pp. 77–101.

[6] The Agda Community. *Agda Standard Library*. Version 1.7.2. (Accessed 2023-11-24). Feb. 2023. URL: https://github.com/agda/agda-stdlib.

[7] Guillaume Allais. "agdarsec–Total parser combinators". In: *of: Boldo, Sylvie, & Magaud, Nicolas (eds), JFLA* (2018), pp. 45–59.

[8] Niccolò Veltri and NM van der Weide. "Guarded recursion in Agda via sized types". In: (2019).

[9] Ambrus Kaposi. "Levels of abstraction when defining type theory in type theory". Talk given at Conference "Celebrating 90 Years of Gödel's Incompleteness Theorems" in Nürtingen, Germany. Slides: https://akaposi.github.io/pres_godel90.pdf. Video: https://youtu.be/8qRrN6ewSYw (Accessed 2023-11-24). July 2021. URL: https://akaposi.github.io/pres_godel90.pdf.

[10] Nicolaas Govert De Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)*. Vol. 75. 5. Elsevier. 1972, pp. 381–392.

[11] Jana Dunfield and Neel Krishnaswami. "Bidirectional typing". In: *ACM Computing Surveys (CSUR)* 54.5 (2021), pp. 1–38.

[12] Thorsten Altenkirch et al. "Quotient inductive-inductive types". In: *International Conference on Foundations of Software Science and Computation Structures*. Springer International Publishing Cham. 2018, pp. 293–310.

[13] Simon Castellan, Pierre Clairambault, and Peter Dybjer. "Categories with families: Unityped, simply typed, and dependently typed". In: *Joachim Lambek: The Interplay of Mathematics, Logic, and Linguistics* (2021), pp. 135–180.

[14] Gordon D Plotkin. *A structural approach to operational semantics*. Aarhus university, 1981.

[15] Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. (Accessed 2023-11-24). Aug. 2022. URL: `https://plfa.inf.ed.ac.uk/22.08/`.

[16] Benjamin C. Pierce et al. *Logical Foundations*. Ed. by Benjamin C. Pierce. Vol. 1. Software Foundations. Version 6.5, http://softwarefoundations.cis.upenn.edu (Accessed 2023-11-24). Electronic textbook, 2023.

List of Codes

1.1 Introductory examples of compilation and evaluation results 3
4.1 Portion from our vocabulary of tokens 11
4.2 Decidability of token equivalence 12
4.3 Mapping strings to our token type 12
4.4 Special tokens that also work as separators 12
4.5 Fallback function for words that are not keywords or separators . . . 13
4.6 Reading natural numbers from lists of characters 13
4.7 Syntax of STLC 15
4.8 Syntax for type annotations 16
4.9 Parsing exact tokens, parenthesised terms and variable names 16
4.10 Parsing type annotations 17
4.11 Parsers of some nodes in our AST 18
4.12 p-subexp and p-exp parsers 19
4.13 Monadic total implementation of the parsing stack 20
4.14 Parsing examples 20
4.15 Syntax for abstract binding trees of STLC 21
4.16 Lemma: an ABT can always be embedded into a bigger context . . . 22
4.17 Arithmetic lemmas needed for scope checking 22
4.18 Scope inference algorithm 23
4.19 Scope checking algorithm 24
4.20 Scope checking examples 24
4.21 Types, contexts and terms in the well-typed syntax 25
4.22 Decidable equality of types 26
4.23 Mapping syntactic types SType to Ty 26
4.24 Helper functions for type checking 26
4.25 Types of functions infer and check 27

4.26 Type inference: some trivial cases 28
4.27 Type inference: if-then-else and addition 28
4.28 Type inference: abstraction, application and variable lookup 29
4.29 Type inference and checking: lists 29
4.30 Type checking: functions and sums 30
4.31 Type inference and checking: annotated terms 31
4.32 The substitution calculus 32
4.33 Quotients of abstraction and application 33
4.34 Terms for De Bruijn indexed variables 33
4.35 Quotients of natural numbers 34
4.36 Quotients of booleans 34
4.37 Proof of semantic equivalence using equational reasoning 35
4.38 Rewriting rules for interpretation 36
4.39 Portions from the standard model 37
4.40 Top-level return types of the elaboration 38
4.41 Combining the elaboration stack 38
4.42 Elaboration example 39
4.43 Type for errors 40
4.44 User-level functions built on top of elaborate 40
4.45 Example: Alpha equivalence and unrolling lambda notation 41
4.46 Example: even and odd functions 41
4.47 Example: xor function 42
4.48 Example: constructing products 42
4.49 Example: destructing products 43
4.50 Example: implementing curry and uncurry 43
4.51 Example: list operations 44
4.52 Example: evaluation of list operations 45
4.53 Example: tree operations and their evaluation 46
4.54 Example: infinite streams of even and odd numbers 46
4.55 Example: arithmetic progression and parametric streams 47
4.56 Example: Machine for summation 48
4.57 Example: Machine that sums odd and even numbers separately . . . 48